Bijlage 34
Perl and TEX a simple application

Gilbert van den Dobbelsteen
`gilbert@login.iaf.nl`

abstract
A simple application where perl is used to extract data from log-files and create output using TEX. The perl-scripts in this article run under perl 4.036 and should also run under perl 5.

1 Introduction

I am a software engineer programming embedded systems. Embedded systems are found everywhere in your environment, From a programmable calculator to dish-washers, coffee-machines, electric razors and GSM telephones.

Most of the time our projects involve communications with another system, usually a PC. Communication software usually involves one or more large state-machines which should ensure reliable transmission and reception. But since it's still software an implementation could be faulty. Timers could be to tight or retransmissions occur to frequently with no cause or the checksum algorithm fails on some weird occasion.

Since communications (and real-time error-free communications in specific) are difficult to debug we use log files where usually the raw data on the link is logged. The log-files are kept simple and straight and usually they are hard to read for normal humans.

Last year I engineered a project for Douwe Egberts Coffee Systems which involved communications with several other parties. The protocol used was a standard protocol used by many parties in the vending industry. The protocol is used to configure the vending machine and to read out the transactions made. Our company made a data-logger and some other party included similar functionality inside their vending machine.

Both parties made a simple PC for verifying the communications protocol. The specifications were not clear on all points so sometimes we needed to make choices in the implementation. After we both finished our product and our tools, we exchanged software and hardware to cross-verify each-others efforts.

That's where the trouble started. Their tool couldn't talk with our product, and vice-versa. So what to do? The log-files where extensive and I didn't had the time to spend hours explaining to the other party what was going on. So I decided to create a perl script which extracted vital information from the log-file and generated a file that could easily be processed by TEX.

I had some experience with perl and text-processing so this would be reasonably easy to do. It seems to me that perl is very well suited for data processing and TEX is a decent tool to create visual output.

2 Log-files

Almost all the programs we develop generate log-files. These are handy when a client calls with some problem. If it can't be supported by phone I ask them to email the log-file. Examining the log-files usually unfolds a bug in the software or a bug in somebody else's software.

Since most applications deal with communications I always have trouble examining the log-files. Sometimes the information is so large it is possible to overlook an error.

Here's an example of the data from the log-file:

```
> 04 05 01 56 A8 FA
< C4 67 13 62 E3 CC 00 00 2A 3F
```

The angular brackets indicate the direction of the transfer. Examining this data is not easy because most log-files are thousands of lines in size.

Perl is not only an excellent tool for processing this type of data, it can do more. It can actually analyze the protocol and point out some positions where the data is not right. This is very handy if the log-file is a few megabytes in size. You could even use colored output (in red?) for pin-pointing potential errors.

3 Basic protocol stuff

Skip this section if you're not interested, it contains some background information.

So what's a communication protocol? How do you transfer data from one system to antoher and make sure no errors occur? Many books have been written on this subject, and many examples are available. I'll stick with some simple things in a point-to-point[1] connection.

First of all you need to detect if the data you receive is correct. The easiest way to do this is to add redundant information. We always use cyclic redundancy checks. This

1. point-to-point: In this article a communication link between two computers.

is a reliable method of determining if received data is correct. Incorrect data is simply discarded.

Then comes the real trouble: When you send data, you want to make sure your data arrives at the other side. How to do that? Although this is easy, many machines I've seen have an incorrect implementation. The solution is to send an acknowledge message back, informing the sender the data arrived OK. The sender then sends the next available data.

Several things can go wrong here. The acknowledgement could get lost due to some error on the link (for example you manually disconnected the modem). The sender is waiting for an acknowledgement and when it does not arrive within a certain amount of time it simply re-transmits the data. This can go wrong too. Suppose you have a very slow link[2]. The acknowledgement arrives, but it is simply too late. The sender retransmits the data and you end up with the same data twice. When the data contains a command such as *turn on the coffee machine* this is no problem. The coffee machine is turned on while it was already on. But if the data contains a command like *pour a can of coffee* you end up with two cans of coffee when in fact you wanted only one. You can imagine what mess this gives since the person requesting a can of coffee expected only one.

How to solve that? Again this is simple. And it is the last thing I'll tell you about protocols. Make sure each data packet carries a unique number. When you re-transmit the data it contains that same number. The acknowledments from the receiver also return the number of the data acknowledged. So now the sender sends *1: pour a can of coffee*, the receiver replies: *ack:1* and pours a can of coffee. If the acknowledgement is lost, the sender re-transmits: *1: pour a can of coffee*. The receiver *knows* it has already seen data #1 so it simply sends back *ack:1* and does *not* pour another can of coffee. Is this simple or what? The unique numbers can easily be generated by a counter, which is incremented each time new data is sent. The receiver also has a counter so it knows which number should be the next to receive. Data arriving with another number is simply acknowledged and discarded.

So that is a simple way to create error-free communications. There are several other aspects but I won't bother you with the details.

4 Processing log-files

See the PERL script below for processing the data.

```
while($line = <STDIN>) {
  if($line =~ /^\>.*/) {
    &process_right(substr($line, 1));
  }
  elsif($line =~ /^\<.*/) {
    &process_left(substr($line, 1));
  }
}
```

The above script processes input from standard input and calls the function `process_right` if the line starts with a > or `process_left` if the line starts with <. Before the call is made, it strips of the first character using `substr` (similar to basic's `MID$`). Don't be alarmed by amount of rubbish present. The $ sign indicates a simple string variable, the & indicates a function call. The forward slashes indicate a regular expression. The rest looks like a C program.

So what to do now? I'll give the details of the `process_right` function, the `process_left` is omitted since it is similar:

```
sub process_right{
  local($data) = @_;
  local(@values);
  local($local_ns,$local_nr);

  $data =~ s/^\s+//; # omit leading white space

  @values = split(/[\s\n]+/, $data);

  # @values is an array with the seperate hexa-
  # decimal values. Convert them to decimal.

  foreach $value (@values){
    $value = hex($value);
  }

  if($values[0] == 1){     # START command?
    print "\\r {START}\n";
  }elsif($values[0] == 2){ # ACK   command?
    $local_nr = $values[1];
    print "\\r {ACK $local_nr}\n";
  }elsif($values[0] == 3){ # NACK  command?
    $local_nr = $values[1];
    print "\\r {NACK $local_ns}\n";
  }elsif($values[0] == 4){ # DATA
    $local_ns = $values[1];
    print "\\r {DATA $local_ns}\n";
  }else{                   # Unknown command
    print "\\r {UNKNOWN}\n";
    &print_data("\\rt", @hexvalues);
  }
}
```

I omitted the various checking functions here which are present in the actual perl-script. This is just to illustrate how to do such things. First the leading white-space is stripped with a regular expression. Then `$data` is converted into

2. I know you use internet so you know what I'm talking about.

an array of `values`. Arrays in perl are arrays of scalars (variables that start with a $-sign). Scalars can be strings or numbers. When you apply an operation, perl automagically converts them to the correct type.

So now we are left with an array of hexadecimal values. This is not too bad, perl provides the `hex` gunction for conversion. The `foreach` statement converts the values to decimal.

After that the program finds out what the command is and prints a converted line on standard out. That's all there is to it. Note: you can use variables directly in print strings, perl expands the variables for you.

5 Output

The output of the script contains lines like this:

```
\STARTPROTOCOL
\l {START}
\lt{Start of protocol}
\r {STACK}
\rt{Ackowledge}
\STOPPROTOCOL
```

So the arrow drawing stuff is left to TEX. These are simple macro's:

```
\startTEX
\def\BOXSIZE{\hsize}
\def\STARTPROTOCOL{\bgroup
  \def\l##1{\hbox to \BOXSIZE{\leftarrowfill
                              \ {\tt ##1}}}
  \def\r##1{\hbox to \BOXSIZE{{\tt ##1}\
                              \rightarrowfill}}
  \def\lt##1{\hbox to \BOXSIZE{\hfill ##1}}
  \def\rt##1{\hbox to \BOXSIZE{##1\hfill}}
  \obeylines }
\def\STOPPROTOCOL{\egroup}
```

6 Example

Below is an example of a complete and correct communication session:

RECEIVER *SENDER*

←———————————————— `START`

Communication start

`STACK` ————————————————→

Acknowledgement

←———————————————— `WhoAreYou`

Sec: 0x2342, Pass: 0x7326

`WhoAreYou Accepted` ————————————————→

Serial#: 385682445

←———————————————— `FINISH`

`ACK` ————————————————→

The `WhoAreYou` command identifies the terminal, and after it is accepted configuration commands can be given.

Here's an extensive example with several errors in it:

RECEIVER *SENDER*

←———————————————— `START`

Communication start

`STACK` ————————————————→

Start Acknowledged

←———————————————— `START`

Communication start (**duplicate!**)

`STACK` ————————————————→

Start Acknowledged (**duplicate!**)

←———————————————— `DATA 1`

WhoAreYou

←———————————————— `DATA 1`

WhoAreYou (**retry!**)

`ACK 1` ————————————————→

`DATA 1` ————————————————→

WhoAreYou Accepted)

←———————————————— `ACK 1`

←———————————————— `DATA 2`

Finish

`NACK 1` ————————————————→

(Finish) not received!

←———————————————— `DATA 2`

Finish

`ACK 2` ————————————————→

As you can see the tool finds errors which are hard to spot by humans. The `STACK` response is not received by the sender so the `START` command is sent again. The `WhoAreYou` command is not received by the receiver so the sender sends a retry. Near the end, the `Finish` command was received but it contained an error. This is signalled by the `NACK`. The sender retries the operation.

7 Conclusion

Perl is a good tool to process ASCII data. When used in combination with TEX, the tool can create nice output. The power of perl is completely unlike C. Although it looks a lot like C it has much more power. In specific the string processing is supurb. For almost anything you want there is a perl internal function available. And if there's not you can roll your own.

I didn't discuss perl 5 which adds many more features and object oriented programming. Perhaps I'll discuss that the next time. Perl is also well equipped for systems programming. It is possible to write a complete web-server in perl.

The program described in this article helped us locating bugs quick and easy. It's a convenient way to look at data. Since the original specifications included similar diagrams, verifying the diagrams was easy too.