Developing Applications for z/OS UNIX

Developing Applications for z/OS UNIX - Course Objectives

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

1. Create programs written in COBOL, PL/I, C, or Assembler that;
   a. are compiled and bound under z/OS batch or a z/OS UNIX shell
   b. run in batch or from a shell script and that can ...
   c. work with z/OS files and files in the Hierarchical File System (HFS)
   d. interact with users at a z/OS UNIX terminal
   e. dynamically call subroutines that are located in a z/OS library or an HFS directory

2. Compile and bind C programs using c89, or Assemble and bind programs using c89, or Assemble programs using the as command, or compile and bind COBOL programs using cob2 or compile and bind PL/I programs using pli

3. Bind programs using the ld command

4. Code programs in C, COBOL, PL/I or Assembler that invoke common C functions to accomplish work, when that is the best way to get the task done

5. Code programs in C, COBOL, PL/I or Assembler that invoke common kernel services to accomplish work, when that is the best way to get the task done

6. Code programs in C, COBOL, PL/I or Assembler that create, set, access, and update environment variables, and that access the parm data passed to a main program

7. Build and use makefiles to manage an application.

Developing Applications for z/OS UNIX - Topical Outline

Day One

Developing Applications for z/OS UNIX - Topical Outline, p.2.

Developing Applications for z/OS UNIX - Topical Outline, p.3.

Developing Applications for z/OS UNIX - Topical Outline, p.4.

This page intentionally left almost blank.

Section Preview

❒ **Introduction**

- ♦ **Applications for z/OS UNIX**
- ♦ **Setting the Stage: A Level Set**
- ♦ **Setting The Stage: Skills To Acquire**
- ♦ **The Ubiquitousness of C**
- ♦ **Class Lab Set Up (Machine Exercise)**

Applications for z/OS UNIX

- **This class is part of a journey**
  - **A trip starting in the landscape of classic mainframe applications environment**
    - COBOL, PL/I, Assembler, C
    - REXX, CLIST
    - Batch / CICS / TSO
  - **Continuing through the waystations of getting current in these classic, evolving, technologies**
  - **Reaching a destination of skills to apply and extend the classic applications to your corporate Intranet, the Internet, and the World Wide Web (IIWWW, pronounced "eee-whew")**
- **The applications we are interested in are traditional (for example, inventory, accounting, human resources) but are run in a new way**
  - **Continuously available, constantly updated, widely shared (yet secured against inappropriate use and malicious damage)**
- **We are also interested in new applications, that is applications involving the internet and the web, and this class lays the foundations for developing those apps**

Applications for z/OS UNIX, 2

❐ **What does it mean to say that an application is "a z/OS UNIX application"? Any or all of ...**

- ♦ **The application accesses HFS files**

- ♦ **The application uses z/OS UNIX services**

- ♦ **The application runs "under" z/OS UNIX in one of the following ways**

    - ✗ As a program or script (shell script, REXX exec, Perl, etc.) submitted as a batch job by running BPXBATCH

    - ✗ As a command or script invoked from an OMVS, telnet, or rlogin session

    - ✗ As a command or script scheduled by the cron daemon or other scheduling software

❐ **For an application on the mainframe to work in the IIWWW environment, the app will have to be a z/OS UNIX application in the above sense**

- ♦ **So we build on our prerequisite material and extend your language skills in this course as part of our journey**

Applications for z/OS UNIX, 3

- ❒ **In this class, we will start with some programs compiled under z/OS and then run them under z/OS UNIX**
  - ♦ **Accessing z/OS files first, then HFS files**

- ❒ **We will explore how calling C functions (even from Assembler, COBOL, and PL/I programs(!)) can help implement z/OS UNIX applications**

- ❒ **We will explore compiling and binding programs under an OMVS session**
  - ♦ **Using commands, then scripts, then** make

- ❒ **We explore using some of the callable z/OS UNIX kernel services that might be of interest**
  - ♦ **Mostly to get familiar with how to use these services and a general picture of what services are available**

Setting The Stage: A Level Set

❒ **In the prerequisite courses, we covered these topics:**

- **Copying executables into the HFS using OPUT and OPUTX**

  ✗ Recall that OPUT will copy executables without change, but may give warning messages if the executable is a load module (no messages if the executable is a program object in a PDSE)

  ✗ OPUTX always does a re-bind of an executable

  ✗ For example, from ispf 6 issue a simple command like this:

```
==> oput tr.pdse(app2a) '/u/scomsto/bin/app2a'
```

- **Running under OMVS, you can execute programs stored in PDSs or PDSEs by using a symbolic link and setting the sticky bit on**

  ✗ Recall this series of commands:

  ➢ **ln -s dumbpgm app1c1**

  ➢ **touch dumbpgm**

  ➢ **chmod 1755 dumbpgm**

  ➢ **export STEPLIB=DEPT.TR.PDSE**

  ➢ **app1c1**

- **Compiled programs run under the shell must be reentrant or you must first set the environment variable _BPX_PTRACE_ATTACH to YES (export _BPX_PTRACE_ATTACH=YES)**

Setting The Stage: A Level Set, 2

- **Running under OMVS, you can execute program objects in the HFS by simply issuing the name of the program**

  - ✗ However, if the program expects DD statements, you must first issue **allocate** commands using the tso command prefix (we did not examine this fact earlier)

    - ➢ These commands may be entered one at a time or in a script; an example from the z/OS UNIX command line:

```
tso "alloc fi(indd) da('stnt.tr.zinputx') shr reu"
tso "alloc fi(reprt) da(myreprt) old reu"
appxxx
```

- **Some other points about running programs under OMVS:**

  - ✗ Files cannot be allocated to SYSOUT; you need to specify a z/OS file or an HFS file

  - ✗ Allocation commands are not inherited by a child process, so you must make sure spawn will try to use the current address space by starting OMVS with the SHAREAS option (the default) or exporting _BPX_SHAREAS with a value of YES:

```
export _BPX_SHAREAS=YES
```

  - ✗ For WTO outputs to display at the terminal you need to export the environment variable _BPXK_JOBLOG with a value of 1:

```
export _BPXK_JOBLOG=1
```

Setting The Stage: A Level Set, 3

❒ **Existing programs that reference standard sequential and VSAM data sets can run against HFS files with no change in code**

- **Simply change DD statements to use these parameters:**
  - ✗ DSNTYPE
  - ✗ FILEDATA
  - ✗ PATH
  - ✗ PATHDISP
  - ✗ PATHMODE
  - ✗ PATHOPTS
- **In addition to specifying DCB parameters where required by OPEN**
- **Note that this does not support random access, record deletions, or changing record size**

❒ **COBOL programs can be modified to dynamically allocate files (z/OS files or HFS files); see our course D704:** Enterprise COBOL Update **for details**

Setting The Stage: A Level Set, 3

- **Recall that you can run UNIX shell scripts or executables using the BPXBATCH and BPXBATSL programs in a batch job**

  - **With executables residing in the HFS**

  - **DD statements can be set up for z/OS and HFS files, including** stdin**,** stdout**,** stderr**, and** stdenv

  - **Additional steps can transcribe** stdout **and** stderr **files to the JES SPOOL (SYSOUT files)**

Setting The Stage: Skills To Acquire

- ❒ **z/OS UNIX applications are likely to need to use these techniques:**
  - ♦ **Reading / writing / updating z/OS files, HFS files, and relational data bases (DB2, Oracle, ...)**
  - ♦ **Interacting with the z/OS UNIX user at the terminal**
  - ♦ **Receiving requests (transactions) from users in XML or CGI formats**
  - ♦ **Responding to requests in XML or HTML or XHTML formats**
  - ♦ **Converting data between encoding schemes (EBCDIC, ASCII, Unicode)**

- ❒ **Some of these techniques are discussed in other courses**
  - ♦ **In this course we focus on working with files and interacting with the z/OS UNIX user**
  - ♦ **With added focus on necessary development / deployment skills (compiles, binds, makefiles, and the like)**

The Ubiquitousness of C

❒ In many senses, the C programming language is the language of UNIX and of the Internet

- ♦ C is based on building up complex functions from simple functions
- ♦ C works easily with networks, transmission streams, and HFS files

❒ To interact with UNIX terminal users, to process HFS files, and to do other tasks useful in the IIWWW environment, using existing features from other programming languages is not always possible

❒ Accomplishing these tasks in COBOL, PL/I, or Assembler may sometimes require calling some combination of

- ♦ LE (Language Environment) services
- ♦ C functions
- ♦ z/OS UNIX kernel functions / services

❒ Sometimes more than one of these options can get the job done

- ♦ Here we focus on using relevant C functions and kernel services where they can be useful
- ♦ Which means we will be discussing how to invoke these facilities directly from COBOL, PL/I, C, and Assembler programs

The Ubiquitousness of C, 2

❒ **So why not just use C for everything? There are a number of reasons:**

- **Since existing application code is already written in other languages, you can leverage existing code by integrating C functionality where it makes sense**

    - ✗ No need to even write a C subroutine: just call the necessary C functions

- **C programming skills are not widely available in the mainframe world**

    - ✗ But COBOL, PL/I, and Assembler skills are - so use the resources available

- **C is a terse language, not particularly self-documenting, and easily mis-understood**

    - ✗ It is not everyone's cup of tea

❒ **So we will emphasize C functions we can use in our immediate (and later) tasks, and how to invoke these functions from programs written in C and from programs written in other languages**

❒ **Note: C programs run under the shell must be AMODE31 or AMODE64, and the SCEERUN library must be available; this is also true for programs that call C functions**

Computer Exercise: Class Lab Set Up

1.) To set up for the lab requires some work. First, from ISPF option 6 issue this command:

```
===> ex '__________.train.library(u520strt)' exec
```

This will invoke a small dialog to create some files we will use for later exercises. The first thing you will see is a prompt for the high level qualifier to use for the data set names; it is set to be your TSO id and this is probably OK. In any case, set the value you want and press <Enter>. At this point the files you need will be created. The file names will begin with your high level qualifier (<hlq>) followed by TR:

<hlq>.TR.CEEDUMP	(space for dump file - just in case)
<hlq>.TR.CNTL	(PDS for JCL)
<hlq>.TR.COBOL	(PDS for COBOL source)
<hlq>.TR.LIBRARY	(PDS for source code and some data)
<hlq>.TR.PDSE	(PDSE for executable programs)
<hlq>.TR.REPRT	(space for output file)
<hlq>.TR.ZINPUTX	(data file)

2.) Next you need to unwind a pax file with scripts we will need later. Get into omvs in your home directory and issue this command:

```
pax -rf "//tr.library(u520pax)"
```

If this is successful you will have **app2**, **data**, **bin**, and **scripts** subdirectories created, with some of these having files in them. We will talk about these as we need them.

3.) Now exit from omvs; in ISPF get into option 3.4 and list your class files; in your TR.CNTL library, submit one of these jobs, depending on what language you prefer to work in:

UCOMPA	for Assembler
UCOMPC	for C
UCOMPCO	for COBOL
UCOMPP	for PL/I

Which ever job you run will compile (or assemble) and bind a source program into your TR.PDSE library.

Computer Exercise: Class Lab Set Up, p. 2.

4.) Again in your TR.CNTL library, submit one of these jobs, which will run the appropriate program and access your z/OS file TR.ZINPUTX and produce a report:

URUNA1	for Assembler
URUNC1	for C
URUNCO1	for COBOL
URUNP1	for PL/I

The expected output is the same for all programs, and a subset is reproduced on the next page.

5.) Finally, in your URUNxx member, replace the ZINPUTX DD statement to point to your data/zinputx file under your home directory; something like this:

```
//ZINPUTX DD PATH='/u/xxxxxx/data/zinputx',PATHOPTS=ORDONLY,
//   FILEDATA=BINARY,RECFM=F,LRECL=100,BLKSIZE=100
```

and run the job again; the results should look the same.

The program names are:

APP2A	Assembler
APP2C	C
APP2CO	COBOL
APP2P	PL/I

the source code for each of these is available in Appendix A of this book.

Computer Exercise: Class Lab Set Up, p. 3.

Expected output:

On the SYSOUT data set:

```
Got to main program APP2x
Leaving program APP2x
```

- **Where *x* is the programming language indicator**

On the REPRT data set:

```
PART03105   Final Flatulence                  35     10.750      376.25
PART03108   Giggling Gigolos                  35     10.900      381.50
PART03111   Marginal Magicians                35     11.050      386.75
.
.
.
PART03732   Rounding Errors                  240     42.100   10,104.00
PART03735   Founding Bearers                 245     42.250   10,351.25
PART03738   Noisy Smells                     245     42.400   10,388.00
```

Section Preview

❒ File Access in z/OS UNIX Applications

- What We Already Know
- C functions for accessing QSAM and HFS files
- COBOL - QSAM access
- COBOL - native access to HFS files
- PL/I - accessing QSAM and HFS files
- Assembler - accessing QSAM and HFS files
- Accessing HFS files under OMVS (Machine Exercise)

What We Already Know

- ❐ **In earlier training or experience you have certainly discovered these facts:**

  - ♦ **Classic applications can access z/OS files using JCL**

    - ✗ The role of JCL is to locate data and tie it to the program while it runs

  - ♦ **These same apps can access HFS files using JCL**

    - ✗ If the file access is restricted to sequential processing of text files

    - ✗ Accessing HFS files with non-text data (for example, packed decimal, binary, floating point) is a bit trickier

- ❐ **You may or may not be familiar with this information:**

  - ♦ **C programs can access z/OS files and HFS files natively with the same code in batch, under TSO, and under omvs**

  - ♦ **COBOL programs can access HFS files natively using line sequential file types, but COBOL cannot use a single FD to access both z/OS files and HFS files natively**

  - ♦ **PL/I programs can access HFS files using the TITLE option of open or by exporting an environment variable named DD_***ddname*

  - ♦ **Assembler programs cannot natively access HFS files at all using classic macro interfaces**

C Functions For Accessing QSAM and HFS Files

❒ **There are, of course, lots of details, options, special cases, and so on - we keep it simple here and trust you can look these up in the docs as you need them**

❒ **As with other languages, a C program must declare a variable that will be used to reference a file, something like this:**

```
    FILE  *zinputx;
    FILE  *reprt;
```

Notes

- **For those not familiar with C, C is case sensitive, so capitalization matters everywhere**

- **The above statements tell the compiler you are working with two file variables**

  - ✗ The variable named **zinputx** is a pointer to a FILE "thing"

  - ✗ The variable named **reprt** is a pointer to another FILE "thing"

    - ➢ the asterisk says "is a pointer to the preceding type"

C Functions For Accessing QSAM and HFS Files, 2

- **To open a file in C, you usually invoke the** fopen() **function**
  - **This function has two arguments**
    - A file specification
    - Open options, coded as a character string
  - **And it returns a single value:**
    - The address to put into the variable that is supposed to point to the FILE "thing"
      - Special case: if the returned value is 0, the open was not successful
        - Note: to test if an address is 0, you check to see if the value is **NULL** (reserved word)

C Functions For Accessing QSAM and HFS Files, 3

❒ **The prototype for** fopen() **in the documentation is:**

FILE *fopen(*const char *filename***,** *const char *mode***);**

- **Which can be read:**

  ✗ the **fopen()** function returns the address of a FILE "thing", and takes two arguments, each a pointer to a constant character string

❒ **Thus, you** could **code:**

```
  zinputx = fopen(file_name, file_options);
```

- **This would invoke the function, passing the arguments, and return the value into our variable which is supposed to hold the address of a FILE thing**

  ✗ Then we could test the value in **zinputx** to see if it is zero (**NULL**) or not

- **But C programmers inevitably combine this into a single statement, like:**

```
if ((zinputx = fopen(filespec, fileoptions)) == NULL)
        { action_to_take_if_error_encountered }
```

- **The parentheses, braces, double equals sign (==) and bold characters (except for the filename zinputx) must be coded as shown (they can wrap across multiple lines, if necessary)**

C Functions For Accessing QSAM and HFS Files, 4

❒ **Now let us examine the two arguments passed to** fopen()

- **First, the** <u>file specification</u>**; this can be any of**

  ✗ A quoted literal HFS file name or z/OS file name:

```
if ((zinputx = fopen("/u/scomsto/data/myinputx", ...
```

  ✗ A string variable containing a [z/OS or HFS] file name (or at least a variable that will contain a file name before the fopen is executed):

```
char infile[] = "/u/scomsto/data/myinputx";
.
.
.
if ((zinputx = fopen(infile, ...
```

  ✗ A literal or variable that will contain the string **DD:***ddname* before fopen is issued:

```
if ((zinputx = fopen("DD:ZINPUTX", ...
```

or

```
char infile[] = "DD:ZINPUTX";
.
.
.
if ((zinputx = fopen(infile, ...
```

- **In this case, the run time DD statement (or corresponding ALLOCATE command) must point to the desired path or dataset**

C Functions For Accessing QSAM and HFS Files, 5

❒ **Now let us examine the two arguments passed to** fopen()**, continued**

- **Second, the** open options**; this can also be literal or character string**

❒ **There are many options, and they may be specified only as needed and in any order; most commonly used options:**

- **r - read file as text (delimited) file (may also be code as** rt**)**
- **w - write file as text (delimited) file (also** wt**)**
- **a - append data to text (delimited) file (also** at**)**
- **rb - read file as binary (non-delimited) file**
- **wb - write file as binary (non-delimited) file**
- **ab - append data to binary (non-delimited) file**
- **recfm= - record format, usual combinations of F, V, U, B, S**
- **lrecl= - logical record length**
- **blksize= - block size**
- **type=record - sequential record I/O; supports both HFS and traditional z/OS files; open must be for rb, wb, or ab; also used for VSAM files, both sequential and direct**

Note

- **A text file may only contain printable characters and control characters; a binary file may contain any bit patterns**

  ✗ Files opened with **type=record** must be opened as binary

C Functions For Accessing QSAM and HFS Files, 6

❒ **Now let us examine the two arguments passed to** fopen()**, continued**

♦ **Second, the <u>open options</u>; this can also be literal or character string, continued**

<u>Examples - as found in our provided programs</u>

```
FILE  *zinputx;
char  zinputx_DD[] = "DD:ZINPUTX";
FILE  *reprt;
char  reprt_DD[] = "DD:REPRT";

if ((zinputx = fopen(zinputx_DD, "rb")) == NULL )
    { action_to_take_if_error_encountered }

if ((reprt = fopen(reprt_DD, "wb, lrecl=76, recfm=fb")) == NULL)
    { action_to_take_if_error_encountered }
```

```
FILE  *in_file;
char  in_name[58];
.
.
.
/*   get  file name into in_name   */
.
.
.
if ((in_file = fopen(in_name, "rb,type=record")) == NULL )
    { action_to_take_if_error_encountered }
```

❒ **Both approaches work for running under z/OS UNIX (omvs), TSO, and in traditional batch**

❒ **The second approach allows you to prompt for the file name (or have it passed as an argument) and it works for both HFS pathnames and for z/OS file names using the "*//dsn*" convention for z/OS file names, as discussed in earlier courses**

C Functions For Accessing QSAM and HFS Files, 7

❒ **So, what to make of the mysterious code described as "*action_to_take_if_error_encountered*"?**

- **As with most error handling routines in any language, there is a large number of possibilities but the most common seems to be what we do:**
  - ✗ Display an error message (in C, use the **printf()** function)
  - ✗ Leave the program, possibly specifying a non-zero return code (in C, use the **exit()** function or **return** statement)

Putting it all together: the whole file declare and open:

```
FILE  *zinputx;
char  zinputx_DD[] = "DD:ZINPUTX";
FILE  *reprt;
char  reprt_DD[] = "DD:REPRT";

if ((zinputx = fopen(zinputx_DD, "rb")) == NULL )
    { printf("Can't open %s\n",zinputx_DD);
      exit(1);
    }

if ((reprt = fopen(reprt_DD, "wb, lrecl=76, recfm=fb")) == NULL)
    { printf("Can't open %s\n", reprt_DD);
      exit(2);
    }
```

```
FILE  *in_file;
char  in_name[58];
.
.
.
/*   get  file name into in_file   */
.
.
.
if ((in_file = fopen(in_file, "rb,type=record")) == NULL )
    { printf("\nCan't open %s for reading.", in_file);
      return -1;
    }
```

C Functions For Accessing QSAM and HFS Files, 8

- ❐ **We will leave a** printf() **discussion until later (idea should be generally clear) and we will not say anything more about** exit()

- ❐ **Now we turn to reading and writing data, which is done using the** fread() **and** fwrite() **functions, respectively**

- ❐ **The prototype for** fread() **in the documentation is:**

  int_bytes **fread(****buffer*, *int_size*, *int_count*, ******file_name)***;**

  - ♦ **Which can be interpreted as saying:**

    - ✗ the **fread()** function returns an integer indicating the number of bytes read

    - ✗ **fread()** is passed four arguments:

      - ➢ Address of the buffer area to put the data
      - ➢ size of chunks to read, as an integer (use maximum record size)
      - ➢ number of chunks to read, as an integer (use **1**)
      - ➢ address of the FILE thing from which data is read

    - ✗ If the return value is zero, it indicates end of file has been reached

C Functions For Accessing QSAM and HFS Files, 9

❒ One common construct of managing to read a file:

```
int  no_bytes;
.
.
.
no_bytes = fread(&in_rec, sizeof(in_rec), 1, zinputx);

while (no_bytes > 0)
   {
/*        code to handle current record  */
.
.
.
no_bytes = fread(&in_rec, sizeof in_rec, 1, zinputx);
   }
```

❒ As an alternative, specify you are reading chunks of 1 byte and specify a maximum record size in the number of chunks:

```
no_bytes = fread(in_data, 1, sizeof in_data, in_file);
```

Notes

- Coding an ampersand (&) followed by a variable name means "the address of" the variable; use this for a structure
- Coding a name for a character string (which is always an array in C), you can omit the ampersand: strings are always managed by addresses
- General rules and details discussed shortly

C Functions For Accessing QSAM and HFS Files, 10

❒ **The prototype for** fwrite() **in the documentation is:**

int_chunks **fwrite(****buffer*, *int_size*, *int_count*, **file_name)***;**

- **Which can be interpreted as saying:**
  - ✗ the **fwrite()** function returns an integer indicating the number of chunks written
  - ✗ fwrite() is passed four arguments:
    - ➢ Address of the buffer area to write from
    - ➢ size of chunks to write, as an integer (use maximum record size)
    - ➢ number of chunks to write, as an integer (use '1')
    - ➢ address of the FILE thing to which data is written
  - ✗ If the return value is less than *int_count*, a write error occurred

❒ **Typically, C programmers ignore the return value, and so are likely to code** fwrite() **functions as if they were statements (verbs) themselves:**

```
fwrite(&out_rec, sizeof out_rec, 1, reprt);
```

- **Instead of:**

```
no_chunks = fwrite(&out_rec, sizeof out_rec, 1, reprt);
```

C Functions For Accessing QSAM and HFS Files, 11

❒ **The prototype for** fclose() **in the documentation is:**

int_success **fclose(****file_name)***;**

- **Which can be interpreted as saying:**

  ✗ the **fclose()** function returns an integer indicating if the close was successful (value of 0) or not (non-zero values; details in the docs)

- **Again, programmers seldom check the return value, so** fclose() **is often coded as:**

```
fclose(zinputx);

fclose(reprt);
```

❒ **Note that code written to be "bullet-proof" will check return values from** fwrite() **and** fclose(), **in order to find out about errors and to handle them**

- **I/O error handling is not discussed in this course**

C Functions For Accessing QSAM and HFS Files, 12

- **Now, to run in batch, specify DD statements for the DD names**
    - **Point to z/OS data sets (DSN=...) or HFS data sets (PATH='/...')**
        - Unless pathname or dataset name is hard coded in the program

- **To run under TSO, use allocate commands for a z/OS file or HFS file (unless hard coded in the program), followed by a TSO CALL command to the program name**

- **To run under omvs, issue TSO allocate commands for a z/OS file or HFS file (unless hard coded in the program), then issue the program name**

COBOL - QSAM and HFS Access

❒ Classic COBOL programming can access both z/OS and HFS files, as follows (note: COBOL is case insensitive except in literals):

Code components

♦ SELECT statement for a file, for example:

```
Environment division.
Input-output section.
File-control.
    select zinputx assign to zinputx
       file status is in-stat.

    select reprt assign to reprt.
```

♦ FD (File Definition) entry for each file, followed by an 01-level record layout (single line or complete structure), for example:

```
Data division.
File section.
fd  zinputx
    block contains 0 records.
01  in-rec   pic x(100).

fd  reprt.
01  out-rec   pic x(76).
```

COBOL - QSAM and HFS Access, 2

❒ **Classic COBOL programming, continued**

Code components, continued

♦ **Probably working-storage for record layout(s) and any file status fields, work fields for calculations, constant data, and so on, for example:**

```
working-storage section.
01  in-record.
    05 in-part-number       pic x(9).
    05 in-description       pic x(30).
    05                      pic x(5).
    05 in-unit-price        pic 9999v999.
    05 in-quantity-on-hand  pic 99999.
    05 in-quantity-on-ord   pic 999.
    05 in-reorder-level     pic 999.
    05 in-switch            pic xx.
    05 in-old-part-no       pic x(9).
    05 in-category          pic x(10).
    05                      pic x(17).

01  reprt-record.
    05                         pic x(1)  value spaces.
    05 reprt-part-number       pic x(9).
    05                         pic x(3)  value spaces.
    05 reprt-description       pic x(30).
    05                         pic x(3)  value spaces.
    05 reprt-quantity-on-hand  pic zz,zz9.
    05                         pic x(3)  value spaces.
    05 reprt-unit-price        pic z,zz9.999.
    05                         pic x(2)  value spaces.
    05 reprt-value             pic zzz,zz9.99.

77  more-records               pic x     value 'Y'.
77  work-value  packed-decimal pic s9(7)v99.

77  in-stat                    pic 99    value 00.
```

COBOL - QSAM and HFS Access, 3

❒ **Classic COBOL programming, continued**

Code components, continued

- **In the procedure division, code to open, read, process, and write, and close files and records, for example:**

```
Procedure division.
mainline.
    open input zinputx output reprt

    perform read-a-record
    perform do-the-work until more-records = 'N'

    close zinputx, reprt

    stop run.

read-a-record.
    read zinputx into in-record
         at end move 'N' to more-records.


do-the-work.
    move in-part-number to reprt-part-number
    .
    .
    .
    compute ...
    write reprt-rec from reprt-record
    perform read-a-record.
```

COBOL - QSAM and HFS Access, 4

- ❒ **Now, to run in batch, specify DD statements for the DD names**
  - ♦ **Point to z/OS data sets (DSN=...) or HFS data sets (PATH='/...')**
    - ✗ Or omit a DD statement and use environment variables in your program to dynamically allocate file(s) you need, as discussed in an earlier course
- ❒ **To run under TSO, use allocate commands for a z/OS file or HFS file (or omit a DD statement and use environment variables in your program to dynamically allocate file(s) you need), followed by a TSO CALL command to the program name**
- ❒ **To run under omvs, issue TSO allocate commands for a z/OS file or HFS file (unless using dynamic allocation techniques mentioned above), then issue the program name**

COBOL - Native Access to HFS Files

❒ **You can define HFS files using the Enterprise COBOL compiler**

- ♦ **In the SELECT statement, specify the organization as LINE SEQUENTIAL**

- ♦ **In the FD specify the record contains from a minimum to a maximum number of characters and one record per block**

- ♦ **In the 01 structure after the FD, specify a record equal to the maximum size**

- ♦ **To connect the internal file to an external file, you can use the ASSIGN TO value as**

  - ✗ A DD name (for a DD statement that uses PATH and other HFS parameters)

  - ✗ Or hard code or dynamically determine the path name prior to open

  - ✗ Or provide an environment variable at run time, as part of a script or just coded as a command under omvs

- ♦ **Open, close, read, and write are as for standard QSAM files in COBOL**

- ♦ **Files declared as line sequential cannot be used to process QSAM files**

❒ **The program fragments on the next two pages demonstrate many of these techniques**

COBOL - Native Access to HFS Files, 2

```
 Identification division.
 Program-id.  app2co3x.
*  Copyright (c) 2010 by Steven H. Comstock     Ver2

 Environment division.
 Configuration section.
 Special-names.
     Sysin is path-input.
 Input-output section.
 File-control.
     select inline assign to inline
       line sequential status is line-seq-stat.

 Data division.
 File section.

 FD  inline
     record varying from 12 to 1000 characters
     block contains 1 records.
 01  line-seq-line       pic x(1000).

 Working-storage section.
 01  Status-stuff.
     02  line-seq-stat     pic 99        value 00.

 01  File-stuff.
     02  file-ptr          pointer.
     02  path-name.
         03                pic x(12)
             value 'INLINE=PATH('.
         03  pathname      pic x(100)  value spaces.
         03                pic xx     value z' '.
     02  rc                pic s9(9)  binary value 0.

 01  inrec     pic x(1000).
 01  rec-work  pic x(1000).
 01  file-name-in    pic x(100)      value spaces.
```

COBOL - Native Access to HFS Files, 3

```
procedure division.
mainline.

     accept file-name-in from path-input
     string file-name-in delimited by space
            ')' delimited by size into pathname

     set file-ptr to address of path-name
     call 'putenv' using by value file-ptr returning rc

     open input inline

     if line-seq-stat = '00'
        perform read-rec
        ...
     else
        display 'Open failed; file status = '
                               line-seq-stat
     end-if

     close inline
     goback.

read-rec.
     read inline into inrec
     .
     .
     .
```

❒ **Remember, these are just code fragments, excerpts from the code**

- **The entire program is found in your TR.COBOL library**

PL/I - Accessing QSAM and HFS Files

❒ **Classic PL/I programming can access both z/OS and HFS files, as follows (note: PL/I is case insensitive except in literals):**

Code components

♦ **A DECLARE statement for each file, for example:**

```
dcl zinputx  file record sequential;
dcl reprt    file record sequential output;
```

♦ **A DECLARE for a variable for each input file to set at end of file, and an ON unit to set the variable:**

```
dcl more_to_do  char(1)  init('y');
.
.
.
on endfile(zinputx) more_to_do = 'n';
```

♦ **Possibly other routines to handle conditions that can arise for OPEN, READ, WRITE, or CLOSE in addition to endfile**

✗ Such as ENDPAGE, ERROR, KEY, RECORD, TRANSMIT, UNDEFINEDFILE; these are not discussed here

PL/I - Accessing QSAM and HFS Files, 2

❒ **Classic PL/I programming, continued**

Code components, continued

- ♦ **DECLARE statement(s) for storage to hold record definitions, for example**

```
dcl 1 inrec,
    2 i_part_no          char(9),
    2 i_description      char(30),
    2 i_rsv_1            char(5),
    2 i_unit_price       pic '9999v999',
    2 i_qty_on_hand      pic '99999',
    2 i_qty_on_ord       pic '999',
    2 i_reorder_level    pic '999',
    2 i_switch           char(2),
    2 i_old_part_no      char(9),
    2 i_category         char(10),
    2 i_rsv_3            char(17);

dcl 1 outrec,
      2 *                char(1)      init (' '),
      2 o_part_no        char(9),
      2 *                char(3)      init (' '),
      2 o_description    char(30),
      2 *                char(3)      init (' '),
      2 o_qty_on_hand    pic 'zz,zz9',
      2 *                char(3)      init (' '),
      2 o_unit_price     pic  'z,zz9v.999',
      2 *                char(2)      init (' '),
      2 o_value          pic 'zzz,zz9v.99';
```

PL/I - Accessing QSAM and HFS Files, 3

❐ **Classic PL/I programming, continued**

Code components, continued

- ♦ **Imperative statements to open, read, write, close, and manage data transfer, for example:**

```
open file(zinputx), file(reprt) output;
read file(zinputx) into(inrec);

do while (more_to_do = 'y');
     o_part_no       =  i_part_no;
    .
    .
    .
     o_value         =  i_qty_on_hand * i_unit_price;
     write file(reprt) from(outrec);
     read file(zinputx) into(inrec);
end;    /* do report  */

close file(zinputx), file(reprt);
```

❐ **Note that a filename can be longer than 8 characters; if so, the implied DD name is the first 8 characters of the file name**

PL/I - Accessing QSAM and HFS Files, 4

❒ **Another possibility is to use the TITLE option of the OPEN statement**

- ♦ **In this case, the value in TITLE can be an alternate DD name, such as:**

```
open file(zinputx) title(infil);
```

- ♦ **Or, the value in the TITLE can be the absolute pathname of an HFS file, preceded by a slash, for example:**

```
open file(zinputx) title('//u/scomsto/indata');
```

PL/I - Accessing QSAM and HFS Files, 5

- ❒ **Now, to run in batch, specify DD statements for the DD names**
  - ♦ **Point to z/OS data sets (DSN=...) or HFS data sets (PATH='/...')**
    - ✗ Unless path hard coded in the program (in the TITLE), for HFS files - then do not need DD statements

- ❒ **To run under TSO, use allocate commands for a z/OS file or HFS file (unless path hard coded in the program), followed by a TSO CALL command to the program name**

- ❒ **To run under omvs, need to establish a value in an environment variable with a name of DD_***ddname*
  - ♦ *ddname* **must be the file name or value in the TITLE option of the open statement, and it must be upper case**
  - ♦ **Using this approach,** do not **issue TSO allocate commands and the files cannot be z/OS files, only HFS files**
  - ♦ **This currently seems to be the only way to access HFS files from PL/I, using native PL/I constructs and verbs, under omvs**

Assembler - Accessing QSAM and HFS Files

❒ **Note that the Assembler programmer has a lot of basic choices to make that influence their code, in particular:**

- ♦ **Will the code be LE-compliant, or not?**
  - ✗ In order to use C functions, the Assembler code must be LE-compliant - so our examples will follow that model
- ♦ **Will the code be reenterant, or not?**
  - ✗ With current z/Architecture machines, reenterant code performs better, so we will make our code reenterant
- ♦ **Will the code run AMODE24, AMODE31, or AMODE64?**
  - ✗ Inter-module communication is faster when running AMODE31, so we will run in that mode
- ♦ **Will the code be loaded above the line or below it?**
  - ✗ To allow for the greatest flexibility, we will have the code loaded above the line

❒ **So our examples in lecture and for the labs will be Assembler code that is LE-compliant, reenterant, running AMODE31 and RMODEANY**

- ♦ **Note that there are some examples in your TR.SOURCE library that use different choices**

Assembler - Accessing QSAM and HFS Files, 2

❒ **Classic Assembler programming can access both z/OS and HFS files, as follows (note: Assembler is case insensitive except in literals, if the correct *PROCESS statement is included):**

Code components

♦ **A DCB statement for each file, for example:**

```
ZINPUTX  DCB    DDNAME=ZINPUTX,MACRF=(GM),DSORG=PS,DCBE=ID
size_ind equ   *-zinputx
REPRT    DCB    DDNAME=REPRT,MACRF=(PM),RECFM=FB,           X
                LRECL=76,DSORG=PS,DCBE=OD
size_out equ   *-reprt
```

♦ **We need the size of the DCBs to use when we move the DCBs below the line**

♦ **DCBEs are needed if you will be running above the line**

✗ DCBs have to be below the line, so they will point to the DCBEs, which can have 31-bit addresses for such items as EODAD (the end of data routine address)

♦ **So we need DCBEs, then, for example:**

```
ID       DCBE   RMODE31=BUFF,EODAD=ENDIN
dcbe_len equ    *-id
OD       DCBE   RMODE31=BUFF
dcbo_len equ    *-od
```

Assembler - Accessing QSAM and HFS Files, 3

- We will need the usual pieces of code to access files

    - Control blocks to represent the files (DCBs and DCBEs as mentioned)

    - I/O areas to get data into and out of

    - OPEN commands to connect the internal control blocks with real, external files

    - GET macro to read records

    - An end of file routine to get control when there are no more input records (point to this using the EODAD option on the DCBE macro)

    - Assembler instructions to process the input records and build an output record

    - PUT macro to write records

    - CLOSE macros to disconnect from the external files (flush buffers, write final blocks and any EOF indicators, and so on)

Assembler - Accessing QSAM and HFS Files, 4

❒ To make an Assembler program reenterant requires some planning, so we start with a CEEENTRY macro and at the end we define areas for storage to be gotten on entry to the program:

```
*PROCESS COMPAT(NOCASE,MACROCASE)
APP2A    CEEENTRY PPA=MESSPA,AUTO=WORKSIZE
         USING WAREAS,13
.
.
.
MESSPPA  CEEPPA
         LTORG
wareas   dsect
         org   *+CEEDSASZ
openl    open  (,,,),mf=l,mode=31
dcbei    dcbe  rmode31=buff,eodad=endin
dcbeo    dcbe  rmode31=buff
*   record layouts for input and output
inarea   ds    0cl100
partno   ds    cl9
.
.
.
outarea  ds    0cl76
         ds    cl1
rptpart# ds    cl9
.
.
.
worksize equ   *-wareas
         CEEDSA
         CEECAA
         END   APP2A
```

Assembler - Accessing QSAM and HFS Files, 5

❒ Also part of the price to pay to be reenterant is the need to have list and execute forms of macros

- ♦ The list form (MF=L) generates control blocks with values
- ♦ The execute form (MF=(E,*list_form*) generates instructions to request a service using the list form data
- ♦ We need to put the list forms into gotten storage so when we invoke the execute forms the services will use the values found in the gotten storage
- ♦ Similarly, this is why we put, for example, DCBEs in our constant areas and in our dsect for the gotten storage: the DCBEs in the dsect ensure we are reserving a large enough area of storage

❒ Putting all these pieces in place, we find the structure of our code looks like this

- ♦ A CEEENTRY macro (note: automatically grabs storage above the line as large as the AUTO parameter requests)
- ♦ A USING instruction
- ♦ A branch around the constants area
- ♦ The constants area
- ♦ Code to initialize the dsect areas
- ♦ GETMAIN to obtain storage below the line
- ♦ Code to move DCBs to the gotten area
- ♦ -- actual program processing

Assembler - Accessing QSAM and HFS Files, 6

❒ **To** put the pieces into some context**, here we show larger fragments, in the order we would have them in a program ...**

```
*PROCESS COMPAT(NOCASE,MACROCASE)
APP2A    CEEENTRY PPA=MESSPA,AUTO=WORKSIZE
         USING WAREAS,13
         bru   the_code
openlsr  open  (,,,),mf=l,mode=31
openl_len equ  *-openlsr
ZINPUTX  DCB    DDNAME=ZINPUTX,MACRF=(GM),DSORG=PS,DCBE=ID
size_ind equ   *-zinputx
REPRT    DCB    DDNAME=REPRT,MACRF=(PM),RECFM=FB,              X
                LRECL=76,DSORG=PS,DCBE=OD
size_out equ   *-reprt
ID       DCBE  RMODE31=BUFF,EODAD=GOBACK
dcbe_len equ   *-id
OD       DCBE  RMODE31=BUFF
dcbo_len equ   *-od
the_code ds    0h
         mvc   openl(openl_len),openlsr
         mvc   dcbei(dcbe_len),id
         mvc   dcbe0(dcbo_len),od
         GETMAIN R,LV=SIZE_IND,LOC=BELOW
         lr    5,1
         mvc   0(size_ind,5),zinputx
         la    8,dcbei
         st    8,0(5)
         GETMAIN R,LV=SIZE_OUT,LOC=BELOW
         lr    6,1
         mvc   0(size_out,6),reprt
         la    8,dcbeo
         st    8,0(6)
         open  ((5),(input),(6),(output)),mf=(e,openl),mode=31
```

- **At this point, we now have two open files; assume we can leave R5 with the address of the input DCB and R6 with the address of the output DCB**

Assembler - Accessing QSAM and HFS Files, 7

- To get an input record, code something like this:

```
         get   (5),inarea
```

- Processing a record uses the appropriate Assembler code, then to write out a record we would code:

```
         put   (6),outarea
```

- Our end of file routine might be simple like just issuing a close:

```
         close ((5),,((6)),mf=(e,openl),mode=31
```

- Note that the execute form of the close can refer to the list form of the open: they both generate the same size and layout of memory

- After close we would probably exit using the CEETERM macro

- Finally, after our executable instructions we would code the dsect type of information we discussed earlier, so we repeat that much on the next page

Assembler - Accessing QSAM and HFS Files, 8

```
MESSPPA  CEEPPA
         LTORG
wareas   dsect
         org   *+CEEDSASZ
openl    open  (,,,),mf=l,mode=31
dcbei    dcbe  rmode31=buff,eodad=endin
dcbeo    dcbe  rmode31=buff
*   record layouts for input and output
inarea   ds    0cl100
partno   ds    cl9
.
.
.
outarea  ds    0cl76
         ds    cl1
rptpart# ds    cl9
.
.
.
worksize equ   *-wareas
         CEEDSA
         CEECAA
         END   APP2A
```

❒ **Now, to run in batch, specify DD statements for the DD names**

- ♦ **Point to z/OS data sets (DSN=...) or HFS data sets (PATH='/...')**

❒ **To run under TSO, use allocate commands for a z/OS file or HFS file, followed by a TSO CALL command to the program name**

❒ **To run under omvs, issue TSO allocate commands for a z/OS file or HFS file, then issue the program name**

Computer Exercise: Accessing HFS Files Under OMVS

Think of this past section as informational for when you do programming in any of the languages we are discussing.

In this lab, you will run the program you worked with in the previous lab, but now you will run it from under OMVS. To do this, you will:

1. copy the program to run from your tr.pdse library into your bin directory (see the discussion on page 11)

2. get into OMVS, and run one of the supplied scripts.

   All the scripts are found in your ~/scripts library, and they are:

runa	- run APP2A
runc	- run APP2C
runco	- run APP2CO
runp	- run APP2P

These scripts set environment variables and issue TSO allocate commands as discussed on page 12, (runp also populates some additional environment variables).

Run the scripts by entering, from the command line:

a dot a space the script name, like:

. run*x*

Computer Exercise: Accessing HFS Files Under OMVS, 2

Expected results:

On the omvs display:

```
runx Ver1 is setting up variables.
Remember to run this with a dot and space before runx.

Got to main program APP2x
Leaving program APP2x
```

In the TR.REPRT data set, the same report you saw on SYSOUT when you ran the program in batch.

If you abend, you will find a dump in TR.CEEDUMP.