UNIX manuals
books

Bell Laboratories Cover Sheet for Technical Memorandum

The information contained herein is for the use of employees of Bell Laboratories and is not for publication (see GEI 13.9-3)

Title: **2dsh — An Experimental Shell For Connecting Processes With Multiple Data Streams**

Date: **August 1, 1980**

TM: **80-9323-3**

Other Keywords: **UNIX shell**
command languages
software tools

Author(s)	Location	Extension	Charging Case:	Filing Case:
Marc J. Rochkind	**WH 1D301A**	**2773**	**49110-055**	**40094-16**

ABSTRACT

The standard UNIX shell *sh* provides the | operator for building a linear pipeline of commands. *2dsh* extends this idea by providing operators for building nonlinear pipelines. For example, this *2dsh* program compares the contents of two directories:

```
(ls dir1, ls dir2) | mdiff
```

(*mdiff* is a modified *diff*(1) that accepts two standard inputs).

To do this with *sh* would require a temporary file:

```
ls dir2 >tmp; ls dir1 | diff - tmp; rm tmp
```

2dsh complements, but does not replace, *sh*. It is intended only for setting up pipelines—not for programming or as a time-sharing command language. Hence, while many *sh*-like facilities are included (parameter substitution, file-name generation, quoting, redirection and inline documents), many others are omitted (flow-of-control statements, command substitution, prompting and special commands).

This memo assumes the reader is thoroughly familiar with *sh*.

Pages Text: 11	Other: 0	Total: 11
No. Figures: 2	No. Tables: 0	No. Refs.: 1

TM-80-9323-3
9/23
MCILROY,M DOUGLAS CM
MH2C526 09/04/80
YOU ARE REFERENCE SOURCE

Bell Laboratories

subject: **2dsh — An Experimental Shell For Connecting Processes With Multiple Data Streams**
Case: **49110-055**
File: **40094-16**

date: **August 1, 1980**

from: **Marc J. Rochkind**
WH 9323
1D301A x2773

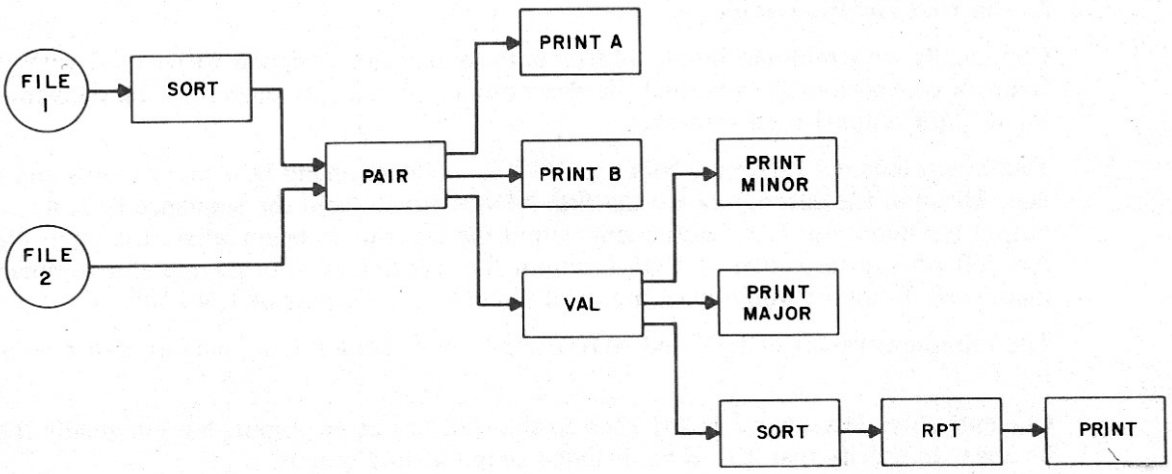
TM: **80-9323-3**

MEMORANDUM FOR FILE

1. Introduction

Consider this problem: You have two files, "file1" and "file2" containing records of type A and type B, respectively. "file1" is to be sorted and then passed along with "file2" through the program *pair* which pairs up type A and B records that match according to some criterion. *pair* has 3 outputs: matched pairs (each pair merged into a single record), type A records that didn't match, and type B records that didn't match. The latter two outputs are to be printed. The good output is to be passed through a data validator, which has three outputs: valid records, records with minor errors, and records with major errors. The erroneous records are to be printed (in separate reports); the valid records are to be sorted, passed through a report generator, and the report is to be printed.

The following diagram illustrates the flow of data:



Now assume that you want to connect the programs with the UNIX* shell, *sh*(1).† Since each program has a single standard input and a single standard output, those programs with multiple inputs (e.g., *pair*) or outputs (e.g., *val* or, again, *pair*) will have to use files for their additional

* UNIX is a trademark of Bell Laboratories.

† The notation "(1)" is a reference to section 1 of the UNIX User's Manual.

inputs and outputs. You'll have to create temporary files, arrange for their removal, ensure that enough disk space is available, and so on. Moreover, the connection of outputs to inputs may not be very clear from the listing of the shell procedure, since only a file name ties the two together.

2dsh is an experimental shell that provides syntax for specifying the connection of processes with multiple inputs and outputs directly. The example problem would be specified to *2dsh* like this:

```
(sort file1,
 <file2
) | pair | (pr -h A | lpr,
              pr -h B | lpr,
              val | (pr -h minor | lpr,
                     pr -h major | lpr,
                     sort | rpt | lpr
                    )
             )
```

Given this specification, *2dsh* will create a process for each command and connect appropriate inputs and outputs with pipes. When *pair* is executed, it inherits 2 input file descriptors and 4 output file descriptors. The first input (file descriptor 0) is the reading end of a pipe connected to the standard output of *sort*; the second input (file descriptor 3) is opened to the file "file2". One of the outputs (file descriptor 2) is the standard error output. The other three (file descriptors 1, 4 and 5) are the writing ends of pipes connected to the two instances of *pr* and to *val*.

The linear pipelines (e.g., the ones connecting instances of *pr* and *lpr*) are set up as they would be with *sh*.

val has one input (file descriptor 0) which is the reading end of the pipe connected to the fourth output of *pair*. *val* has 4 outputs: the standard error output and 3 pipes (file descriptors 1, 4 and 5) connected to 2 instances of *pr* and to *sort*.

2. Inherited File Descriptors

Commands with multiple inputs and/or outputs that are designed to be used with *2dsh* must follow a convention for inherited file descriptors that is an extension of the convention (0 for input, 1 for output) used with *sh*.

Two environmental variables, NIN and NOUT, tell a command how many inputs and outputs it has. Its input file descriptors are the first NIN numbers from the sequence 0, 3, 4, ... Its first output file descriptor is 1; subsequent output file descriptors begin where the input file descriptors left off, up to a total of NOUT output file descriptors. For example, a command with 2 inputs and 3 outputs would have inputs of 0 and 3, and outputs of 1, 4 and 5.

The minimum values of NIN and NOUT are 1, so existing filters (such as *sort*) may be used as is.

Currently, file descriptor 2 is left open to the standard error output, but eventually there might be a way to specify that it is to be included as the second output.

So that programmers need not know what the actual file descriptors are, two functions may be used to get arrays of file descriptors:

```
int *fdin, *fdout, *getfdin(), *getfdout();
...
fdin = getfdin();
fdout = getfdout();
```

After these calls, each array contains the number of file descriptors in its first element, and the file descriptors themselves in subsequent elements. For example, if fdin[0] is 4, the input file

descriptors are given by fdin[1], fdin[2], fdin[3] and fdin[4].

3. Explanation of Syntax

The *2dsh* | operator causes the connection of the command *group* on its left to the group on its right. If each group is a single command, as in

```
who | wc
```

then the meaning is the same as with *sh*.

However, either group may consist of several commands separated by commas, with the entire group in parentheses. Such a group specifies something about the group on the *other* side of the |: the number of its inputs or outputs. For example, in

```
(a, b) | c
```

the command *c* has two inputs because the group on its left consists of two commands, *a* and *b*.

Similarly, in

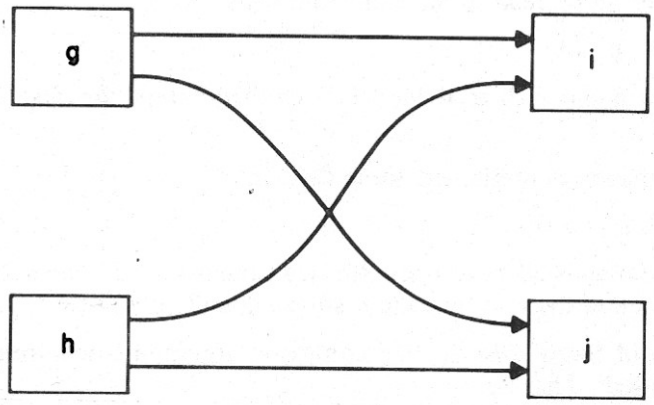
```
d | (e, f)
```

the command *d* has two outputs.

And in

```
(g, h) | (i, j)
```

the commands *g* and *h* each have two outputs, to *i* and *j*. And *i* and *j* each have two inputs, from *g* and *h*. There are a total of 4 pipes, as the following diagram shows:



The order of multiple inputs and outputs is significant and preserved by *2dsh*.

The syntax for connecting commands is recursive. For example, in

```
a | (b | c, d) | e
```

the command *a* inputs to *b* and *d*, *b* inputs to *c*, and *c* and *d* input to *e*.

2dsh itself has no limit on the complexity of a specification, but since it has only 20 file descriptors to play with, it can't get enough pipes for any group-to-group connection bigger than 4 by 3. That is, it can do

```
(a, b, c, d) | (e, f, g)
```

but not

```
(a, b, c, d) | (e, f, g, h)
```

Long linear strings of commands are no problem, since *2dsh* is careful about conserving its resources (mostly file descriptors), but some UNIX systems limit the number of processes available to a user.

4. Redirection

Three built-in commands may be used to redirect input and output. The command < causes the group on the *other* (repeat, *other*) side of the | to have its corresponding input redirected. Similarly, the command > causes the group on the other side to have an output redirected. Finally, >> is an appending version of >. Each of these built-in commands takes a single file name argument. For example,

```
echo hello there | >output
```

to *2dsh* is like

```
echo hello there >output
```

to *sh*.

In

```
(a, <f1, b, <f2) | c | (>f3, d) | e
```

the command *c* has 4 inputs: the output of *a*, file "f1", the output of *b* and file "f2". *c* also has 2 outputs (in addition to the standard error output): to file "f3" and to *d*. But *e* has only *one* input: from *d*. Since ">f3" cannot be an input to *e*, it is ignored when the inputs to *e* are counted.

The same file can be read by several commands. In

```
<f1 | (a, b, c)
```

commands *a*, *b* and *c* all read file "f1" (with different file descriptors, so each sees the entire file).

Several commands can write the same file. In

```
(a, b, c) | >>f1
```

the three commands all have open file descriptors to "f1", and all three file descriptors are initially positioned at the end, but the results will still be a mess.*

I'm thinking of fixing *2dsh* so that command arguments beginning with a redirection character are special-cased. That is,

```
a >f1 | (b, c)
```

would be taken as

```
a | (>f1, b, c)
```

This would make *2dsh* more like *sh*.

A fourth redirection built-in command is << which is similar to the *sh* facility for inline documents. *2dsh* treats << exactly like it does <, except that the data comes from a *2dsh*-generated temporary file instead of from a file named by the user.

5. Faking Multiple Inputs

Several UNIX commands would work well with *2dsh* if they took multiple inputs. Obvious examples are *cat*(1), *comm*(1), *diff*(1), *join*(1) and *sort*(1). Eventually, *2dsh* versions of these (actually, they could go either way, depending on whether NIN were defined), should be written, but meanwhile the command *mfake* may be used to fake the multiple inputs. *mfake* reads

* UNIX 3.0 has an open mode that causes the file pointer to be set to the end of the file prior to each write, thus avoiding the mess.

each input in its entirety and writes it to a temporary file. Then it invokes the command given as its arguments (like *time*(1)) and removes the temporaries when the command terminates. Since each input has to be read before the command is invoked, *mfake* can't simulate the ability of a real *2dsh* command to start processing as soon as it sees some data (this is no problem with *sort*, but it hurts with, say, *paste*(1)).

mfake assumes that its argument command accepts "—" as the first input file and leaves the standard input alone. The "—" convention is used by most UNIX commands of interest.

For example, the following prints detailed information on who is logged in:

```
(who | sort,
 </etc/passwd | tr ':' ' ' | sort
) | mfake join
```

If *mfake* is invoked under the name "m*cmd*", it acts as though "mfake *cmd*" had been invoked; hence, the following commands are really just links to *mfake*:

```
mjoin
msort
mdiff
mcat
mcmp
mpaste
```

The example of Section 1 may be recoded like this:

```
(sort file1,
 <file2
) | pair | (pr -h A,
             pr -h B,
             val | (pr -h minor,
                    pr -h major,
                    sort | rpt
                   )
            ) | mfake lpr
```

Here *mfake* would have 5 inputs, and *lpr* would be given 5 arguments: "—" and 4 temporary file names.*

6. 2dsh Compatible Commands

The rich set of filters available on UNIX contributes to the power of *sh* programming at least as much as *sh* itself does. Similarly, one needs a resonably complete set of multiple-stream filters for *2dsh* to be useful. At present, this set is quite small, but I hope that it will grow through *2dsh*-user contributions. Many existing filters (such as those mentioned in Section 5) can be modified or generalized; others might be invented specifically for use with *2dsh*.

As a start, I've programmed one new command, *mgrep*, which takes regular expressions as arguments and has a single input. There is one output for each regular expression. Each input line is written to each output whose regular expression matches it. For example, the following lists C files in "c.out", object files in "o.out" and all files in "all.out":

```
ls | mgrep '\.c$' '\.o$' '.' | (>c.out, >o.out, >all.out)
```

mgrep requires the number of regular expression to match the number of outputs.

---

* Actually, you can't use "mfake lpr" because *lpr* doesn't accept "—" as a file name. This glitch could be easily fixed by adding a "files only" option to *mfake*.

7. Forcing Multiple Connections

Sometimes you want two commands to be connected multiply, but you can't use the normal *2dsh* syntax. For example, the following prints C file names and object file names in two columns:

```
ls | mgrep '\.c$' '\.o$' | (cat, cat) | mfake paste
```

The "(cat, cat)" part is necessary to match the two outputs of *mgrep* with the two inputs of *paste*. Otherwise, *mgrep* would complain, and *paste* wouldn't know it had two inputs.

Instead of using *cat* for this purpose, the no-op command — may be used:

```
ls | mgrep '\.c$' '\.o$' | (—, —) | mfake paste
```

This is also more efficient, since the two *cat*s are actually executed, whereas the — command just causes pipes to be bridged.

The no-op is also needed for unbalanced connections:

```
ls | mgrep '\.o$' '\.c$' | (—, sort) | mfake paste
```

8. Bidirectional Connections: Tags

Suppose you have an *ed* preprocessor, called *pre*, that takes input from the terminal, processes it, sends it to *ed*, processes the response and prints it. The following won't work, because each *pre* would be a different process:

```
pre | ed | pre
```

However, you can tag a command and refer to the tag later:

```
p: pre | ed | p
```

Here *ed* inputs from and outputs to the same process running *pre*. Unfortunately, *pre* also has its standard input and output routed to and from *ed*, because an explicit input (or output) has priority over a standard one. So you might try this:

```
— | p: pre | ed | p | —
```

but this won't do because the first output from *pre* is to *ed*, but it should be the second output. Turning it inside out does the trick:

```
(—, e: ed) | pre | (—, e)
```

Bidirectional connections are not only tricky to set up—they are also prone to deadlock unless the participating processes follow the appropriate protocol. The problem results from the fact that reading an empty pipe puts a process to sleep. For example, if *pre* attempts to read too much from *ed*, it will deadlock, because *ed* will not output anything more until *pre* has given it more input.

To avoid deadlock, one of these techniques can be used:

- Precede each transmission by a count or other indication of the amount of data to be read.
- Exchange fixed data objects (one line, one structure, etc.).
- Follow each transmission with an end-of-message flag.

The last technique can often be used with *ed* by operating it with prompting turned on; *pre* reads until it sees the prompt character.

Deadlock can also occur with unidirectional connections. For example, assume that process A is connected to process B by two pipes, and that B reads from its inputs alternately. Deadlock can occur if A attempts to write all of the data destined for one of its outputs without writing any data to the other output, since A will be put to sleep when the first pipe fills. B can't empty the first pipe because it is sleeping on the empty second pipe.

There is no simple solution to this deadlock problem, nor any easy way to detect it (without modifying the operating system). Users must bear in mind that *2dsh* only does the plumbing; it has no responsibility for the way the plumbing is used.

9. Debugging

To help see what the connections are, a **—d** option may be used. The output for the example at the end of Section 5 is:

```
sort(0):
        0 inputs:
        1 outputs: pair(2)
<(1):
        0 inputs:
        1 outputs: pair(2)
pair(2):
        2 inputs: sort(0) <(1)
        3 outputs: pr(3) pr(4) val(5)
pr(3):
        1 inputs: pair(2)
        1 outputs: mfake(10)
pr(4):
        1 inputs: pair(2)
        1 outputs: mfake(10)
val(5):
        1 inputs: pair(2)
        3 outputs: pr(6) pr(7) sort(8)
pr(6):
        1 inputs: val(5)
        1 outputs: mfake(10)
pr(7):
        1 inputs: val(5)
        1 outputs: mfake(10)
sort(8):
        1 inputs: val(5)
        1 outputs: rpt(9)
rpt(9):
        1 inputs: sort(8)
        1 outputs: mfake(10)
mfake(10):
        5 inputs: pr(3) pr(4) pr(6) pr(7) rpt(9)
        0 outputs:
```

Parenthesized numbers identify processes; they are useful in distinguishing between the same process occurring twice and two different processes executing the same program.

The **—d** option may be used to determine that the example

```
— | p: pre | ed | p | —
```

is wrong:

```
pre(1):
        2 inputs: stdin ed(2)
        2 outputs: ed(2) stdout
ed(2):
        1 inputs: pre(1)
        1 outputs: pre(1)
```

and that the example

```
(—, e: ed) | pre | (—, e)
```

is right:

ed(1):
    1 inputs: pre(2)
    1 outputs: pre(2)
pre(2):
    2 inputs: stdin ed(1)
    2 outputs: stdout ed(1)

10. sh-like Features

2dsh has many features similar to those of *sh*. The following subsections, which follow the *sh*(1) manual page in the UNIX User's Manual, describe the similarities and differences. Most *sh* features that are not implemented will never be implemented; *2dsh* is intended to be neither a time-sharing, interactive shell nor a programming language.

10.1 Commands

Arguments are like *sh*.

Pipelines are as described above. Newlines are normally ignored.

A list is a sequence of pipelines separated by semicolons (the other *sh* separators are not implemented). *2dsh* waits for the entire pipeline to complete before processing the next pipeline or terminating.

No flow-of-control commands (**for**, **case**, **if** and **while**) are implemented. Parentheses have a different meaning. Curly braces are not implemented.

10.2 Comments

Comments have a **#** in column 1.

10.3 Command Substitution

Not implemented.

10.4 Parameter Substitution

The forms **$***digit*, **$***parameter* and **${***parameter***}** are implemented. Parameter values are taken from the environment only; there is no way to assign values to them within *2dsh*. Parameter replacement text is rescanned.

The special parameters **#**, **–**, **?**, **$** and **!** are not implemented.

No parameters (HOME, PATH, etc.) are used by *2dsh* (but the operating system uses PATH).

10.5 Blank Interpretation

As in *sh* (I think!).

10.6 File Name Generation

As in *sh*.*

10.7 Quoting

As in *sh*, except that the accent grave (`) is not quotable with a backslash (\).

10.8 Prompting

Not implemented (*2dsh* is not intended for interactive use).

10.9 Input/Output

The forms < *word*, > *word*, >> *word* and <<[–] *word* are implemented (but they occur in a different place; see above). A document is quoted by quoting the entire word in single quotes.

---

* I used a file-name-generation routine written by Steve Bourne.

10.10 Environment

As in *sh*, but note that many associated features are absent (**export**, **set**, **—k**, assignment, etc.).

10.11 Signals

As in *sh* (but note that **&** is not implemented).

10.12 Execution

As in *sh*.

10.13 Special Commands

Not implemented.

10.14 Invocation

No *sh* flags or other *sh* invocation features are implemented. *2dsh* has a **—d** flag for printing a summary of the pipeline architecture.

11. Examples

This section presents some simple *2dsh* examples.

11.1 Extended Who

This example augments the output of *who* with information from the password file:

```
(who | sort,
 awk '—F:' '
 {
        print $1 "\t" $5
 }' /etc/passwd | sort
) | mjoin
```

Typical output is:

```
3722el tty22 Jun 2 08:47 3722-LIPINSKI,E.T(WH2244)W016
3722gac tty23 Jun 2 10:23 3722-CZUPRAWSKI,G(WH8883)W163
3722jk tty10 Jun 2 08:56 3722-KOLB,J.(WH3441),W016
3722tjm tty15 Jun 2 10:13 3722-MACPHERSON,T(WH2135)W045
3722wam tty31 Jun 2 08:55 3722-MEYER,W(WH8902)W246
```

11.2 Three-Column Listing

This example lists file names in three columns: object files, C files and files whose names end in a digit:

```
ls | mgrep '\.o$' '\.c$' '[0—9]$' | (—, —, —) | mpaste
```

Typical output is:

```
2dshl.o        2dshmain.c     demo2
2dshmain.o     expand.c       demo3
2dshy.o        getfd.c        demo4
expand.o       getopt.c       demo5
getfd.o        mfake.c        demo6
getopt.o       mgrep.c        demo7
mfake.o        p1.c           demo8
mgrep.o        p2.c           p1
prcs.o         prcs.c         p2
whatfd.o       whatfd.c
```

11.3 Conversation

This example connects two programs, *p1* and *p2*, conversationally:

```
(<<—' !'
hello
there
everyone
!

,
p: p2
) | p1 | (—,
          p
         )
```

Program *p1* is:

```
#include <stdio.h>

main()
{
        int *fdin, *fdout, *getfdin(), *getfdout();
        char s[200];
        FILE *in, *out;

        fdin = getfdin();
        fdout = getfdout();
        in = fdopen(fdin[2], "r");
        out = fdopen(fdout[2], "w");
        setbuf(in, NULL);
        setbuf(out, NULL);
        while (gets(s)) {
                fputs(s, out);
                fputc('\n', out);
                fgets(s, sizeof(s) — 1, in);
                printf("%s", s);
        }
}
```

Program *p2* is:

```
#include <stdio.h>

main()
{
        char s[200];

        setbuf(stdin, NULL);
        setbuf(stdout, NULL);
        while (gets(s))
                printf("`%s' echoed\n", s);
}
```

The output is:

```
`hello' echoed
`there' echoed
`everyone' echoed
```

12. Conclusion

2dsh is still quite new, so its utility and significance are not fully known. I built it originally because I was disatisfied with several relational database systems that required the user to make up numerous temporary relations—this seems to me to be like programming in assembly

language. Also, systems that assume that a relation exists in its entirety at some instant cannot handle streams of indefinite length.

Explicitly telling a shell what the interconnections are has other advantages. One of them is that a single option to *2dsh* could specify tracing: the data flowing through each pipe would be recorded on a temporary file for later study. This is trivial to add to *2dsh*; doing it with a conventional shell would require discovering the interconnections, which is impossible in general.

Another advantage is optimization: two frequently combined commands could be repackaged to share an address space, and *2dsh* could use the combined form when possible. Again, this is impossible for a conventional shell.

Another optimization is replacing pipes with other interprocess communication mechanisms, such as shared memory or FIFOs. Since a *2dsh* specification only states *what* interconnections are to be made, but not *how*, alternate interconnection mechanisms could be used. The changes to *2dsh* to substitute another mechanism besides pipes would be fairly easy. In fact, the connections between processes could even be across machines, but this facility would be far from easy to implement, since the underlying hardware and operating system software is underdeveloped.

13. Related Work

Many people before me have thought about adding nonlinear plumbing to *sh*, but *2dsh* is the only such working shell of which I am aware. Note, however, that I did not solve the problem of adding syntax to *sh*; *2dsh* is a totally new shell that is definitely not an extension of *sh*.

After I completed *2dsh* and wrote the first draft of this memorandum, Doug McIlroy directed me to some notes he wrote over a year ago.* His proposal and *2dsh* are remarkably similar, in syntax and semantics. I take the existence of our two independent inventions as evidence that *2dsh* is a step in the right direction.

WH-9323-MJR-mjr

Marc J. Rochkind

* M. Douglas McIlroy. *A Notation for Arboreal Plumbing*. (Unpublished memorandum.)