Creating and Using DLLs in z/OS

Creating and Using DLLs in z/OS - Course Objectives

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

1. Code DLLs and DLL applications using one or more of these compilers:
   - Enterprise COBOL
   - XL C/C++ for z/OS
   - Enterprise PL/I

   or Assembler language

2. Use packages in PL/I programs, if PL/I is available.

Creating and Using DLLs in z/OS - Topical Outline

DLLs - Assembler
- Assembler Language support for DLLs - original
- Assembler Support: HLASM 1.5+ and z/OS 1.6+
- The CEEENTRY macro
- The CEEPDDA macro
- The CEEPLDA macro
- The CEEPCALL macro
- Assembler DLL considerations

Conclusions
- Loading DLLs - Tradeoffs
- Subroutine Linkages - Options
- Computer Exercise: Creating and Using DLLS 136

Languages Selection

- ❒ **This course is multi-lingual, but we don't talk about programming languages you will not be encountering**
- ❒ **So here is the time for you to specify which languages you are interested in exploring during this class**
- ❒ **Based on your selection(s) we will omit parts of lecture and labs that are not relevant to your work**

Language

_____ **Assembler**

_____ **C**

_____ **COBOL**

_____ **PL/I**

Section Preview

- **Introduction to the class**
  - **Interesting Applications**
  - **Coding Notes For Examples in the Class**
  - **Setting Up for the Labs (Machine Exercise)**

Interesting Applications

- **Applications that are simple can be written as self-contained single programs as an on-line transaction or a batch job-step**

- **But interesting (read: complex) applications often need to be written as a mainline (driver) program with one or more subroutines**
  - **The mainline calls subroutines as needed**
  - **And subroutines can in turn call other subroutines**

- **A good design point is to compartmentalize each subroutine to perform a single function**
  - **If that function can be broken down into sub-pieces, put those pieces into separate subroutines**
  - **This way, updates and maintenance are localized and simplified**

Interesting Applications, 2

- **Typically when a program (mainline or subroutine) calls a subroutine, the caller passes data to the callee**
  - **The called program then accesses the passed data, and may change the passed data**
  - **The called program may also return a value to the caller**
- **Life is sweet and simple if all programs are written in a single language**
  - **But this is often not the case:**
    - ✗ High level language programs, written in COBOL or PL/I, say, may need to call subroutines that were written in Assembler to accomplish some function that cannot be done in the high level language
    - ✗ Conversely, many functions are accomplished more simply in a high level language than in Assembler
    - ✗ Certain computations may be done more naturally in PL/I or C (engineering applications often need to work with math functions and imaginary numbers, for example, tasks not well suited to COBOL)
    - ✗ The person writing the subroutine may prefer to code in a particular language that is not the same as the language of the calling program

Interesting Applications, 3

- **In this class we explore the mysteries and details of coding applications written using Dynamic Link Libraries (DLLs)**

- **This includes programs written in these languages**
  - **Assembler**
  - **COBOL**
  - **PL/I**
  - **C**

- **We examine invoking DLL routines written in the same language as the invoker and invoking routines written in different languages from the invoker**

- **We are specifically focused on the most current compilers and running in the z/OS LE environment**
  - **We assume you are proficient in at least one of the four languages discussed, but that you may not be familiar with how to work in all of them**
    - ✗ So we have provided enough details and clues to enable you to succeed in the labs that use languages that you might not be fluent in

Coding Notes For Examples in the Class

- ❒ **We assume you are using the most recent versions of compilers / the Assembler, z/OS, Language Environment, and the program binder**
  - ♦ **However, most of the discussion is relevant to earlier versions of each of these products**
  - ♦ **A newer version of a product may be available several months before these materials are updated**
  - ♦ **Where it is especially critical, versions and levels of products will be specified**
- ❒ **We are concerned with having lots of correct coding examples**
  - ♦ **And we want them to be complete enough for you to use these examples as models / starting points back on the job**
  - ♦ **But, we do not want to clutter up examples with lines of code that should be clear to experienced programmers**
  - ♦ **For example, we will not show declarations of data items unless it is necessary for clarity**
  - ♦ **To simplify the examples, therefore, we have put on these following pages assumptions you can make about unshown segments of a program**

Coding Notes For Examples in the Class - Assembler

- **In Assembler examples, we will not show standard save area linkage code unless it is required to demonstrate some aspect of the example**
  - **We will not show the LE Assembler macros, but we will specify if an Assembler example is LE conforming or not, if it makes a difference in behavior**
  - **Generally speaking, everything discussed here works for LE-conforming Assembler, while non-LE conforming Assembler can:**
    - ✗ Call LE COBOL subroutines directly with a lot of overhead or call intermediate routines to first establish the LE environment
    - ✗ Call LE PL/I subroutines only using intermediate routines to first establish the LE environment
    - ✗ Call LE C subroutines only using intermediate routines to first establish the LE environment
- **We will not necessarily show the target of branch instructions, if the content of the code is not central to the example**
- **The following data names may be used in examples, assuming definitions as shown:**

```
fc        dc    12x'00'       for LE feedback
dest      dc    f'2'          for LE message routing
dblwrd    dc    d'0'          for conversions
```

Coding Notes For Examples in the Class - COBOL

- In COBOL examples, we will not show any divisions not necessary for understanding of an example
  - We assume familiarity with COBOL program structure
- We will not necessarily show the target of "perform" statements, if the content of the code is not central to the example
- The following data names may be used in examples, assuming definitions as shown:

```
01  fc      pic x(12)  value low-values.
01  dest    pic s9(9)  binary value 2.
```

Coding Notes For Examples in the Class - PL/I

- In PL/I examples, we will not show any code not necessary for understanding of an example
  - We assume familiarity with PL/I program structure
  - We will not generally show declarations for builtin functions nor LE service routines
- We will not necessarily show the target of "call" statements, if the content of the code is not central to the example
- The following data names may be used in examples, assuming definitions as shown:

```
dcl  fc    char(12)           init(low(12));
dcl  dest  fixed  binary(31)  init(2);
```

Coding Notes For Examples in the Class - C

- **In C examples, <u>we will not show any code not necessary for understanding of an example</u>**
  - **We assume familiarity with C program structure**
  - **All C examples may or may not also apply to C++**
  - **We will not generally show all #includes, unless necessary to demonstrate some aspect of the example; you need to ensure you have all necessary #include statements in any code you write; be sure to check these:**
    - ✗ #include <leawi.h> for LE services support
    - ✗ #include <decimal.h> for packed decimal support
- **We <u>will not necessarily show the target of function references</u>, if the content of the code is not central to the example**
- **Examples use standard C notations; but actual code in the labs uses trigraphs, mostly: "??(" for "[" and "??)" for "]"**
- **The following data names may be used in examples, assuming definitions as shown:**

```
_FEEDBACK  fc;
long int  dest = 2;
long int i;
long int j;
long int k;
```

Computer Exercise: Setting Up for the Labs

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

First, you need to run M525STRT, 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.

From ISPF option 6, on the command line enter:

```
===> ex '__________.train.library(m525strt)' 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 all your JCL
<hlq>.TR.COBOL	for all COBOL source code
<hlq>.TR.SOURCE	for all other source code
<hlq>.TR.LOAD	for load modules
<hlq>.TR.PDSE	for program objects
<hlq>.TR.DEFSF	for side files for DLLs

Note: There is a single large lab at the end of the lecture sections, with a variety of opportunities for experiment. The plan is to lecture on all the languages of interest and then have lab for the remainder of the class time.

Section Preview

- **DLLs - Dynamic Link Libraries**
  - **DLLs**
  - **Creating DLLs**
  - **Using DLLs**
  - **DLLs - Binder**
  - **The INCLUDE statement**
  - **DLL Applications - Finding the DLLs**
  - **DLLs - Vocabulary**

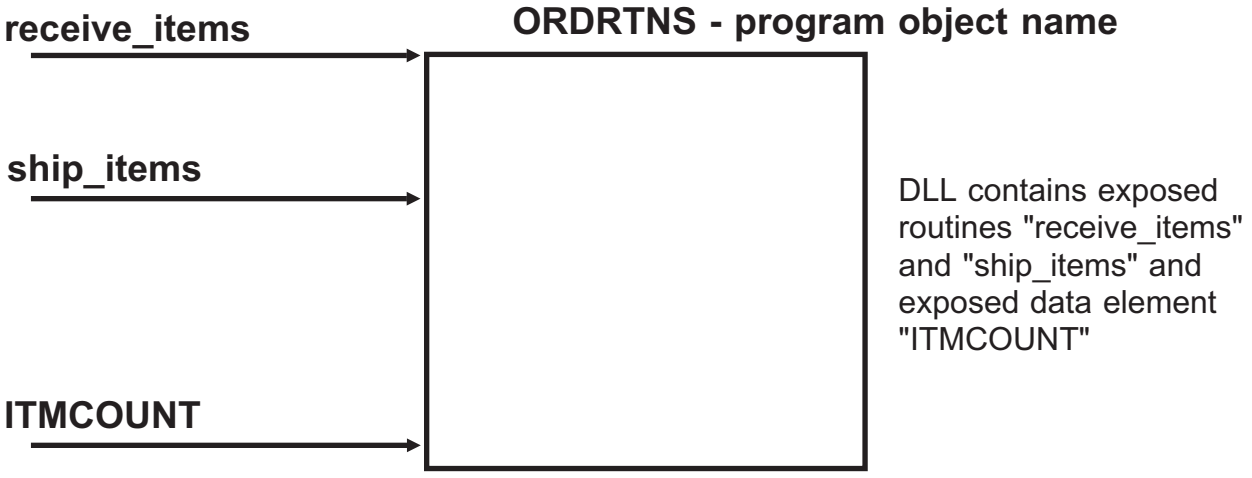
DLLs - Dynamic Link Libraries

- **The name and concepts of DLL come from the UNIX world, where they needed to accomplish what z/OS does with dynamic calls and external data - but with some additional capabilities**

- **A DLL is a program object with one or more programs whose entry points are exposed - available for being invoked dynamically**

  - **In addition, a DLL can have data items (variables) exposed, much like shared EXTERNAL data we discussed in the prerequisite course, but DLL variables can be shared across module and language boundaries**

- **Conceptually, you might envision a DLL to look something like this:**



- **z/OS has support for creating and using DLLs with the various languages we're discussing, so you could develop a library of DLLs and then new applications can access the routines / variables as needed**

DLLs, continued

- **Consider DLLs to be a generalization of, and extension to, the concepts of dynamic calls and shared external data, including...**
  - **Support for long names, and case-sensitive names**
  - **Sharing across load module boundaries - DLLs are shared at the enclave level: all threads in an enclave have access to any DLL in the enclave**
    - ✗ Therefore DLLs must be reentrant, which implies any statically linked subroutines included in the DLL must also be reentrant in order for the whole program object to be reentrant
  - **Sharing across language boundaries**
    - ✗ Currently, z/OS provides DLL support in COBOL, C, C++, PL/I, Assembler, and Java
    - ✗ All these languages support exporting routines; however, not all support exporting variables
  - **A single load brings in many routines and variables at once**

- **DLL support is provided for batch, TSO, CICS, IMS and z/OS UNIX**

Creating DLLs

❒ **To create a DLL, compile each of the routines that will belong to the DLL, specifying which entry points and external data should be exposed (also called exported)**

- ♦ **This creates object code, of course**
- ♦ **And it provides information for building function descriptors and variable descriptors at run time**

❒ **Although there is a compiler option like 'DLL' for C and COBOL, you don't need to specify it for the lowest level routines - routines that don't call any further DLL routines**

- ♦ **The DLL compiler option tells the compiler to make all dynamic calls DLL-type linkages**
  - ✗ That is, to access dynamically called programs through function descriptors instead of standard linkages
  - ✗ This implies that dynamic calls to routines that don't export their entry points (that is, to non-DLL routines) may have problems: you should not mix the two styles in a single application
- ♦ **For languages that support it, the DLL compiler option also tells the compiler to access external data through variable descriptors instead of traditional mechanisms**

Creating DLLs, 2

❒ **Pass the object code of each of the modules that will constitute your DLL to the program binder with appropriate parameters**

- **This creates a load module or program object and a defintion side file (also called a definition side deck) which you save until later**

- **If you use the prelinker and program binder, your output can go to a PDS; if you just use the program binder, your output can go to a PDSE or HFS file**

  ✗ From DFSMS version 1.4 on, you don't need the prelinker, and so we will just discuss the binder approach here

❒ **Next, combine all the program objects and definition side files into a single program object: your DLL**

- **Note that a DLL may have just a single module in it**

- **Note also that an application may use multiple DLLs, and that a DLL program can also use another DLL program**

- **Finally, note that all components of a DLL application must have the same AMODE, since no AMODE switching is done across DLL linkages**

Using DLLs

- A program that uses a DLL is called a DLL application
  - This can be a main program or a subroutine
- To create a DLL application program, compile with at least the DLL option (and the RENT option, since DLL linkages must be reentrant)
- Code invocations of DLL functions as standard dynamic invocations (that is, dynamic CALLs or function references)
  - The DLL will be automatically loaded and its functions are accessible as if for dynamic calls; this is called auto-load or trigger-load-on-reference for variables and trigger-load-on-call for functions
- Alternatively, you can invoke special services to explicitly load the DLL and then to dynamically locate the functions and variables your application program is interested in
  - In some cases this is required: for example, to access a DLL exported variable from a non-DLL COBOL program requires calling these functions to load the module and then to obtain the address of the variable

Using DLLs, continued

- **DLL variables should only be initialized in one place**
  - **Although any module in the enclave can change it, you need to be careful in multi-threading environments that you don't lose any updates**

- **It turns out, one of the tricky things about DLLs is to make sure you use them dynamically, not statically**
  - **That, is, you may access DLL functions and variables statically but the objective is usually to have dynamic invocation**
  - **But sometimes the Automatic Call feature of the Binder includes the DLL code in the calling module / program object without your being aware**
    - ✗ Unless you take care to look at the Binder map

- **We'll point out situations where you might have potential problems as we go along**

DLLs - Binder

For each program that is to be combined into a DLL...

- ❒ **Link with options RENT and DYNAM(DLL), and maybe also UPCASE(YES)**
  - ♦ **Also supply a DD statement named SYSDEFSD that points to a place for resulting IMPORT statements to be placed (this is what is in a definition side file: an IMPORT statement for each program and variable name being exposed)**
    - ✗ Fixed block 80 byte records; block size up to 32760
    - ✗ A sequential file, member of a PDS or PDSE, or an HFS file
  - ♦ **If the current module calls another DLL module, the SYSIN DD statement at bind time should contain the IMPORT statements created when that DLL module was linked**
    - ✗ You can point to the file or simply code your own
- ❒ **Note that there is a related binder option: CASE({UPPER|MIXED})**
  - ♦ **CASE(MIXED) should be used if any exported symbols include lower case letters**
  - ♦ **COMPAT(PM3) is the lowest level of binder compatibility to request**
- ❒ **Of course, your installation may have established all or some of these as defaults when the binder was installed**

DLLs - Binder, continued

- **After automatic call processing resolves any external references it can, all DLL-type references that have not been resolved are compared against names available in IMPORT statements**
  - For each match, the reference is considered resolved, with the tacit understanding that real resolution will occur dynamically at run-time
  - If there are still some unresolved references, if UPCASE(YES) is in effect, one extra pass is made over DLL names that are longer than 8 characters: replace '_'s with '@'s, fold to upper case, replace leading 'IBM' with 'IB$', 'CEE' with 'CE$', 'PLI' with 'PL$', and truncate to 8 characters and then see if match can be made

DLLs - Binder, continued

- **Each time you bind with option DLL, the binder puts out a definition side file, to the SYSDEFSD DD statement**
  - **If you specify a sequential data set name, or an HFS name, that is the name used for the definition side file**
  - **If you specify** *PDS_name*(*name*) **or** *PDSE_name*(*name*)**, the member** *name* **is used**
  - **If you specify** *PDS_name* **or** *PDSE_name***, without a member name, the member name will be name specified on the binder NAME control statement**
- **When you bind with option DYNAM(DLL), the binder builds an import / export table as part of the program object that identifies where imported functions and variables are to be found and what functions and variables are to be exported from this program object**
  - **On the binder listing this shows up as class B_IMPEXP**
- **Note that program objects are not externalized by IBM: that is, the internal format is not publicly documented**
  - **Rather, there are callable services available to access and change the attributes or properties of a program object**

DLLs - Binder, continued

❒ **You can code your own IMPORT statements in the binder control statement stream, if you want**

Syntax

IMPORT {CODE|DATA|CODE64|DATA64} *dll_name*,*import_name*

Where

- *dll_name* **is the name of the DLL containing the function or variable being exposed**
  - ✗ If this is a member of a PDS or PDSE, max 8 characters
  - ✗ If it is an alias name in a PDSE, the limit is 1024 characters
  - ✗ If this is an HFS file, max of 255 characters
- *import_name* **is the name of the function or variable that is exposed in the DLL**
  - ✗ Maximum of 32767 characters, case sensitive
- **You can use commas or spaces (or both) between the operands**
- **Alternatively, in the binder SYSLIN or SYSIN file you can have an INCLUDE statement that points to a member in a PDS or PDSE, that contains the IMPORT statements you want**
  - ✗ **This can also point to an HFS file name**
- **For 64-bit applications, you can specify CODE64 or DATA64**

The INCLUDE Statement

❒ **Additional features have been added to the INCLUDE statement over the years, to support not only DLLs, but also z/OS UNIX**

The full syntax of this statement now is:

INCLUDE [{-ATTR|-NOATTR}[,]{-IMPORTS|-NOIMPORTS}][,]
[{-ALIASES|-NOALIASES}][,]
{*ddname***[(***member_name***,...)][,...] | *pathname***[,...]}**

Where

- **-ATTR | -NOATTR - indicate if module attributes should be copied from the included module and applied to the module being built; default is -NOATTR**
- **-IMPORTS | -NOIMPORTS - indicate if IMPORT info should be copied from the included module and applied to the module being built; default is -NOIMPORTS; Note: starting with z/OS 1.6, the default is -IMPORTS**
- **-ALIASES | -NOALIASES - indicate if module aliases should be copied from the included module and applied to the module being built; default is -NOLIASES**
- **After any of the above options, specify one of:**
  - ✗ A ddname or comma-separated list of ddnames
  - ✗ A ddname with one or more membernames in parentheses
  - ✗ A pathname or comma-separated list of pathnames

DLL Applications - Finding the DLLs

- **Assuming you're trying to access DLL functions and variables dynamically, your DLL app will not have any references to the functions and variables you will be using in it's ESD**
  - **Therefore, the Binder won't try to include the code in with the app's code**
- **But - you do need to tell the system how to find what you're looking for at run time**
- **There are two alternatives**
  - **If your app accesses DLL functions and variables using helping services (*dllload*, *dllqueryfn*, *dllqueryvar*), these services tell the system which load module (program object) to load and which function or variable to access**
    - ✗ All will be well because you make it happen; you do not need IMPORT binder statements in this case
    - ✗ Note that as of z/OS 1.6, there are alternative DLL services: dlopen, dlsym, dlclose, and dlerror; these are part of the updated Single UNIX Speciification standard
    - ✗ Although the older services are now deprecated, existing code may use them; we will examine both sets of services
  - **If you use language-specific dynamic access techniques, then at Bind time you will need to include the necessary IMPORT statements**
    - ✗ This information is stored in your program object, but the actual referenced code or data are not stored in your program

DLLs - Vocabulary

❒ **It is helpful to pay attention to the language here**

- **A DLL is a program object (could be a load module in some cases, but most commonly a program object) that is composed of one or more functions and variables**
  - ✗ Typically, many source modules have been compiled and bound to build a DLL
  - ✗ Conceptually, a collection of subroutines
  - ✗ Typically, each subroutine is first compiled and bound separately, and then a final bind step builds the ultimate DLL
- **A DLL application is a program that uses a DLL**
  - ✗ Invokes the functions available or accesses the variables
- **Again note that a DLL may use another DLL**
  - ✗ And in this situation, a program object may be both a DLL and a DLL application