Using Fortran programs as CGI scripts

[who]	[how]	[what]	[where]
Introduction	How the library routines work	Documentation provided on this page	Source code distribution

---

Introduction

In the last weeks (time of writing is 10 February 1999) I encountered a few posts on comp.lang.fortran and the comp.infosystems.www.* newsgroups dealing with usage of Fortran programs as back end for CGI executables called from HTML forms. Worth mentioning among these the examples by D.Bickel

If you do not know what CGI are refer to the NCSA tutorial.

Since it was at least a couple of years that the idea of a general library for using CGI in Fortran programs (I mean directly, not via an intermediate shell script as I did so far) was lurking in my mind, the above postings instigated me to put my hands on the keyboard and write down the code.

One more proof that Real Programmers Do It in Fortran :-)

L.Chiappetti - 11 February 1999
modified cgiheader.f 18 October 2000
modified Bickel's link 20 May 2002

How to do it

The library addresses the following problems :

- Retrieving the query string generated by the HTML form. This is handled in both the POST and the GET method. The main difference is that the POST method reads from standard input a known number of bytes passed in an environment variable. The GET method passes the query string in an environment variable without any indication of the length.
  The routine `CGIRETRIEVE` loads the query string in common and is called transparently by the first call of one of the following two parsing routines.
- The query string contains a list of variable names and values in a compact, encoded form, and must be parsed. Two parsing routines are provided.
  - The routine `GET_NEXT_VARIABLE` can be called repeatedly to parse all variables in the order in which they occur, returning the name and value.
  - The routine `GET_NAMED_VARIABLE` can be called to parse a known named variable at a time in random order, returning the value if such variable is found.
- The string components in the query string are URL-encoded (see RFC...). The decoding is handled transparently by routine `URLDECODE`, while the reverse `URLENCODE` is needed for service purposes to check the match of variable names.
- once you have decoded the variables, **you** can do whatever processing you like ...
  ... that's not **my** business :-)
- to produce HTML output (or text, or any other output) you must write a valid HTML header on standard output, this is done by routine `CGIHEADER`
- the rest of the HTML code can be written by plain WRITE statements, however a little utility routine `CGITOKEN` is provided to write strings of the form <any>string</any>.

Material provided

The material I provide here (without any express or implied guarantee etc. etc.) for demonstration purposes is :

- some ripped-off HTML forms of my own, in which I replaced the original CGI script with a test script which essentially lists all the variables in the form, and demonstrates a few features. These forms are :

  - a simple POST method example
  - the same example with GET method
  - a longer example of POST and GET method demonstrating errors one can get if the query string is longer than the buffer destined to hold it (I know how to deal with it by dynamic memory allocation, but that will complicate the tutorial, and I intend to keep it private so far)
    Note that at the moment I'm keeping the size of the buffers in cgi.inc very small (256 characters) just to demonstrate the error. Behaviour may change without notice. With more suitable larger values no error will be generated by an empty form, but could be generated by filling it. Also the error is signalled only by the POST method, while GET silently truncates the query.

- an HTML commented copy of the source of the test program, which is also provided in source form
- source code of the CGI library routines, with some additional comments

---

Source code distribution

Since the relevant material consists of quite a few lines of code, I consider simple to give access to each source file, more than preparing gzipped tar files or anything like that. You should just retrieve via Web (Save As...) all the source files listed below and read the relevant comments. Then compile the routines and place them in a relocatable library, or anyhow link the test program, or any CGI program of yours against them in the way you are accustomed to.

Portability issues

Not all routines listed below have been written ad-hoc. A few of them (those "below the red line" are taken by libraries I developed for other purposes). The latter have either been extensively used on several OSs (starting from VAX/VMS to SunOS, Ultrix, HP-UX, Digital Unix and Solaris). The few routines for which a separate version is necessary for different systems are clearly indicated.

However note I did **not** do any extensive tests on my new CGI routines. I just tried them on my DU 3.2 Alpha with NCSA httpd 1.5.2. While I did not expect any portability issue in most of my code, I must say I was pleasantly surprised to see that the POST method worked with plain Fortran i/o. In the past I noticed in fact that the POST method (which uses stdin to communicate between form and CGI) sometimes had problems :

1. in some cases httpd seemed to append a carriage-return (control-M) to the end of the query string, which then makes its way into the value of the last variable
2. in other cases on Digital machines, if the last string on stdin (or any file) is not terminated by a line-feed it's ignored by a Fortran `READ`

Source code is here !

Item	Source File Name	Operating Sytem	Comments
demo main	prova.f	all	here
include file	cgi.inc	all	here
subroutine	cgiheader.f	all	here
subroutine	cgiretrieve.f	all	here
subroutine	cgitoken.f	all	here
subroutine	get_named_variable.f	all	here
subroutine	get_next_variable.f	all	here
subroutine	urldecode.f	all	here
subroutine	urlencode.f	all	here
additional routines			
function	true_length.f	all	here
function	bindex.f	all	here
subroutine	upcase.f	all	here
subroutine	get_global_default.f	all	here
subroutine	z_get_global.f	Unix VAX/VMS	**READ THIS !**
C routine	zc_getenv.c	Unix only	here

Brief comments on CGI routines

- include file cgi.inc
  defines a couple of `COMMON` blocks (separate for character and numbers), containing a `CGISTRING` buffer used to hold the query string, and an additional work buffer. These are sized by a `PARAMETER` statement.
  Additionally contain the CGI method used, the Content-length where defined, and the debug flag used to allow some writes in the other routines.
  Additionally some commonly used HTML strings can be defined as `PARAMETER`s at the end of the file.

  You do not need to refer to this in the main program, unless you want to set the `CGIDEBUG` flag to `.TRUE.`

- get_named_variable.f

  Use this routine to get the value of a variable you know by name.
  Since the name may contain special characters, it has to be URL-encoded before trying to match its `name` in the query string. One looks for a string `encodedname=value&` and then extracts the `value` and URL-decodes it. In the process of encoding and decoding, as well as copying to the target string, truncation may occur. This and other errors are returned as non-zero error codes (documented in the source). Negative error codes are presumably severe, while positive ones can probably be considered as warnings (this includes the case of a null value, code 1).

- get_next_variable.f

  Use a sequence of calls to this routine to get the name and value of each variable in order. The routine looks for `name=value&` strings, extracts and URL-decodes the `name`, and then extracts and URL-decodes the `value`. In the process of decoding and copying to the target strings, truncation may occur. This and other errors are returned as non-zero error codes (documented in the source) similar to the previous routine. In addition a code of -1 means the end of the query string has been encountered.

---

both above routines call the next one internally

- cgiretrieve.f

  deals with POST and GET method and retrieves the run string. Any call after the first one are no-op.

  The `REQUEST_METHOD`, `CONTENT_LENGTH` and `QUERY_STRING` are retrieved from environment variables, while for the POST method the query string is read from stdin with an A format. Detection of a method other than POST or GET is an irrecoverable error (although I noted that browsers or httpd daemons tend to prevent this forcing a valid method).

- cgiheader.f
  Self-explanatory routine used to produce the header for the HTML output. Note the way of producing a null line separating header and body.

- cgitoken.f
  Self-explanatory routine

- urldecode.f
  Called internally by the `get_xxx_variable` routines, will replace pluses with blanks and `%hh` hexadecimal encodings with the corresponding character.

- urlencode.f
  Called internally by `get_named_variable` to perform the reverse encoding of the above

---

The routines below are not part of CGILIB, but of other libraries of mine. A link to them is provided, since it is simpler than making the entire libraries available.

- portable service routines

  true_length.f returns the length of the non-blank part of a string (trimmed length).

  bindex.f is a variant of the intrinsic `INDEX` function, which returns the absolute offset in a

string of subsequent occurrences of a substring.

upcase.f is a trivial routine converting a string to upper case.

- system-dependent environment interface

  get_global_default.f is a system-independent front-end for environment variable retrieval. It returns an user-specified default value even it the variable is not present. It calls in turn the next, system-dependent routine

  z_get_global.f is the Unix system-dependent incarnation of a routine used to retrieve an environment variable. There is also a VAX VMS incarnation. The Unix one in turn calls a zc_getenv.c C jacket routine, while the VMS one is standalone.

  For reasons too long to explain here, this approach was preferred to calling routines `getenv` or similar provided on some systems in their Fortran library. It shall be noted also that the routines are overdone for general use, so **you should replace them with whatever you like or simplify them** : in particular you should at least remove the reference to routine `z_initenv`, and the code which looks for a variable named `XAS_NAME` before `NAME` (such thing is irrelevant and unnecessary in CGIs, while it was peculiar of the XAS environment for which the routine was originally written).
  Similarly you should replace the references to voscommon.inc (defines a couple of variables to hold error codes, replace with local variables) and errors.inc (assigns system-independent parameter constant values to error codes).

In this HTML version of this source code I use the following colour conventions :

- black italics indicates comments
- **red bold** indicates mandatory code for a typical CGI program
- red indicates typical code for a CGI program
- green indicates local peculiarities of this program
- **blue bold** indicates calls to CGI library items which are not normally done in main programs but demonstrate features of the underlying routines.

I provide also direct links to uncommented (at least, HTML-uncommented, ample Fortran comments inside) source code of all routines.

```
C
C       this is the test Fortran CGI program referred by the demos in
C       http://sax.lambrate.inaf.it/~lucio/WWW/cgi-lib/
C
```

```
C
C       these are just local variables
C

        CHARACTER prova*8,var*16,val*16,name*8
        INTEGER   ierr
```

```
C       these constants holding HTML useful strings are not here, they are instead
C       in the include file cgi.inc, where further strings can be added ad libitum
C

*       CHARACTER RED*20,BLU*20,GRN*20,BLK*7
*       PARAMETER (RED='<font color=#ff0000>')
*       PARAMETER (BLU='<font color=#0000ff>')
*       PARAMETER (GRN='<font color=#00ff00>')
*       PARAMETER (BLK='</font>')
```

```
C       this include file is referred here just to set the CGIDEBUG flag which
C       allows some debug WRITE to appear. The include file also refers to an
C       important COMMON block used internally to store the query string etc.
C       but need not to be used in the main program unless you want to play around
C       with the CGIDEBUG flag
C


        INCLUDE    'cgi.inc'
        CGIDEBUG=.TRUE.
```

```
C       these two utilities are better called first and in this order. The first is
C       actually mandatory to route the standard output where and how it has to be.
C       You select the kind of output : text/html, text/plain or whatever.
C       The second call just outputs <title>CGI program return<title>
C


        CALL CGIHEADER('text/html')
        CALL CGITOKEN('title','CGI program return')
```

```
C       this very important call retrieves the query string in POST or GET
C       method as appropriate. It is not necessary to call it in the main. It will
C       be called automatically by the first GET_NAMED_VARIABLE or GET_NEXT_VARIABLE
C       call to load the string in common. It is called here to place DEBUG information
C       first on the screen.
C

        CALL CGIRETRIEVE
```

```
C       what follows is just an example of decoding the query string for known
C       variables called by name. This works with the provaget and provapost forms,
C       but is likely to give "variable not found" if you try it with any other form
C       (including yours).
C
C       It also demonstrates HTML output. This can be generated directly with WRITE
C       statements, or by utility routines. I have made only one, CGITOKEN, which
C       outputs a string enclosed in <any>...</any> clauses.
C

        write(*,*)'<hr>'
        CALL CGITOKEN('h1','parsing 3 vars by name in 8-char buffer')

C
C       Named variables can be called in any order, provided the name matches a
C       variable in the form. Here string PROVA will contain the returned value, and
C       since this is just 8 character long, may demonstrate a truncation error.
C
C       Here I demonstrate the use of a variable name with imbedded URL-encoded
C       characters (look at the debug output). The variable is called 'tar$get'.
C       An error code 2 indicates truncation.
C       The same variable is retrieved untruncated in the example loop below.
C

        NAME='tar$get'
        CALL GET_NAMED_VARIABLE(NAME,PROVA,IERR)
        CALL CGITOKEN('h3',
     .  blu//name//blk//'is '''//red//prova//blk//'''')
        write(*,*)' error code' ,ierr

C
C       Here I demonstrate the successful use of a plain variable name 'decs'
C

        NAME='decs'
        CALL GET_NAMED_VARIABLE(NAME,PROVA,IERR)
        CALL CGITOKEN('h3',
     .  blu//name//blk//'is '''//red//prova//blk//'''')
        write(*,*)' error code' ,ierr

C
C       Here I try to retrieve a non-existing variable 'pippo', error code -1
C

        NAME='pippo'
        CALL GET_NAMED_VARIABLE(NAME,PROVA,IERR)
        CALL CGITOKEN('h3',
     .  blu//name//blk//'is '''//red//prova//blk//'''')
        write(*,*)' error code' ,ierr
```

```
C       what follows is a more regular example of decoding the query string for all
C       variables in the order they appear. For each it will return name and value.
```

```
C
C       Some usual demo of HTML output.
C

        write(*,*)'<hr>'
        CALL CGITOKEN('h1','parsing all variables in order')

C
C       a loop retrieving each variable in turn until end of them
C


 111    CONTINUE
        CALL GET_NEXT_VARIABLE(val,var,IERR)
        IF(IERR.NE.-1)THEN

         CALL CGITOKEN('h3','variable '//blu//val//blk)
         write(*,*)'has value ',red,var,blk
         write(*,*)' error code' ,ierr

         GOTO 111
        ENDIF
```

```
C       ok that's all
C

       CALL CGITOKEN('h1',grn//'end of parsing'//blk)

        STOP
        END
```

```
C       You can have a look at individual routines
C
*       include 'cgiheader.f'
*       include 'cgitoken.f'
C
C       the next two parse the query string from var=val&var=val&...&var=val
C
*       include 'get_named_variable.f'
*       include 'get_next_variable.f'
*       include 'cgiretrieve.f'
C
C       the next two are service routines to handle URL-encoding. Strings are
C       encoded by http browsers with blanks replaced by '+' and most "funny"
C       characters replaced by hexadecimal %hh sequences. Decoding reconstructs
C       the original string, e.g. a(b) from a%28b%29 . Encoding does the reverse
C       and is necessary only to match named variables, which YOU provide in
C       unencoded form as NAME arguments to GET_NAMED_VARIABLE.
C
*       include 'urldecode.f'
*       include 'urlencode.f'
```