#SHAREorg

SHARE

IEWBIND and IEWBFDAT – Learning to Use the Binder APIs Hands-on Lab

Barry_Lichtenstein@us.ibm.com

February 2013

Session# 12929

SHARE in San Francisco 2013

Introduction

- This lab will provide the opportunity to work with the binder APIs in High Level Assembler and/or C
  - A working knowledge of coding in Assembler and/or C is required
- There are exercises demonstrating features of
  - The regular binder APIs (IEWBIND)
  - The fast data access APIs (IEWBFDAT)
  - The C/C++ APIs (IEWBNDD/IEWBNDDX)

Introduction …

- The exercises can be completed working in either batch, TSO or UNIX
  - Again a working knowledge of the chosen environment is required (such as the ability to allocate datasets, write JCL to invoke procs and program)
  - Some sample commands are provided along the way
- The handout is just a sort of reference summary for the various APIs and buffer formats we'll be looking at, taken from the Program Mangement: Advanced Facilities book, but organized differently. The handout is also available in the SHARE proceedings for this session under 12929 RefSum.pdf
- Pick and choose what exercises to do, you will not probably not have time to complete everything
- Instructor(s) are available to help with any questions!

Setup

- Each workstation is tagged with a userid##
  - SHARA01 – SHARA16
  - passwords match the userids
- Some content will be available in
  - Datasets: **'SHARE.BINDLAB.**'**
  - UNIX directory: **/sharelab/binderLab**

Setup …

- We'll use a couple of datasets under each userid
  - userid##.BINDLAB.ASM
  - userid##.BINDLAB.C
  - userid##.BINDLAB.LOADLIB
  - userid##.BINDLAB.PDSELIB
- Run 'SHARE.BINDLAB.JCL(ALLOCS)' to allocate them
  - If they already exist you can run 'SHARE.BINDLAB.JCL(CLEANS)'
    - but you'll lose all the existing content!!!
- Note that DSNTYPE=LIBRARY creates PDSEs for Program Objects while DSNTYPE=PDS is for Load Modules. Both must be RECFM=U and LRECL=0.

Setup …

- Binder ship 4 samples which we will use. They are all printed in the Program Management: Advanced Facilities as well:
  - IEWAPBND
    - BAGETE – a sample of using the regular binder APIs in High Level Assembler
  - IEWAPCCC
    - A sample of using the C/C++ APIs; test_binder_api() and test_fdata_api() show examples of both regular and fast data access in a C program
  - IEWAPFDA
    - A sample of using the fast data access APIs in High Level Assembler
  - IEWAPCOM
    - A sample of JCL which will compile or assemble, bind and run each of the above samples

Setup …

- Several other C source programs (earlier version previously discussed in prior SHARE presentations) are made available in the dataset **'SHARE.BINDLAB.C'** and in UNIX in **/sharelab/binderLab**. These are also available in the SHARE proceedings for this session as 12929- Additional Samples.zip.pdf – it's really a zip file of the program source, just rename it to get rid of the .pdf suffix when you download it!
  - dumpClassText (C) *– compare to 'SYS1.SAMPLIB(IEWAPCCC)'*
    - Single-source program which can be compiled for either regular or fast-data APIs
    - Uses GETN API to retrieve all binder class names and display information about them
    - Shows how to deals with new information available only in newer buffer formats
    - Optionally uses GETD API to retrieve and write out the text of the entire class (all elements)
  - getE (C) *– compare to 'SYS1.SAMPLIB(IEWAPBND)'*
    - Single-source program which can be compiled for either regular or fast-data APIs
    - Uses GETE API to retrieve all ESDs
      - *You can optionally pass any of the "filters" for which ESDs you want*
    - It formats almost all the contents of the ESD buffers
      - *It handles programs of any size (unlimited number of ESDs)*
  - idModSect (C)
    - Adds an IDRU to the "module section" – binder section x'00000001' (can't be done with IDENTIFY!)
  - addAlias (C)
    - Adds an ALIAS to an existing MVS program
    - Uses INTENT=ACCESS (this can't be done any other way)

Setup …

- We will use several pieces of JCL from ‘SHARE.BINDLAB.JCL’
- The do not use the userid## for the prefix, so…
- The first time you look in SDSF for output, do these commands:
  - SET DISPLAY
  - PREFIX *
- Now you will see your job output in SDSF ST

Exercise #1a
BAGETE (IEWAPBND)

- This exercise will familiarize you with the regular APIs. We'll learn about the ESD buffer and the GETN, GETE and GETD APIs.
- This program uses GETN to get all the section names and then gets all the ESD records for each section.
- It writes each ESD record to a single record in file MYDDN.
  - It's hex but you do see the ESD types in character format.
- It writes ALL binder messages to SYSPRINT
- First copy 'SYS1.SAMPLIB(IEWAPBND)' to your own dataset or UNIX
  - If a dataset, you can use 'SHARE.BINDLAB.ASM(BAGETE)
  - If UNIX, you can use /shareuser/userid##

Exercise #1a
BAGETE (IEWAPBND)

- Make sure you can assemble, bind and run it as-is
  - This program is not itself exploiting any program object features so can be created as a load module
  - To “make” (assemble and bind) the executable into ‘userid##.BINDLAB.LOADLIB(BAGETE)’
    - In UNIX, you can use /sharelab/binderLab/make_baGetE
    - In JCL, you can use ‘SHARE.BINDLAB.JCL(MBAGETE)’
  - To run
    - In JCL use ‘SHARE.BINDLAB.JCL(RBAGETE)
    - Examine the SDSF output, MYDDN and SYSPRINT

Exercise #1b
BAGETE (IEWAPBND)

- Now we'll see how the binder APIs use variable format string arguments
  - Change and then re-make the BAGETE program so that
    - Instead of using LPALIB, it uses your own library (call it MYLIB)
    - Instead of using the member name IFG0198N, use the member MYABC
  - Run 'SHARE.BINDLAB.JCL(MMABC)' to "make" the program 'userid##.BINDLAB.PDSELIB(MYABC)'
    - This program exploits program object features so must be stored into either a PDSE or UNIX filesystem
    - The source and object files are in /sharelab/binderLab, take a look at them:
      - *a.c, b.c, c.c, myabc.c*
  - Now run BAGETE… what happens? …

Exercise #1b … BAGETE (IEWAPBND)

- … In the SDSF output file MYDDN you'll see messages like this:

```
IEW2370W 2007 NO DATA EXISTS FOR CLASS = B_ESD AND SECTION = MYABC#T.
IEW2370W 2007 NO DATA EXISTS FOR CLASS = B_ESD AND SECTION = C#T.
IEW2370W 2007 NO DATA EXISTS FOR CLASS = B_ESD AND SECTION = B#T.
IEW2370W 2007 NO DATA EXISTS FOR CLASS = B_ESD AND SECTION = A#T.
IEW2370W 2007 NO DATA EXISTS FOR CLASS = B_ESD AND SECTION = B#C.
IEW2370W 2007 NO DATA EXISTS FOR CLASS = B_ESD AND SECTION = C#C.
IEW2370W 2007 NO DATA EXISTS FOR CLASS = B_ESD AND SECTION = A#C.
```

- Notice what's wrong??? The problem happens because C is a case-sensitive langauge
  - The binder is somehow using uppercase names for the section names
    - This is because the binder is uppercasing the section names from the GETN calls it subsequently uses on the GETD calls
  - How to fix this?
    - Change the BAGETE program to pass the CASE=MIXED parameter to the SETOption call
    - NOTE: Regular binder APIs have lots of options but Fast data access APIs have no options !!!

Exercise #1c
BAGETE (IEWAPBND)

- The very beginning prolog of this program claims that it uses GETE to get the ESD records. But in fact, you'll see that it uses GETD.
  - We will now change the program to use GETE to help understand how they are different and get a little insight into that 2D program object model
  - So go ahead and change the GETD to a GETE and rebuild and rerun… what happens? You get this error:

```
IEW2392E 214E CLASS B_ESD SPECIFIED ON GETE REQUEST IS NOT A TEXT CLASS.
```

  - What does *that* mean?!

Exercise #1c … BAGETE (IEWAPBND)

- GETE is a specific call to get ESD records so it is *always* getting them from the B_ESD class. The class specification on GETE identifies which classes to select for the given section.
- GETD is a generic call to GetData of whatever class is specified. So the existing GETD call is using the B_ESD buffer because it wants ESDs.
- So now change the program to use B_TEXT or C_CODE for the CLASS= and see how that works!
- Try one more thing. Add the RECTYPES parameter to the GETE call. Since it's exclusively for ESD records, something that GETE allows you to do that GETD does not, is to limit the types of ESD records. Try getting just (LD) types and see how that looks.

Exercise #1c … BAGETE (IEWAPBND)

- Notice you'll see some messages:

```
IEW2348W 210F GETE FOUND NO DATA MEETING SUPPLIED SELECTION CRITERIA.
```

- This is because some of the sections do not have any of the record type you requested for the specified class.

Exercise #1d
BAGETE (IEWAPBND)

- Look at the mapping of the ESD record buffer IEWBESD
- Remember that the same buffers are used for all “flavors” of the APIs
  - Though C manages the buffers for you, and does not give you back addressability to the header, but instead directly to the entries.
- Remember the layout of the buffer (also in the handout)? Records at the top, strings at the bottom… So how do we find the *names* of the symbols? Take a look at IEWBESD and find where they are. Notice they are sort of like variable length strings, but not quite – because they have a length but then a pointer to the name.

Exercise #1d … BAGETE (IEWAPBND)

- Extra Credit!
  - Change BAGETE to print the symbol name preceding each ESD record.
  - For a guaranted A+, also print the Target Section and the Resident Class. Make sure to account for the possibility that they may not exist!
  - What are these things? Comments in the IEWBESD help to explain it.
    - What is the Target Section?
      - *These are for ER (external reference) ESDs. The target section the section that contains the resolved external reference (where it was found).*
    - What is the Resident Class?
      - *For ERs this is the rest of the information about where it was found.*
      - *For Label Definitions (LD) and Part Definition (PD) ESDs, those are really offsets within a binder class, so the resident class is the class in which it's defined. There is no target section, because the containing section is the SD that preceded this LD or PD.*

Exercises 2
C/C++ APIs

- Using the C/C++ can greatly simplify coding binder API calls
  - Binder implements them in an LE Dynamic Link Library so your application will not have to statically link in all the API interface code
  - Binder provides both a NOXPLINK and XPLINK DLL for better performance
  - Binder DLL manages
    - Loading and Deleting IEWBIND
    - Allocating, initializing and Freeing all the buffers
    - Buffer mappings are provided in **/usr/include/__iew_api.h**
      - *Buffer headers are not defined, only the entries*
    - Simplified approach to API & buffer versions based on z/OS release

Exercise 2a
C/C++ APIs

- The shipped sample ‘SYS1.SAMPLIB(IEWAPCCC)’ shows the use of GETN and GETC in the regular APIs and correspondingly GN and GC in the fast data access APIs.
  - Take a look at the Name List buffer IEWBBNL and the CUI buffer IEWBCUI to see the information it contains
  - Let’s make IEWAPCCC and run it

Exercise 2a … C/C++ APIs

- **In UNIX:**
  - Source a simple script to set up c89
    - This only needs to be done once per login, or each time you want to change the setup
    - Make sure you type ". " in front of the command!

```
. /sharelab/binderLab/setup              # for NOXPLINK with the UNIX DLL
. /sharelab/binderLab/setup - X # for XPLINK with the UNIX DLL
. /sharelab/binderLab/setup D   # for NOXPLINK with the dataset DLL
. /sharelab/binderLab/setup D X          # for XPLINK with the dataset DLL
```

  - Copy the source from 'SYS1.SAMPLIB(IEWAPCCC)' to UNIX

```
cp "//'sys1.samplib(iewapccc)'" /shareuser/userid##/iewapccc.c
```

  - Make it

```
/sharelab/binderLab/make_apCCC
```

  - Run it

```
iewapccc /bin/ld
```

Exercise 2a … C/C++ APIs

- **In batch:**
  - Copy the source from ‘SYS1.SAMPLIB(IEWAPCCC)’ to a dataset
    - Use 3.3 to copy from *’sys1.samplib(iewapccc)’*
    - To *’userid##.BINDLAB.C(iewapccc)’*
  - Make it
    - Use ’SHARE.BINDLAB.JCL(MAPCCC)’
  - Run it
    - Use ’SHARE.BINDLAB.JCL(RAPCCC)’

Exercise 2a … C/C++ APIs

- After you run IEWAPCCC, look at the SYSPRINT output:
  - SYSPRINT in SDSF
  - The UNIX file /shareuser/userid##/temp
- The binder writes all it's error messages to SYSPRINT, but sometimes you want to produce error messages only and suppress the SYSPRINT. This is what binder does with SYSTERM.
  - NOTE: This is only for the regular binder API
- What's missing in IEWAPCC is that it does not write the SYSTERM file. It turns on the TERM=Y option but never creates the file. In batch it's always available, but with the APIs you need to explicitly tell the binder.

Exercise 2a … C/C++ APIs

- Fix it!
  - As an exercise in using the binder file lists, modify IEWAPCCC to create SYSTERM – both for a batch invocation and UNIX invocation
  - Look for where the SYSPRINT files list entry is and add a second entry for SYSTERM
  - Remember to update the files list count on the __iew_create_list() call!
  - Make it
  - Run it
  - Check your output.
    - Is SYSTERM or some new UNIX file (whatever you've named it) there now?

Exercise #2b
C/C++ APIs

- Two of the examples (previously described) in **/sharelab/binderLab** are designed to be compiled to use *either* the regular binder APIs *or* the fast data access APIs
  - getE.c
  - dumpClassText.c
- Fast data access is enabled by turning on the C feature test macro **FD**
- While there are many differences, this demonstrates the parallels between the capabilities of the regular APIs and fast data access APIs
  - Remember some key points about fast data access:
    - *Only for Program Objects*
    - *Read-only, you cannot rebind or save the program*

Exercise #2b
C/C++ APIs

- **In UNIX:**
  - Source the setup shell script as in the previous exercise

```
. /sharelab/binderLab/setup              # for NOXPLINK with the UNIX DLL
. /sharelab/binderLab/setup - X          # for XPLINK with the UNIX DLL
. /sharelab/binderLab/setup D            # for NOXPLINK with the dataset DLL
. /sharelab/binderLab/setup D X          # for XPLINK with the dataset DLL
```

  - Copy the source

```
cp /sharelab/binderLab/getE.c /shareuser/userid##
```

Exercise #2b
C/C++ APIs …

- **In UNIX …**
  - Make the program
    - `make –u getE` `# for the regular APIs`

      or

    - `CFLAGS=-DFD make –u getE` `# for the fast data access APIs`
  - Run it!

    ```
    getE /sharelab/binderLab/myabc
    ```

  - Run AMBLIST against the same program and compare it's output to that of getE:

    ```
    " echo LISTLOAD" | amblist /sharelab/binderLab/myabc > myabc.amblist
    ```

Exercise 2b … C/C++ APIs

- **In batch:**
  - Copy the source from ‘SHARE.BINDLAB.C(GETE)’ to a dataset
    - `Use 3.3 to copy from`
      `’share.bindlab.c(gete)’`
    - `To`
      `’userid##.BINDLAB.C(gete)’`
  - Make it
    - `Use ’SHARE.BINDLAB.JCL(MGETE)’`
  - Run it
    - `Use ’SHARE.BINDLAB.JCL(RGETE)’`
    - RGETE also ran AMBLIST against the same program, compare it’s output to that of getE

Exercise #2b
C/C++ APIs …

- Play with it
  - While you can pass binder parms using –P, it's not too useful since parms don't change the ESDs
    - Crucial in order to be able to rebind programs and get consistent results
  - Another of the filters provided with GETE (unlike GETD) is OFFSET. Offset will limit the output to symbols at or before the specified offset. Run GETE again and look at the output:
    - -O '0x100'

Exercise #2b
C/C++ APIs …

- Play with it …
  - Try passing different section names
    - Notice there is a section which has the name '$PRIV000003'
      - *This is the $SUMMARY section, it contains the definitions of the parts that is used at run-time*
      - *Try to pass this $PRIV000003 section name*
        - *You cannot! Why?*

Exercise #2b
C/C++ APIs …

- What's the problem?
  - The binder convention is to format so-called private symbol names as **$PRIVxxxxxx** (where the x's are hex digits)
  - These are called "private" because while they are external symbols they have no name, so there is no way to reference them.
  - That is, you could not code: DC V($PRIV000010)
    - z/OS v1r13 introduced the ability to refer to them from the binder
      - *For CHANGE and REPLACE control statements*

Exercise #2b
C/C++ APIs …

- What's the problem?
  - In fact the private names are "handed out" by the binder each time it's called, in order. So a symbol with one name one time may get a different one another.
  - In $PRIVxxxxxx the xxxxxx is the actual name. In the binder it is a full word binary number. Thus the name is 4 bytes long.
  - Binder APIs have no problem working with these names, because they are just variable length strings.

Exercise #2b
C/C++ APIs …

- Fix it!
  - However the binder C/C++ APIs have a hard time because C is a string-oriented language.
  - So for the most part it's very convenient to use C strings for symbol names
  - But now we have a problem. If you look at the getE.c code you'll see that it sees these names and formats them by prefixing the "$PRIV" part.

Exercise #2b
C/C++ APIs …

- Fix it! …
  - It's a little stickier to pass the name to the binder call however since the C/C++ APIs expect strings and if you pass a binary number, it might (*it will!*) contain a binary 0 and end the string.
  - So there is a utility function called **__iew_api_name_to_str()**
    - This is a "common" utility, for both regular and fast data access
      - *Because the buffer contents (and so this problem) is identifical for both*

Exercise #2b
C/C++ APIs …

- Fix it! …
  - Add some code to getE.c to allow a name like '$PRIVxxxxxx' to be passed in
  - Using **__iew_api_name_to_str()** have the correct symbol passed to the binder APIs
  - Make it!
  - Run it!
  - Check your results by passing to the getE call:
    - -S '$PRIV000003'

Exercise 3
FDEMO (IEWAPFDA)

- This shipped sample assembler program is a demo of all the different fast data access Request Code interface function calls
  - So it's a coding example of every call
  - As well as IEWBUFF macro calls for corresponding API buffers
  - And it takes an input file of "commands" so that you can build a command file (quite analagous to the binder SYSLIN)

Exercise #3
FDEMO (IEWAPFDA) …

- **In UNIX …**
  - Copy the source from 'SYS1.SAMPLIB(IEWAPFDA)' to UNIX

```
cp "//'sys1.samplib(iewapfda)'" /shareuser/userid##/iewapfda.s
```

  - Make the program

```
/sharelab/binderLab/make_apFDA
```

  - Run it!
    - This demonstrates the ability to use dynamic allocation in UNIX with BPXWDYN and also program redirection using /dev/fdNN character special files
    - Note that it relies on having _BPX_SHAREAS=YES exported

```
/sharelab/binderLab/alloc_apFDA /sharelab/binderLab

iewapfda
```

    - Because of the redirection this becomes an interactive invocation. Use the fast data access request code functions, or XX to terminate:

```
SJ myabc
GN CLASSES
GD
XX
```

Exercise #3
FDEMO (IEWAPFDA) …

- **In batch:**
  - Copy the source from ‘SYS1.SAMPLIB(IEWAPFDA)’ to a dataset
    - `Use 3.3 to copy from`
      `’sys1.samplib(iewapfda)’`
    - `To`
      `’userid##.BINDLAB.ASM(iewapfda)’`
  - Make it
    - `Use ’SHARE.BINDLAB.JCL(MAPFDA)’`
  - Run it
    - `Use ’SHARE.BINDLAB.JCL(RAPFDA)’`

Exercise #3
FDEMO (IEWAPFDA) …

- Note that the GD implementation in this sample has hard-coded the use of B_TEXT
- Refering to the slide depicting the 2 dimensional model of data in a program object, note that this only is a single column of data within a program. A real program might have several different text classes of data (for example many LE conforming programs will also have C_CODE).
- Modify the IEWBFDAT sample to instead use C_CODE
  - If you're up to it, change the program so it can optionally be passed the class name on the GD "command"
    - Or better yet, to accept multiple class names
- Remake and rerun and compare the output

Exercise #3
FDEMO (IEWAPFDA) …

- Finally for extra credit
  - Change IEWBFDA to write out the first 32 bytes of data returned by each GD call (you can assume there's at least 32 for this exercise)
  - Refer to the IEWBTXT text buffer mapping (including in the hand-out)
    - Note that addressability will have already been provided by virtue of the use of the IEWBUFF macro calls
    - Note that there is an error in this documentation, the field identifid as TXT_ARRAY is actually TXT_ENTRY (this is the naming convention for the entries used by all the binder API buffers)