CGIDEV2 Tutorial

1. Getting acquainted

If you are not yet familiar with CGI programming and with CGIDEV2 service program, please read the following:

- Benefits of CGIDEV2 Service program
- Giovanni's flyer on CGI
- Giovanni's editorial on CGI

2. Basic Demos

Our demos are the best way to learn in practice how to develop RPG-ILE CCGI programs. After reading about CGIDEV2 service program functions, we recommend you **visit** them.

3. Set up commands

You should not develop your CGI's in one of the libraries (e.g. CGIDEV2) downloaded from our site. Installing a refresh would delete your developments.
You should instead develop your CGI's in your (source and object) libraries. The problem of copying all the pieces needed for development and for execution of your CGI's is solved through some commands. We have commands to easy your setups:

- **cgidev2/setcgilib** to set up your source and production libraries for CGI development
- **cgidev2/crtcgisrc** to create a running example of CGI source.

Then we have prepared for you some ILE-RPG source members that you may **include** in your sources, thus saving coding time.

4. CGI services available through CGIDEV2 service program

For a comprehensive walk through all the features of CGIDEV2 service program, you may go to Mel Rothman's Readme page.

We recommend however you go through the following scheme, which provides you with a smooth learning path:

Main functions

- Read input from client browser
- Map input string into program variables
- Multiple occurrences of an input variable
- Use an externally defined HTML script to write HTML output

Other functions

- Handling cookies
- Message handling
- Maintain and retrieve page counts
- Retrieve environment variables
- Other environment variables functions
- Timing functions
- IFS subprocedures
- Uploading PC files
- CGI debug file
- Debugging functions

Coding facilities

- Data conversion functions
- Execute a command
- Override a database file

***Dynastatic* pages**

- Write HTML to a stream file

Program state support

- Using user spaces
- Create a random string

Persistent CGI support

- Get a random integer
- Assign a Session ID ("handle")

5. CGI debugging tips

Though debugging always comes last in a tutorial, it should rank first in a programmer's interest. This is why we dedicate a **separate page** to this topic.

6. Error number values

In some circumstances some subprocedures of the service programs may end with non-zero return codes. Usually such error codes are also reported in the CGIDEBUG file. The guess would be about the meaning of such codes. To find out about them:

- Go to the following page of the **iSeries Information Center**: Errno Values for UNIX-Type Functions.

7. Performance tips

In order to obtain the best performance from your CGI's, you may want to adopt the following tips:

1. use a named activation group (a different activation group for each CGI, see FAQ number 26)
2. return without setting LR on

   Note. By returning with LR set to *off provides you with another great performance advantage. The next time the program is called and tries to reload the external HTML, the service program will find that the HTML is still in core, and will skip its loading, thus saving some relevant amount of time.
3. open files just the first time through, never close them
4. each time the program runs
   - re-initialize variables
   - do not rely on files being positioned on the first record; reposition with **SETLL**, or **SETGT**, or any other appropriate way
5. After programs are thoroughly tested, use **CALLP SetNoDebug(*ON)** to turn all debugging off.

   Note. SetNoDebug sets a global variable in the service program. If multiple CGI programs are running in the same named activation group, all those programs are affected in the same way by the most recent execution of SetNoDebug.

8. About persistent CGI

So far, without mentioning them, we have being talking about *non persistent CGI*. Traditionally, CGI programs are not persistent, that is, a CGI program, the next time it is called, does not remember (or, better, must not remember - as no guarantee exists that it is called by the same client as before) what values where left in its variables, and where files are positioned. This fact of course has a relevant impact in program design, because the values of the variables which are important for a program at the time it is invoked must be saved in the previous response HTML (usually as hidden fields) and sent with the program invocation request.

Since V4R3, however, *persistent CGI* have been implemented for OS/400 HTTP.
A persistent CGI returns without raising LR, that is, the next time it is called, it finds its status as it was left.
The obvious problem with persistent CGI, is that for a given program there are potentially several copies waiting to be called back, and each copy is related to a different user. You would not like to have such responsibilities be randomly mixed up. To solve this problem, it is a program responsibility to give a "ticket" (technically speaking, a "handle") to each next program invocation.
The next problem is that a persistent CGI cannot wait forever to be called back. An HTTP controlled timeout will kill any persistent CGI session waiting over a given number of seconds. At this point, the user may need to re-start his transaction cycle from the beginning.

Because of this second problem, the implementation of persistent CGI is not very frequent and left to cases where a COMMIT technique is mandatory.

Our recommendation is **not to use persistent CGI**, unless strictly necessary. Persistent CGI

- do not perform faster than non-persistent CGI
- require much more skill and testing
- may not provide a user-friendly solution

However, should you like to know more on this subject, this is your page.

9. ZIP and UNZIP commands

Library CGIDEV2 includes commands ZIP and UNZIP.
These commands allow to compress and decompress IFS stream files using standard *.zip* files.
The requests entered from commands ZIP and UNZIP are transformed to QSHELL commands, that

are run in batch mode.
Results from ZIP and UNZIP commands can optionally be displayed.

Read input from client browser

<center>⬅️M➡️</center>

1. The input string
2. GET and POST methods
3. Procedure to read the input string

1. The input string

There are three ways a remote client browser may send input to a CGI program:

1. **From an HTML form**. As an example:

   html   **which shows as**

   ```
   <form method="post" action="/cgidev2p/hello1.pgm">
   Your first name:
   <input type="text" name="firstname"
           size="10" maxlength="30"><br>
   Your last name:
   <input type="text" name="lastname"
           size="10" maxlength="30"><br>
   <center>
   <input type="submit" value="Send">
   </center>
   </form>
   ```

   Your first name: George
   Your last name: Brown
   Send

   In this case, when the remote user, after entering data in the two input fields, presses the Send button, the string sent to program `hello1` is:

   firstname=*George*&lastname=*Brown*

   where *George* and *Brown* are the values entered into the two input fields of the form.

2. **From an anchor tag (<a>...) in an HTML script**. As an example, the following anchor

   ```
   <a href="/cgidev2p/hello1.pgm?firstname=George&lastname=Brown">Say
   hello to George Brown</a>
   ```

   will send the same string to program `hello1`.

3. **From the command line (location line) of the browser**.
   For instance, if in this line one types

   ```
   http://.../cgidev2/hello1.pgm?firstname=George&lastname=Brown
   ```

   the same string will be sent to program `hello1`.

2. The GET and the POST methods

There are two ways an input string may be sent to a CGI program.

1. With the GET method.
   This is implicitly done when the sending is performed either through an anchor tag (`<a> href=...</a>`) or through the browser command line.
   The GET method may also be used in a `form` tag. This is usually done for test purposes.
   In fact, when using the GET method, the input string is **always visible** in the browser command line.
   The GET method has some restrictions, which do not exist for the POST method:
      i. The input string (containing parameters after character "**?**"; usually called *query string*) has a maximum length of about 3 thousand characters. The maximum length depends on the type of browser.
      ii. The values of the parameters can just be alphanumeric strings: no special characters, no inbedded spaces. Special characters and inbedded spaces must be replaced by URL escaped sequences.
2. With the POST method.
   This is commonly done in a `form` tag.
   In fact, when using the POST method, the input string is **not visible** to the end user.

Though these two methods have implications on the way a CGI should retrieve its input string, Mel Rothman's service program provides procedures which would take care to retrieve the input string whichever way it was sent.
Therefore your CGI programs are not sensitive to the method used by the remote browser.

3. Procedure to read the input string

In order to acquire the input buffer sent from the browser, your CGI program must use the **zhbGetInput** subprocedure

ZhbGetInput uses the server's QzhbCgiParse API to get the browser's input and places it into a set of internal, dynamically allocated arrays for subsequent high performance use by the **ZhbGetVarCnt**, **ZhbGetVar**, **ZhbGetVarUpper** input variable parsing procedures.

Warning. For the QzhbCgiParse API to work properly, the CGIConvMode must contain value %%EBCDIC/EBCDIC%% (not the value %%MIXED/MIXED%%). If that does not happen, ZhbGetInput writes an error message into the debugging file and allows the program to continue until it fails.
To use the *zhbGetInput* correctly you must add the following Apache HTTP directive:

```
                    CGIConvMode %%EBCDIC/EBCDIC%%
```

This is how you can use the ZhbGetInput subprocedure in your CGI program:

```
 * Prototype definitions and standard system API error structure
 /copy CGIDEV2/qrpglesrc,prototypeb
 /copy CGIDEV2/qrpglesrc,usec
 * Number of variables
DnbrVars          s              10i 0
 *
 * Saved query string
Dsavedquerystring...
D                 s          32767    varying
 *
          ... etc. ...
 * Get the input buffer sent from the browser
C                 eval      nbrVars =
C                           zhbGetInput(savedquerystring:qusec)
```

or you can use the following code:

```
 * Prototype definitions and standard system API error structure
 /copy CGIDEV2/qrpglesrc,prototypeb
 /copy CGIDEV2/qrpglesrc,usec
 * Predefined variables
 /copy CGIDEV2/qrpglesrc,variables3
          ... etc. ...
 * Get the input buffer sent from the browser
 /copy CGIDEV2/qrpglesrc,prolog3
```

For a live example, please see the source of the template3 program.

Map input string into program variables

Sample input string:

```
cgiinp01=George&cgiinp02=Brown
```

Once a CGI has read the input string sent from the remote browser, it must understand what the request is. To do this there must be some routine that scans the input (query) string for all the possible keywords and saves their values into program variables, so that the program may then test them and process the user request. This scan_and_break operation is commonly referrred to as "**parsing**".

The following *parsing* subprocedures are available:

1. **"zhbGetVar" parsing procedure**
2. **"zhbGetVarptr" parsing procedure**
3. **Other zhbGet... subprocedures**
4. **Receiving names of input variables**

1. "zhbGetVar" parsing procedure

Parsing procedure **zhbgetvar** lets you retrieve fields from the input string one at a time into program-defined fields.
If you want to uppercase a field while retrieving it from the input string, you may use the parsing procedure **zhbgetvarupper**.
If you want to lowercase a field while retrieving it from the input string, you may use the parsing procedure **zhbgetvarlower**.

This is how you can use the zhbGetVar subprocedure in your CGI program:

```
 * Prototype definitions and standard system API error structure
 /copy CGIDEV2/qrpglesrc,prototypeb
 /copy CGIDEV2/qrpglesrc,usec
 * Number of variables
DnbrVars          s              10i 0
 *
 * Saved query string
Dsavedquerystring...
D                 s              32767    varying
 *
 * Client input variables
D custname        s              40
D emailadd        s              40
D state           s               2
            ... etc. ...
 * Get input
C                 eval      nbrVars =
C                           zhbgetinput(savedquerystring:qusec)
 * Parse variables from QUERY_STRING environment variable:
 * Customer name
C                 eval      custname = zhbgetvar('custname')
 * E-mail address
C                 eval      emailadd = zhbgetvar('emailadd')
```

```
 * State
C                    eval      state = zhbgetvar('state')
```

For a complete example see the source of program TEMPLATE3.

2. "zhbGetVarPtr" parsing procedure

Parsing procedure **zhbgetvarptr** returns a pointer to an input variable. This is useful when an input variable's length might exceed ZhbGetVar's maximum size of 32767.
The maximum length of such a variable is 64000.
If the input variable is not found or length is 0, returns *null .

This is how you can use the zhbGetVarPtr subprocedure in your CGI program:

```
 * Prototype definitions and standard system API error structure
 /copy CGIDEV2/qrpglesrc,prototypeb
 /copy CGIDEV2/qrpglesrc,usec
 * Number of variables
DnbrVars           s             10i 0
 *
 * Saved query string
Dsavedquerystring...
D                  s             32767    varying
 *
 * Pointer returned from zhbGetVarPtr
D ReturnVarP       s               *
 * Variables for zhbGetVarPtr
D  varnamein       s             50
D  occurrence      s             10i 0
D  varLenOut       s             10i 0
          ... etc. ...
 * Get input
C                    eval      nbrVars =
C                              zhbgetinput(savedquerystring:qusec)
 * Retrieve the pointer to input variable named 'longstring':
C                    eval      occurrence = 1
C                    eval      ReturnVarP = zhbGetVarPtr('longstring':
C                              occurrence:
C                              varLenOut)
```

Note. Do not use this subprocedure for reading a file being uploaded from the browser. See Uploading PC files.

3. Other zhbGet... subprocedures

- **ZhbCountAllVars**: returns the number of occurrences of all variables in the input string (*ZhbGetInput* must have been run before calling this subprocedure)
- **ZhbGetVarDetails**: returns the following information on the user-specified nth input variable (out of those counted with *ZhbCountAllVars*): variable name, variable occurrence number, indicator (char 0/1) whether variable was found.

```
D  nbrInpVars      s             10i 0
D  ThisVarVal      s           1000a
D  ThisOccur       s             10i 0
D  ThisVarName     s             50
D  ThisVarOccur    s             10i 0
D  FoundInd        s               n
 * Get input
C                    eval      nbrVars =
C                              zhbgetinput(savedquerystring:qusec)
```

```
 * Example of retrieving
 * the number of occurrences of all variables in the CGI input
C                   eval      nbrInpVars = ZhbCountAllVars
 * Example of retrieving
 * detailed information for all the input variables
C                   if        nbrInpVars > 0
C     1             do        nbrInpVars   ThisOccur
C                   eval      ThisVarVal =
C                             ZhbGetVarDetails(ThisOccur:
C                             ThisVarName:
C                             ThisVarOccur:
C                             FoundInd)
C                   enddo
C                   endif
```

4. Receiving names of input variables

There may be cases where the CGI program cannot predict the names of the input variables, and must find out what they are.
Such cases can be easily solved by using, in sequence, the following subprocedures:

a. ZhbGetInput
b. ZhbCountAllVars
c. ZhbGetVarDetails

Handle multiple occurrences of an input variable from the browser

What is a multiple input variable from the browser

Suppose that a CGI expects to receive from the browser an input string like the following

itemno=00727&qty=1&itemno=00932&qty=7&itemno=01858&qty=15

That could be an order of three items, each with a different quantity.
In other words, the CGI receives *multiple occurrences* of input variables `imtemno` and `qty`.

CGIDEV2 service program provides the following procedures:

- **ZhbGetVarCnt** returns the number of multiple occurrences of a given input field
- **ZhbGetVar** returns the value of one of the multiple occurrences of a given input field
- **ZhbGetVarUpper** returns the value of one of the multiple occurrences of a given input field *in uppercase characters*

Example:

```
 * Prototype definitions and standard system API error structure
 /copy CGIDEV2/qrpglesrc,prototypeb
 /copy CGIDEV2/qrpglesrc,usec
 * Number of variables
DnbrVars          s              10i 0
 * Saved query string
Dsavedquerystring...
D                 s              32767    varying
 * Return code
D rc               s              10i 0 inz(0)
 * Variables for retrieving multiple occurrencies of "itemno" and "qty"
D itemcount        s              10i 0
D varocc           s              10i 0
 *
D itemno           s               5a
D qty              s               3a
 *          ... etc. ...
 * Get input
C                 eval       nbrVars =
C                            zhbgetinput(savedquerystring:qusec)
 * Example of multiple occurences
C                 eval       itemcount = ZhbGetVarCnt('itemno')
C                 IF         itemcount > 0
C     1           do         itemcount    varocc
C                 eval       itemno = ZhbGetVar('itemno':varocc)
C                 eval       qty    = ZhbGetVar('qty':varocc)
C                 exsr       ProcessItem
C                 enddo
C                 ENDIF
```

These procedures are able to process an unlimited number of occurrences of the same input variable.

For an **example** of retrieving multiple occurrences of an input variable, please check out program boatsch2 in the YachtWorld demo.

Externally defined HTML

"The same power and flexibility as an externally defined Display File!"

To provide an answer from your program to the web user, your program should send out an HTML string.

In a RPG program, without using Mel's service program, you should prepare your html string and write it out using the **Write to Stdout (QtmhWrStout) API**. Preparing such a string in a program, while not easy, suffers of unflexibility. Any time you want to change something in the output HTML (texts, fonts, alignments, href, gifs, etc.), you must change your program.

In a normal AS/400 program you would have greater flexibility just using an externally defined display file.

Mel's service program provides a technique as flexible as DDS, but even simpler to use.
This technique is called

Externally defined HTML

There are two ways you can develop external HTML

1. using a source physical file
2. using IFS stream files

---

1. Using a source physical file

This is how you implement it (it takes more to tell than to use it):

1. Create a source physical file named HTMLSRC (record format name MUST be HTMLSRC) with record length 240 (MUST not be more than 240). You may then rename this source physical file, if you want; its record format would still be HTMLSRC.
   Note 1. A file HTMLSRC is automatically created in your object (production) library when you use command **setcgilib**.
2. Add a member named as you like (we recommend to specify HTML for the SEU type).
3. Divide your source into **sections**.
   - In a sense, a section is the same as a record format in a display file.
     A section is a single piece of html your program will output under given conditions.
     Examples of sections could be
       - starting html, starting body, defining title and headers
       - a table start
       - a table row
       - a table end
       - body and html end
   - A section is identified by a source record containing
       - **/$section_name** starting in column 1, where *section_name* can be up to 20 characters (prefix **/$** is the standard one for section identification; however, the developer may define other prefixes).
       - **/$top**, for instance, identifies the beginning of section **top**.
   - The section which is issued as first must start as follow

     ```
     /$section_name
     ```

```
Content-type: text/html

<html>
```

Please pay attention to the blank line just before the one containing <HTML>. If you miss it, the client browser may not interpret your html!

- HTML texts in your sections may contain *variables*. A trivial case is that of a table row (subfile line) with table definitions *<TD>* (subfile line fields) containing variables, such as item and price

    - You specify a variable in your html text by using the following syntax
      ```
      /%variable_name%/
      ```
      where *variable_name* can be up to 30 non-case sensitive characters. Delimiters **/%** and **%/** are the standard ones. However, the developer may decide to use his own delimiters. There is no naming convention for variables. For instance, they do not need to have the same name of their corresponding database fields, though if you do so you may improve understanding of your programs.

- Your CGI program will

    1. Read the external html source member into memory using Mel's service program subprocedure <u>gethtml</u>.
    2. Fill the html `/%variables%/` (which are character fields in your RPG) from e.g. database records fields using <u>updhtmlvar</u> subprocedure.
    3. Call subprocedure <u>wrtsection</u> to output html sections.

- **Section** and **variable delimiters**

    1. **Section name delimiters**.
       A section is identified by the following sequence starting in column 1 of a dedicated line:
       xxxsection_nameyyy
       where
         - `xxx` is the section name starting delimiter (10 char max) that you would mention in the RPG program when calling subprocedure <u>gethtml</u>.
           The default section name starting delimiter is `/$`. This default section name starting delimiter may be omitted when calling subprocedure <u>gethtml</u>.
         - *section_name* is the name of the section (mandatory) that you would mention when calling subprocedure `wrtsection`. Section name must be an alphanumeric string up to 20 characters.
         - `yyy` is the section name ending delimiter (optional, 10 char max). If specified, you must mention it in the RPG program when calling subprocedure `gethtml`.

       Notes on section name delimiters
         1. Default section name starting delimiter `/$` may cause problems when the external HTML is on the IFS. This is because character $ may not be correctly converted for the CGISRVPGM2 service program.
         2. We suggest you use the following:
            "**<! -- Sec_**" as section name starting delimiter
            "** -->**" as section name ending delimiter
            Example:
            <! -- Sec_top -->
            for section *top*.
            This approach defines the HTML section as a comment and is therefore transparent to the HTML editors.

    2. **Variable name delimiters**
       The default delimiters for a variable name are
       ```
       /% and %/
       ```
       User-defined delimiters can also be used, provided thay do not exceed 10 characters. Non-default variable delimiters must be specified as parameters to the <u>gethtml</u> subprocedure.

- The following **restrictions** apply:

Description	Maximum
Source record length	240 bytes (228 bytes for source data)
Number of records	32,764
Number of unique substitution variables (each may appear multiple times in the source)	16,221
Number of occurrences of substitution variables in the source file member	32,767
Substitution variable name length	30 characters
Substitution variable value	1,000 characters
Substitution variables' delimiters length	10 characters
Number of sections	200
Section name length	20 characters
Section name delimiters' length	10 characters

2. Using IFS stream files

Instead of writing the external HTML in members of a source physical file, you could decide to create PC files (stream files) in one or more directories of the OS/400 Integrated File System (IFS). This approach may have some advantages:

1. the restriction of 228 characters per line no longer applies (the other restrictions stay the same)
2. it would be possible design external HTML scripts using HTML authoring tools, such as Microsoft Front Page, IBM Websphere Studio, or other.

Performance would be the same as for members of a physical file.

Directories containing externally defined HTML:

- do not require any HTTP directive to be accessed
- for performance reasons, it is recommended to store the external HTML files in directories different from the ones accessed from the HTTP server (such as the directories containing static HTML pages and/or images).

Examples

1. Giovanni's html skeleton member using default section name delimiters
2. Mel's html skeleton member using user-defined section name delimiters

How to provide HTML responses using externally defined HTML

This page shows how to

1. **define the variables needed for these operations**
2. **read into memory an externally defined html script**
3. **assign values to variables of this external html**
4. **write html sections and send the output buffer**
5. **use special output procedures**

1. Variables

We recommend you use

1. command cgidev2/setcgilib to setup the development and execution environments in your source library and object (production) library
2. command cgidev2/crtcgisrc to create a sample source for a new CGI program. You can compile an run such program.

If you do so, you'll find out that the variables are already defined by copying in other source members:

```
/copy mysrclib/qrpglesrc,hspecs
/copy mysrclib/qrpglesrc,hspecsbnd
          ... etc. ...
/copy mysrclib/qrpglesrc,prototypeb
/copy mysrclib/qrpglesrc,usec
/copy mysrclib/qrpglesrc,variables3
          ... etc. ...
```

2. Read skeleton html member into memory

Use subprocedure

- **getHtml** to load into memory **a single externally defined html source member**
- **getHtmlifs** to load into memory **a single externally defined html IFS (stream) file**
- **getHtmlifsMult** to load into memory in one shot **multiple externally defined html IFS (stream) files**

Examples for default section and variables delimiters

- section name start delimiter: /$
- variable name start delimiter: /%
- variable name end delimiter: %/

1) getHtmlxxx

```
C                    eval      HtmlSrcLib = 'CGIDEV2'
C                    eval      HtmlSrcFil = 'DEMOHTML'
C                    eval      HtmlSrcMbr = 'EXERCISE'
C                    callp     gethtml(HtmlSrcFil:HtmlSrcLib:HtmlSrcMbr)
```

```
C                    callp     gethtml('DEMOHTML':'CGIDEV2':'EXERCISE')
```

2) getHtmlifs

```
C                    callp     gethtmlifs('/CgidevExtHtml/talk2ifs.html')
```

See the example in CGI TEMPLATE4.
Note that the directory to be specified as a parameter of the gethtmlifs procedure is the **real** directory path, **not** the alias mentioned in a HTTP **pass directive**.
The recommendation is -in any case- not to use for the IFS html code directories accessed also by the HTTP server.

3) getHtmlifsMult

This subprocedures allows to load into memory multiple externally defined html files. All the sections and records in all the files are read into dynamic storage as though they resided in a single file. If a section name appears more than once, only the first occurrence is used.
This feature allows to maintain as separate HTML files frequently used pieces of HTML code, such as headers, footers, navigation bars, etc.. In several cases, breaking html code into separate modules may greatly reduce both development and maintenance times.

```
 * Indicators for GetHtmlIfsMult subprocedure
D IfsMultIndicators...
D                 ds
D  NoErrors                     n
D  NameTooLong                  n
D  NotAccessible                n
D  NoFilesUsable                n
D  DupSections                  n
D  FileIsEmpty                  n
   ... etc. ...
 * Read externally defined output html files
C                    eval      IfsMultIndicators = gethtmlifsmult(
C                              '/CgiDevExtHtml/StdTop.Html +
C                              /CgidevExtHtml/StdRunTime.Html +
C                              /CgidevExtHtml/StdMsg.Html +
C                              /CgidevExtHtml/StdPssr.Html +
C                              /CgidevExtHtml/Talk2Stuff.Html +
C                              /CgidevExtHtml/StdEnd.Html')
```

See the example in CGI TEMPLATE5.

Note 1. Subprocedures **getHtmlifs** and **getHtmlifsMult** can also load HTML source members as part of the /QSYS.LIB/... directory.
See the following example:

```
 * Read externally defined output html files
C                    eval      IfsMultIndicators = gethtmlifsmult(
C                              '/CgiDevExtHtml/StdTop.Html +
C                              /QSYS.LIB/MYLIB.LIB/HTMLSRC.FILE/X.MBR')
```

Note 2. When using **getHtmlifs** or **getHtmlifsMult** subprocedure, make sure that the IFS files containing the external HTML code can be read from the HTTP server user profile QTMHHTP1 (the one adopted

when running CGI). This can be done in one of the following ways (in the examples below, we assume that QPGMR owns the files, but this is not relevant at all):

```
              Data    --Object Authorities--
User      Authority Exist  Mgt  Alter  Ref
*PUBLIC   *RX
QPGMR     *RWX          X    X     X    X
```

or

```
              Data    --Object Authorities--
User      Authority Exist  Mgt  Alter  Ref
*PUBLIC   *EXCLUDE
QPGMR     *RWX          X    X     X    X
QTMHHTP1 *RX
```

Examples for user-defined delimiters (see also this page).

First example:

- section name start delimiter: "`<!-- Sec_`"
- section name end delimiter (optional): "` -->`"
- variable name start delimiter: "`<var400>`"
- variable name end delimiter: "`</var400>`"

```
C                     eval      HtmlSrcLib = 'CGIDEV2'
C                     eval      HtmlSrcFil = 'HTMLSRC'
C                     eval      HtmlSrcMbr = 'TALK2'
C                     callp     gethtml(HtmlSrcFil:HtmlSrcLib:HtmlSrcMbr:
C                               '<!-- Sec_':
C                               ' -->':
C                               '<var400>':
C                               '</var400>')
```

Second example:

- section name start delimiter: "`<as400>`"

```
C                     callp     gethtmlifs('/CgidevExtHtml/talk2ifs.html':
C                               '<as400>')
```

```
 * Indicators for GetHtmlIfsMult subprocedure
D IfsMultIndicators...
D                ds
D  NoErrors                    n
D  NameTooLong                 n
D  NotAccessible               n
D  NoFilesUsable               n
D  DupSections                 n
D  FileIsEmpty                 n
   ... etc. ...
 * Read externally defined output html files
C                     eval      IfsMultIndicators = gethtmlifsmult(
C                               '/CgiDevExtHtml/StdTop.Html -
C                               /CgidevExtHtml/StdRunTime.Html -
C                               /CgidevExtHtml/StdMsg.Html -
C                               /CgidevExtHtml/StdPssr.Html -
C                               /CgidevExtHtml/Talk2Stuff.Html -
C                               /CgidevExtHtml/StdEnd.Html':
C                               '<as400>')
```

3. Assign values to html variables

This must be done using subprocedure **updHtmlVar**.
This subprocedure assigns a value to **all the instances** of a given html variable name.
Note 3. The value to be assigned must be a character string (numeric fields must be converted or edited to character strings, see examples number 4 and 5).

<div align="center">Inputs</div>

- variable name
- variable value
- action (optional)
  - '1' = replace this variable if it exists, otherwise add it (default)
  - '0' = clear arrays and write this one as the first
- trim instructions (optional)
  - %trim - trim left and right (default)
  - %triml - trim left only
  - %trimr - trim right only
  - %trim0 - don't trim

Examples:

1. Clear all html variables and assign to html variable `custname` the value contained in character field `CusNam`:

```
C                   callp     updHTMLvar('custname':CusNam:'0')
```

2. Assign to html variable `custaddr` the value contained in character field `CusAdr`:

```
C                   callp     updHTMLvar('custaddr':CusAdr)
```

3. Assign to html variable `custaddr` the value contained in character field `CusAdr` without trimmimg it:

```
C                   callp     updHTMLvar('custaddr':CusAdr:'1':'%trim0')
```

4. Assign to html variable `ordertotal` the value contained in numeric field (9 2) `TotThisOrd` using an edit code:

```
C                   callp     updHTMLvar('ordertotal':
C                             %editc(TotThisOrd:'J'))
```

[About edit codes](#)

5. Assign to html variable `ordertotal` the value contained in numeric field (9 2) `TotThisOrd` using an edit word:

```
C                   callp     updHTMLvar('ordertotal':
C                             %editw(TotThisOrd:'  ,   ,   0.  '))
```

See this [living example](#).

For vary large output variables (max 16 Mb) you may use subprocedure **updHtmlVar2**.
In this subprocedure, instead of passing the name or the value of the substituting variable, you will pass a

pointer to it and its length.

<div align="center">Inputs</div>

- variable name
- pointer to the substituting string
- length of the substituting string
- action (optional)
  - '1' = replace this variable if it exists, otherwise add it (default)
  - '0' = clear arrays and write this one as the first
- trim instructions (optional)
  - %trim - trim left and right (default)
  - %triml - trim left only
  - %trimr - trim right only
  - %trim0 - don't trim

Example:

```
D varPointer       s                  *
D varLength        s                 10i 0
C                    callp     updHtmlVar2('longdata':
C                              varPointer:varLength)
```

4. Write html sections and send the output buffer

This must be done using subprocedure **wrtsection**.
One can write one or multiple sections in a single call.
Important.
Once all sections have been written, do not forget to **send the output buffer to the client**. This must be done by writing a pseudo section named ***fini**.

Example:

```
 * Write a single section
C                    callp     wrtsection('start')
 * Write two sections
C                    callp     wrtsection('part1 part2')
 * Send the output html to the remote browser
C                    callp     wrtsection('*fini')
```

Note 4. Section names are not case sensitive. A section name is a 50 char max alfanumeric string. Only english letters are supported.

Note 5. Subprocedure `wrtSection` supports two optional parameters:

- **NoNewLine** indicator.
  Set it to *on to tell `wrtsection` NOT to insert a newline character x'15' at the end of each HTML output line. This may be useful when binary data are being sent to the browser.
- **NoDataString** value.
  What to do when a substitution variable is encountered and no value has been set up with UpdHtmlVar.
  If not specified, output variables left without substitutions display the default "**Missing Data**".

Examples:

```
D NoDataString    c                      '<b>*** no substitution! ***</b>'
 * Write a section without newline characters
C                    callp     wrtsection('mysection':*on)
 * Write a section with non-default warning for
 * missing substitution variables
C                    callp     wrtsection('mysection':*off:
```

```
C                              NoDataString)
```

5. Special output procedures

Subprocedure wrtNoSection
writes data for the browser without using substitution variables or sections.
This subprocedure can be used when a large block of data is to written. This is more likely to happen when writing non-textual data such as images.
The following example shows how to read an external IFS file (an HTML page, an image, or what you need) and how to insert it into the CGI output buffer:

```
D IfsInpBuff      s             10000a   varying
D InpBuffLen      s               10i 0
D IfsObj          s              255a
D FileHandle      s               10i 0
D ReturnInt       s               10i 0
D BytesIn         s               10i 0
* ... ... ...
* Read an IFS object into "IfsInpBuff"
C                   eval      IfsObj = '/mypath/mysubpath/myobj.xxx'
*   1-Open the IFS file
C                   eval      FileHandle = open(%trim(IfsObj)
C                               : O_RDONLY + O_TEXTDATA)
*   2-Read the IFS file
C                   eval      BytesIn = read(FileHandle
C                               : %addr(IfsInpBuff)
C                               : %size(IfsInpBuff))
*   3-Close the IFS file
C                   eval      ReturnInt = close(FileHandle)
C                   eval      IfsInpBuff = %trim(IfsInpBuff)
* Insert the string read into the CGI output buffer
C     ' '         checkr    IfsInpBuff    InpBuffLen
C                   callp     WrtNoSection(%addr(IfsInpBuff):
C                               InpBuffLen)
```

Subprocedure clrHtmlBuffer
clears any HTML output that has been buffered but has neither been sent to the browser nor written into a stream file. This is useful when program logic dictates you need to output something other than what has already been buffered.

```
C                   callp     clrHtmlBuffer
```

Subprocedure getHtmlBytesBuffered
- Returns the number of bytes in the output HTML buffer.
- This number is incremented each time output is written with WrtSection or WrtNoSection.
- It is reset to 0 when either WrtSection('*fini') or WrtHtmlToStmf is run.
- If this number is allowed to grow to more than 16 MB, the CGI program will fail.
Could be useful to stop creating an output page when its size exceeds a reasonable limit.

```
D buffsize        s               10i 0
C                   eval      buffsize = getHtmlBytesBuffered
```

Subprocedure getHtmlBufferP
- Returns the pointer to the output HTML buffer and the number of bytes used in the output HTML buffer.
- This is expecially useful for debugging.

```
 D OutBuffer       s           32767    based(OutbufferP)
 D OutBufferInfo   ds
 D  OutBufferP                      *
```

```
D  OutBufferLen                 10u 0
C                  eval         OutBufferInfo=callp(GetHtmlBufferP)
```

Handling Cookies

1. About Cookies

Cookies are a mechanism for storing persistent data on the client. As HTTP is a stateless protocol, cookies provide a way to maintain information between client requests.

In a sense, a cookie may be though of as a small data area on the client.

A cookie has the following properties:

Name	*mandatory*	Identifies the cookie (as if it were the name of a data area)
Value	*mandatory*	The contents of the cookie (as if it were the value of a data area). **Note** that **Netscape Navigator does not support blanks in the cookie value**. If that happens, the *Set-Cookie* string is trimmed right at the first blank met. Therefore it may be needed to substitute all the blanks in a cookie value with some other character before creating the cookie; in such a case, the opposite operation should be performed after retrieving the cookie.
Expiration	*optional*	The date until which the cookie should survive. If the expiration is not specified, the cookie expires when the user session ends.
Domain	*optional*	The domain under which the cookie can be retrieved. If the domain is not specified, the web browser should assume the host name of the server generating the cookie.
Path	*optional*	The path under which the cookie can be retrieved. If the path is not specified, the web browser should assume the path of the page generating the cookie.

Cookies are stored and retrieved by the web browser.

Whenever a web page is loaded, the web browser makes available all the unexpired cookies that:

1. match the domain of the page
   (for instance, *www.ibm.com* or *195.183.8.2*)
2. are in the path of the page
   (for instance, to page */cgidev2o/exhibiu8.htm* are made available all the cookies with path "/" and all the cookies with path "*/cgidev2o*").

For further details on the rules controlling access to cookies, read Determining a Valid Cookie.

2. Creating a Cookie with a CGI - Basic approach

To create a cookie you must provide, in the external html, a *Set-Cookie* http header, as follow:

```
/$top
Content-type: text/html
Set-Cookie: name=value [;EXPIRES=dateValue] [;DOMAIN=domainName] [;PATH=pathName] [;SECURE]
                                        (mandatory blank line)
<html>
  ... etc. ...
```

For detail explanation about the *Set-Cookie* http header, please refer to the following page from Netscape.

3. Retrieving a cookie in a CGI - Basic approach

A CGI can **retrieve** *all the available cookies* through the environment variable **HTTP_COOKIE**. The cookies made available are all those compliant with the domain name and the path of the CGI. The following example illustrates the value returned from the environment variable HTTP_COOKIE in a case where two cookies were found:

- the first cookie has *name=IBM* and *value=HG1V432*
- the second cookie has *name=Easy400* and *value=JohnVincentBrown*

```
                        IBM=HG1V432; Easy400=JohnVincentBrown
```

Note 1. As all the available cookies are returned in a single string, it is a program responsibility to retrieve from this string the cookies it might be interested in.

Note 2. The value of a cookie may contain *escaped* characters. An escaped character is the ASCII hexadecimal representation of an ASCII character. For instance, `%3D` is an escaped character and is the ASCII hexadecimal representation of ASCII character "=".
Escaped characters are generated by the web browser when storing a cookie. This is done to eliminate conflicts with regulare string separator and control characters.
Now, it is a responsibility of the CGI program to convert any ASCII escaped characters --in the value of a retrieved cookie-- to the corresponding EBCIDIC characters.

See our example about creating and retrieving a cookie in a CGI through this approach.

4. Creating/retrieving a cookie in a CGI - Advanced approach
Service program **cgidev2/cgisrvpgm2** provides two subprocedures to help managing cookies in a CGI:

- **crtCookie** allows for an easier construction of the *Set-Cookie* http header
- **getCookieByName** retrieves a given cookie from the `HTTP_COOKIE` environment variable.

```
/$top
Content-type: text/html
Expires: 0
/%setmycookie%/

<html>
... etc. ...
```

```
     ** Variables used to build the http header "Set-Cookie"
     ** through subprocedure "CrtCookie"
  D  SetMyCookie    s           1000    varying
  D  CookieNam      s           1000    varying
  D  CookieVal      s           4000    varying
  D  RetCode        s             10i 0
  D  Domain         s           1000    varying
  D  Path           s           1000    varying
  D  Secure         s              n
  D  Expires        s              z
  D  xdocloc        s            512
   ** Other variables
  D TimeNow         s              z
  D r1              s             10i 0
  D r2              s             10i 0
   *================================================================
   * Main line
   *================================================================
   /free

         // Get broswer input
         nbrVars=zhbgetinput(savedquerystring:qusec);

         // Load external html, if not loaded yet
```

```
        gethtml('DEMOHTML':'CGIDEV2':'COOKIE2':'/$');

        // Create the Set-Cookie header
        exsr CrtMyCook;

        // Start the output html
        updhtmlvar('setmycookie':SetMyCookie);
        wrtsection('top');

        // Retrieve cookie current value and display it
        exsr RtvMyCook;
        updHtmlVar('cookienam':CookieNam);
        updHtmlVar('cookieval':CookieVal);
        if CookieVal=' ';
           wrtsection('cookieno');
        else;
           wrtsection('cookieyes');
        endif;

        // End the output html
        UpdHtmlVar('timenow':%trim(%char(TimeNow)));
        wrtsection('endhtml *fini');

        return;

   /end-free
    *=======================================================================
    * Create a cookie
    *   Name:    ThreeMonths
    *   Value:   current timestamp
    *   Domain:  current CGI domain
    *   Path:    /
    *   Secure:  no
    *   Expires: three months from now
    *=======================================================================
   /free

        Begsr CrtMyCook;

        //Retrieve the server domain into variable "Domain"; trim off the port number
        exsr RtvDomain;
        //Reset the domain to blank. The WEB browser assumes the host name of the server
        // generating the cookie
        Domain=' ';

        //Set cookie name, cookie value and cookie path
        CookieNam='ThreeMonths';
        CookieVal=randomString(10);
        Path='/';

        //Set cookie expiration date & time
        TimeNow=%timestamp;
        Expires=TimeNow+%months(3);

        //Create the Set-Cookie header
        SetMyCookie=CrtCookie(CookieNam:CookieVal:RetCode:Domain:
                    Path:*off:Expires);

        Endsr;

   /end-free
    *=======================================================================
    * Retrieve the server domain
    * The server domain is the one the URL of a document starts with,
    * As an example, in the URL
    *       http://www.easy400.net/easy400p/maindown.html
    * the server domain is
    *       www.easy400.net
    *
    * HOW TO SET THE DOMAIN OF THE COOKIE
    * 1-APPROACH NUMBER ONE (deprecated)
    * Usually, there is no easy way through which your CGI can find out
    * what the server domain is.
    * One way I found, is to have the document URL retrieved from some javascript
    * and have it passed in an input variable of the form invoking the CGI.
    * Example:
    *   <form name=cookie2 method=post action="/cgidev2p/cookie2.pgm">
    *   <script language=javascript>
```

```
*    document.write("<input type=hidden name=xdocloc value='"+document.location+"'>")
*    </script>
*        ....
*    </form>
*    In this way the document URL is passed in the input variable "xdocloc".
*    NOTE, however, that if a port number is specified, the port number is returned
*    with the URL and it should not be part of the domain.
* 2-APPROACH NUMBER TWO (suggested)
*    The easiest way is to specify no domain for the cookie. When this is done, the
*    WEB browser assumes as domain of the cookie the name of the host creating the cookie.
*
* Though this subroutine uses approach number ONE to retrieve the domain name for the
* cookie, the program sets the domain name for the cookie to blank, thus making the
* WEB browser default the cookie domain to the name of the host creating the cookie.
*
*=======================================================================
/free

       Begsr RtvDomain;

       Domain=' ';
       xdocloc=zhbgetvar('xdocloc');         //document location ("http://domain:port/...")

       //Remove the URI ("/...") and the port number (if any)
       r1=%scan('http://':xdocloc);
       if r1=1;
          r2=%scan('/':xdocloc:8);
          if r2>8;
             Domain=%subst(xdocloc:8:r2-8);
             r1=%scan(':':Domain);
             if r1>1;
                Domain=%subst(Domain:1:r1-1);
             endif;
          endif;
       endif;

       Endsr;

/end-free
*=======================================================================
* Retrieve a cookie of given name
* Returns a string containing the current value of the cookie,
*        or blanks if cookie not found
*=======================================================================
/free

       Begsr RtvMyCook;

       CookieNam='ThreeMonths';
       CookieVal=GetCookieByName(CookieNam);

       Endsr;
```

See our example about creating and retrieving a cookie in a CGI through this approach.

Handling HTML messages

When errors are found in the user's input, it is necessary to send one or more messages to the browser. Formatting and outputting such messages can be tedious, repetitive, and error-prone.

You can more easily perform this task, using Mel's service program HTML message support. Do the following:

1. In the external HTML source member create a set of sections to be used for message outputting. The standard set is available in `CGIDEV2/HTMLSRC` member `TALK2`. The standard set must
   1. contain section names `MsgStart, MsgL1, MsgL2, MsgL3, MsgEnd`
   2. a `/%msgtext%/` variable must exist in sections `MsgL1, MsgL2, MsgL3`

Look at the example from `TALK2`:

```
/$top
Content-type: text/html

<HTML>
  <HEAD>
    <TITLE>Response To Talk To Us</TITLE>
    <style TYPE="text/css">
        <!--
         .centeredtitle { color: Blue; font-weight: Bold; font-size: 24pt; text-align: center }
         .emphasize {color: Blue; font-weight: Bold; }
         .indent40 {margin-left: 40px; }
         .messagestart, .message1, .message2, .message3  {color: Red; font-weight: Bold; }
          .messagestart { font-size: 16pt }
          .message1 { margin-left: 20px; text-indent: -12px; font-size: 12pt }
          .message2 { margin-left: 40px; text-indent: -10px; font-size: 10pt }
          .message3 { margin-left: 60px; text-indent: -10px; font-size: 10pt }
        -->
    </style>
  </HEAD>
<BODY>
     <!--  ... etc. ... -->
/$MsgStart
<div class=messagestart>Errors:</div>

/$MsgL1
<div class=message1>- /%msgtext%/</div>

/$MsgL2
<div class=message2>- /%msgtext%/</div>

/$MsgL3
<div class=message3>- /%msgtext%/</div>

/$MsgEnd
<div class=messagestart><hr></div>
```

2. Optionally you may use subprocedure **CfgMsgs** to override the default externally described HTML names:
   ○ message text field name (msgtxt),
   ○ starting section name (msgstart),
   ○ level 1 section name (msgl1),
   ○ level 2 section name (msgl2),
   ○ level 3 section name (msgl3),
   ○ ending section name (msgend).
3. Use subprocedure **ClrMsgs** to clear the messages stored in the service program's arrays and to set their count to zero
4. Use subprocedure **AddMsg** to add a message's text and formatting level (1, 2, or 3) to the service program's arrays
5. Use subprocedure **GetMsgCnt** to retrieve the number of messages currently stored in the service program's arrays. It can be used to condition calling `WrtMsgs`.
6. Use subprocedure **WrtMsgs** to write the messages in the arrays to standard output. If no messages are there, nothing is done.

For examples of using HTML messages, please look at the source of RPG CGI program TEMPLATE.

Increment and Edit Page Counter (Number of times a page is visited)

You may maintain and retrieve a counter for the number of times a CGI program of yours is accessed (a page is visited) by using Mel's service program subprocedure **CountP**

Read through to learn how to do it.

1. File CGICOUNT

File CGICOUNT is used to optionally score accesses to given CGI pages. You implicitly duplicate it from library **cgidev2** to your object (production) library when you use command setcgilib.

2. Subprocedure countP

Use this procedure to increment and to retrieve the number of times a given page was accessed.

Example:

```
/copy mysrclib/qrpglesrc,prototypeb
/copy mysrclib/qrpglesrc,usec
/copy mysrclib/qrpglesrc,variables3
          ... etc. ...
 * Get updated counter for program "mylib/hello1" into field "counter"
C                   eval      rc = docmd('OVRDBF FILE(CGICOUNT) +
C                             TOFILE(mylib/CGICOUNT) +
C                             SECURE(*YES)')
C                   eval      counter=countp('mylib     HELLO1     ')
```

Retrieve HTTP server environment variables

You may retrieve the value set by the server for a particular HTTP environment variable by using Mel's prototyped procedure **GetEnv**. This procedure calls **QtmhGetEnv API**.

QtmhGetEnv API provides information about the following environment variables:
(for a complete list of the available environment variables, see this page)

Environment variable	Meaning
AUTH_TYPE	If the server supports client authentication and the script is a protected script, this environment variable contains the method that is used to authenticate the client. For example: Basic
CGI_ASCII_CCSID	Contains the ASCII CCSID the server used when converting CGI input data. If the server did not perform any conversion, (for example, in %%BINARY%% mode), the server sets this value to the DefaultNetCCSID configuration directive value
CGI_MODE	Contains the CGI conversion mode the server is using for this request. Valid values are %%EBCDIC%%, %%MIXED%%, %%BINARY%%, or %%EBCDIC_JCD%% (for more information, see HTTP Server for AS/400 Webmaster's Guide). The program can use this information to determine what conversion, if any, was performed by the server on CGI input data and what format that data is currently in
CGI_EBCDIC_CCSID	Contains the EBCDIC CCSID under which the current server job is running (DefaultFsCCSID configuration directive). It also represents the current job CCSID that is used during server conversion (if any) of CGI input data
CONTENT_LENGTH	When the method of POST is used to send information, this variable contains the number of characters. Servers typically do not send an end-of-file flag when they forward the information by using stdin. If needed, you can use the CONTENT_LENGTH value to determine the end of the input string. For example: 7034
CONTENT_TYPE	When information is sent with the method of POST, this variable contains the type of data included. You can create your own content type in the server configuration file and map it to a viewer. For example: Application/x-www-form-urlencoded
GATEWAY_INTERFACE	The version of the CGI specification with which the server complies. Format: CGI/revision
HTTP_ACCEPT	MIME content types the browser will accept.
HTTP_COOKIE	All the cookies available to the current page.
HTTP_HOST	Contains the HTTP host URL. Example: *www.easy400.net*
HTTP_REFERER	Reference to the page or frame the current page or frame was linked from
HTTP_USER_AGENT	String identifying the Web client. Includes name and version of the browser, request made through a proxy, and other information.
IBM_CCSID_VALUE	The CCSID under which the current server job is running.
PATH_INFO	The extra path information following the path information required to identify the CGI program name.
PATH_TRANSLATED	The server provides a translated version of PATH_INFO, which takes the path and does any virtual-to-physical mapping to it.
QUERY_STRING	Anything that follows the first ? in the request URL. The string is encoded in the standard URL format of changing spaces to '+" and encoding special characters with '%xx' hexadecimal encoding.
REMOTE_ADDR	The IP address of the remote host making the request
REMOTE_HOST	The hostname making the request.
REMOTE_IDENT	User ID of the remote user.
REQUEST_METHOD	The method with which the request was made. For HTTP, this is GET or POST.
REMOTE_USER	If you have a protected script and the server supports client authentication, this environment variable contains the user name that is passed for authentication
SCRIPT_NAME	A virtual path to the program being executed, used for self-referring URLs.
SERVER_ADDR	The server's IP address
SERVER_NAME	The server's hostname, DNS alias, or IP address as it would appear in self-referring URLs
SERVER_PORT	The port number to which the request was sent.
SERVER_PROTOCOL	The name and revision of the information protocol this request came in with. Format: protocol/revision
SERVER_SOFTWARE	The name and version of the information server software answering the request (and running the gateway). Format: name/version. For example: IBM-Secure-ICS/AS/400 Secure HTTP Server

We provide a live example of retrieving environment variables:

- program ENVVAR retrieves the most common environment variables.

Example: the environment variable *SERVER_PROTOCOL*

```
 /copy mysrclib/qrpglesrc,hspecs
 /copy mysrclib/qrpglesrc,hspecsbnd
 * Variables common to all CGIs
 /copy mysrclib/qrpglesrc,prototypeb
 /copy mysrclib/qrpglesrc,usec
 /copy mysrclib/qrpglesrc,variables3
 *           ... etc. ...
 * Server's Protocol
C                   eval      S_Protocol =getenv('SERVER_PROTOCOL':
C                             qusec)
```

Other environment variable functions

Besides **getenv** (retrieve environment variable), two more environment variable functions are available:

1. contlen

This procedure returns as a 4 byte integer the contents of the CONTENT_LENGTH environment variable. This environment variable contains the number of characters of the input string (query string) when the POST method is used. Example:

```
 * Variables common to all CGIs
 /copy mysrclib/qrpglesrc,prototypeb
 /copy mysrclib/qrpglesrc,usec
 /copy mysrclib/qrpglesrc,variables3
 *          ... etc. ...
 * Retrieve query string length
C                   eval      inactln = contlen
```

2. putenv

In some circumstances, one may need to change the value of an existing environment variable, or to create a new environment variable.
This is useful for communication between programs running in the same job, such as your program and the Net.Data language environment.
Example:

```
 * Variables common to all CGIs
 /copy mysrclib/qrpglesrc,prototypeb
 /copy mysrclib/qrpglesrc,usec
 /copy mysrclib/qrpglesrc,variables3
 *          ... etc. ...
 * Set to blank environment variable QUERY_STRING
C                   callp     putenv('QUERY_STRING=':qusec)
```

Timing functions

The following timing functions allow to compute the number of seconds elapsed since a given moment. They are generally used to compute the response time of a program.

1. TimerStart

Use subprocedure `timerStart` to start computing the elapsed time.
Example:

```
 *    Set timer for calculating execution time
C                   callp     TimerStart()
```

1. TimerElapsed

Use subprocedure `timerElapsed` to receive the number of seconds elapsed since the last `timerStart`.
Example:

```
 * Program timing variable
D sec             s               15p 6
C                   eval      sec = TimerElapsed()
C                   callp     updhtmlvar('runtime':%editc(sec:'N'))
C                   callp     wrtsection('runtime')
```

IFS subprocedures

- **ChkIfsObi2: check an IFS object**
- **ChkIfsObi3: check an IFS object**
- **LoadStmf: load a stream file**

ChkIfsObi2: check an IFS object

Subprocedure `chkIFSObj2` checks whether an IFS object exists and can be accessed. If so, it also returns some information about it.

Note 1. This procedure was named `chkIFSObj2` to distinguish it from subprocedure `chkIFSObj` made available by Giovanni B. Perotti for freeware IFSTOOL.

chkIFSObj2

- Checks IFS object's existence and optionally returns its type, size, and error information.
- No authority to the object is required to use this subprocedure.
- *X authority is required for all subdirectories in the object's path. If this authority is lacking, the object is not accessible.
- If the object is found and is accessible, ChkIfsObj2 returns *on. Otherwise, it returns *off. See parameters, below, for more details.
- If you don't care about the object's type or size or error details, all parameters except the first are optional.

Required parameter group:

Returned value:	indicator	*on =object exists and can be accessed *off =object does not exist or cannot be accessed
Parameters:	Null terminated string of complete path to the object	
	11a (optional)	object type (if successful)
	10i 0 (optional)	number of bytes (if successful)
	10i 0 (optional)	error number - if successful, contains 0 - otherwise, contains C's error number
	256a (optional)	error text - if successful, contains a zero length string - otherwise, contains the C message text associated with C's error number

Coding examples:

-

```
D indicator        s                 n
D ifsObj           s              256a   inz('/home/joe/x.y')
D objType          s               11a   varying
D objSize          s               10i 0
D rc               s               10i 0
D errText          s              256a   varying
C                 eval            indicator = ChkIfsObj2(%trim(ifsObj):
C                                 objType:objSize:
C                                 rc:errText)
```

-

```
* if you only want to find out if the object is accessible
C                   eval      indicator = ChkIfsObj2('/home/joe/x.y')
* if you also want the object's type
C                   eval      indicator = ChkIfsObj2('/home/joe/x.y':
C                             objType)
* if you also want the object's size
C                   eval      indicator = ChkIfsObj2('/home/joe/x.y':
C                             objType:objSize)
* if you also want C's errno & description when a failure occurs
C                   eval      indicator = ChkIfsObj2('/home/joe/x.y':
C                             objType:objSize:
C                             rc:errText)
```

ChkIfsObi3: check an IFS object

Subprocedure `chkIFSObj3` checks whether an IFS object exists and can be accessed. If so, it also returns more information about it than subprocedure *chkIfsObj2*.

chkIFSObj3

- Checks IFS object's existence and optionally returns its type, size, creation timestamp, codepage, CCSID and error information.
- No authority to the object is required to use this subprocedure.
- *X authority is required for all subdirectories in the object's path. If this authority is lacking, the object is not accessible.
- If the object is found and is accessible, ChkIfsObj2 returns *on. Otherwise, it returns *off. See parameters, below, for more details.
- If you don't care about the object's type, size, creation timestamp, codepage, CCSID or error details, all parameters except the first are optional.

Required parameter group:

Returned value:	indicator	*on =object exists and can be accessed *off =object does not exist or cannot be accessed
Parameters:		Null terminated string of complete path to the object
	11a (optional)	object type (if successful)
	10i 0 (optional)	number of bytes (if successful)
	z (optional)	creation timestamp (if successful)
	5u 0 (optional)	codepage (if successful)
	5u 0 (optional)	CCSID (if successful)
	10i 0 (optional)	error number - if successful, contains 0 - otherwise, contains C's error number
	256a (optional)	error text - if successful, contains a zero length string - otherwise, contains the C message text associated with C's error number

Coding examples:

-
```
D indicator       s              n
D ifsObj          s            256a   inz('/home/joe/x.y')
D objType         s             11a   varying
D objSize         s             10i 0
D objCrtStamp     s               z
D objCodepage     s              5u 0
D objCCSID        s              5u 0
D rc              s             10i 0
```

```
D errText          s              256a   varying
C                  eval       indicator = ChkIfsObj3(%trim(ifsObj):
C                             objType:objSize:
C                             objCrtStamp:
C                             objCodePage:objCCSID:
C                             rc:errText)
```

-
```
* if you only want to find out if the object is accessible
C                  eval       indicator = ChkIfsObj3('/home/joe/x.y')
* if you also want the object's type
C                  eval       indicator = ChkIfsObj3('/home/joe/x.y':
C                             objType)
* if you also want the object's size
C                  eval       indicator = ChkIfsObj3('/home/joe/x.y':
C                             objType:objSize)
* if you also want the object's creation stamp
C                  eval       indicator = ChkIfsObj3('/home/joe/x.y':
C                             objType:objSize:
C                             objCrtStamp)
* if you also want the object's codepage
C                  eval       indicator = ChkIfsObj3('/home/joe/x.y':
C                             objType:objSize:
C                             objCrtStamp:
C                             objCodepage)
* if you also want the object's CCSID
C                  eval       indicator = ChkIfsObj3('/home/joe/x.y':
C                             objType:objSize:
C                             objCrtStamp:
C                             objCodepage:CCSID)
* if you also want C's errno & description when a failure occurs
C                  eval       indicator = ChkIfsObj3('/home/joe/x.y':
C                             objType:objSize:
C                             objCrtStamp:
C                             objCodepage:CCSID:
C                             rc:errText)
```

LoadStmf: load a stream file

Subprocedure `loadStreamFile` loads a stream file in memory.
This may be useful when the stream file data must be processed. An example could be writing the stream file data to the html output buffer through the `wrtNoSection` subprocedure.

Warnings:

1. Stream files exceeding the 16 MB size (16,776,704 byte) cannot be loaded in memory.
2. The user program is responsible, after calling this subprocedure, for releasing the memory dynamically acquired by the subprocedure (see the example below).

Required parameter group:

Return code:	0 = successful operation
	-1 = IFS object not found
	-2 = not a stream file
	-3 = stream file size is 0
	-4 = stream file size exceeds 16 Mb
	-5 = cannot allocate memory
	-6 = stream file cannot be opened
Input parameters:	• *Stmf*: path and name of the stream file
	• *DataType*:
	○ BIN - do not perform any CCSID converstion, take the data as they are
	○ TEXT - convert the data to the CCSID of the job

Output parameters:	• *DataLength*: length of stream file data
	• *DataPointer*: pointer to the memory area containing the stream file data

Example:

```
D stmf            s             1024    varying
D rc              s               10i 0
D dataLength      s               10i 0
D dataPointer     s                 *
D data            s             1000    based(dataPointer)
 /free
      stmf='/cgidev/conf/httpd.conf';
      rc=LoadStreamFile(stmf:'TEXT':DataLength:DataPointer);
      if rc=0;
         // ... process the stream file data in memory, pointed by "dataPointer" ...
         dealloc(n) DataPointer;  // release the allocated memory
      endif;
      return;
```

Uploading PC files

On May 20, 2009, CGISRVPGM2 subprocedure *zhbGetInput()* has been added the ability to upload PC files.

The work was done by **Ron Egyed**, *RJE Consulting Inc, New Port Richey (FL), U.S..*

The way this feature works is very simple:

- If a <form ...> includes the parameter **enctype="multipart/form-data"**, it can upload PC files.
- A nice way to let the user browse the PC an pick up the file to be uploaded is that of using
  `<input type="file" name="`***namexxx***`" size="..." device="files">`
  where ***namexxx*** is a name of your choice.
- When the form is submitted, a copy of the PC file is sent to the HTTP server
- As soon as the CGI program runs subprocedure *zhbGetInput()*, the file is uploaded to IFS directory **/tmp**
- Subprocedure *zhbGetInput()* provides two input variables that can be received by subprocedure *zhbGetVar()*:
  - the first, named ***namexxx***, contains the name of the PC file (e.g. *mytext.txt*).
    Note. Usually browsers, when uploading a file to a server, transfer just the PC file name, not its path. This is done for security reasons. However, some browser may also transfer the file path, which would then be shown in this variable (e.g. *C:\mypath \mytext.txt*). This is for instance done by Internet Explore when the following option is enabled (default case): *Tools-> Internet Options-> Security-> Customized level-> Include local path during a file upload to a server*.
  - the second, named ***namexxx*_tempfile**, contains the path and the name of the IFS file uploaded from the PC file. Files are always uploaded to IFS directory **/tmp** and are assigned unique names (e.g. */tmp/mytext_395935_20110128094216294000.txt*). It is then up to the CGI program to rename the uploaded stream file and to move it to the appropriate IFS directory.
  Please note that ***namexxx*** is the name you have assigned to the input variable in your form.
- The CGI program is of course able to receive via subprocedure *zhbGetVar()* any other input parameter sent from the form.

A sample program taking advantage of the file-upload feature is the ILE-RPG CGI program CGIDEV2/UPLOAD.
To run it click here
To display its source click here
To display its external HTML click here.

For more details on this technique, take a look at the Easy400 FUPLOAD utility, this page.

Validating a file upload request

Though the ability to upload files sounds great, there might be a need to restrict it to some users or to some file types (extensions).

There are two ways to perform such a validation:

1. Client validation

In principles, this is the best approach, as the validation process takes place on the client and is immediate. What you need is some JavaScript function validating the file to be uploaded. This Javascript function must then be made available in the external HTML of your upload program.

The only validation that makes sense on the client side is on the extension of the file to be uploaded.

As an example, the external HTML of CGI program CGIDEV2/UPLOAD contains a JavaScript validation function named ***ValidateExtension()***. This function works on three arguments:

 a. the name of the file to be uploaded
 b. a constant (possible values 'yes' or 'no') telling whether extension validation should take place
 c. an array of allowed extensions

Parameters b) and c) should be customized according to the installation needs. However, as this may be an hazard, a special command - **cgidev2/updalwext** - has been developed to customize these parameters, which are then set in the script from program UPLOAD as output variables.

2. Server validation

A file upload goes through two stages:

 a. The PC file(s) are trasmitted to the server along with any other input field. This is done by the HTTP server.
 b. The application program (the CGI program) takes care of receiving the input variables and the input file(s). This occurs when CGIDEV2/CGISRVPGM2 subprocedure **ZhbGetInput()** is invoked. Usually ZhbGetInput() would copy the input file(s) to IFS stream file(s) in directory */tmp*.

   However, before creating an IFS stream file, subprocedure **ZhbGetInput()** checks whether an *Exit Point Validation User Program* is available and, if so, asks it to validate the file.

   - If the validation is successful, the PC file is uploaded to an IFS stream file.
   - If the validation fails, no IFS stream file is created, the name of the file is returned as

     ```
     *** NOT VALIDATED ***
     ```
     and an error message is written to the CGIDEBUG file.

What you could then do is to write such a Validation User Program and make it available for the appropriate Exit Point. This is how you do it:

i. Writing the upload validation program

   - The validation program receives two parameters, the qualified name (path, file name and extension) of the IFS stream file to be created and a return code:

   ```
   D UPLOADVAL       pr
   D  filename                  1024    varying
   D  retcode                     10i 0
   D UPLOADVAL       pi
   D  filename                  1024    varying
   D  retcode                     10i 0
   ```

   - A value **0** (zero) of the return code means that the PC file passed the validation, a value **-1** means that the validation was not passed (failed).
   - Most sensitive items for validation are the file extension and the user name (the user name, to be available, requires user validation through the appropriate HTTP directives).
   - The validation program could as well return a different qualified name for the IFS stream file to be created.
   - As an example, you could look at program CGIDEV2/UPLOADVAL, press here to display its source. Please note that this validation program accepts only files with extension *csv*.

ii. Making the upload validation program available to the Exit Point

   1. Run command **cgidev2/updexitp**. The following screen appears:

   ```
                         Update Exit Points
   ```

```
Type option, press Enter.
  2=Change

  Exit point              User program
_ FILE-UPLOAD-001



  F3=End
```

2. Type **2** in front of the `FILE-UPLOAD-001` exit point name to receive the following screen:

```
                Update Exit Points

 Exit point . . . . . . . . FILE-UPLOAD-001
 User program . . . . . . . _____
   Library  . . . . . . . .   _____

 F3=End  F12=Cancel
```

Then type the name and the library name of your upload validation program.
Just as an example you could specify program CGIDEV2/UPLOADVAL.
We suggest to create your own file upload validation program in some library of yours.
Never change CGIDEV2 programs, nor develop objects in library CGIDEV2:
when installing the next CGIDEV2 release your changes would disappear.

CGI debug file

In debugging your CGI programs you may hit cases where you would need to see how

- input string to your program
- HTML output from your program

look like.

In Mel's service program facilities exist to trace input/output html and here is how you can implement them to debug your CGI programs.

1. File CGIDEBUG

Mel's service program may trace the input string and the response html on a file named **CGIDEBUG**. You implicitly duplicate it from library **cgidev2** to your object (production) library when you use command setcgilib.

2. Start/Display/End CGIDEBUG

- You start html trace with command
      mylib/CGIDEBUG ACTION(*ON)
- You display html trace with command
      mylib/CGIDEBUG ACTION(*DSPDATA)
- You end html trace with command
      mylib/CGIDEBUG ACTION(*OFF)
- You clear html trace with command
      mylib/CGIDEBUG ACTION(*CLRDATA)

Warning

At the end of your debugging session you **must remember to end your html trace**. If you leave the trace on, sooner or later you will fill up file CGIDEBUG, and your CGI will start bumping out!!!

3. Clear the CGIDEBUG file

You may need to clear the CGIDEBUG files from time to time. You cannot use the `clrpfm` command, because these files are locked by the HTTP server.
You should instead use command
    `cgidev2/cgidebug *CLRDTA` or
    `cgidev2/clrdebug`   (use F4 to specify the library name)
after adding CGIDEV2 to the library list.

A tip

CGIDEBUG is far from being your ultimate debugging facility for CGIs. Please read page CGI debugging tips.

Debugging functions

If you decide to use the CGI file debug trace facilities, you may find usefull some Mel's service program functions that allow your CGI's to write their own specific pieces of information to this file.

The following procedures are available for use in your CGI programs (use opcode *callp* to invoke them:

1. **isdebug**
   returns a 1-char value to indicate whether debugging is on ('1') or off ('0').
2. **wrtdebug**
   writes into the debugging physical file, CGIDEBUG, the text passed to it as a parameter. WRTDEBUG is used by several of the service program's subprocedures. You can use it, as desired. No output is generated unless debugging output has been turned on by the `CGIDEBUG *ON` command or the optional parameter, `force`, has been set to `*ON`.
3. **wrtjobdbg**
   writes the qualified job name, current date, and current time into the debugging file.
4. **wrtpsds**
   receives the program status data area and unconditionally writes it in a formatted manner into the debugging file.
5. **SetNoDebug**
   turns off all conditionally or unconditionally debugging, thus improving the performance of a CGI.

Examples

```
/copy mysrclib/qrpglesrc,hspecs
/copy mysrclib/qrpglesrc,hspecsbnd
* Variables common to all CGIs
/copy mysrclib/qrpglesrc,prototypeb
/copy mysrclib/qrpglesrc,usec
/copy mysrclib/qrpglesrc,variables3
*          ... etc. ...
* Example of using
*   wrtdebug(text:force)
C                 callp     wrtdebug(PgmName +
C                           ' execution time (seconds) ' +
C                           %trim(%editw(sec:'     0 .   '))):*on)
 * Example of writing qualified job name to debug file.
 *   Note that the force parameter is set to *on
C                 callp     wrtjobdbg(*on)
&nspp;* Example of sending psds data to cgidebug physical file
C                 callp     wrtpsds(psds)
```

In your CGI programs you may use a **program status data structure** and a **program status subroutine** to trap program status error and to

- notify the client user that an error has occurred
- use function **wrtpsds** to format and write the contents of the program status data structure to the CGIDEBUG file.

See how this is implemented in program `CGIDEV2/TEMPLATE`.

```
F                                       infsr(*pssr)
 * Prototype definitions and standard system API error structure
 /copy cgidev2/qrpglesrc,prototypeb
 /copy cgidev2/qrpglesrc,usec
 *
 * For program status data structure and program status subroutine
D psds            sds
D   psdsdata                       429
D pssrswitch      s               1    inz(*off)
D wrotetop        s               1    inz(*off)
 *
 ************************************************************************
 * Program status subroutine
 ************************************************************************
C     *pssr         begsr
 * If have already been in pssr, get out to avoid looping
C                   if        pssrswitch=*on
C                   eval      *inlr = *on
C                   return
C                   endif
 * Set on switch to indicate we've been here
C                   eval      pssrswitch=*on
 * Write HTML sections (top if not already done, pssr, and *fini)
C                   if        wrotetop=*off
C                   callp     wrtsection('top')
C                   endif
C                   callp     wrtsection('pssr endhtml *fini')
 * Send psds data to cgidebug physical file
C                   callp     wrtpsds(psds)
C                   eval      *inlr = *on
C                   return
C                   endsr
```

Data conversion functions

Mel's service program includes several data conversion functions you may take advantage from.

You may find examples about these functions by scanning through PDM the source file QRPGLESRC in library CGIDEV2.

A. **Conversion procedures from module XXXDATA**
   1. Subprocedure **char2hex** converts a character string to its hexadecimal representation.
   2. Subprocedure **hex2char** converts a character string in hexadecimal format to its character representation.
   3. Subprocedure **chknbr** accepts a character string and an optional parameter specifying the maximum number of digits to the left of the decimal point. Additional optional parameters are available to request that any errors found should be formatted as messages and added to the service program's message arrays, the text that should be used to describe the field in the messages, and whether a message should be sent if the field's value is less than zero.
   Chknbr returns a structure containing seven indicators. The indicators and their meaning when *on are:
      1. one or more errors occurred
      2. non-numeric characters (includes minus sign in wrong place)
      3. multiple decimal points
      4. multiple signs (both leading and trailing)
      5. zero length input or no numeric characters
      6. too many digits to the left of the decimal point
      7. no errors, but value is less than 0.
   Note 1. Indicator 7 *on does not set indicator 1 *on.
   4. Subprocedure **c2n** converts a character string to a floating point number. It is recommended that you check the string with chknbr before calling c2n.
   5. Subprocedure**c2n2** converts a character string to a packed 30.9 number. c2n2 performs faster than c2n and has none of c2n's floating point precision problems. Therefore, it is recommended that you use c2n2 instead of c2n. It is recommended that you check the string with chknbr before calling c2n2.
   6. Subprocedure**xlatWCCSIDs** uses CCSIDs to translate variable length strings up to 32767 characters in length.
   If optional parameters fromCCSID and toCCSID are specified, they are used for the translation. Otherwise, translation between ASCII and EBCDIC is performed using the CCSIDs found in the CGI_EBCDIC_CCSID and CGI_ASCII_CCSID environment variables. Input parameter, toebcdic, is used to determine whether translation is from ASCII to EBCDIC (*on) or from EBCDIC to ASCII (*off).

   7. Subprocedure **uppify** converts all the characters in a string to upper case.
   An optional CCSID parameter may be used to support non-english language characters.
   Examples:
```
D  ccsid                          10i 0
 * If you want the best possible performance and the
 * English language characters are sufficient, do
 * not use the CCSID parameter.
C                   eval charstring =  uppify(charstring)
 * To convert to uppercase a non-english language
 * character string, you must pass the correct CCSID
 * as second parameter.
 * This takes 2 times as long as using no CCSID.
 * For instance, if the character string is in swedish language:
C                   eval ccsid = 278
C                   eval charstring =  uppify(charstring:ccsid)
```

```
 * A value 0 for the CCSID parameter instructs the uppify
 * procedure to use the job CCSID.
 * This takes 3 times as long as using no CCSID.
C                   eval charstring =  uppify(charstring:0)
```

8. Subprocedure **lowfy** converts all the characters in a string to lower case.
An optional CCSID parameter may be used to support non-english language characters.
Examples:

```
D  ccsid                     10i 0
 * If you want the best possible performance and the
 * English language characters are sufficient, do
 * not use the CCSID parameter.
C                   eval charstring =  lowfy(charstring)
 * To convert to uppercase a non-english language
 * character string, you must pass the correct CCSID
 * as second parameter.
 * This takes 2 times as long as using no CCSID.
 * For instance, if the character string is in swedish language:
C                   eval ccsid = 278
C                   eval charstring =  lowfy(charstring:ccsid)
 * A value 0 for the CCSID parameter instructs the uppify
 * procedure to use the job CCSID.
 * This takes 3 times as long as using no CCSID.
C                   eval charstring =  lowfy(charstring:0)
```

B. **Conversion procedures from module XXXDATA1**
   ○ When using the GET method to send input to a CGI program, non-alphanumeric characters in the query string must be replaced by so called "escape sequences".
   An escape sequence is made of
      ■ an escape character "%"
      ■ followed by two characters which represent the hexadecimal value of the corresponding ASCII character.
   For instance, if the query string contains the following input
   ```
   cusname=Van der Meer
   ```
   then each of the two spaces in "Van der Meer" must be replaced by the escape sequence %20 (as the ASCII representation of a space character is x'20').

   The tool to replace non-aplhanumeric characters with the corresponding escape sequences is subprocedure **UrlEscSeq**:

   ```
   D inpString       s          32767    varying
   D outString       s          32767    varying
    *
   C                 eval      outString=UrlEscSeq(inpString)
   ```
   A "trim right" indicator can be optionally passed to subprocedure *UrlEscSeq*.
   If it is not passed, or if it is passed and it is *on, the input string is trimmed right before being processed.
   If it is passed and it is *off, the input string is not trimmed right (trailing blanks are converted to escape sequences %20):

   ```
   D inpString       s          32767    varying
   D outString       s          32767    varying
   D trimRightInd     s                   n
    /free
        trimRightInd=*off;
        outString=UrlEscSeq(inpString:trimRightInd);
   ```
   You may use CGI program TSTESCSEQ to display the result of converting an input string to an output string containing escaped sequences. In this example, the input string is trimmed right before being processed.

   ○ Subprocedure **UrlUnEscSeq** may be used to convert back a string containing escape sequences, for

instance to convert the string "Van%20der%20Meer" to "Van der Meer":

```
D inpString       s            32767     varying
D outString       s            32767     varying
 *
C                 eval         outString=UrlUnEscSeq(inpString)
```

C. **Conversion procedures from module XXXWRKHTML**
- o When displaying database fields in a HTML page, it may happen that some data containing special HTML characters are interpreted as HTML tag delimiters thus generating ununderstandable strings. On the other way, multiple consecutive blanks in a field are displayed as a single space, which in some cases may be unappropriate.
  The following three subprocedures allow to convert special characters and blanks into their corresponding HTML character entities, in order to display field data exactly as they are on databases. All procedures require
  - ▪ the input and the ouput fields be defined with *varying* length not exceeding 32767

1. Subprocedure **encode**
   converts the following special characters to their HTML entity equivalents:

"	is converted to	"
&	is converted to	&
<	is converted to	<
>	is converted to	>

2. Subprocedure **encodeBlanks**
   converts blanks to non-breaking spaces ( ).
   This procedure is an alternative to use the traditional HTML tags <pre> and </pre>.

   Examples:

   ▪
   ```
   D VarOutput       s            1000     varying
     ................
    /free
        read record;
        dow not %eof;
            VarOutput=%trimr(recordField);
            VarOutput=Encode(VarOutput);
            VarOutput=EncodeBlanks(VarOutput);
            updhtmlvar('htmlvar':VarOutput);
            wrtsection('TableRow');
            read record;
        enddo;
    /end-free
   ```

   ▪
   ```
    /free
        read record;
        dow not %eof;
            updHtmlVar('htmlvar':EncodeBlanks(
                Encode(%trimr(recordField))));
            wrtsection('TableRow');
            read record;
        enddo;
    /end-free
   ```

3. Subprocedure **encode2**
   allows to translate special characters to their corresponding named entities as documented in a user specified stream file. If not provided, the stream file defaults to /cgidevexthml /encod2arr.txt .

Required parameter group:

returned value:	65528 char max,	input string with special characters converted to HTML named entities	
inputs:	1891 char max,	input string with special characters to be converted to HTML named entities	
	10i 0,	return code:	0 =successful
			-1 =file error. Could be any of the following:
			0. file not found
			0. file not accessible (authority, etc.)
			0. file empty
			0. file contains no valid records (at run time, a detailed message is sent to the CGIDEBUG debugging file)
	256 char max, (optional)	"entities" stream file, the file that contains the arrays of characters and character entities. If omitted, file `/cgidevexthtml/encode2fil.txt` is used.	

Note 2. Click here to display stream file `/cgidevexthtml/encode2fil.txt` .

Note 3. Click here to run CGI `dspencode2`. This program uses procedure `encode2` to test an "entity" stream file.

Note 4. To customize the arrays:

- Never modify, move, or rename the default stream file `/cgidevexthtml /encode2fil.txt` .
- Copy the default stream file to an IFS file of your own.
- Make sure QTMHHTP1 has *RX authority to your file.
- Modify your file:
    - Record format, one record per line
    - Comment records:
        - positions 1 -2 must be //
    - Data records:
        - Position 1 = the character to be encoded
        - Positions 2 - 9 = the character entity to be substituted for the character. If these positions are blank, the record is ignored.
        - Remainder of record = blanks or comments
- Use your file in the EntitiesFile parameter.

Coding examples:

-

```
D inputString      s             1891     varying
D outputString     s            65528     varying
D dftFile          s              256
D rc               s               10i 0 inz(0)
 /free
      dftFile='/cgidevexthtml/encode2fil.txt';
      outputString=encode2(inputString:rc:dftFile);
```

-

```
 /end-free

* Passing a literal
C                   eval      result = encode2('':rc)
* Passing a varying field.
C                   eval      vfield = ''
C                   eval      result = encode2(vfield:rc)
* Passing from a fixed length field
C                   eval      ffield = ''
C                   eval      result = encode2(%trimr(ffield):rc)
* Passing an expression
C                   eval      result = encode2('abc' +
```

```
C                                          %trimr(ffield) +
C                                          vfield + 'xyz':rc)
```

D. **Conversion procedures from module XXXCVTSTG**

○ Subprocedure *CvtStg (Convert String)* can be used to convert a memory string from one CCSID to another CCSID. For instance you could convert a memory string from CCSID 1208 (Unicode, UTF-8 Level 3) to CCSID 37 (EBCDIC USA, Canada, Netherlands, Portugal, Brazil, Australia, New Zealand). This is how you use it:

```
 *  "InpCCSID" is the CCSID of the input string
 *           (the string to be converted)
 *  "InpBufP" is a pointer to the input string
 *  "InpBufLen" is the length on the input string
 *  "OutCCSID" is the CCSID of the output string
 *           (the string to receive the converted value)
 *  "OutBufP" is a pointer to the output string
 *  "OutBufLen" is the length of the output string
 *           (this length must be large enough
 *            to contain the converted value)
 *  "OutDtaLen" is the length of the converted value
 *            once it is converted in the output string
D  InpCCSID       s             10u 0
D  InpBufP        s               *
D  InpBufLen      s             10u 0
D  OutCCSID       s             10u 0
D  OutBufP        s               *
D  OutBufLen      s             10u 0
D  OutDtaLen      s             10u 0
 /free
      // Convert a memory string from CCSID 1208 to CCSID 37
      InpCCSID=1208;
      OutCCSID=37;
      CvtStg(InpCCSID:InpBufP:InpBufLen;
             OutCCSID:OutBufP:OutBufLen:OutDtaLen);
```

Execute a command
using DoCmd subprocedure

This subprocedure allows an RPG-ILE program to run a CL command.

Example: Override a database file

```
 * Variables common to all CGIs
 /copy mysrclib/qrpglesrc,prototypeb
 /copy mysrclib/qrpglesrc,usec
 /copy mysrclib/qrpglesrc,variables3
 *          ... etc. ...
 * Override database file CTRDVY
C                   eval      rc = docmd('OVRDBF FILE(CTRDVY) +
C                             TOFILE(CGIDEV2/CTRDVY) +
C                             SECURE(*YES)')
```

Note

On return from **docmd**, variable **rc** (return code) contains

- **0**, if the command was executed
- **1**, if the command failed.

Write HTML to a stream file
(*Dynastic* pages)

There may be cases, where a page created from a CGI (**dynamic page**)

- is frequently accessed
- requires some relevant processing, thus providing a rather long response time
- and its contents may change at unpredictable times

Examples of such cases could be:

- our page about "our subscribers per country"
- many other *statistics* pages

One interesting way to serve such requests is to have some **dynamic** processes generating **static** pages:

- user response is much faster
- computer load much lower

- the generation of a static page may be triggered by an event or scheduled at a regular time interval

Implementing such a process is quite easy: you have to write a batch program similar to a CGI and gear it to some event or schedule.

Here is how you may write such a program:

1. Develop the external HTML as usual for a CGI program, but
     i. do not insert the initial http header
        `Content-type: text/html`
        (you are going to build a static page)
2. Develop the program as if it were a CGI program using our CGISRVPGM2 service program (you may use our command cgidev2/crtcgisrc to create an initial sample CGI source)
     i. avoid reading and parsing the input string
        (drop our cgidev2/crtcgisrc generated statement
        /copy .../qrpglesrc,prolog1/2/3
)
     ii. instead of sending the html buffer to the client with the instruction
        callp   wrtsection('*fini')
        save it as an IFS stream file using the subprocedure **WrtHtmlToStmf()**:

```
D  Stmf            s            512    varying
D                                      inz('/web/dynapage1.html')
D  CodePage        s             10i 0 inz(819)
D  rc              s             10i 0
 * If "codepage" parameter omitted, code page is assigned by the
 * system
C                  if         codepage > 0
C                  eval       rc = WrtHtmlToStmf(Stmf:CodePage)
C                  else
C                  eval       rc = WrtHtmlToStmf(Stmf)
C                  endif
```

Notes:
     i. The filename portion in the "stream file" variable supports a maximum length of 245

bytes.
  ii. Create the program with **`actgrp(*new)`**.
  iii. Subprocedure WrtHtmlToStmf() clears the output buffer before returning to the user program. This is done to allow the user program to create more than one stream file without overlaying previous output data.
  iv. Should you need instead to *append* the html buffer to an existing stream file, you must use subprocedure **AppHtmlToStmf()**:

```
D  Stmf            s             512    varying
D                                       inz('/web/dynapage1.html')
D  rc              s              10i 0
C                   eval       rc = AppHtmlToStmf(Stmf)
```

Example of a dynamic/static page:

- Please browse the source of our example program. It would generate a page containing some random integers. To run this program:
  1. `addlible cgidev2`
  2. enter command
     `CGIDEV2/RANDOMNBRS STMF('/cgidev/randomnbrs.htm') CODEPAGE(819)`
- Check out the generated static page.

Error codes.

If subprocedure **wrtHtmlToStmf** does not complete normally, it returns a non-zero error code (in field "rc"). Error codes are documented in an IBM Infocenter separate page.

Supporting program state through User Spaces

1. **Introduction**
2. **User spaces**
3. **User space procedures**
4. **Sample program**

1. Introduction

Non-persistent CGI are "stateless". In other words, the browser connects to the server, makes a request (calls a CGI), receives the response, then disconnects from the server. On the next request from the same browser, there is no guarantee that the same program has mantaned the state (variables and record positioning) it was left with at the end of the previous request.

In such a stateless situation, the developer has to implement some "tricks" to restore some variables on the next call to the CGI, such as

- using hidden input fields in the HTML forms
- writing and retrieving cookies

Sometimes, however, when designing complex transactions, these rather simple tricks may not be sufficient to fulfill state requirements and a programmer would be tempted to use "persistent" CGI instead. However, persistent CGI have their own problems.

In those cases, better methods for storing and retrieving state information are required.

2. User spaces

iSeries user spaces objects are ideal for this purpose:

- Each user space can hold up to 16 MB of information
- System APIs are provided to create a user space, change its attributes (including making it automatically extendible, retrieve a pointer to it, etc.)
- Once addressability to a user space (a pointer) has been established, one can map, using based variables, many types of data into it, including data structures
- Saving and restoring user state information can be accomplished as follow:
    - at the start of the transaction, create a uniquely named user space,
    - use based variable(s) to map the user's data into the space,
    - send an HTTP response to the user, including a hidden field containing the user space name,
    - when the user makes a request using the form that contains the hidden user space name, use that name to retrieve a pointer to the user space, thus restoring addressability to the user space's contents using the same based variables that were used to store them.

3. User space procedures

1. **Subprocedure "CrtUsrSpc" - Create an User Space**
   It creates a randomly named, automatically extendible user space in a user-specified library.

The user space's contents are initialized to all x'00's.

Parameters

- User space library (input)
  - If the library not found, CrtUsrSpc sets the user space name to blanks and MsgId to CPF9810
  - If the requestor does not have change authority to the library, CrtUsrSpc sets the user space name to blanks and MsgId to CPF2144
- Pointer to user space (output)
  - Set to null if the user space is not created
- Message ID (output)
  - blank if no errors
  - else, message id of error

Optional Parameters

- Public authority (input)
  - If not passed, it is set to *EXCLUDE
- Text (input)
  - If not passed, it is set to 'Created by CGIDEV2' plus timestamp
- Initial size (input)
  - If not passed, it is set to 12288
- Extended attribute
  - If not passed, it is set to blanks

Returns

- If successful
  - User space name
- Otherwise
  - Blanks

Errors in system APIs

- If any of the called system APIs fails, a message is forced into the CGIDEBUG file.

Example:

```
 * Input variables
D AnEntry         s              40    varying
 * User space
D UsrSpcName      s              10
D UsrSpcLib       c                    'CGIDEV2USP'
* Message ID for CrtUsrSpc
D MsgId           s               7
* State related variables (user space contents)
D State           ds                   based(StateP)
D  Count                         10i 0
D  Entries                     1000    varying
C                 eval       UsrSpcName= CrtUsrSpc(
C                               UsrSpcLib : StateP : MsgID)

```

In this example, a user space is created in library CGIDEV2USP and its name is loaded into variable "UsrSpcName".

2. **Subprocedure "RtvUsrSpcPtr" - Retrieve Pointer to User Space**

Parameters

- User space name (input)
- User space library (input)

Returns

- ○ If successful
  - ■ Pointer to the user space
- ○ Otherwise
  - ■ Null pointer

Errors in system APIs

- ○ If any of the called system APIs fails, a message is forced into the CGIDEBUG file.

Example:

```
C                  eval      StateP = RtvUsrSpcPtr(UsrSpcName:
C                            UsrSpcLib)
C                  if        StateP = *null
                   ... ...
C                  endif
```

In this example, the contents of the user space are made accessible via data structure "State".

To update the contents of the user space, one should just update the data structure "State":

```
C                  eval      AnEntry = ZhbGetVar('AnEntry')
C                  if        AnEntry <> ''
C                  eval      Count = Count + 1
C                  eval      Entries = Entries + '<br>' + AnEntry
C                  endif
```

4. Sample program

CGI program STATE demonstrates the use of user space to maintain program state information.
In this programs, the user may enter, one at a time, several inputs. The inputs are saved in a user
space. The user space contents are displayed to the user.

- HTML source of this example
- RPG source of this example
- Run this example

Generate a random string
using randomString subprocedure

←M→

Subprocedure **randomString** returns a string up to 1024 random characters. The caller controls the number of characters returned, the contents and case of the first character, and the contents and case of the remaining characters.
This procedure can be used for assigning temporary names for user spaces, stream files, file members, etc.

Parameters:

- Number of characters to return (0 - 1024)
  If 0, a null string is returned.
  If > 1024, 1024 characters are returned.
- First character (if not passed, defaults to *mixedDigit)
  - *upperLetter (upper case letter only)
  - *lowerLetter (lower case letter only)
  - *mixedLetter (upper or lower case letter only)
  - *upperDigit (upper case letter or digit)
  - *lowerDigit (lower case letter or digit)
  - *mixedDigit (upper or lower case letter or digit)
  - *digit (digit only)
- Remaining characters (if not passed, defaults to *mixedDigit)
  - same choices as first character
- UpperChars - characters that are "upper case")
  - If not passed or has length = 0, defaults to 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
- LowerChars - characters that are "lower case")
  - If not passed or has length = 0, defaults to 'abcdefghijklmnopqrstuvwzyz'
- DigitChars - characters that are "digits")
  - If not passed or has length = 0, defaults to '0123456789'

For more information, see CGIDEV2/QRPGLESRC member XXXRANDOM.

Example:

```
D UsrSpcName      s              10
C                   eval      usrSpcName = RandomString(
C                             10:'*UpperLetter':'*UpperDigit')
```

Generate a random integer
using Random subprocedure

This subprocedure returns a random integer between two user-specified values. A random integer is expecially useful to create a session ID ("handle") for persistent CGI programs.

Click here to see a live example of a CGI program using this subprocedure,
here to display the source of such sample program.

Example

```
/copy mysrclib/qrpglesrc,hspecs
/copy mysrclib/qrpglesrc,hspecsbnd
* Variables common to all CGIs
/copy mysrclib/qrpglesrc,prototypeb
/copy mysrclib/qrpglesrc,usec
/copy mysrclib/qrpglesrc,variables3
* Variables required by subprocedure "random"
DMyRandom         S              10u 0
DMyLow            S              10u 0
DMyHigh           S              10u 0
*         ... etc. ...
* Asks for a random integer
*  between 0 and 9, to be returned in variable "MyRandom"
C                   eval      MyLow = 1
C                   eval      MyHigh = 9
C                   eval      MyRandom = random(MyLow:MyHigh)
```

Assign a session ID

A persistent CGI, each time it is called, must compute a unique **session identifier** to be used as a *handle* to recall it back on the next user transaction. Click here to know more about persistent CGI requirements.

A random integer generation function may help in computing such unique session ID. Click to see how to compute a random integer.

Mel's service program includes also function **getSessionID**. This function returns a 15-character string made of

- the six characters of the job number
- nine random digits.

```
/copy mysrclib/qrpglesrc,hspecs
/copy mysrclib/qrpglesrc,hspecsbnd
* Variables common to all CGIs
/copy mysrclib/qrpglesrc,prototypeb
/copy mysrclib/qrpglesrc,usec
/copy mysrclib/qrpglesrc,variables3
*          ... etc. ...
* Example of computing a Session ID
C                   eval      sessionid = getsessionid
```

Commands to prepare libraries and sources for CGI development

1. Prepare your CGI libraries

Use command **cgidev2/setcgilib** to set up your source and production libraries for CGI programs:

```
          Set lib.s for CGI development (SETCGILIB)
CGI source library . . . . . . . _____   Name
CGI production library . . . . . *SRCLIB    Name, *SRCLIB
```

CGI source library: The name of the library which will contain the sources of the CGI programs. This command creates the following objects in this library:

1. Command, panel group, and program **COMPILE** to be used to regenerate your modules and programs any time you have to change something in your external data structure
2. the following source files, if missing
   - QDDSSRC
   - QRPGLESRC
   - QCLSRC
   - QCMDSRC
   - QPNLSRC

Note that source file QRPGLESRC will contain the following members (to be copied into your CGI sources):

- **hspecs** (H specifications for module compilation)
- **hspecsbnd** (H specifications for binding, so that you do not need any longer to specify the *bnddir* keyword in your *crtpgm* commands)
- **prototypeb** (prototypes for requesting services from Mel's service program
- **usec** (data structure for return codes from API's)
- **variables3** (variables common to CGI's)
- **prolog3** (get the input string sent from the remote browser)

Note that it is your responsibility to maintain CL program **compile** any time you develop a new CGI module or program.

CGI production library: The name of the library which will contain the CGI programs. If you specify *SRCLIB, the source library is also the production library.
This command creates the following objects in this library:

- Source file **HTMLSRC** for your html skeleton output members
- Command CGIDEBUG, with file CGIDEBUG and data area CGIDEBUG, to let you debug your html inputs and outputs.

IFS directory: An IFS directory with the same name of the CGI production library is created.
Three subdirectories are also created:

- **/***production_library***/css** for your .css files
- **/***production_library***/html** for your .html files
- **/***production_library***/graphics** for your graphical objects (icons, images, etc.)

HTTP directives: This command will also ask whether you want to generate original or Apache HTTP directives. If you did sign on with a user profile that has *change authority over HTTP instance control files (QUSRSYS/QATMHINSTC, QUSRSYS/QATMHTTPC), you will then be presented a list of HTTP instances (original or Apache) to choose from. Once you make a choice, the selected HTTP instance is updated with the basic HTTP directives needed to make your static or dynamic pages supported. We give examples for the two cases:

- **Original HTTP directives**

```
Map  /myprdlibh/*  /QSYS.LIB/MYPRDLIB.LIB/HTMLSRC.FILE/*
Pass /QSYS.LIB/MYPRDLIB.LIB/HTMLSRC.FILE/*
Pass /myprdlib/*
Exec /myprdlibp/*  /QSYS.LIB/MYPRDLIB.LIB/*  %%EBCDIC/EBCDIC%%
```

where *myprdlib* is the library name you specify for the production library. These four directives work as follow:
1-The Map directive allows you to specify the shortcut name */myprdlibh/* (instead of /QSYS.LIB /MYPRDLIB.LIB/HTMLSRC.FILE/) in your html scripts (thus saving keystrokes and related errors)
2-The first Pass directive allows HTTP to access members (containing static pages) in your HTMLSRC file
3-The second Pass directive allows HTTP to access files in a root directory named as your production

library: you could use such a directory to maintain images and static pages as well
4-The Exec directive allows CGIs in the production library to be executed. The %%EBCDIC/EBCDIC%% parameter allows the correct execution of `zhbGetInput` procedure (high performance procedure to read the input string from the remote browser).

○ **Apache HTTP directives**

```
AliasMatch /myprdlibh/(.*)\.htm /QSYS.LIB/MYPRDLIB.LIB/HTMLSRC.FILE/$1.mbr
Alias /myprdlibh/ /QSYS.LIB/MYPRDLIB.LIB/HTMLSRC.FILE/
Alias /myprdlib/ /myprdlib/
ScriptAliasMatch /myprdlibp(.*).pgm /qsys.lib/myprdlib.lib/$1.pgm
<Directory /QSYS.LIB/MYPRDLIB.LIB>
  AllowOverride None
  Options None
  order allow,deny
&%nbsp; allow from all
</Directory>
<Directory /myprdlib>
  AllowOverride None
  Options None
  order allow,deny
  allow from all
</Directory>
```

where *myprdlib* is the library name you specify for the production library. These directives work as follow:

1-The first directive defines a short path *myprdlibh/*.htm* through which one may invoke static pages in file MYPRDLIB/HTMLSRC using extension .htm

2-The second directive defines the short path *myprdlibh* which maps to MYPRDLIB/HTMLSRC

3-The third directives informs that IFS path "/myprdlib" can be used

4-The fourth directive allows execution of CGI programs in library MYPRDLIB. They must be invoked through their pseudo-path *myprdlibp*

5-The first Directory group allows object in library MYPRDLIB to be accessed (basically: static pages in MYPRDLIB/HTMLSRC and CGI programs in library MYPRDLIB)

6-The second Directory group allows IFS files in directory /myprdlib to be retrieved.

2. Create a sample CGI source

Use command **cgidev2/crtcgisrc** to create a sample ILE-RPG CGI source and the related HTML in source file HTMLSRC. The CGI is able to manage both the input (via ZhbGetInput and ZhbGetVar) from the client and the output to the client. You may easily create a module, create a CGI program, and run it. This will speed out your initial CGI developments.

```
         Create sample CGI RPG source (CRTCGISRC)
ILE-RPG CGI source member  . . .  _____  Name
CGI source library . . . . . . . _____  Name
CGI production library . . . . . *SRCLIB____   Name, *SRCLIB
```

ILE-RPG CGI source member: The name of source member to be created in file QRPGLESRC in the source library.

CGI source library: The name of the library which will contain the sources of the CGI programs.

CGI production library: The name of the library which will contain the CGI programs.

About persistent CGI

Table of Contents

1 - Introduction

Before OS/400 Release V4R3, CGI programs could only be run as *non-persistent*.
A non-persistent CGI program is reloaded at every browser request.
Because of this, there is only one way a non-persistent CGI program can know the values its variables had when it provided an html response to a client browser.
This is done by saving variable values in fields of the output html (usually "hidden" fields in an html form), so that they are sent back to the program with the next browser request.

Starting with OS/400 Release V4R3, CGI programs can be run as *persistent*.
Persistent CGI is an extension to the CGI interface that allows a CGI program to remain active across multiple browser requests and maintain a session with that browser client.
This allows

- the program state to be maintained
- files to be left open
- long running database transactions to be committed or rolled back on end-user input.

The AS/400 CGI program must be created using a named activation group which allows the program to remain active after returning to the server.
The CGI program notifies the server it wants to remain persistent using the "Accept-HTSession" CGI header as the first header it returns in the output html. This header defines the session ID associated with this instance of the CGI program, is taken off from the http server, and is not returned to the browser.
Subsequent URL requests to this program must contain the session ID as the first parameter after the program name.
The server uses this ID to route the request to that specific instance of the CGI program.
The CGI program should regenerate this session ID for each request.
Though not mandatory, it is strongly recommended that you use the Secure Socket Layer (SSL) for persistent and secure business transaction processing.

2 - Apache HTTP Directives for persistent CGI programs

There are three Apache HTTP directives for persistent CGI jobs.

1. **MaxPersistentCGI - Maximum number of persistent CGI jobs**
   Use this directive to set the maximum number of persistent CGI jobs that you want to have active at one time. The default is 50.

   Example

MaxPersistentCGI 10

2. **PersistentCGITimeout - Default timeout value for persistent CGI jobs**
   This directive specifies the number of *seconds* that your server waits for a client response before ending a persistent CGI session. The CGI program can override the value that you specify on a request-by-request basis. The default timeout value is 300 seconds.

   Example

PersistentCGITimeout 120

3. **MaxPersistentCGITimeout - Maximum timeout value for persistent CGI jobs**
   This is the maximum number of *seconds* that a CGI program can use when overriding the PersistentCGITimeout directive. The default timeout value is 1800 seconds.

   Example

MaxPersistentCGITimeout 3600

Notes on persistent CGI running under Apache

i. Persistent CGI running under Apache must use the POST method, not the GET method
ii. The *ScriptAliasMatch* directive for executing persistent CGI programs MUST HAVE the following format
```
ScriptAliasMatch /cgidev2p/(.*)  /qsys.lib/cgidev2.lib/$1
```
The following format WOULD NOT WORK:
```
ScriptAliasMatch /cgidev2p/(.*).pgm   /qsys.lib/cgidev2.lib/$1.pgm
```

---

3 - Tips for developing output html skeleton members
(when using Mel's service program technique)

1. **Accept-HTSession** Header
   The first html section issued by your program should start as follow

/$*section_name*
Accept-HTSession:/%HANDLE%/
Content-type: text/html
<HTML>

   where /%HANDLE%/ will be substituted with the "unique session ID" computed by your program (see the next topic).
   The "Accept-HTSession" header will not be sent to the client browser. It will be detected and taken off by the http server. The "unique session ID" will be associated with your program instance, so that the next request from the client browser mentioning it will cause your program instance be re-activated.

2. **HTTimeout** Header
   This header defines the amount of time, in minutes, that a CGI program wants to wait for a subsequent request.
   If not specified, the value specified on the HTTP **PersistentCGITimeout** directive is used.
   If specified, it overrides the HTTP PersistentCGITimeout directive,
   but the server will not wait longer than the time specified on the HTTP MaxPersistentCGITimeout directive.
   This allows individual CGI programs to give users more time to respond to lengthy forms or explanations. However, it still gives the server ultimate control over the maximum time to wait.

   Example

HTTimeout:50

   This header must be preceded by an **Accept-HTSession** header; if not, it is ignored.
   If you omit this header, the default time-out value for the server is used.

3. To enable the client browser to re-activate your program instance, your html URL link should be specified in the following way

   /path/cgi_name/handle/rest_of_path

   where

- ○ *path* is the path to your CGI persistent program
- ○ *cgi_name* is the name of your program followed by *.pgm*
- ○ *handle* is the "unique session ID" you already put in the "Accept-HTSession" header
- ○ *rest_of_path* is the parameter string (if any) expected by your program

Examples:

```
<FORM METHOD="POST" ACTION="/path/mypgm.pgm/ /%HANDLE%/>
<INPUT TYPE="HIDDEN" NAME="action" VALUE="go">
   . . .
</FORM>
```

```
<A HREF="/path/mypgm.pgm//%HANDLE%/?action=go> . . . </A>
```

4 - Tips for developing persistent CGI RPG programs

1. Also a CGI persistent program, after returning the ouput html to the browser should *return* to the server. This is different from a traditional non CGI program, where the program sits after an EXFMT instruction.
   - ○ do not set on the LR indicator, when you want the program to remain active for further requests
   - ○ set on the LR indicator when you want the program be no longer active. In this case, make sure the browser receives some html response, otherwise the end user will wait until a script-timeout is issued from the http server.
2. When receiving control from the http server, the persistent CGI program should test some variable of its own to establish the state it was left in.
3. Have the program itself regenerating every time a new *session ID* (also called "**handle**") to be inserted in two points of the output html:
   - ○ the "Accept-HTSession" header
   - ○ the URL to call again that program.
   In building a new session ID, you may use a random number obtained through Mel's service program random subprocedure.
4. When creating a persistent CGI program, be sure to specify a *named activation group* in the parameter **ACTGRP**. As an example, the name of the activation group could be the same for all CGIs in an application.

5 - Sample persistent CGI RPG programs

Mel Rothman's demo

GO **What day of the week?**
It computes its *Session ID ("Handle")* using the **getSessionID** subprocedure.

View

the bootstrap HTML
the externally defined HTML source
the RPG source of the CGI

Giovanni Perotti's demo

GO **Sample persistent RPG CGI program**
It computes its *Session ID ("Handle")* just using the **random** subprocedure.

View

the externally defined HTML source
the RPG source of the CGI

CGI debugging tips

What to do when the time comes to debug a CGI program

That time will come soon, just after starting.

> # Error 500
>
> CGI Program failed

is waiting for you.
So, better be prepared.

Note - In this page it is assumed that you run the Apache HTTP server.

1- Setup

Whenever a program check is met, Apache HTTP server jobs bump out, and all the diagnostics are then in their joblogs. So, first of all, HTTP server jobs must have a readable joblog. This is how you do it:

1. `chgjobd qhttpsvr/qzhbhttp log(4 00 *seclvl) inqmsgrpy(*dft)`
2. `endtcpsvr *http`
3. `strtcpsvr *http`

Second, you must change the "script time-out" value in your http directives, by specifying a time value high enough to let you complete your debugs without having the server job terminating because of an excessive wait time on the server response.

1. `edtf '/www/`*http_instance_name*`/conf/httpd.conf'`
2. add or change the following directive
   `TimeOut` *nn*
   by specifying af least `1800` (30 minutes) for *nn* .
3. `endtcpsvr *http`
4. `strtcpsvr *http`

If the HTTP server job does not bump out and you want to debug a CGI program of yours, you must first find out which HTTP job is going to serve your request.
To find out the HTTP job servicing your request

1. `wrkactjob sbs(qhttpsvr) job(`*instance_name*`)`
2. press F10 several times to reset counts
3. run any HTTP request *that would not fail*
4. press F5 to refresh the counts
5. the HTTP job servicing your request is the CGI one displaying some CPU time.

2- HTTP instance configuration

1. If your browser displays **error 404**, that may just mean that your HTTP directives are not

working

2. If you have installed library **cgidev2**, clear the cash of your web browser and try to run the following:

```
http://.../cgidev2o/hello.htm
```

(replace ... with the TCP address of your AS/400)

- ○ If you get no response, but an *Error 400* message, it may mean that your HTTP server is not serving your request. Some of the causes:
  - Your HTTP server is not active; you can check it by entering command `wrkactjob sbs(qhttpsvr)`
  - Your HTTP server is active, but some other active HTTP server is locking port 80 for its own exclusive use. This happens for instance when you also run `DOMINO`. You may then try to change the port of your HTTP configuration in the following way:
    1. Enter command
       edtf '/www/*http_instance_name*/conf/httpd.conf'
       to edit the HTTP server instance directives
    2. Look for a `Listen` directive.
    3. Change the port number on the `Listen` directive.
       For instance, instead of port 80, assign port 7777.
    4. Re-start the HTTP server instance
    5. Re-try the URL by entering
       ```
       http://...:7777/cgidev2o/hello.htm
       ```

3. If you get **Error 500** in running

```
http://.../cgidev2o/hello.htm
```

do the following:
- ○ make sure that library CGIDEV2 and program CGIDEV2/HELLO1 are authorized to the *public for *use
- ○ make sure that the configuration directives of your HTTP server contain the directives needed to run CGIDEV2, and that this HTTP server was re-started after installing such directives (to install CGIDEV2 HTTP directives, run command command `CGIDEV2/HTTPCFG`)
- ○ make sure you installed the last available PTF Cumulative for product 57xxDG1 (IBM HTTP Server for iSeries)

3- Error 500 trivial cases

Error 500 is raised in four cases:

1. **The mandatory HTTP header is missing**.
   The HTTP headers are statements at the beginning of the output generated from a CGI for the browser.
   The fist mandatory HTTP header must be:
   Content type: ...
   In absense of such HTTP header, you get ERROR 500 from Apache. Another common HTTP header is
   Expires: 0
2. **The mandatory HTTP headers separator is missing**.
   The HTTP headers must end with a sequence of two Carriage-Return-Line-Feed (CRLF).
   This is implemented by adding an *empty line* after the last HTTP header.
   In absense of such a separator, you get ERROR 500 from Apache.
3. **The output buffer is not sent to the browser**
   The CGI program returns without sending the output buffer via the statement
   callp wrtsection('*fini')
4. **The CGI program faces a program exception and is terminated**. This is the topic discussed hereafter.

4- Error 500 non trivial case

If the HTTP instance job servicing your CGI request was terminated due to a program exception, you should first look at its joblog (output queue QEZJOBLOG).

In many cases, error messages provide enough information to enable program correction.
For more complex cases, you need to debug your CGI program.

To perform debug, you must have compiled your ILE modules with the option
`DBGVIEW(*SOURCE)` or **`DBGVIEW(*LIST)`**
If you have done so,

1. `strsrvjob` ..... the http server job which you expect to process your next CGI request
2. `strdbg` *your_CGI_program* `updprod(*yes)`
   Note- To be quicker, you may use command **EDBG** (Enhanced Debug) available with
   CGIDEV2 (use F4 to prompt parameters).
3. Some basics, when the source of the initial module is displayed
   - on the command line,
     `F` *string*
     to find a *string*
   - F16 to find next
   - move a cursor to a source line and press F6 to add a breakpoint
4. Rerun your CGI program.
   Hopefully it will stop at some breakpoint of yours, you may tell from your debug session
5. More hints on debug
   - position the cursor on a variable and press F11 to display its value
   - you can do the same on the command line by entering
     `eval` *variable_name*
   - if you want to display the value of a variable in hexadecimal format, on the command
     line enter
     `eval` *variable_name*`:x`
   - to change the value of a variable, on the command line enter
     - `eval` *variable_name* `= '`*value*`'`
       if it is a character variable
     - `eval` *variable_name* `=` *value*
       if it is a numeric variable
   - to execute one step of coding, press F10
   - to resume execution till the next breakpoint, press F12
   - if your module calls another module, and you want to debug this latter one
     - add a breakpoint on the call statement
     - when execution stops on this statement, press F22 to display and add
       breakpoints to this latter module

By sure you will become the most proficient debugger in your neighbourhood!