A Web-Server in Forth

Bernd Paysan

October 27, 2001

Abstract

An HTTP-Server in Gforth is presented as an opportunity to show that you can do string-oriented things with Forth as well. The development time (a few hours) shows that Forth is an appropriate tool for this kind of work and delivers fast results.

This paper was originally presented at the Forth Tagung 2000 conference in Hamburg, later translated to English (proofreading and corrections by Chris Jakeman), and for the presentation on EuroForth 2001 I added the description of the transparent proxy extension.

1 Introduction

Since I have always given bigFORTH/MINOS-related presentations in the last few years, I'll do something with Gforth this time. Gforth is another tool you can do neat things with, and in contrast to what you here elsewhere, Forth is suitable for almost anything. Even a web-server.

In this age of the "new economy", the Internet is important. Everybody is "in there" except Forth, which hides in the embedded control niche. There isn't any serious reason for that. The following code was created in just a few hours of work and mostly operates on strings. The old prejudice, that Forth was good at biting bits, but has troubles with strings, is thus disproved.

1.1 Motivation

What do you need a web-server for in Forth? Forth is used for measurement and control in remote locations such as the sea-bed or the crater of a volcano. Less remotely, Forth may be used in a refrigerator and, if that stops working, things soon get messy. So a communication thingy is built in.

How much better would it be if instead of "some communication thingy built in", there was a standard protocol. HTTP is accessible from the web-cafe in Mallorca, or from mobile yuppie toys such as PDAs or cell phones. Perhaps one should build such a web-server into each stove and into the bath, so that people can use their cell phone on holidays to check repeatedly (every three minutes?) if they *really* turned their stove off.

Anyway, the customer, boss or whoever buys the product, wants to hear that there is some Internet-thingy build in, especially if one isn't in *e*-Business already. And the costs must be zero too, because there's really no money in the "new economy".

But let's take this slowly, step by step.

2 A Web Server, Step by Step

Actually, you had to study the RFC[1]-documents. The RFCs in question are RFC 945 (HTTP/1.0) and RFC 2068 (HTTP/1.1), which both refer to other RFCs. Since these documents alone are much longer than the source code presented below (and reading them would take longer than writing the sources), we will defer that for later. The web server thus won't be 100% RFC conforming (i.e. implement all features), and conforms only as far as necessary for a typical client like Netscape. However additions are easy to achieve.

A typical HTTP-Request looks like this:

```
GET /index.html HTTP/1.1
Host: www.paysan.nom
Connection: close

```

(Note the empty line at the end). And the response is

```
HTTP/1.1 200 OK
Date: Tue, 11 Apr 2000 22:27:42 GMT
Server: Apache/1.3.12 (Unix) (SuSE/Linux)
Connection: close
Content-Type: text/html

<HTML>
...
```

This looks quite trivial, so let's start. The web server should run under Unix/Linux. That takes one problem out of our hands - how we get to our socket - since that's what *inetd*, the Internet daemon, does for us. We only need to tell it on which port our web server expects data, and enter that into the file `/etc/inetd.conf`:

```
# Gforth web server
gforth stream tcp nowait.10000 wwwrun
    /usr/users/bernd/bin/httpd
```

[1] Request For Comments — the documents of Internet standards are called like that.

We won't replace the default web server just yet (something might not work straight away), so we shall need a new port and that one goes into the file `/etc/services`:

```
gforth  4444/tcp # Gforth web server
```

When we do a restart or a `killall -HUP inetd` inetd will realize the changes and starts our web server for all requests on port 4444. What we need next is an executable program. Gforth supports scripting with `#!`, as common for scripting languages in Unix. In the line below, the blank is significant:

```
#! /usr/local/bin/gforth

warnings off
```

We better disable any warning. Let's load a small string library (see attachment):

```
include string.fs
```

We shall need a few variables for the URL requested from the server, the arguments, posted arguments, protocol and states.

```
Variable url       \ URL
Variable posted    \ POST args
Variable url-args  \ URL args
Variable protocol  \ stores the protocol
Variable data      \ true: return data
Variable active    \ true for POST
Variable command?  \ true: request line
```

A request consist of two parts, the request line and the header. Spaces are separators. The first word in a line is a "token" indicating the protocol, the rest of the line, or one/two words are parameters.

Since we can process a request only once the whole header has been parsed, we save all the information. Therefore we define two small words which take a word representing the rest of a line and store it in a string variable:

```
: get ( addr -- )  name rot $! ;
: get-rest ( addr -- )
  source >in @ /string dup >in +!
  rot $! ;
```

As told above, we have header values and request commands. To interpret them, we define two wordlists:

```
wordlist constant values
wordlist constant commands
```

But before we can really start, the URL might contain spaces and other special characters, what to do with them? HTTP advises to transmit these special characters in the form %xx, where xx are two hex digits. We thus must replace these characters in the finished URL:

```
: rework-% ( add -- ) { url }
  base @ >r hex
  0 url $@len 0 ?DO
      url $@ drop I + c@ dup '% = IF
          drop 0. url $@ I 1+ /string
          2 min dup >r >number r> swap - >r 2drop
      ELSE  0 >r  THEN  over url $@ drop + c!  1+
  r> 1+ +LOOP  url $!len
  r> base ! ;
```

So, that's done. But stop! URLs consist of two parts: path and the optional arguments. Separator is '?'. So first split the string into two parts:

```
: rework-? ( addr -- )
  dup >r $@ '? $split url-args $! nip r> $!len ;
```

So we've defined the basics and can start. Each requests fetches a URL and the protocol, splits the URL into path and arguments and replaces the special character glyphs by the real characters (but those in the arguments remain as we don't yet know what should happen to them). Finally, we must switch over to another vocabulary, since the header follows after the request.

```
: >values ( -- )  values 1 set-order command? off ;
: get-url ( -- )  url get protocol get-rest
  url rework-? url rework-% >values ;
```

So now we can define the commands. According to the RFC, we only need GET and HEAD, POST is then a bonus.

```
commands set-current

: GET   get-url data on  active off ;
: POST  get-url data on  active on  ;
: HEAD  get-url data off active off ;
```

And now for the header values. Since we need a string variable for each value, and otherwise want only to store the string, we build that with `CREATE-DOES>`. Again: we need a variable *and* a word, which stores the rest of the line there. In two different vocabularies. The latter with a colon behind.

Fortunately, Gforth provides `nextname`, an appropriate tool for this. We construct exactly the string we need and call `VARIABLE` and `CREATE` afterwards

```
: value:  ( -- )  name definitions
  2dup 1- nextname Variable values set-current nextname
  here cell - Create , definitions  DOES> @ get-rest ;
```

And now we set to work and define all the necessary variables:

```
value: User-Agent:
value: Pragma:
value: Host:
value: Accept:
value: Accept-Encoding:
```

```
value: Accept-Language:
value: Accept-Charset:
value: Via:
value: X-Forwarded-For:
value: Cache-Control:
value: Connection:
value: Referer:
value: Content-Type:
value: Content-Length:
```

There are some more (see RFC), but those are all we need at the moment.

2.1 Parsing a Request

Now we must parse the request. This should be completely trivial, we could just let the Forth interpreter chew it but for one little caveat:

1. Each line ends with CR LF, while Gforth under Unix expects lines to end with an LF only. We thus must remove the CR. And

2. each header ends with an empty line, not some executable Forth word. We thus must read line for line with `refill`, remove CRs from the line end, and look then if the line was empty.

```
Variable maxnum

: ?cr ( -- )
  #tib @ 1 >= IF  source 1- + c@ #cr = #tib +!  THEN ;
: refill-loop ( -- flag )
  BEGIN  refill ?cr  WHILE  interpret  >in @ 0=  UNTIL
  true  ELSE  maxnum off false  THEN ;
```

So, the key things are done now. Since we can't let the Forth interpreter loose on the raw input stream *stdin*, we preprocess the stream ourselves. We initialize a few variables which we need to interpret anyway, and steal some code from `INCLUDED`:

```
: get-input ( -- flag ior )
  s" /nosuchfile" url $! s" HTTP/1.0" protocol $!
  s" close" connection $! infile-id push-file
  loadfile ! loadline off  blk off commands
  1 set-order  command? on  ['] refill-loop catch
```

Waiiiit! The request isn't done here. The method `POST`, which was added as bonus, expects the data now. The length fortunately is stored as base 10 number in the field "`Content-Length:`".

```
  active @ IF
      s" " posted $! Content-Length $@ snumber? drop
      posted $!len posted $@ infile-id read-file
      throw drop
  THEN  only forth also  pop-file ;
```

2.2 Answer a Request

OK, we've handled a request, and now we must answer. The path of the URL is unfortunately not as we want it: we want to be somehow Apache compatible, i.e. we have a "global document root" and a subdirectory in the home directory of each user, where he can put his personal home page. Thus we can't do anything else but look at the URL again and finally check, if the requested file really is available:

```
Variable htmldir

: rework-htmldir ( addr u -- addr' u' / ior )
  htmldir $!
  htmldir $@ 1 min s" ~" compare 0=
  IF    s" /.html-data" htmldir dup $@
        2dup '/ scan nip - nip $ins
  ELSE  s" /usr/local/httpd/htdocs/"
        htmldir 0 $ins  THEN
  htmldir $@ 1- 0 max + c@ '/ =
  htmldir $@len 0= or
  IF  s" index.html" htmldir dup $@len $ins  THEN
  htmldir $@ file-status nip ?dup ?EXIT
  htmldir $@ ;
```

Next, we must decide how the client should render the file — i.e. which MIME type it has. The file suffix is all we need to decide, so we extract it next.

```
: >mime ( addr u -- mime u' )
  2dup tuck over + 1- ?DO
  I c@ '. = ?LEAVE  1-  -1 +LOOP
  /string ;
```

Normally, we'd transfer the file as is to the client (transparent). Then you tell the client how long the file is (otherwise, we'd have to close the connection after each request). We open a file, find its size and report that to the client.

```
: >file ( addr u -- size fd )
  r/o bin open-file throw >r
  r@ file-size throw drop
  ." Accept-Ranges: bytes" cr
  ." Content-Length: " dup 0 .r cr r> ;
: transparent ( size fd -- ) { fd }
  $4000 allocate throw swap dup 0 ?DO
      2dup over swap $4000 min
      fd read-file throw type
      $4000 - $4000 +LOOP  drop
  free fd close-file throw throw ;
```

We do all the work with `transparent`, using `TYPE` to send the file in chunks to support "keep-alive" connections, which modern web browsers prefer. The creation of a new connection is significantly more "expensive" than to continue with an established one. We benefit on our side also, since starting Gforth again isn't for free either. If the connection is keep-alive, we return that, reduce maxnum by one, and report to the client how often he may issue further requests. When it's the last request, or no further are pending, we send that back, too.

```
: .connection ( -- )
  ." Connection: "
  connection $@ s" Keep-Alive"
  compare 0= maxnum @ 0> and
  IF  connection $@ type cr
      ." Keep-Alive: timeout=15, max="
      maxnum @ 0 .r cr -1 maxnum +!
 ELSE  ." close" cr maxnum off  THEN ;
```

Now we just need some means to recognize MIME file suffixes and sned the appropriate transmissions. For the response, we must also first send a header. We build it from back to front here, since the top definitions add their stuff ahead. To make the association between file suffixes and MIME types easy, we simply define one word per suffix. That gets the MIME type as string. `transparent:` does all that for all the file types that are handled using `transparent`:

```
: transparent: ( addr u -- )
  Create  here over 1+ allot place
  DOES>  >r  >file
  .connection ." Content-Type: "
  r> count type cr cr
  data @ IF  transparent
  ELSE  nip close-file throw  THEN ;
```

There are hundreds of MIME types, but who wants to enter all of them? Nothing could be easier than this, we steal the MIME types that are already known to the system, say from `/etc/mime.types`. The file lists the mime type on the left paired with the file suffixes on the right (sometimes none).

```
: mime-read ( addr u -- )
  r/o open-file throw  push-file
  loadfile !  0 loadline ! blk off
  BEGIN  refill  WHILE  name
      BEGIN  >in @ >r name nip  WHILE
          r> >in ! 2dup transparent:
      REPEAT
      2drop rdrop
  REPEAT
  loadfile @ close-file pop-file throw ;
```

One more thing we need: for active content we want to use server side scripting (in Forth, of course). Since we don't know the size of these requests in advance, we don't report it but close the connection instead. That relieves us of the problem of cleaning up the trash the user is creating with his active content (that's Forth code!).

```
: lastrequest
  ." Connection: close" cr maxnum off
  ." Content-Type: text/html" cr cr ;
```

So let's start with the definition of MIME types. Get a new wordlist. Active content ends with `shtml` and is `included`. We provide a few special types and the rest we get from the system file mentioned above. For unknown file types, we need a default type, `text/plain`.

```
wordlist constant mime
mime set-current

: shtml ( addr u -- )  lastrequest
  data @ IF  included
  ELSE  2drop  THEN ;

s" application/pgp-signature" transparent: sig
s" application/x-bzip2" transparent: bz2
s" application/x-gzip" transparent: gz
s" /etc/mime.types" mime-read

definitions

s" text/plain" transparent: txt
```

2.3 Error Reports

Sometimes a request goes wrong. We must be prepared for that and respond with an appropriate error message to the client. The client wants to know which protocol we speak, what happened (or if everything is OK), who we are, and in the error case, a error report in plain text (coded in HTML) would be nice:

```
: .server ( -- ) ." Server: Gforth httpd/0.1 ("
  s" os-class" environment? IF  type  THEN  ." )" cr ;
: .ok  ( -- ) ." HTTP/1.1 200 OK" cr .server ;
: html-error ( n addr u -- )
  ." HTTP/1.1 " 2 pick . 2dup type
  cr .server  2 pick &405 =
  IF ." Allow: GET, HEAD, POST" cr  THEN
  lastrequest
  ." <HTML><HEAD><TITLE>"  2 pick . 2dup type
  ." </TITLE></HEAD>" cr
  ." <BODY><H1>" type drop ." </H1>" cr ;
: .trailer ( -- )
  ." <HR><ADDRESS>Gforth httpd 0.1</ADDRESS>" cr
  ." </BODY></HTML>" cr ;
: .nok ( -- )  command? @
  IF  &405 s" Method Not Allowed"
  ELSE  &400 s" Bad Request"  THEN
   html-error
  ." <P>Your browser sent a request that this server "
  ." could not understand.</P>" cr
  ." <P>Invalid request in: <CODE>"
  error-stack cell+ 2@ swap type
  ." </CODE></P>" cr .trailer ;
: .nofile ( -- )  &404 s" Not Found" html-error
  ." <P>The requested URL <CODE>" url $@ type
  ." </CODE> was not found on this server</P>"
  cr .trailer ;
```

2.4 Top Level Definitions

We are almost done now. We simply glue together all the pieces above to process a request in sequence - first fetch the input, then transform the URL, recognize the MIME type, work on it including error exits and default paths. We need

to flush the output, so that the next request doesn't stall. And do that all over again times, until we reach the last request. I added a feature to allow transparent redirects (i.e. subdirectories are fetched from another web server using a proxy).

```
: http ( -- )
  get-input  IF  .nok  ELSE
  IF  url $@ 1 /string rework-htmldir
      dup 0< IF  drop .nofile
      ELSE  .ok  2dup >mime
          mime search-wordlist
          0= IF  ['] txt  THEN
          catch IF  maxnum off THEN
  THEN  THEN  THEN
  outfile-id flush-file throw ;

: httpd  ( n -- )  maxnum !
  BEGIN  ['] http catch
         maxnum @ 0= or  UNTIL ;
```

To make Gforth run that at the start, we patch the boot message and then save the result as a new system image.

```
script? [IF]  :noname &100 httpd bye ;
              is bootmessage  [THEN]
```

2.5 Scripting

As a special bonus, we can process active content. That's really simple: We just write our HTML file as usual and indicate the Forth code with "<$" and "$> " (the space for the closing parenthesis is certainly intentional!). Let's define two words, $>, and to get the whole thing started, <HTML>:

```
: $> ( -- )
  BEGIN  source >in @ /string s" <$" search  0= WHILE
      type cr refill  0= UNTIL  EXIT  THEN
  nip source >in @ /string rot - dup 2 + >in +! type ;
: <HTML> ( -- )  ." <HTML>" $> ;
```

That's quite enough, we don't need more. The rest is all done by Forth, as in the following example:

```
<HTML><HEAD>
<TITLE>GForth <$ version-string type
$>  presents</TITLE></HEAD><BODY>
<H1>Computing Primes</H1><$ 25 Constant #prim $>
<P>The first <$ #prim . $> primes are: <$
: prim?  0 over 2 max 2 ?DO
     over I mod 0= or  LOOP  nip 0= ;
: prims ( n -- )  0 swap  2
  swap 0 DO  dup prim? IF  swap
             IF  ." , "  THEN  true swap
  dup 0 .r 1+  1  ELSE  1+  0  THEN
  +LOOP  drop ;
#prim prims $> .</P>
</BODY>
</HTML>
```

3 Redirection via a Proxy

A feature that was added later was redirecting accesses to certain directories to another web host. The intention was to create a web-page that allowed to take a snapshot of all online resources about Forth with a simple `wget` on the top-level. The changes to the toplevel word are minimal — we just add a check for redirection, and perform a redirect word if that check succeeds.

```
Defer redirect? ( addr u -- t / f )
Defer redirect ( -- )
:noname 2drop false ; IS redirect?
: http ( -- )
  get-input  IF  .nok  ELSE
  IF  url $@ 1 /string 2dup
      redirect? IF redirect 2drop ELSE
      rework-htmldir
      dup 0< IF  drop .nofile
      ELSE  .ok  2dup >mime
          mime search-wordlist
          0= IF  ['] txt  THEN
          catch IF  maxnum off THEN
  THEN  THEN  THEN  THEN
  outfile-id flush-file throw ;
```

The first thing we need is a `open-socket ( addr u port -- fd )` function. It will take a hostname and a port, and delivers a file descriptor.

```
require unix/socket.fs
```

Since the HTTP protocol expects carriage return and line feed, `write-line` is not sufficient. I add a `writeln` that provides the necessary line end.

```
Create crlf #cr c, #lf c,
: writeln ( addr u fd -- )
  dup >r write-file throw crlf 2 r> write-file throw ;
```

3.1 Requesting a Page

We will only pass the user agent through, not the other variables. A more advanced variant would do that, too.

```
: request ( host u request u proxy-host u port -- fid )
    open-socket >r r@ write-file throw
    s"  HTTP/1.1" r@ writeln
    s" Host: " r@ write-file throw r@ writeln
    s" Connection: close" r@ writeln
    s" User-Agent: " r@ write-file throw
    User-Agent @ IF
        User-Agent $@ r@ write-file throw
        s" via Gforth Proxy 0.1"
    ELSE  s" Gforth Proxy 0.1"  THEN
    r@ writeln s" " r@ writeln r> ;
```

There are two ways to access a web page: Either directly, or via a proxy. Many firewalls prevent people from direct access, so we have to support a intermediate proxy. Fortunately, the request is the same, just the target is replaced:

instead of directly calling the target on port 80, we call the proxy on it's port (typically 3128). A good sysop will provide the proxy under the name `proxy`, if yours is less friendly, replace the string in the proxy variable. Also, the typical port used varies, 3128 is default for squid, 8080 is another commonly used port.

```
Variable proxy       s" proxy" proxy $!
Variable proxy-port  3128 proxy-port !
```

The two possible ways to talk just differ in how they call `request`.

```
: proxy-open ( host u request u -- fid )
    proxy $@ proxy-port @ request ;
: http-open ( host u request u -- fid )
    2over 80 request ;
```

3.2 Gathering the Response

Now we have to gather the response. It's the same format our own server delivers, but there can be more answers. Most of the responses will be filtered out, too. I use the same technique to digest the response as I used above to digest a request. There's a wordlist for the first line of the response, and a second one for the variables. They are defined with `response:` ⟨*name*⟩, and just like the normal protocol variables, they will be visible with : for assignment in the `response-values` wordlist, and without (for access) in the standard wordlist.

```
wordlist Constant response
wordlist Constant response-values
Variable response$
: response:  ( -- )  name
    Forth definitions 2dup 1-
    nextname Variable
    response-values set-current
    nextname here cell - Create ,
DOES> @ get-rest ;
: >response  response-values 1 set-order ;
```

The responses we expect are either HTTP version 1.0 or 1.1:

```
response set-current
: HTTP/1.1 response$ get-rest >response ;
: HTTP/1.0 response$ get-rest >response ;
```

And the following are the accepted variables:

```
Forth definitions
response: Allow:
response: Age:
response: Accept-Ranges:
response: Cache-Control:
response: Connection:
response: Proxy-Connection:
response: Content-Base:
response: Content-Encoding:
response: Content-Language:
response: Content-Length:
response: Content-Location:
response: Content-MD5:
response: Content-Range:
response: Content-Type:
response: Date:
response: ETag:
response: Expires:
response: Last-Modified:
response: Location:
response: Mime-Version:
response: Proxy-Authenticate:
response: Proxy-Connection:
response: Public:
response: Retry-After:
response: Server:
response: Transfer-Encoding:
response: Upgrade:
response: Via:
response: Warning:
response: WWW-Authenticate:
response: X-Cache:
response: X-Powered-By:
Forth definitions
```

I now can reuse some of the code from above to handle the response, since the basics are already defined in `refill-loop`.

```
: get-response ( fid -- ior )  push-file
  loadfile !  loadline off  blk off
  response 1 set-order
  ['] refill-loop catch
  only forth also  pop-file ;
```

3.3 Handling the Respose Data

After getting in the header, I have to handle the data. Unlike our web-server, we can not be sure that the target will deliver the data just as is. It's possible that it transfers it with a given size or until the connection is closed (just like we do), or with a chunked protocol, where each chunk is preceeded by a line specifying the size. The dispatcher is `read-data`, it selects between the three methods depending on Content-Length and Transfer-Encoding.

```
Variable data-buffer
: clear-data ( -- )
    s" " data-buffer $! ;
: add-chunk ( u fid -- u' )
    swap data-buffer $@len dup >r +
    data-buffer $!len
    data-buffer $@ r@ /string
    rot read-file throw
    dup r> + data-buffer $!len ;
: read-sized ( u fid -- )
    add-chunk drop ;
: read-to-end ( fid -- )
    >r BEGIN  $1000 r@ add-chunk
              $1000 <> UNTIL  rdrop ;
: read-chunked ( fid -- )
   base @ >r hex >r
    BEGIN  pad $100 r@ read-line throw
```

```
    WHILE
        pad swap s>number drop dup
    WHILE  r@ add-chunk drop
        pad 1 r@ read-line throw
    nip 0= UNTIL
    ELSE  drop  THEN  THEN
    rdrop r> base ! ;
: read-data ( fid -- ) clear-data >r
    Content-Length @ IF
        Content-Length $@ s>number drop
        r> read-sized  EXIT  THEN
    Transfer-Encoding @ IF
        Transfer-Encoding $@
        s" chunked" compare 0= IF
            r> read-chunked  EXIT  THEN
    THEN
    r> read-to-end ;
```

3.4 Forwarding the Response

After receiving the data from the original site, we forward it (as a single piece with a now defined size) to the receiver. First, we write the header, second, we write the data.

```
: write-response ( -- )
    .ok
    ." Connection: close" cr
    ." Accept-Ranges: bytes" cr
    ." Content-Type: "
    Content-Type $@ type cr
    ." Content-Length: "
    data-buffer $@len 0 .r cr cr ;
: write-data ( -- )
    data-buffer $@ type ;
```

The toplevel definition to handle a request then takes the file handle to the proxy or the web-server, asks it to deliver the page, and forwards the result.

```
: handle-request ( fid -- )
    dup >r get-response throw
    r@ read-data r> close-file throw
    write-response write-data ;
```

3.5 Redirection Definitions

Redirects also use a vocabulary structure to convert pseudo-directories into real target addresses. You can add a subdirectory by using a vocabulary, and a tail directory (that marks a redirection) with `redirect:`. This word takes a server name and a request string (the request string typically contains the server name, I leave the work to extract it to the user). There's a trick to parse '/' as separator for the interpreter: I just replace it with the line-feed character, which is treated as white-space, but impossible to embed into any normal line. It must therefore be reverted to '/' again to form the rest of the path.

```
wordlist Constant redirects
Variable redir$
Variable host$
: redirect: ( "path" host<"> url<"> -- )
  Create
  '" parse here over char+ allot  place
  '" parse here over char+ allot  place
DOES> ( -- addr u )
  data @ IF s" GET " ELSE s" HEAD " THEN
  redir$ $!
  count 2dup host$ $! +
  count redir$ $+!
  source >in @ /string dup >in +!
  2dup bounds ?DO
     I c@ #lf = IF  '/ I c!  THEH  LOOP
  redir$ $+! redir$ $@ ;
```

I have deferred two words above, the one asking if there's a redirection necessary, and the other to perform the redirection itself. The first converts the '/' to line-feeds (as said above), and evaluates the resulting string in the `redirects` context. If there's a redirection found, the `redir$` variable will contain something. If the line can't be interpreted, we can stop here. There's a bit of a danger here, because literals still might be converted.

```
: (redirect?) ( addr u -- t / f )
  htmldir $! htmldir $@ bounds ?DO
     I c@ '/ = IF  #lf I c!  THEN  LOOP
  redirects 1 set-order redir$ $off
  htmldir $@ ['] evaluate catch
  IF  false  ELSE  redir$ @ 0<>  THEN ;
```

The remaining request is simple. Just call `proxy-request`, and turn off `maxnum`, so that the handler will stop. And we have to assign the deferred words.

```
: (redirect) ( -- )
  host$ $@ redir$ $@ proxy-request
  maxnum off ;
' (redirect?) IS redirect?
' (redirect) IS redirect
```

The example given below will add subdirectory `systems` to the web directory. There, the subdirectory `bigforth` will redirect requests to `bigforth.sourceforge.net`.

```
redirects set-current
get-order redirects swap 1+ set-order
Vocabulary systems
also systems definitions
redirect: bigforth bigforth.sourceforge.net"
...http://bigforth.sourceforge.net/"
previous previous definitions
```

4 Outlook

That was a few hundred lines of code — far too much. I have delivered an "almost" complete Apache clone. That won't be necessary for the sea-bed or the refrigerator. Error handling is ballast, too. And if you restrict to single connection (performance isn't the goal), you can ignore all the protocol variables. One MIME type (text/html) is sufficient – we keep the images on another server. There is some hope that one can get a working HTTP protocol with server-side scripting in one screen.

Appendix: String Functions

Certainly we need some string functions, it doesn't work without. The following string library stores strings in ordinary variables, which then contain a pointer to a counted string stored allocated from the heap. Instead of a count byte, there's a whole count cell, sufficient for all normal use. The string library originates from bigFORTH and I've ported it to Gforth (ANS Forth). But now we consider the details of the functions. First we need two words bigFORTH already provides:

```
: delete   ( addr u n -- )
  over min >r  r@ - dup 0>
  IF  2dup swap dup  r@ +  -rot swap move
  THEN  + r> bl fill ;
```

`delete` deletes the first n bytes from a buffer and fills the rest at the end with blanks.

```
: insert ( string u buffer u -- )
  rot over min >r  r@ - ( left over )
  over dup r@ +  rot move   r> move  ;
```

`insert` inserts as string at the front of a buffer. The remaining bytes are moved on.

Now we can really start:

```
: $padding ( n -- n' )
  [ 6 cells ] Literal + [ -4 cells ] Literal and ;
```

To avoid exhausting our memory management, there are only certain string sizes; $padding takes care of rounding up to multiplies of four cells.

```
: $! ( addr1 u addr2 -- )
  dup @ IF  dup @ free throw  THEN
  over $padding allocate throw over ! @
  over >r  rot over cell+ r> move 2dup !
  + cell+ bl swap c! ;
```

`$!` stores a string at an address. If there was a string in before, this string will be released.

```
: $@ ( addr1 -- addr2 u )  @ dup cell+ swap @ ;
```

`$@` returns the stored string.

```
: $@len ( addr -- u )  @ @ ;
```

`$@len` returns just the length of a string.

```
: $!len ( u addr -- )
  over $padding over @ swap resize throw
  over ! @ ! ;
```

`$!len` changes the length of a string. Therefore we must change the memory area and adjust address and count cell as well.

```
: $del ( addr off u -- )
  >r >r dup $@ r> /string r@ delete
  dup $@len r> - swap $!len ;
```

`$del` deletes u bytes from a string with offset off.

```
: $ins ( addr1 u addr2 off -- ) >r
  2dup dup $@len rot + swap $!len
  $@ 1+ r> /string insert ;
```

`$ins` inserts a string at offset off.

```
: $+! ( addr1 u addr2 -- )  dup $@len $ins ;
```

`$+!` appends a string to another.

```
: $off ( addr -- )  dup @ free throw off ;
```

`$off` releases a string.

As a bonus there are functions to split strings up.

```
: $split ( addr u char -- addr1 u1 addr2 u2 )
  >r 2dup r> scan dup >r dup  IF  1 /string  THEN
  2swap r> - 2swap ;
```

`$split` divides a string into two, with one char as separator (e.g. '? for arguments)

```
: $iter ( .. $addr char xt -- .. ) { char xt }
  $@ BEGIN  dup  WHILE  char $split
            >r >r xt execute r> r>  REPEAT  2drop ;
```

`$iter` takes a string apart piece for piece, also with a character as separator. For each part a passed token will be called. With this you can take apart arguments — separated with '& — at ease.