Writing z/OS CGIs in Assembler

Writing z/OS CGIs in Assembler - Course Objectives

On successful completion of this class, the student, with the aid of the appropriate reference materials, should be able to:

1. Code, assemble, bind, debug, deploy, and maintain CGIs for the z/OS environment, written in Assembler language

2. Handle GET and POST requests: analyze and take action, as appropriate
   * Parse and decode a QUERY_STRING value for GET
   * Gather in the stdin data for POST
     - Save a file as is or translated to EBCDIC on the mainframe, for POST

3. Produce responses that are dynamically created HTML pages or redirection to existing pages

4. Access environment variables

5. Access DB2 data (optional: depends if DB2 installed and lab set up done)

6. Access VSAM KSDS data by primary key or alternate index

7. Put out HTML encoded in UTF-16, to provide a truly international aspect to your website

8 Submit jobs to the batch from a CGI (optional; may not be appropriate in all environments).

Note: Generally speaking, the comments here about HTML also apply to XHTML; but our focus is on using HTML 5

Note: This course supports the HTTP server provided free with z/OS and the ported Apache server (see page 4).

Writing z/OS CGIs in Assembler - Topical Outline

Writing z/OS CGIs in Assembler - Topical Outline, p.2.

HTTP Servers on z/OS

- There are several HTTP servers available for z/OS, but these are the known free choices (identified by the value found in the environment variable SERVER_SOFTWARE):

  - IBM HTTP Server/V5R3M0 - this server comes free automatically with z/OS and is based on early standards (still works fine, though)

  - IBM Apache Server - since late 2008, IBM provides this Apache server already ported, along with some ported tools; this is free but must be separately ordered

- There is also WebSphere Application Server (WAS) which comes with the same Apache server - but WAS is not free

- There is also a free Tomcat server from Dovetailed Technologies; it is Java-centric (http://www.dovetail.com/products/tomcat.html)

- For simplicity, we assume you are using one of the free available servers, which we shall refer to as "the HTTP server" (for the first server in the list above) or "Apache" for the second

  - Technically, of course, these are all HTTP servers, but we're looking for a shorthand to be both concise and accurate

  - Finally, since the behavior of these servers is largely the same, you can take "the server" to be shorthand for "either the HTTP server or the Apache server".

Section Preview

- ❐ **General Program Structure and Techniques**
  - ♦ **General Program Structure**
  - ♦ **Redirect Using printf**
  - ♦ **Redirect Using bpx1wrt**
  - ♦ **Watching for Errors**
  - ♦ **Deploying Your CGI**
  - ♦ **Setting Up for Labs (Machine Exercise)**

General Program Structure

- ❐ **The main work of a CGI is writing out HTML pages to** stdout**, which are then intercepted by the HTTP server and transmitted to the requesting client**

- ❐ **There are two basic choices for writing to** stdout **from Assembler language**

  - ♦ **<u>Use the callable service bpx1wrt</u> - which uses a classic MVS, z/OS approach in its parameters**

  - ♦ **<u>Call the C function printf()</u> - a viable alternative; not as natural in Assembler language as using bpx1wrt, but it handles certain formating chores more easily than using bpx1wrt**

    - ✗ Especially if you have numeric data to display, or need character string data to be constructed from strings whose length you don't know in advance

    - ✗ Under the HTTP server, your CGIs can mix these two approaches, but you cannot do so, apparently, under the Apache server

- ❐ **CGIs written in Assembler language must be reentrant, and probably should be LE-conforming**

  - ♦ **If a program is not LE-conforming, it cannot invoke the C functions** *printf()* **and** *getenv()***, nor the LE functions such as CEEGTSTG and CEEENV that are useful in this work**

  - ♦ **Still, you can call kernel services such as bpx1wrt even if your program is not LE-conforming**

    - ✗ Except there is no kernel callable service for working with environment variables

General Program Structure, 2

❒ **Reentrant LE-conforming programs typically start like this:**

```
*PROCESS COMPAT(NOCASE,MACROCASE)
TCAREDP CEEENTRY PPA=MESSPPA,AUTO=WORKSIZE
```

- **The PROCESS statement allows the Assembler to process code and macros coded in mixed case**

- **The CEEENTRY macro names the program, identifies the PPA location, and specifies how large a DSA (Dynamic Save Area) should be allocated**

  - ✗ This generates CSECT, AMODE, and RMODE statements as well as basic save area linkages

  - ✗ We use the DSA to hold variable work areas - requesting storage here eliminates using GETMAIN, STORAGE, CPOOL, or CEEGTSTG to get storage for your modifyable data areas

    - ➢ This storage is obtained in the program's stack storage, so it is automatically freed when you leave the program

General Program Structure, 3

❒ **Next, you might have these statements**

```
          using wareas,13
          bru   the_code      branch around data areas
*
*    CONSTANTS, WORK AREAS, ETC.

          DC    C'Ver3 of TCAREDP'
           . . .
```

- **The** using **provides addressability to a DSECT that describes any modifyable data areas**

- bru **is one of the conditional relative branch instructions: branch relative unconditional**

  - ✗ Since bru uses short relative addressing, "the_code" can be up to 64K bytes away from the bru instruction itself

- **Next, define all your non-modifyable data areas as well as the source for initializing your modifyable data areas**

  - ✗ The example just shows an eyecatcher, useful in some dump reading situations

    - ➢ Also useful for ensuring you are working with the correct version of a program, if you are rigorous about keeing the version number updated

      - ➤ It took the course author three tries to get it right

Redirect Using printf

❒ **Now, for a redirecting CGI, you might have these data items:**

```
loc      dc     C'Location: ../~scomsto'
         dc     c'/customer.html.ascii'
         dc     x'1500'
blank    dc     x'1500'

the_code ds     0h
```

Notes

- **The line labeled "loc" contains the text needed in a redirect header**
  - ✗ The ../~scomsto says "back out of the current directory" (the two dots) "then go into your user id's web pages directory" (/~ followed by your z/OS UNIX ID)
    - ➢ You will need to replace this with a siimlar construct using your actual ID, as described for you in the first lab writeup
  - ✗ And /customer.html.ascii is the name of the page to redirect to in this directory
    - ➢ Notice the full name is spread across two lines; since they assemble consecutively, the resulting string is what we need
- **The x'1500' following represents an EBCDIC new line (NL) character followed by a null (the NL is used by the server to indicate end of the current line; the null is needed by C type functions that work with strings)**
- **The line called "blank" is used to delimit the header set; this is required by the server**
- **After "the_code" is where to code the actual logic ...**

Redirect Using printf, 2

- **Now, a simple redirecting CGI only requres a Location header and a blank line, so code:**

```
    call printf,(loc),vl,mf=(e,plist)  point to file
    call printf,(blank),vl,mf=(e,plist) blank line
```

  - **These two lines write to stdout using the C** printf **function**

  - **In each case, passing a single parameter: the address of a null-terminated string**

    - Which happens to end in an NL character before the null

    - The null delimits what **printf** should write (everything up to the null), and the server will see the two lines (Location header and blank line)

  - **The "vl" indicates there is a variable number of parameters being passed, and the Assembler will turn on the end-of-list bit in the last entry**

  - **Finally, the mf=(e,plist) parameter says this is the execute form of a macro, where the the list form is at the location called "plist"**

    - This approach is used for generating reentrant code: the list form of the macro is in the modifyable data areas DSECT, while the execute form will take input values and put them into the list form, then call the service, passing the address of the arguments in the list form

Redirect Using printf, 3

❒ **Next, to return to z/OS, use the CEETERM macro, first placing a return code value into R15:**

```
         la      15,0
         CEETERM   RC=(15),MODIFIER=0,mf=(e,realterm)
```

- **Again use the execute form of a macro and point to the list form**

❒ **Next code a Program Prologue Area, which contains some LE program management control fields - note that the name is the name coded on the CEEENTRY macro at the start of the program:**

```
MESSPPA    CEEPPA
           LTORG
```

- **The LTORG is an Assembler instruction telling the Assembler to gather any literal pool data areas at this point**

  ✗ It is there so that no literals get swallowed into the addressability of the following DSECT lines (next page)

Redirect Using printf, 4

❒ **The final lines of code ...**

```
wareas   dsect
         org    *+CEEDSASZ
plist    call   ,(0,0,0,0,0,0,0,0),mf=l
realterm ceeterm rc=(15),modifier=0,mf=l
worksize equ    *-wareas
         CEEDSA
         CEECAA
         END    TCAREDP
```

- **Beginning at "wareas" is your modifyable storage area**

- **CEEDSASZ is a label generated by the CEEDSA macro, indicating how much storage to reserve for the LE DSA**

  ✗ So the "org" places your modifyable storage after the DSA that LE needs

- **In this case, there are just have two items: the list form of a call and a CEETERM macro**

- **The CEEDSA and CEECAA generate LE-required control areas and symbols in DSECTs**

- **And, of course, the END statement is the traditional Assembler END statement**

Redirect Using printf, 5

❒ **Finally, to put it all into one place:**

```
*PROCESS COMPAT(NOCASE,MACROCASE)
TCAREDP CEEENTRY PPA=MESSPPA,AUTO=WORKSIZE
         using wareas,13
         bru   the_code       branch around data areas
*
*     CONSTANTS, WORK AREAS, ETC.

        DC    C'Ver3 of TCAREDP'
loc     dc    C'Location: ../~scomsto'
        dc    c'/customer.html.ascii'
        dc    x'1500'
blank   dc    x'1500'

the_code ds     0h
         call printf,(loc),vl,mf=(e,plist)  point to file
         call printf,(blank),vl,mf=(e,plist) blank line
         la    15,0
         CEETERM  RC=(15),MODIFIER=0,mf=(e,realterm)
MESSPPA  CEEPPA
         LTORG
wareas   dsect
         org   *+CEEDSASZ
plist    call  ,(0,0,0,0,0,0,0,0),mf=l
realterm ceeterm rc=(15),modifier=0,mf=l
worksize equ   *-wareas
         CEEDSA
         CEECAA
         END   TCAREDP
```

- **Pretty straightforward, and easy to build on**

- **Now let's look at the same function using calls to the z/OS UNIX kernel service** bpx1wrt

Redirect using BPX1WRT

- **To use BPX1WRT, the general structure is pretty much the same, but with these issues to address**

  - **The bpx1wrt service requires seven parameters, passed in the classic MVS, z/OS style:**

    - File descriptor number; use a fullword binary 1 for **stdout**

    - Address of a pointer to the buffer containing the data to write

    - Address to a pointer to a buffer ALET (Address space or data space where buffer is); specify zeros to indicate the current address space, which is what we want

    - Bytes to write - fullword binary integer containing the length of the data you want to put out

    - Return value from function: -1 indicates write failed; otherwise returns the actual number of bytes written

    - Return code and reason code; each fullwords; not meaningful unless return value is -1

  - **Because bpx1wrt has so many arguments, the macro generates a lot of instructions to set up addresses before actually calling**

    - You can put a single execute form call in a generalized routine and do a BRAS to that routine for every write you need to have

  - **This service <u>must not</u> have trailing nulls in its parameters**

Redirect using BPX1WRT, 2

❒ **So the pieces you end up with that are different from using printf:**

- **Defining constants; here we have:**

```
loc     dc   C'Location: ../~scomsto/'
        dc   C'customer.html.ascii'
blank   dc   x'15'
len_loc equ  *-loc
stdout  dc   f'1'
```

  - ✗ Note: no trailing nulls are needed for this service's parameters
  - ✗ Also, you will need the length of the string being sent; we get this with the equ for **len_loc**
  - ✗ And, finally, a constant that we will reference in the call to bpx1wrt to direct the lines to **stdout**

Redirect using BPX1WRT, 3

- <u>Defining our DSECT area</u>; **at the end of our code, after our LTORG and before our CEEDSA, we have:**

```
wareas   dsect
         org   *+CEEDSASZ
plist    call  ,(0,0,0,0,0,0,0,0),mf=l
realterm ceeterm rc=(15),modifier=0,mf=l
* bpx1wrt data items
            ds   0f
bpx1_stuff  ds   0cl24
buffer_ptr  ds    f
buffer_alet ds    f
num_bytes   ds    f
return_co   ds    f
reason_co   ds    f
return_val  ds    f
worksize equ      *-wareas
```

- **Much of this is familiar; the differences:**

  - ✗ The list form of the call ("plist"); note you can use this list form for any call that has seven or fewer parameters to pass; you can add additional parameters in plist if you need to

  - ✗ The "bpx1_stuff" items; this list contains a field for every argument in the bpx1wrt call, except for the first argument (which uses the "stdout" item defined in the non-modifyable area)

Redirect using BPX1WRT, 4

- The executable code looks like this:

```
the_code ds     0h
          xc    bpx1_stuff,bpx1_stuff

          la    2,loc
          st    2,buffer_ptr
          la    2,len_loc
          st    2,num_bytes
          bras  2,common_write

          la    2,blank
          st    2,buffer_ptr
          la    2,l'blank
          st    2,num_bytes
          bras  2,common_write


          la     15,0

          CEETERM  RC=(15),MODIFIER=0,mf=(e,realterm)

 common_write ds 0h
         call  bpx1wrt,(stdout,buffer_ptr,              x
               buffer_alet,num_bytes,                   x
               return_val,return_co,                    x
               reason_co),vl,mf=(e,plist)
         br    2
```

Notes

- Notice the first line of the code where the call parameters are intialized to zeros
- You can see the pattern clearly about how to invoke the common routine to write to stdout ("common_write")
- The 'x's at the right have to be in column 72 of our code

Redirect using BPX1WRT, 5

❐ Again, putting it all together:

```
*PROCESS COMPAT(NOCASE,MACROCASE)
TCAREDB CEEENTRY PPA=MESSPPA,AUTO=WORKSIZE
        using wareas,13
        bru   the_code       branch around data areas

        DC    C'Ver3 of TCAREDB'
loc     dc    C'Location: ../~scomsto/'
        dc    c'/customer.html.ascii'
blank   dc    x'15'
len_loc equ  *-loc
stdout  dc   f'1'

the_code   ds    0h
           xc    bpx1_stuff,bpx1_stuff

           la    2,loc
           st    2,buffer_ptr
           la    2,len_loc
           st    2,num_bytes
           bras  2,common_write

           la    2,blank
           st    2,buffer_ptr
           la    2,l'blank
           st    2,num_bytes
           bras  2,common_write

           la   15,0
           CEETERM   RC=(15),MODIFIER=0,mf=(e,realterm)

 common_write ds 0h
          call  bpx1wrt,(stdout,buffer_ptr,            x
                buffer_alet,num_bytes,                 x
                return_val,return_co,                  x
                reason_co),vl,mf=(e,plist)
          br    2
```

Redirect using BPX1WRT, 6

❒ **Again, putting it all together: (page 2)**

```
MESSPPA  CEEPPA
         LTORG
wareas   dsect
         org    *+CEEDSASZ
plist    call   ,(0,0,0,0,0,0,0,0),mf=l
realterm ceeterm rc=(15),modifier=0,mf=l
* bpx1wrt data items
            ds   0f
bpx1_stuff  ds   0cl24
buffer_ptr  ds    f
buffer_alet ds    f
num_bytes   ds    f
return_co   ds    f
reason_co   ds    f
return_val  ds    f
worksize equ    *-wareas
         CEEDSA
         CEECAA
         END    TCAREDB
```

❒ **For both of these CGIs, the redirect address can be a fully specified URI, for example:**

```
C'Location: http://192.168.1.231/~scomsto'
```

♦ **For the first sample program (TCAREDP)**

```
C'Location: http://192.168.1.231/~scomsto/'
```

♦ **For the second sample program (TCAREDB)**

Watching for Errors

- **Debugging CGIs is generally quite awkward**
  - **The environment is complex**
  - **Often the HTTP server tries to continue on, even after a CGI has abended**

- **So one step you can take in your code is to watch for errors**

- **For example, both printf and bpx1wrt might not be successful**
  - **Check R15 after a call to** printf **- if it is any negative number, there was a problem**
  - **Check the return code after a call to bpx1wrt - if it is -1, there was a problem**

- **But what to do in these cases? You can't write a message, since these are the routines used to write messages! Well, you can try several approaches**
  - **Call the LE service CEEMOUT to write to** stderr
  - **Call bpx1wrt but write to** stderr **(use a fullword 2)**
  - **Put a non-zero value in the program return value**
  - **Call CEE3ABD or CEE3AB2 if you are LE-conforming, or ABEND if you are not**
  - **Insert an instruction (*e.g.*: DC H'0'') in the flow where it will abend in the middle of code you suspect, to force an Abend**

- **If you have errors in other routines, at least you can use bpx1wrt or** printf **to write out HTML text to the client that gives some indication of the situation**

Watching for Errors, 2

- In our code samples and labs we will not do extensive error checking, in order to focus on functionality

  - But in a number of places we will demonstrate error checking and handling, so you can see some of the ways of dealing with errors

- When trying to debug CGIs, it is often helpful to look at the HTML source the CGI has emitted up to the point of the error

  - Using your browser to look at a page put out by your CGI, right click on a blank spot of the page

  - In most browsers a pop-up menu will include an option like "View page source"

  - Selecting this will show you the HTML your CGI wrote out, perhaps giving you some clues where things went wrong

- There is a pretty good tool for examining HTTP traffic; it's called HTTPLook, it's shareware and you can download it from

  - http://www.brothersoft.com/httplook-download-25677.html
  - Caution: download then run the install program; you will see a dialog about installing the BrotherSoft Extreme with some check boxes; close the dialog; when it prompts you to continue or exit setup, choose exit; wait a while and then you will see the setup dialog for HTTPLook - now install this program

Deploying Your CGI for Testing

❒ **Once you have your CGI coded, you need to Assemble and bind and put the code in the correct place for it to be found by the HTTP server when it is called for**

❒ **So the steps are:**

- **Assemble and bind using JCL (we shall bind into a PDSE named** *<your_id>***.TR.PDSE)**

  ✗ Or, if you are working under the shell, use the **c89** or **as** commands to Assemble and bind into your CGI directory

- **If working outside of the shell, you need to copy your executable load module into your CGI directory; use ISPF 6 like this:**

```
===> oput tr.pdse(tcaredp) '/u/scomsto/CGI/tcaredp'
```

  ✗ Note that for the second operand, <u>case is important</u>; also you need to specify the name of the directory set up for your CGIs instead of the directory shown, of course

- **Either way, your last step here is to ensure the CGI program has the right permission bits; if you are not in the shell already, issue the omvs command from ISPF 6 then issue these commands:**

```
cd CGI
chmod 755 tcaredp
```

  ✗ Note that you only have to do this the first time you put each CGI into your directory; later, if you replace it, the permission bits are remembered

Computer Exercise: Setting Up For Labs

This machine exercise is designed to provide setup for all the remaining class exercises.

In order to work with CGIs, a lot of pieces have to be in place:

* You must have the IP address or system name of your host where the CGIs will run; this can be internal (your intranet, behind your firewall) or external (your internet presence, accessible by browsers from outside your organization):

  ____________________________ (system name or IP address)

* You must have a z/OS UNIX ID, part of what's called an OMVS segment as part of your security package; this includes a user id for logon (a character string that is usually lower case), and a UID (an integer) to identify you to the user database, a home directory (usually of the form **/u/***user_id*), and some other information: ___________ (your user id)

* You must have a TSO id also (which we assume to be your z/OS UNIX userid in upper case); normally the password for both ids is the same.

* You must know your installation's choice for the directory where web pages should be stored; often it is **public_html** under your home directory; that is: **/u/***user_id***/public_html**, but not always:

  ________________________ (web page directory)

* You need to know the name of the directory where your CGIs should reside; it is often called **CGI** and is under your home directory; that is: **/u/***user_id***/CGI** but it does not have to be so:

  _______________ (CGI directory)

* Finally, you need to know the mapping id that the server will use to direct CGI requests to your CGI directory; for example, in our shop, our configuration file has the entry:

```
Exec     /SCOMSTO/*     /u/scomsto/CGI/*
```

  which says requests for any file in SCOMSTO should resolve to files in /u/scomsto/CGI, my CGI directory; SCOMSTO is my CGI mapping id.

  ______________ (CGI mapping id)

Computer Exercise, p.2.

For this lab, you have two parts: 1) the set up work and then 2) a small lab that will build on the lecture and test the set up at the same time.

The set up

Run uc06strt, a supplied REXX exec that will prompt you for the high level qualifier (HLQ) you want to use for your data set names; the exec uses a default of your TSO id, and that is usually fine. Then the exec creates data sets and copies members you will need. Then there is still some work to do.

From ISPF option 6, on the command line enter:

```
===> ex '__________.train.library(uc06strt)' exec
```

A panel displays for you to specify the HLQ for your data sets, with your TSO id already filled in. Press <Enter> and you get a panel telling you setup has been successful. Press <Enter> again and you are back to the ISPF command panel

The allocated data sets:

<hlq>.TR.CNTL	for your JCL (and it also contains some archive files and other data as members)
<hlq>.TR.SOURCE	for your source code
<hlq>.TR.PDSE or	for program objects
<hlq>.TR.LOAD	for load modules

Computer Exercise, p.3.

Next, get into OMVS, and cd to your html directory and issue these commands:

```
umask 000
pax -r -f "//tr.cntl(uc06html)"
```

this unwinds the testing HTML pages and some data.

While you are in this directory, create a sub-directory we will use in a later lab:

```
mkdir PDFs
```

Also while you are in this directory, you should oedit the file **AsmCGI_Labs.html** as follows:

* change all occurrences of SCOMSTO to the mapping id for your CGI directory

* If you have access to a corporate logo image file, you can change the <img > tag to point to that logo.

Next, change to your CGI directory, and issue this command:

```
pax -r -f "//tr.cntl(cgis)"
```

this unwinds your style sheet (discussed later).

Computer Exercise, p.4.

The lab.

Exit OMVS and get into edit of your source PDS. There are two members there that do redirects:

TCAREDB - redirect using bpx1wrt
TCAREDP - redirect using printf

Modify each of these so that "scomsto" is changed to your id

Now Assemble and bind each of these. To do this, edit your TR.CNTL library, member ASMCGIA. This JCL Assembles and binds programs into your TR.PDSE library. The

```
//   SET  O=
```

line should have the name of the member to Assemble and bind. So Assemble and bind each of these two programs.

Once you have clean Assemblies and binds, deploy the executables from your TR.PDSE or TR.LOAD library to your CGI directory. (see page 22 for hints)

Finally, test your work by pointing your browser on your workstation to your AsmCGI_Labs.html page and runnng the Assembler language programs listed in the first test option.

Conventions used in this course:

192.168.1.231 - internal IP address used by course author for development and testing; always replace with your system name or IP address

scomsto - UNIX id used by the author; always replace with your UNIX id

public_html - directory for user HTML pages; always replace with your HTML directory

~scomsto - mapping id used to get to your HTML directory; replace with your mapping id

SCOMSTO - mapping id used to get to your CGI directory

CGI - actual directory for user CGIs to run from; always replace with your CGI directory mapping id

/s-css/* - directory for style sheets referenced by CGIs; maps to /u/scomsto/CGI/* ; always replace with your CGI stylesheet mapping (more later)

Conventions used in this course, 2:

CGI program names used in all our language-specific CGI courses: **TC**xfffs where:

T	comes from The Trainer's Friend
C	indicates this is a CGI
x	indicates the programming language; one of: A - Assembler B - COBOL C - C P - PL/I X - REXX
fff	mnemonic for the function, *e.g.*: RED for REDIRECT
s	indicate method used to write to stdout; one of: B - BPX1WRT P - printf() D - display (COBOL) K - put skip (PL/I) S - say (REXX) E - echo (shell script) R - print (Perl, Java, php) X - EXECIO (REXX)

In a few cases, we may not follow this naming convention but it will usually help you keep straight which program is which.

Section Preview

❒ **Basic Processing**

- **Emitting Headers**
- **Emitting HTML**
- **Accessing environment variables**
- **Displaying environment variables**
- **Stylesheets and CGIs**
- **Writing out HTML pages (Machine Exercise)**

Emitting Headers

- ❐ **Every CGI must emit**
  - ♦ **One or more HTTP headers**
  - ♦ **A blank line**
  - ♦ **Some content**
    - ✗ Usually an HTML page
      - ➢ Perhaps also some log or trace information or error messages

- ❐ **We saw with the redirect example a single header (Location) and a blank line**
  - ♦ **If no content is supplied with a redirect header, the z/OS HTTP server supplies a little content to help the transmission protocol be maintained**

Emitting Headers, 2

- When there is more to emit than a Location header, most typically emitting a Content-type header is required

  - Using a content type of text/html, add two NL characters to send the header line and corresponding blank line

  - Using printf, add a trailing null, so define:

```
charset1  dc    C'Content-type: text/html ',x'15'
blank     dc    x'1500'
```

  - And write to stdout with:

```
          call printf,(charset1),vl,mf=(e,plist)
```

  - Using bpx1wrt, define:

```
charset1  dc    C'Content-type: text/html ',x
blank     dc    x'1515'
```

  - And write out with:

```
          la    2,charset1
          st    2,buffer_ptr
          la    2,l'charset1+2
          st    2,num_bytes
          bras  2,common_write
```

Emitting HTML

- There may be some work to do before writing out the main HTML, but at some point, put out these lines:

```
<!DOCTYPE html>
<htm>
<head>
<link rel=stylesheet href=/s-css/cgi-style1.css
   type=text/css >
```

  - Then a title element, then the end of the <head> section, then start your <body>

  - After the detail lines (body), bring closure with </body> and </html> before ending your CGI

- Notice the link to a stylesheet

  - This is optional, of course, and there are some issues regarding style sheets, CGIs, and the HTTP server - which we address later in this section

  - But having the ability to work with a stylesheet is pretty essential with HTML 5

Emitting HTML, 2

- **Since every HTML page starts out the same, we have provided a subroutine, TTFPREA, you can call to generate these first lines for you**

  - **It takes no parameters, just call it, using:**

```
         call ttfprea,,vl,mf=(e,plist)
```

- **This saves the time and coding to get your basic HTML page starting lines out of the way**

  - **It also allows us to encapsulate the location-specific information in the link to the stylesheet into only one place**

Notes

  - **TTFPREA uses bpx1wrt to write out html**

  - **Because calling bpx1wrt and printf don't seem to mix when running under Apache, we have also provided subroutine TTFPREP, which uses printf to write out html**

    - ✗ Call TTFPREP with the same syntax as for TTFPREA above

    - ✗ For class labs, we only use this routine in one place, but you may find a use for it elsewhere

Emitting HTML, 3

❒ **After the headers ttfprea emits, for your next lines, you will want to have something like this defined:**

```
title    dc    c'<title>Display Environment variables </x
               title>',x'15'
head_end dc    c'</head>',x'15'
body_beg dc    c'<body>',x'15'
h2_tag   dc    c'<h2>Assembler - Standard CGI variables x
               </h2>',x'15'
br_tag   dc    c'<br>',x'15'
body_end dc    c'</body>',x'15'
html_end dc    c'</html>',x'15'
```

Notes

- **The items labeled "title" and "h2_tag" demonstrate continuing a literal item: code up through 71, put a non-blank character in 72, and begin the continuation exactly in column 16**

  ✗ We do this instead of simply having two consecutive DC statements because we will want to use the length attribute, for example: **LA 2,L'TITLE**
  and with two separate statements, only the length of the first statement would be picked up

  ✗ Actually, in this example we didn't have to continue, except for typographical limitations: both sets actually fit on one line each

- **If you will be using** printf **for output, each x'15' should be x'1500'**

Emitting HTML, 4

❒ **Putting out lines using printf would be something like:**

```
         call printf,(title),vl,mf=(e,plist)
         call printf,(head_end),vl,mf=(e,plist)
         call printf,(body_bet),vl,mf=(e,plist)
         call printf,(h2_tag),vl,mf=(e,plist)
```

♦ **And so on; the same work using bbx1wrt would be:**

```
         la    2,title
         st    2,buffer_ptr
         la    2,l'title+1
         st    2,num_bytes
         bras  2,common_write

         la    2,head_end
         st    2,buffer_ptr
         la    2,l'head_end+1
         st    2,num_bytes
         bras  2,common_write

         la    2,body_beg
         st    2,buffer_ptr
         la    2,l'body_beg+1
         st    2,num_bytes
         bras  2,common_write

         la    2,h2_tag
         st    2,buffer_ptr
         la    2,l'h2_tag+1
         st    2,num_bytes
         bras  2,common_write
```

Accessing Environment Variables

- ❐ **A simple redirect response is not very interesting: we only write out HTTP headers, not even any HTML**
  - ♦ **In the next section we explore more complex requests, focusing there on GET requests**

- ❐ **In order to find out what request has been made, a CGI generally needs to access various environment variables**

- ❐ **There are three possible techniques here:**
  - ♦ **Follow memory control block chains - complex and error prone**
  - ♦ **Use the LE callable service CEEENV - excellent, but not available before z/OS 1.8, so can be a problem in some environments**
  - ♦ **Call the relevant C function,** getenv **- a viable alternative for all releases and compiled languages**

- ❐ **In this course we will demonstrate both of the last two approaches**

Accessing Environment Variables - CEEENV

- **All LE-conforming languages may call the CEEENV service (introduced in z/OS 1.8)**

Syntax

CEEENV *request*, *name_len*, *name*, *val_len*, *value*, *fc*

Input	*request*:	**a(fullword binary); "1" indicates "locate value"**
Input	*name_len*:	**a(fullword binary containing length of variable name)**
Input	*name*:	**a(string containing variable name) (not null-temrinated)**
Output	*val_len*:	**a(fullword where length of value is returned)**
Output	*value*:	**a(string containing the value)**
Output	*fc*:	**a(12 byte feedback code area)**

Examples

```
call  ceeenv,(f1,nL,Name,vL,Value,fc),vl
```

- **Or, in keeping with our reentrant style:**

```
call  ceeenv,(f1,nL,Name,vL,Value,fc),vl,mf=(e,plist)
```

- **In the reentrant case, at least, parameters nL, Name, vL, and Value will have to be filled in before call**

- **In both cases, check fc afterwards, to ensure the call was successful**

Accessing Environment Variables - CEEENV, 2

❐ So, to flesh it out a little in the style we have been working with ...

- In our non-modifyable area we add:

```
f1        dc   f'1'  - request retrieve value
zeros     dc   3f'0' - for fc compares
varname1  dc   c'QUERY_STRING'
```

- Then the prep and call might look like this:

```
          la    2,varname1
          st    2,Name
          la    2,l'varname1
          st    2,nL
          xc    fc,fc
          call  ceeenv,(f1,nL,Name,vL,Value,fc),    x
                vl,mf=(e,plist)
          clc   fc,zeros  check for success
          jne   no_val
*
*  if get here, Value contains address of string
*  and vL contains length of string
```

- Finally, our modifyable DSECT area would include these new items:

```
Name      ds    f
nL        ds    f
Value     ds    f
vL        ds    f
fc        ds    cl12
```

Accessing Environment Variables - getenv

- **The** getenv **C function takes as input a null-terminated string containing the name of the environment variable you are interested in**

  - **And returns in R15 either the address of the null-terminated string containing the value of the variable, or binary zeros if the variable does not exist**

  - **So we would set up the variable name as:**

```
varname1 dc    c'QUERY_STRING',x'00'
```

  - **And call the function this way:**

```
         call  getenv(varname1),vl,mf=(e,plist)
         ltr   3,15
         jz    no_val
* get here and r3 contains the address of the
* null-terminated value of the environment variable
```

- **Now, let's take a look at how we might display the value we've found - or how to deal with a variable with no value**

  - **Our approach is to write out some HTML**

    - ✗ And we will demonstrate using both printf and bpx1wrt

Displaying Environment Variables

- ❒ **Let's suppose for a minute that you are only interested in displaying the value in an environment variable**

  - ♦ **Which could be the case during development, debugging, or our next lab(!)**

- ❒ **We can use either** printf **or bpx1wrt to display a variable, regardless of how we got to the value**

  - ♦ **However, if you use getenv, the result is returned as a null-terminated string - easy to use from printf**

    - ✗ If you use ceeenv, you get back the string and its length - easy to use from bpx1wrt

  - ♦ **We will assume from now on that you have used getenv, since that is available in older systems and ceeenv is only available in newer systems**

    - ✗ Converting from ceeenv usage to getenv usage is left as an exercise for the student

  - ♦ **For now we can assume we have R3 pointing to the value of an environment variable**

    - ✗ And that value is a null-terminated string

Displaying Environment Variables Using printf

❒ **To use the** printf() **function to display a string, you usually pass a message string which includes a "%s" everywhere you want the function to fill in a string value**

- **Followed by a pointer to a null-terminated string for each %s in your message string (matching is done in order from left to right)**

- **So building on previous work, we would have this in our non-modifyable storage:**

```
varname1  dc    c'QUERY_STRING',x'00'
var_msg   dc    c'%s = %s<br>',x'00'
err_msg1  dc    c'%s: ** variable not set **<br>',x'00'
```

- **That is, both the variable display message and the error message are HTML with text followed by a break**

  ✗ Since we are using getenv and printf, we need to terminate the strings by null characters

- **Note the %s entries; now if we are successful we issue:**

```
     call printf,(var_msg,varname1,(3)),vl,mf=(e,plist)
```

- **and if we are unsuccessful we could do:**

```
     call printf,(err_msg,varname1),vl,mf=(e,plist)
```

❒ **Note that** printf **always writes to** stdout

Displaying Environment Variables Using BPX1WRT

- **To use bpx1wrt for this task takes a little more work: you must build up the pieces of the message yourself in a buffer and then write the buffer to** stdout
  - **And we start by using the** strlen() **function to extract the length of the variable we currently have the address of in R3**
  - **To put it in context:**

```
varname1  dc    c'QUERY_STRING',x'00'
stdout    dc    f'1'
spaces    dc    cl256' '
err_msg   dc    c'** variable not set **'
move_var1 mvc   line_out+l'varname1+4(0),0(3)
   .
   .
   .
           call  getenv,(varname1),mf=(e,plist)
           ltr   3,15       c(3)=a(QUERY_STRING)
           jz    no_val

           call  strlen,((3)),mf=(e,plist)
           lr    4,15       c(4)=L'QUERY_STRING
           mvc   line_out,spaces
           mvc   line_out(l'varname1),varname1
           mvi   line_out+l'varname1+2,c'='

           bctr  4,0
           ex    4,move_var1
           la    1,line_out+l'varname1+3 point after =

           la    2,l'varname1+5(4)
           st    2,num_bytes
           la    2,line_out
           st    2,buffer_ptr
           bras  2,common_write
```

Displaying Environment Variables Using BPX1WRT, 2

- **And down in our modifyable areas we see, in addition to our earlier items:**

```
line_out     ds   cl256
```

- **IMPORTANT NOTE: Here we have deliberately set up for a maximum of 256 characters in a message**
  - But if a **printf()** with variables or an **ex** of a **mvc** generates a longer string, you could do damage to other data items - a variation of the infamous buffer overrun problem
  - SO YOU MUST KNOW YOUR DATA AND PLAN ACCORDINGLY

Displaying Environment Variables Using BPX1WRT, 3

❒ **Now, if a variable is not set when using bpx1wrt, we might have:**

```
no_val     ds     0h
           mvc    line_out,spaces
           la     6,line_out       target
           lr     7,4              bytes to move
           lr     8,2              source (variable name)
           lr     9,4              bytes to move
           lr     2,4              save var name length
           mvcl   6,8
           mvi    2(6),c'='
           la     6,4(6)          point after varname1 =

           mvc    0(l'err_msg,6),err_msg
           la     2,l'err_msg+5(2)
           st     2,num_bytes
           la     2,line_out
           st     2,buffer_ptr
           bras   2,common_write
```

Writing to stderr

❒ **When a message really should go to** stderr **instead of** stdout**, you have two choices, again:**

- **Use the LE CEEMOUT callable service**

- **Use** bpx1wrt **with a routing to** stderr **(fullword '2') instead of** stdout **(fullword '1')**

❒ **If you will be using** bpx1wrt**, you can use the** sprintf() **C function to format a buffer in the same way that** printf() **does; for example:**

```
         call  sprintf,(work_out,err_msg1,(2)),          x
               vl,mf=(e,plist)
         la    2,work_out
         st    2,buffer_ptr
         la    2,l'err_msg1+1(6)
         st    2,num_bytes
         call  bpx1wrt,(stderr,buffer_ptr,               x
               buffer_alet,num_bytes,                    x
               return_val,return_co,reason_co),          x
               vl,mf=(e,plist)
```

- **Where work_out is a buffer in our modifyable area and everything else is stuff from before**

❒ **Lines written to** *stderr***, however, end up in the cgi-error log for the HTTP server, not always easy to get to**

❒ **Note that you can mix and match the use of** printf(), getenv(), ceeenv, strlen(), bpx1wrt, sprintf(), ceemout **all in the same [LE-conforming] program, as needed, under the HTTP server**

- **You only need to be aware of the formats of arguments**
- **The Apache server does not let you mix and match so freely**

bpx1wrt vs. printf()

- ❒ **As with most techniques in programming, these two approaches for writing to** stdout **each have their own pros and cons**

bpx1wrt

- ♦ **This is a z/OS UNIX kernel command; not dependent on C-specific interfaces**
- ♦ **More arguments (so more set up)**
- ♦ **Strings must not be null-terminated**
- ♦ **May build up a line by bpx1wrt-ing each piece separately; required if you intend to format one or more of the pieces**
- ♦ **May appear in LE-conforming or non-LE-conforming programs**
- ♦ **May use to write to** stderr **also**

printf()

- ♦ **C-specific**
- ♦ **Fewer arguments**
- ♦ **String arguments (both in the message string and in any strings passed to match up to %s formats) must be null-terminated**
- ♦ **Formating is done based on your arguments and format indicators**
  - ✗ So must have all the pieces in place before calling printf()
- ♦ **Requires program to be LE-conforming**
- ♦ **Always writes to** stdout

Stylesheets and CGIs

- Generally, a static HTML page is served from a particular directory, and CGIs are run out of a different directory

- When a CGI references a stylesheet and it is a relative reference (for example: <link ... href="cgi-style1.css" type="text/css">), the stylesheet is presumed to be found in the CGI directory

  - But because of the configuration values normally set up, files found in the CGI directory are presumed to be executables and the server tries to run the stylesheet instead of just pass it on to the server

Stylesheets and CGIs, 2

❒ **To fix this, you need to provide a mapping in your configuration files, and there are several ways to go, including these two:**

- **Create a string that maps to a common, shared directory for styles; here's an example:**

```
Pass   /t-css/*   /usr/lpp/testing/*
```

  ✗ Then in your CGI your link to a stylesheet might be:

```
<link ... href="/t-css/cgi-style1.css" ... >
```

- **Create a string that maps to one of your directories, maybe even your CGI directory; for example:**

```
Pass   /s-css/*   /u/scomsto/CGI/*
```

  ✗ Then in your CGI your link to a stylesheet might be:

```
<link ... href="/s-css/cgi-style1.css" ... >
```

❒ **Of course, this is normally done by a systems person, and not lightly because it requires recycling the HTTP server**

- **So you build one mapping per person, and have each person work in their own directory   or**

- **You build a single, shared mapping and everyone uses a shared directory   or a combination**

Computer Exercise: Writing Out HTML Pages

In this exercise we work on displaying some environment variable values, and laying the base for our future work. All the source code here is in your TR.SOURCE library. To Assemble and bind, use member ASMCGIA in your TR.CNTL library, just change the value of the SET O= line to point to the code to work with.

The source code to work with:

TTFPREA - writes out the first HTML headers, as discussed on page 33
TTFPREP - same as above but uses printf() instead of bpx1wrt

TCAVARP - uses printf() for writing to stdout; calls TTFPREP
TCAVARB - uses bpx1wrt for writing to stdout; calls TTFPREA

The last two programs output a page that displays the values of the environment variables QUERY_STRING and SERVER_SOFTWARE.

Your tasks:

Change TTFPREA and TTFPREP, fixing up the IP / system name and userid; then Assemble and bind these programs.

Change TCAVARP or TCAVARB (or both, if you are so inclined) to add displays of the contents of REMOTE_ADDR and REMOTE_USER. Assemble and bind.

Deploy TCAVAR*x*, as discussed earlier.

Get into OMVS, cd to your html library, and oedit **test_display.html**, changing both occurrences of SCOMSTO to be your CGI directory mapping id.

Test your work by pointing your browser to AsmCGI_Labs.html:

* Select option 2, which takes you to test_display.html

* This page asks for a userid and password (you can use anything, as they are not checked - yet) and for you to select the name of the CGI you want to test; Fill these items in and select Submit; you should see the output from your CGI.

Take some time and study the outputs, especially for QUERY_STRING.