1. Introduction.

A designer requires a variety of component specifications when creating schematics, printed circuit boards (PCBs) and manufacturing documentation. This program is a specification server, called `compsrv`, that executes commands sent via a socket. A client sends this server a command and receives a reply. Commands and replies use ASCII strings.

A couple of things that I like about this client-server approach:

- The interaction between the client and server is achieved using ASCII strings. Building commands and parsing strings is trivial.
- Communication is via a socket. The client and server need not be on the same machine.
- Upgrading or testing a new server is performed by stopping the old server and starting a new server.
- Manual testing and debug of the server can be performed with a simple terminal program.

To use this program:

1. Start the component server.
2. Open a socket to the server.
3. Send a command string to the server.
4. Receive and parse the returned string.

☡ *This program was built using a literate programming tool I am working on which is called* `pweb`*. My tool is modeled after the Knuth tool called* `cweb`*. Non-catastrophic infelicities should be considered bugs. Sections are numbered, hyperlinks to sections are in red and a list of sections is at the end of the document.*

2. Operation.

Currently the only specifications this server retrieves are datasheet filenames.

This server sends back a list of datasheet filenames (or an error message) for each `get_datasheet` message that is received. The required command arguments are `manufacturer` and `manufacturer_part_number`.

The message string consists of a command string, followed by a vertical bar followed by an argument list. The argument list is one or more key-value pairs with the key and value separated by an equal sign. Pairs are separated by verical bars.

`get_datasheet | manufacturer=`⟨*name*⟩ `| manufacturer_part_number=`⟨*part number*⟩ `\r\n`

3. Component Specifications Hash.

The component specifications hash is used to store and retrieve component specifications. The key for the hash is built using the name of the manufacturer and a part number for the component. To retrieve a component the `manufacturer` name must match exactly. If the `manufacturer_part_number` does not match exactly then regular expressions associated with the specified `manufacturer` will be tested.

The hash table is populated using data contained in an ASCII file. A data record consists of a group of non-blank lines followed by a blank line. Each non-blank line contains a single key-value pair with the key and value separated by an equal sign. Table 1 lists the valid keys. Listing 1 shows an example datafile.

Key	Description	Notes
`manufacturer`	Company name	One value per record
`manufacturer_part_number`	Part number	
`regex`	Perl regular expression	Used to match part numbers if there are no exact matches for `manufacturer_part_number`.
`datasheet`	Datasheet filename	One or more values per record

Table 1: Key-value Pairs for the Hash Table

Listing 1: Example Datafile

```
[datasheet]
manufacturer=TI
manufacturer_part_number=74ACT14
manufacturer_part_number=74LS14
datasheet=/local/pub/dataheets/texas-instruments/74xx14-datasheet.pdf
datasheet=http://ti.com/74xx14-datasheet.pdf
regex=^74\D*14$
```

4. Shebang, a couple of my favorite pragmas and the usual suspects. `IO::Socket` and `Net::hostent` are used to create a socket, receive commands and send replies.

```
#!/usr/bin/perl -w
use strict;
use warnings;
use Carp;
use Data::Dumper;
use Getopt::Long;
use IO::Socket;
use Net::hostent;
```

5. Get the program command line options. The options are listed in table 2.

Option	Default Value	Description
`port`	`9000`	communication port
`datafile`	`compsrv.data`	file containing component specifications

Table 2: Command-line Options

```
my %Opt;
GetOptions("port=i"      ⇒  \$Opt{port},
           "datafile=s"  ⇒  \$Opt{datafile});
$Opt{port} = 9000                unless defined $Opt{port};
$Opt{datafile} = 'compsrv.data'  unless defined $Opt{datafile};
```

6. Create the server object.

```
my $Server = IO::Socket::INET→new( Proto     ⇒  'tcp',
                                   LocalPort ⇒  9000,
                                   Listen    ⇒  SOMAXCONN,
                                   Reuse     ⇒  1);
```

```
die "can't setup server" unless $Server;
print "[Server $0 accepting clients at http://localhost:$Opt{port}/]\n";
```

7. Create and initialize hashes. The `%Subs` hash contains subroutine references for each of the acceptable commands. The `%Comp` hash contains the component specifications that are read from the datafile.

```
my %Subs;
my %Comp;
⟨initialize the command subroutine hash 10⟩;
&read_compsrv_data($Opt{datafile}, \%Comp);
```

8. Main loop. The main loop listens on the assigned address and port and runs the appropriate subroutine for each received command.

```
my $Client;
while ($Client = $Server→accept()) {
    $Client→autoflush(1);
    my $hostinfo = gethostbyaddr($Client→peeraddr);
    printf "[Connect from %s]\n", $hostinfo→name || $Client→peerhost;
    while ( <$Client>) {
        if (s/\\\s*$//) {    # Remove the continuation backslash and
            $_ .= <>;        # append the next line to $_ then
            redo unless eof; # restart the loop block after the conditional
        }
        print "(compsrv) skipping received string that does not contain a \\r\\n\n" unless /\r\n/;
        next unless s/\r\n//s;
        s/^\s*//; # Remove leading spaces
        s/\s*$//; # Revove trailing spaces
        print($Client "error: received string had zero length\r\n"), next unless length;
        ⟨parse the received line and run the appropriate subroutine 9⟩;
    }
    close $Client;
}
```

9. The message string consists of a command string followed by a argument list. The argument list is specified as key-value pairs with the key and value separated by an equal sign. The command and argument list is separated by a vertical bar.

The following procedure splits the message string into a command and argument list. An error is returned if the command is undefined or there is no subroutine defined for the command.

```
⟨parse the received line and run the appropriate subroutine 9⟩≡
my ($cmd, @arg) = split /\s*\|\s*|\s*=\s*/;
printf("cmd = %s\n", defined $cmd ? $cmd : 'undef');
print($Client "error: undefined command in received string\r\n"), next unless defined $cmd;
print($Client "error: no procedure for command $cmd\r\n"), next unless defined $Subs{$cmd};
no strict qw(subs);
print $Client $Subs{$cmd}→(@arg);
use strict qw(subs);
```

10. Command processing. Each allowed command that is received calls a subroutine that is defined in the `%Subs` hash. The key in the hash is the command name and the value is a reference to the subroutine to be called.

```
⟨initialize the command subroutine hash 10⟩≡
my %Subs = (get_datasheet  ⇒  \&get_datasheet,
            help           ⇒  \&help);
```

11. The help command returns the name of each allowed command followed by a short description.

```
sub help {
    my $helpmsg = ⇐ 'END';
get_datasheet ... returns the name of the manufacturers datasheet
help ............ command summary
END
    return("$helpmsg\r\n")
}
```

12. The `get_datasheet` commands returns a list of datasheet filenames for the specified component. The elements in the list are separated by vertical bars.

The following conditions return an error message:

- no command parameters
- missing parameter value
- manufacturer name was not defined
- manufacturer part number was not defined
- unknown manufacturer (the name was not in the datafile)
- no datasheet found

```
sub get_datasheet($) {
    my (@arg) = @_;
    return("error: no defined parameters for get_datasheet command\r\n")
        if $#arg ≡ -1;
    return("error: missing value for parameter $arg[-1]\r\n")
        unless $#arg % 2;
    my %arg = (@arg);
    return("error: value for manufacturer was not defined\r\n")
        unless defined $arg{manufacturer};
    return("error: value for manufacturer_part_number was not defined\r\n")
        unless defined $arg{manufacturer_part_number};
    my $comp = $Comp{datasheet};
    return("error: unkown manufacturer '$arg{manufacturer}'\r\n")
       unless defined $comp→{$arg{manufacturer}};
    $comp=$comp→{$arg{manufacturer}};
    my @datasheets;
    if (defined $comp→{$arg{manufacturer_part_number}}) {
      @datasheets = @ { $comp→{$arg{manufacturer_part_number}} };
    } else {
        $comp=$comp→{_regex};
        foreach my $regex (keys %$comp) {
            next unless $arg{manufacturer_part_number} =˜ /$regex/;
            @datasheets = @ { $comp→{$regex} };
```

```
                last;
            }
        }
        return(sprintf("%s\r\n", join('|', @datasheets))) unless $#datasheets ≡ -1;
        return("error: no datasheet for $arg{manufacturer} $arg{manufacturer_part_number}\r\n");
    }
```

13. The component data hash (`%Comp`) is populated by parsing the contents of an ascii file.

```
    sub read_compsrv_data {
        my ($filename, $hashref) = @_;
        my $datatype;
        my %rec;
        open(IN, "$filename") or die "Could not open $filename for input: $!";
        while (<IN>) {
            s/\#.*//; #  Remove comments
            s/^\s*//; #  Remove leading spaces
            s/\s*$//; #  Revove trailing spaces
            &add_datasheet_rec($datatype,\%rec), next unless length; #  Skip empty lines
            last if /^__END__$/; #  Skip lines after the end marker
            if (s/\\\s*$//) {         #  Remove the continuation backslash and
                $_ .= <IN>;           #  append the next line to $_ then
                redo unless eof(IN); #  restart the loop block after the conditional
            }
            $datatype = $1, next if /^\s*\[(.*)\]\s*$/;
            next unless defined $datatype;
            ⟨split current line into key-value pairs and add to the current record 14⟩;
        }
        close(IN);
        &add_datasheet_rec($datatype,\%rec);
    }
```

14. Add values to the current record.

```
    ⟨split current line into key-value pairs and add to the current record 14⟩≡
    my @key_value_pairs = split /\s*=\s*|\s*\|\s*/;
    while (@key_value_pairs) {
        my ($k, $v) = splice @key_value_pairs, 0, 2;
        next unless defined $k;
        next unless defined $v;
        $rec{$k} = [$v], next unless defined $rec{$k};
        push @ { $rec{$k} }, $v;
    }
```

15. The `add_datasheet_rec` subroutine adds a datasheet record to the component specification hash `%Comp`. The format of the record is:

`datasheet` → ⟨*mfg*⟩ → `_regex` → ⟨*regex*⟩ → ⟨*list of datasheets*⟩

`datasheet` → ⟨*mfg*⟩ → ⟨*part number*⟩ → ⟨*list of datasheets*⟩

```
    sub add_datasheet_rec ($$) {
        my ($datatype, $ref) = @_;
        return unless defined $datatype;
        return unless $datatype eq 'datasheet';
        return unless defined $ref→{manufacturer};
        my $mfg = $ref→{manufacturer}[0]; #  should be only one
        my $ds  = [ @ { $ref→{datasheet} } ];
        foreach my $mfg_pn (@ { $ref→{manufacturer_part_number} }) {
            $Comp{datasheet}{$mfg}{$mfg_pn} = $ds;
        }
        foreach my $regex ( @ { $ref→{regex} } ) {
            $Comp{datasheet}{$mfg}{_regex}{$regex} = $ds;
        }
        %$ref = ();
    }
```

16. Style.

Adapted from the Perl Cookbook, First Edition, Recipe 12.4

- Names of functions and local variables are all lowercase.
- The program's persistent variables (either file lexicals or package globals) are capitalized.
- Identifiers with multiple words have each of these separated by an underscore to make it easier to read.
- Constants are all uppercase.
- If the arrow operator (`->`) is followed by either a method name or a variable containing a method name then there is a space before and after the operator.

17. License.

No-Fee Software License Version 0.2

Intent

The intent of this license is to allow for distribution of this software without fee. Usage of this software other than distribution, is unrestricted.

License

Permission is granted to make and distribute verbatim copies of this software provided that (1) no fee is charged and (2) the original copyright notice and this license notice are preserved on all copies.

Permission is granted to make and distribute modified versions of this software under the conditions for verbatim copying provided that the entire resulting derived work is distributed under terms of a license identical to this one.

This software is provided by the author "AS IS" and any express or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed. In no event shall the author be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this software, even if advised of the possibility of such damage.

18. Code Sections

⟨parse the received line and run the appropriate subroutine 9⟩ Used in section 8
⟨initialize the command subroutine hash 10⟩ Used in section 7
⟨split current line into key-value pairs and add to the current record 14⟩ Used in section 13