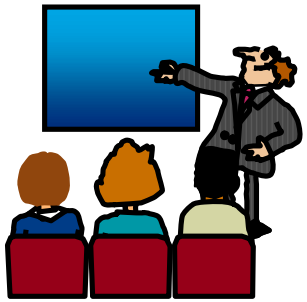
Using Dynamic Link Libraries (DLLs) with Language Environment

Session 8130 / 8286

Barry Lichtenstein
IBM Poughkeepsie
BarryL@us.ibm.com



SHARE in Anaheim, CA, Feb-Mar 2005

Agenda

- Terms
- DLL Overview
- How to Write DLL-enabled code
- How to Build DLLs and DLL Applications
- Executing with DLLs
- More than you ever wanted to know about DLL Linkage

Terms

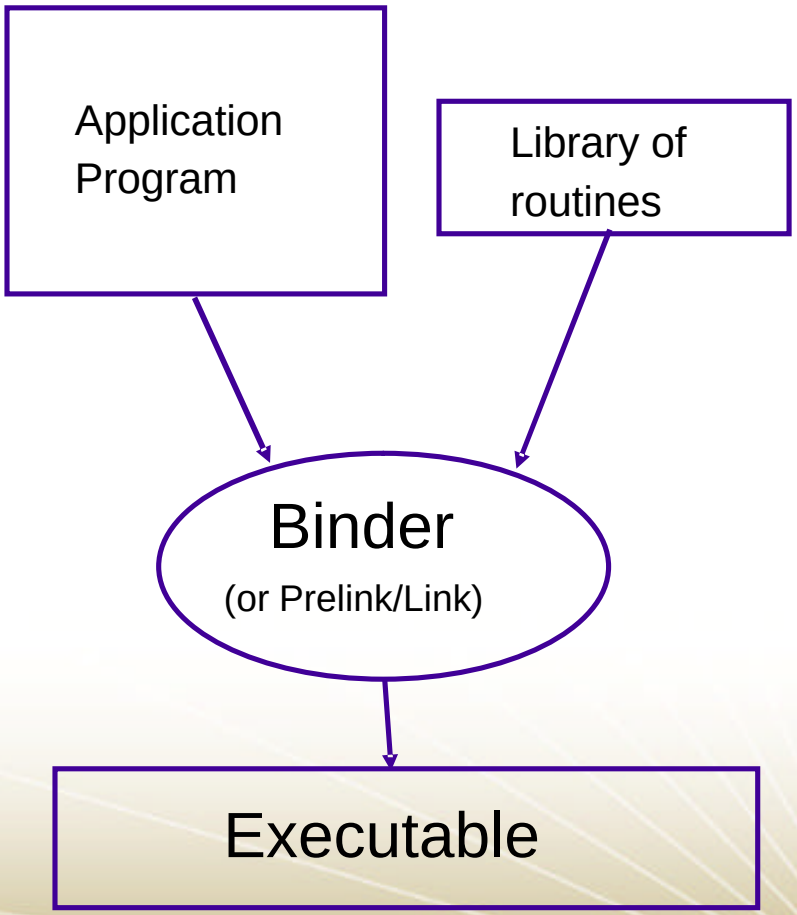
DLL-enabled code	C or COBOL code compiled with the DLL compiler option, or C++ or PL/I (VA or later) code, or assembler
non-DLL code	C or COBOL code compiled without the DLL compiler option
DLL	Collection of variables and/or functions, accessible to other applications
DLL Application	An executable that uses a DLL (can itself also be a DLL, matter of "perspective")
Export	Variable or function in a DLL made available to DLL applications
Import	Variable or function defined in a DLL and used by a DLL application
Explicit DLL load	Run time loading of DLL via explicit source level calls
Implicit DLL load	Load-on-call loading of DLL via implicit reference to imported var or function

Compiler Support for DLLs

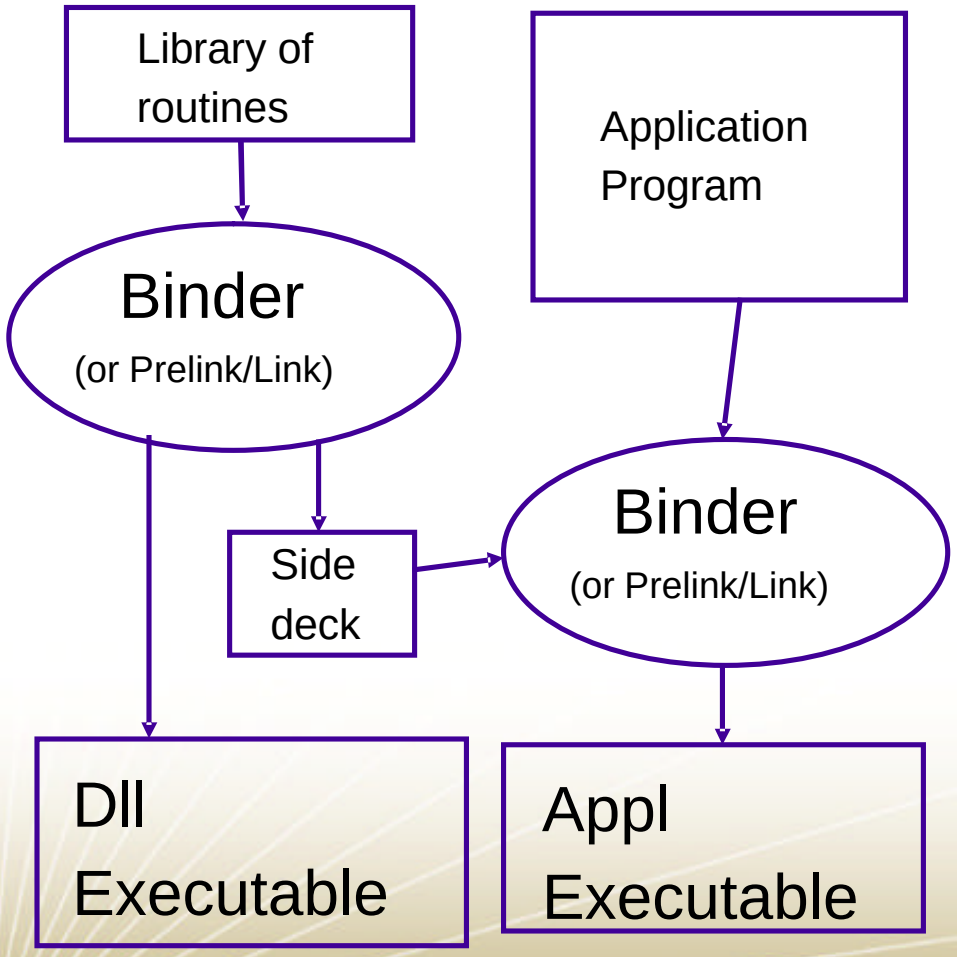
- z/OS and OS/390 C and C++
- Enterprise COBOL for z/OS and OS/390
- COBOL for OS/390 & VM
- Enterprise PL/I for z/OS and OS/390
- VisualAge PL/I for OS/390
- High Level Assembler (z/OS Release 5)

Static versus Dynamic Linking (or Binding)

Static Binding



Dynamic Binding



Building an Executable (DLL or DLL App)

An executable is either a Program Object or a Load Module

- Program object
  - Resides in a PDSE or the HFS
  - Can NOT be in a PDS
  - Created by the Binder
  - ZOSV1R6 (PM4SUB2) is the newest form
  - PDSEs can't be put in LPA, must be in Dynamic LPA
  - HFS files can be loaded as Shared Libraries (more later)
- Load Module
  - Resides in PDS
  - Created by either the Prelinker/Linkage Editor or Binder (PM2 or earlier)

A few words about Rentrancy...

- Reentrancy allows more than one user to share a single (read-only) copy of a load module.
  - Uses less storage (eg. LPA), less paging
  - Bind with REUS=RENT (or use the Prelinker)
- If Not reentrant, each application must load its own (writeable) copy.
- Two "types" of reentrancy:
  - Natural Reentrancy
    - Your code does not alter any of its static data.
  - Constructed Reentrancy
    - Compiler generates a separate "Writeable Static Area" (WSA)
    - specify RENT compiler option (C/COBOL/Enterprise PLI)
    - C++ implies constructed reentrancy
    - Assembler requires LIST and EXECUTE format expansions

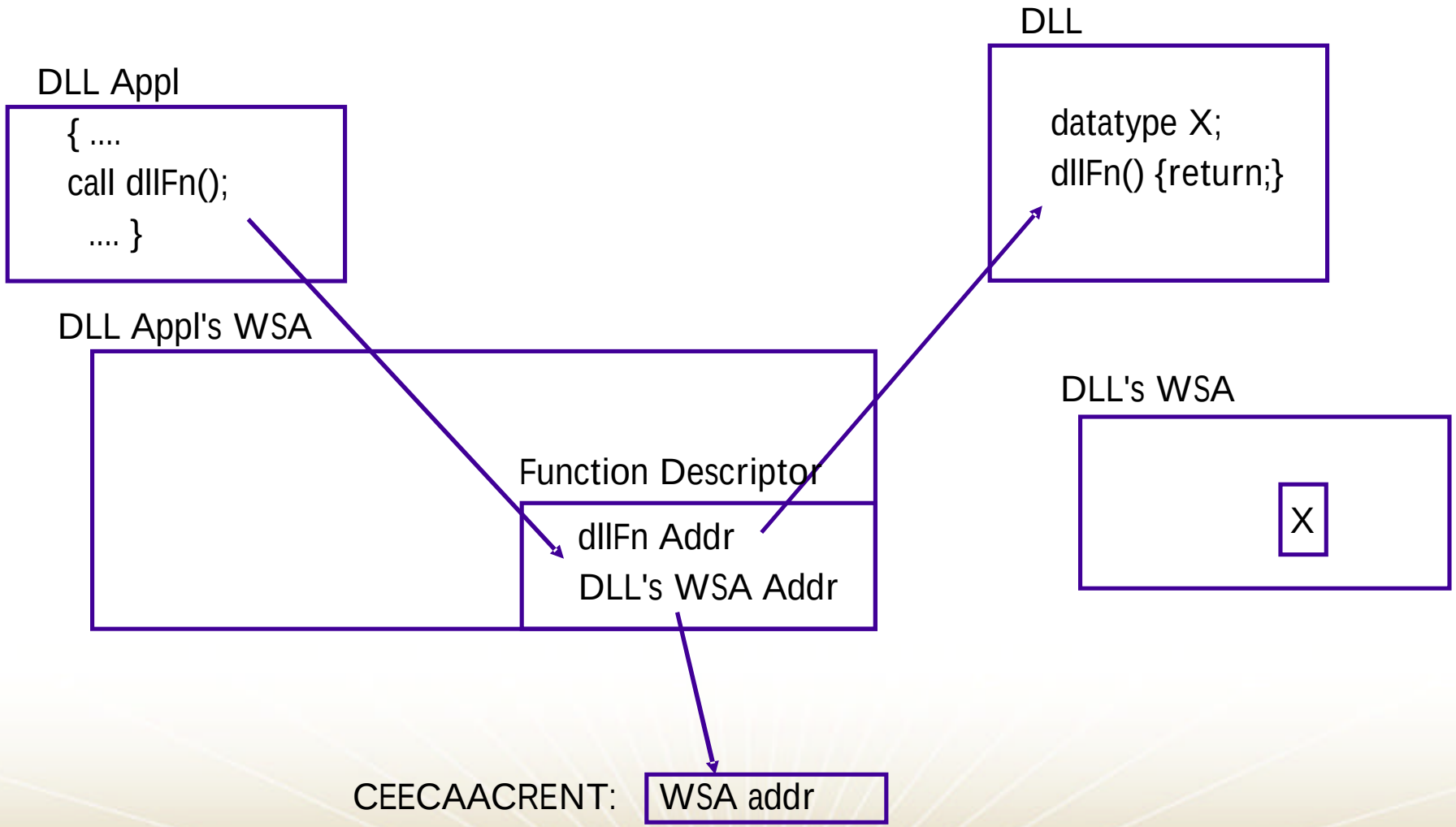
...because DLLs must be Reentrant

- Applications sharing DLLs in the same enclave expect to also share static storage (which includes exported data)
- Applications sharing DLLs expect to reduce storage overhead.

What's so special about DLL-enabled code?

- The currently-executing Program Object requires addressability to its own WSA
- CEECAACRENT is set to WSA address of function being called
  - Must be updated when a function in a DLL is called!! (although not usually restored when function returns)
- There are now *two* pieces of information required to call a function
  - Addresses of entry point and function's WSA
- This is achieved through use of a Function Descriptor

What's so special about DLL-enabled code?



High-level Structure of your Application

- Primarily a packaging issue
  - Determine how parts of application relate to each other or to other DLLs
  - Does it make sense to group certain functions and/or data together and separate?
- Advantages:
  - Easier Maintenance (no rebind of application if DLL changes)
  - Smooth Inter-Language Communication (ILC)
  - Better performance (reduced storage and paging)
- Disadvantages:
  - Slower performance (linkage overhead)
  - Compatibility Issues with non-DLL code

Writing Your C/C++ Code

- How to Identify Functions/Variables to be Exported
  - C
    - #pragma export selected functions and/or variables (define data items to be exported at global scope), or EXPORTALL compiler option
  - C++
    - _Export keyword to export selected functions and/or variables (define data items to be exported at global scope), or EXPORTALL compiler option

Writing Assembler DLL Code

- How to Identify Functions/Variables to be Exported (using the Language Environment Assembler macros)
  - High Level Assembler
    - Exported Entry Point

```
DLLFUNC  CEEENTRY MAIN=NO,PPA=DLLPPA,EXPORT=YES
DLLFUNC  ALIAS C'dllfunc'
```

    - Exported Data

```
         CEEPLDA DllVar,REG=9
         LA      R8,456
         ST      R8,0(R9)
   . . .
         CEEPDDA DllVar,SCOPE=EXPORT
         DC      A(123)
         CEEPDDA END
```

Writing Your COBOL/PLI DLL Code

- How to Identify Functions to be Exported
  - COBOL
    - EXPORTALL compiler option
    - Note: COBOL EXTERNAL data is outside the realm of DLLs, shared by all COBOL routines in run unit (as usual)
  - PL/I
    - Specify OPTIONS(FETCHABLE) on the PROCEDURE statement

    or
    - DLLINIT compiler option (applies OPTIONS(FETCHABLE) to all external procedures)

Writing Your DLL Code

- If there are functions (or variables) in the DLL that you do not want exported:
  - C/C++, HLASM, and PL/I allow you to export selectively,
  - Compile non-exported functions separately with NOEXPORTALL, or
  - Delete those entries from your side deck (be careful when editing the side decks in the HFS, they're FB 80 with no newline characters).

Writing Your C/C++ DLL Application Code

- Identify functions/variables to be Imported
  - C/C++: Prototype with external scope
    - extern int dllvar;
    - extern int dllfunc(int);

Writing Your Assembler DLL Application Code

- Identify functions to be Imported
  - HLASM:
    - Calling an Imported Entry Point Data with *no parameters*:

```
CEEPCALL Bif1,MF=(E,)
```

    - Calling an Imported Entry Point Data with *parameters*:

```
CEEPCALL Bif4,(PARM1,PARM2,PARM3,PARM4),VL,MF=(E,PARMS)
```

```
. . .
PARM1    DC    A(15)
PARM2    DC    A(33)
PARM3    DC    A(45)
PARM4    DC    A(57)
. . .
         CEEDSA
PARMS    CEEPCALL ,(0,0,0,0),MF=L
AUTOEND  DS    0D
AUTOSZ   EQU   AUTOEND-CEEDSA
```

Writing Your Assembler DLL Application Code (cont…)

- Identify variables to be Imported
  - HLASM:
    - Imported Data

```
* Getting the address of imported var Biv3, which looks like:
*      struct { int Biv3_part1 ; int Biv3_part2 ; } Biv3;
*
          CEEPLDA Biv3,REG=9
*
* Set value of imported variable Biv3_part1 to 123
*
          LA    R8,123
          ST    R8,0(,R9)
*
* Set value of imported variable Biv3_part2 to 456
*
          LA    R8,456
          ST    R8,4(,R9)
. . .
          CEEPDDA Biv3,SCOPE=IMPORT
```

Writing Your COBOL/PLI DLL Application Code

- Identify functions/variables to be Imported
  - COBOL: Use CALL literal or CALL identifier where the target program name is an exported DLL function
    - literal case gets resolved from side deck ("preferred")
    - identifier causes DLL load and DLL function query using value of identifier (ie. DLL name and DLL function name must be the same)
  - PL/I: Identify DLL routine with ENTRY declaration
    - Execution of a CALL statement or function reference ("preferred")
    - Appearance of an ENTRY constant in a FETCH statement
      - ENTRY name on FETCH (ie. DLL name) must match DLL PROC name

(Implicit) C Example

DLL Application (SIGGY)

```
extern int sigcaught;
int dllsig(int);

main() {
    int rc;
    ...
    rc = dllsig(SIGUSR1);
    ...
    raise(SIGUSR1);
    if (sigcaught) {
        printf("signal %d caught\n", sigcaught);
    }
    ...
}
```

DLL (SIGDLL)

```
int sigcaught = 0;
void dllcatch(int);

int dllsig(int sig) {
    struct sigaction sa;
    int rc;

    errno = 0;
    sa.sa_sigaction = dllcatch;
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);
    rc = sigaction(sig, &sa, NULL);
    return errno;
}

void dllcatch(int signum) {

    sigcaught = signumber;
}
```

COBOL Example

DLL Application (MyCobolAppl)

```
IDENTIFICATION DIVISION.
PROGRAM-ID. 'MyCobolAppl'.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
DATA DIVISION.
FILE SECTION.
WORKING-STORAGE SECTION.
01 TODAYS-DATE-YYYYMMDD  PIC 9(8).
PROCEDURE DIVISION.
    Display 'MyCobolAppl: Entered'
    MOVE FUNCTION
                 CURRENT-DATE(1:8) TO

        TODAYS-DATE-YYYYMMDD
    Call 'MyCobolDllFn' using
                 todays-date-yyyymmdd
    Display 'MyCobolAppl: Done'
    GOBACK
    .
```

DLL (MyCobolDllFn)

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 'MyCobolDllFn'.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
 DATA DIVISION.
 FILE SECTION.
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01 TODAYS-DATE-YYYYMMDD  PIC 9(8).
 PROCEDURE DIVISION using
                  todays-date-
yyyymmdd.
     Display 'MyCobolDllFn: Entered'
     Display 'MyCobolDllFn:Todays
                  date is '
                  todays-date-yyyymmdd
     Display 'MyCobolDllFn: Done'
     GOBACK
     .
```

PL/I Example

DLL Application (VAPLIDLA)

```
*PROCESS OFFSET SOURCE LIST;

  VAPLIDLA:PROC OPTIONS(MAIN);

    DCL (DLLFN1, DLLFN2) EXT ENTRY;

    DISPLAY('Start of VAPLIDLA.');

    CALL DLLFN1();
    CALL DLLFN2();

    DISPLAY('End of VAPLIDLA.');

  END;
```

DLL (VAPLIDLL)

```
*PROCESS OFFSET SOURCE LIST DLLINIT;

  DLLFN1:PROC;
    DISPLAY('DLLFN1: Entered.');
  END;

  DLLFN2:PROC;
    DISPLAY('DLLFN2: Entered.');
  END;
```

Assembler Example

DLL Application

```
DLLAPPL  CEEENTRY MAIN=YES,PPA=DLLPPA
* (skipping reg equates)
* Get addr of imported var Biv1
         CEEPLDA Biv1,REG=9
* Store 123 into Biv1
         LA    R8,123
         ST    R8,0(,R9)
* Call imported function Bif1
         CEEPCALL Bif1,MF=(E,)
         SR    R15,R15
RETURN   DS    0H
         CEETERM RC=(R15),MODIFIER=0
*
         CEEPDDA Biv1,SCOPE=IMPORT
DLLPPA   CEEPPA
         LTORG
         CEEDSA
         CEECAA
         END   DLLAPPL
```

DLL

```
DLLFUNC  CEEENTRY MAIN=NO,PPA=DLLPPA,
              EXPORT=YES
DLLFUNC  ALIAS C'Bif1'
* Get addr of (exported) var Biv1
         CEEPLDA Biv1,REG=9
* Double the value of Biv1
         L     R8,0(R9)
         SLA   R8,1(0)
         ST    R8,0(,R9)
*
         SR    R15,R15
RETURN   DS    0H
         CEETERM RC=(R15),MODIFIER=0
*
         CEEPDDA Biv1,SCOPE=EXPORT
         DC    A(1)
         CEEPDDA END
DLLPPA   CEEPPA
         CEEDSA
         END   DLLFUNC
```

Other Externals for DLL support

Explicit C run-time functions (in <dll.h>)

- Load a DLL: dllhandle* dllload(char *)
- Get a DLL function address: void (*dllqueryfn(dllhandle *, char *))();
- Get a DLL variable address: void* dllqueryvar(dllhandle *, char *)
- Free a DLL: int dllfree(dllhandle *);

Environment variable

Specify path name(s) to search for DLLs in the HFS:

LIBPATH <:>pathname<:pathname><:>

New Single UNIX ® Specification Externals for DLL support

Explicit C run-time functions (in <dlfcn.h>)

- Load a DLL: `void* dlopen(const char * file, int mode);`
  - file can be 0 for a global symbol object for symbol lookup
  - mode can force DLL preloading, and hide symbols from symbol lookup
- Get a DLL function or variable address: `void * dlsym(void *, const char *);`
- Free a DLL: `int dlclose(void *);`
- Get diagnostics if any of the above fail: `char * dlerror(void);`
- Introduced in z/OS R6. Previous functions are deprecated.
- Also see www.opengroup.org

Explicit C Example

DLL Application (SIGGY)

```
#include <dll.h>
main() {
    int rc;
    dllhandle *handle;
    typedef int (*fp)(int);
    fp fptr;
    int *vptr;

    handle = dllload("SIGDLL");
    ...
    fptr = (fp)dllqueryfn(handle,"dllsig");
    ...
    rc = fptr(SIGUSR1);
    ...
    raise(SIGUSR1);
    vptr = (int *)dllqueryvar(handle,"sigcaught");
    if (*vptr) {
        printf("signal %d caught\n", *vptr);
    }
    rc = dllfree(handle);
}
```

DLL (SIGDLL)

```
int sigcaught = 0;
void dllcatch(int);

int dllsig(int sig) {
    struct sigaction sa;
    int rc;

    errno = 0;
    sa.sa_sigaction = dllcatch;
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);
    rc = sigaction(sig, &sa, NULL);
    return errno;
}

void dllcatch(int signum) {

    sigcaught = signumber;
}
```

Explicit High Level Assembler Example

DLL Application (ASMDLLE) (1 of 5)

```
*
*    Symbolic Register Defs and Usage
R1        EQU   1   Parm list addr
R9        EQU   9   Work register
R14       EQU   14  Return point addr
R15       EQU   15  Entry point addr
*
ASMDLLE   CEEENTRY PPA=APPLPPA
*
*    Load a DLL load module
*
****     dlltoken = dllload(dllname)
*
          LA    R1,DLLNAME
          ST    R1,PARMNAME
          LA    R1,PARMNAME
          L     R15,DLLLOAD
          BASR  R14,R15
          LTR   R15,R15
          ST    R15,DLLTOKEN
          JZ    REPORT
```

DLL (CDLL12)

```
#include <stdio.h>
#pragma export(dllvar)
#pragma export(dllfunc)

int dllvar;

int dllfunc(int parm)
{
  printf("Inside dllfunc...\n");
  printf("  passed parameter: %d\n",parm);
  printf("  value of dllvar:  %d\n",dllvar);
  return 0;
}
```

Explicit High Level Assembler Example

DLL Application (ASMDLLE) (2 of 5)

```
*
*  Locate a DLL variable;
*  If successful, change value to 7
*
**** var_ptr = dllqueryvar
*                (dlltoken,var_name)
*
         LA    R1,VAR_NAME
         ST    R1,PARMNAME
         LA    R1,PARMLIST
         L     R15,DLLQUERYVAR
         BASR  R14,R15
         LTR   R15,R15
         ST    R15,VAR_PTR
         JZ    REPORT
         LA    R9,7
         ST    R9,0(R15)
```

DLL Application (ASMDLLE) (3 of 5)

```
*
*  Locate a DLL function;
*  If successful, call the function
*
**** fn_ptr = dllqueryfn(dlltoken,
*                        fn_name)
*
         LA    R1,FN_NAME
         ST    R1,PARMNAME
         LA    R1,PARMLIST
         L     R15,DLLQUERYFN
         BASR  R14,R15
         LTR   R15,R15
         ST    R15,FN_PTR
         JZ    REPORT
         CALL  (15),(5)
```

Explicit High Level Assembler Example

DLL Application (ASMDLLE) (4 of 5)

```
*
*    Free the DLL
*
****      dllfree(dlltoken)
*
          XR     R1,R1
          ST     R1,PARMNAME
          LA     R1,DLLTOKEN
          L      R15,DLLFREE
          BASR   R14,R15

          SR     R15,R15
RETURN    DS    0H
         CEETERM RC=(R15),MODIFIER=0
*
REPORT    DS    0H
*                   Report error here
          J      RETURN
*
*
```

DLL Application (ASMDLLE) (5 of 5)

```
VAR_PTR   DS    F          * address of DLL variable
FN_PTR    DS    F          * address of DLL function
* C RTL DLL functions (from dll.h)
          EXTRN    @@DLLOAD     * dllload()
DLLLOAD   DC     A(@@DLLOAD)
          EXTRN    @@DLLQRV     * dllqueryvar()
DLLQUERYVAR DC   A(@@DLLQRV)
          EXTRN    @@DLLQRF     * dllqueryfn()
DLLQUERYFN DC    A(@@DLLQRF)
          EXTRN    @@DLLFRE     * dllfree()
DLLFREE   DC     A(@@DLLFRE)
         CNOP 2,4
DLLNAME   DS  0CL7              * NULL-terminated
          DC  CL6'CDLL12'       * ... DLL name
          DC  X'00'
         CNOP 2,4
VAR_NAME  DS  0CL7              * NULL-terminated
          DC  CL6'dllvar'       * ... variable name
          DC  X'00'
         CNOP 2,4
FN_NAME   DS  0CL8              * NULL-terminated
          DC  CL7'dllfunc'      * ... function name
          DC  X'00'
          DS  0F
PARMLIST  DS  0CL8          * Parms to dll*() calls
DLLTOKEN  DC  A(0)
PARMNAME  DC  A(0)
APPLPPA   CEEPPA
          LTORG
          CEEDSA
          CEECAA
          END      ASMDLLE
```

Compiling Your DLL and DLL Application

- C
  - DLL and DLL Application:
    - Compile with options: DLL, RENT, LONGNAME, and possibly EXPORTALL for DLLs
- C++
  - No special options (always DLL-enabled code), except possibly EXPORTALL for DLLs

Compiling Your DLL and DLL Application

- COBOL
  - DLL:
    - Compile with options: DLL, RENT, EXPORTALL
  - DLL Application:
    - Compile with options: DLL, RENT, NOEXPORTALL
- PL/I
  - DLL:
    - Compile with options: DLLINIT, RENT (Enterprise PL/I only)
  - DLL Application:
    - RENT (Enterprise PL/I only)
- High Level Assembler
  - Assemble with GOFF option

Assembler DLL macro summary

Export/Call requires: • GOFF • >=PM3/4	non-XPLink	XPLink	AMODE 64
Function Prolog (EXPORT=)	CEEENTRY EDCPRLG	EDCXPRLG	CELQPRLG
Function Eplilog	CEETERM EDCEPLG	EDCXEPLG	CELQEPLG
DSA mapping	EDCDSAD	CEEDSA DSASZ includes arguments	CEEDSA DSASZ includes arguments
Call (I) Function (CBN, CBR only)	CEEPCALL	EDCXCALL	CELQCALL
Declare (I/E) Variable	CEEPDDA declare/DC to define WSA var, optional I/E	=	=
Refer to (I/E) Variable	CEEPLDA load reg w/@WSA var, I/(local)E	=	=

Definition Side Deck

- A side deck is a flat file created for a DLL to represent its exported variables and functions
- Contents of side deck are Binder IMPORT control statements
- Variable and function names are produced in alphabetic order
- Side deck(s) for DLLs used by an applicaion must be specified at link-edit time
  - Beginning z/OS R6 Binder will use remembered IMPORTs (available with PO4 and later formats)
- Side decks are not used for explicit loading
- Side decks may be edited if desired to remove unneeded external references
  - Be careful!! Side decks are FB 80, with NO newline chars!!

Sample Definition Side Deck

- From previous sample DLL:

```
IMPORT CODE, 'SIGDLL','dllcatch'
IMPORT CODE, 'SIGDLL','dllsig'
IMPORT DATA, 'SIGDLL','sigcaught'
```

Binding Your Code

- Bind your DLL with the DYNAM=DLL and REUS=RENT binder options
  - Writes the DLL definition side deck to SYSDEFSD DD
  - Creates the necessary internal structures to export
- Bind your DLL Application with the DYNAM=DLL and REUS=RENT binder options
  - INCLUDE the side deck(s) from the DLL in the set of object modules to bind (not resolved during autocall)
  - After final autocall, unresolved DLL references are picked up from the side deck(s) in order of appearance
- If an executable must reside in a PDS, you must first Prelink

Characteristics of DLLs

- Compilers, Binder (or Prelinker), and LE all cooperate to provide DLL support
- DLLs are shared at the LE enclave level
  - One copy of the DLL WSA exists!
- DLLs can be loaded multiple times, using a mixture of implicit and explicit loading
- DLLs are logically freed at LE enclave termination
- C++ Static Constructors are run when a DLL is first loaded, and Static Destructors are run when a DLL is physically deleted

When are DLLs loaded?

- A DLL Application that implicitly references an imported variable causes the containing DLL to be transparently preloaded during LE initialization
  - To avoid run-time check
- A DLL Application that implicitly references only imported functions will cause the DLL to be transparently loaded the first time one of those functions are called
- Explicit DLL loads occur at time of dllload() call

Where are DLLs loaded from?

- The DLL name is unambiguous...
  - ...if it starts with a ./ or contains a single / anywhere in the name
    - it's an HFS filename, and the load occurs from the HFS
  - ...if it starts with a //
    - it's an MVS member name, and the load occurs from MVS search order
- If the DLL name is ambiguous...
  - If POSIX(ON), search HFS first, then MVS
  - If POSIX(OFF), search MVS first, then HFS

Where are DLLs loaded from?

- From MVS data sets via ?LOAD
  - DLL name must not be longer than 8 characters, will be converted to uppercase
  - Search order is:
    - Job Pack Queue, STEPLIB / JOBLIB, LPA, Link List
- From HFS via BPX1LOD
  - DLL name is case sensitive
  - If envar LIBPATH is set to list of directories, then those directories are searched, otherwise current directory is searched
  - If name is found in HFS, check for external link or sticky bit first

LE DLL Load Mechanism

DLL Appl A

```
main() {
...
dllfunc1();
...
dllfunc2();
}
```

DLL Appl A WSA

A(CEETLOC)
dllfunc1_id
A(CEETLOC)
dllfunc2_id

Two DLL Application's (residing in separate program objects) are loading, in this case, into the same enclave.

Their WSA's (obtained when they were loaded) contain function descriptors recognized by the Binder as representing imported functions, and have been initialized accordingly.

DLL Appl B

```
applB() {
...
dllfunc2();
...
}
```

DLL Appl B WSA

A(CEETLOC)
dllfunc2_id

LE DLL Load Mechanism

DLL Appl A

```
main() {
...
dllfunc1();
...
dllfunc2();
}
```

DLL Appl A WSA

A(CEETLOC)
dllfunc1_id
A(CEETLOC)
dllfunc2_id

DLL Appl A calls dllfunc1, which resides in DLL X (resolved at Bind time via side deck).

DLL Appl B

```
applB() {
...
dllfunc2();
...
}
```

DLL Appl B WSA

A(CEETLOC)
dllfunc2_id

LE DLL Load Mechanism

DLL Appl A

```
main() {
...
dllfunc1();
...
dllfunc2();
}
```

What is LE doing?
Call LE "trigger DLL load on Call" routine.

DLL Appl B

```
applB() {
...
dllfunc2();
...
}
```

DLL Appl A WSA

A(CEETLOC)
dllfunc1_id
A(CEETLOC)
dllfunc2_id

DLL Appl B WSA

A(CEETLOC)
dllfunc2_id

LE DLL Load Mechanism

DLL Appl A

```
main() {
...
dllfunc1();
...
dllfunc2();
}
```

What is LE doing?

Determine if the DLL is already loaded in this enclave.

DLL Appl B

```
applB() {
...
dllfunc2();
...
}
```

DLL Appl A WSA

A(CEETLOC)
dllfunc1_id
A(CEETLOC)
dllfunc2_id

DLL X

```
dllfunc1() { ... }
dllfunc2() { ... }
```

DLL Appl B WSA

A(CEETLOC)
dllfunc2_id

LE DLL Load Mechanism

DLL Appl A

```
main() {
...
dllfunc1();
...
dllfunc2();
}
```

What is LE doing?

Has DLL been loaded into this enclave before?

No -- Obtain and Init WSA

DLL Appl B

```
applB() {
...
dllfunc2();
...
}
```

DLL Appl A WSA

A(CEETLOC)
dllfunc1_id
A(CEETLOC)
dllfunc2_id

DLL X

```
dllfunc1() { ... }
dllfunc2() { ... }
```

DLL X WSA

DLL Appl B WSA

A(CEETLOC)
dllfunc2_id

LE DLL Load Mechanism

DLL Appl A

```
main() {
...
dllfunc1();
...
dllfunc2();
}
```

What is LE doing?
Resolve all descriptors (and variable references) to DLL X (only) in DLL Appl A's WSA

DLL Appl B

```
applB() {
...
dllfunc2();
...
}
```

DLL Appl A WSA

A(dllfunc1)
A(DLL X WSA)
A(dllfunc2)
A(DLL X WSA)

DLL X

```
dllfunc1() { ... }
dllfunc2() { ... }
```

DLL X WSA

DLL Appl B WSA

A(CEETLOC)
dllfunc2_id

LE DLL Load Mechanism

DLL Appl A

```
main() {
...
dllfunc1();
...
dllfunc2();
}
```

What is LE doing?
Complete call to dllfunc1().

DLL Appl B

```
applB() {
...
dllfunc2();
...
}
```

DLL Appl A WSA

A(dllfunc1)
A(DLL X WSA)
A(dllfunc2)
A(DLL X WSA)

DLL X

```
dllfunc1() { ... }
dllfunc2() { ... }
```

DLL X WSA

DLL Appl B WSA

A(CEETLOC)
dllfunc2_id

LE DLL Load Mechanism

DLL Appl A

```
main() {
...
dllfunc1();
...
dllfunc2();
}
```

dllfunc1() returns directly to DLL Appl A.

DLL Appl B

```
applB() {
...
dllfunc2();
...
}
```

DLL Appl A WSA

A(dllfunc1)
A(DLL X WSA)
A(dllfunc2)
A(DLL X WSA)

DLL X

```
dllfunc1() { ... }
dllfunc2() { ... }
```

DLL X WSA

DLL Appl B WSA

A(CEETLOC)
dllfunc2_id

LE DLL Load Mechanism

DLL Appl A

```
main() {
...
dllfunc1();
...
dllfunc2();
}
```

Subsequent calls and references to this DLL are direct.

DLL Appl B

```
applB() {
...
dllfunc2();
...
}
```

DLL Appl A WSA

A(dllfunc1)
A(DLL X WSA)
A(dllfunc2)
A(DLL X WSA)

DLL X

```
dllfunc1() { ... }
dllfunc2() { ... }
```

DLL X WSA

DLL Appl B WSA

A(CEETLOC)
dllfunc2_id

LE DLL Load Mechanism

DLL Appl A

```
main() {
...
dllfunc1();
...
dllfunc2();
}
```

Subsequent calls and references to this DLL are direct.

DLL Appl B

```
applB() {
...
dllfunc2();
...
}
```

DLL Appl A WSA

A(dllfunc1)
A(DLL X WSA)
A(dllfunc2)
A(DLL X WSA)

DLL X

```
dllfunc1() { ... }
dllfunc2() { ... }
```

DLL X WSA

DLL Appl B WSA

A(CEETLOC)
dllfunc2_id

LE DLL Load Mechanism

DLL Appl A

```
main() {
...
dllfunc1();
...
dllfunc2();
}
```

Subsequent calls and references to this DLL are direct.

DLL Appl B

```
applB() {
...
dllfunc2();
...
}
```

DLL Appl A WSA

A(dllfunc1)
A(DLL X WSA)
A(dllfunc2)
A(DLL X WSA)

DLL X

```
dllfunc1() { ... }
dllfunc2() { ... }
```

DLL X WSA

DLL Appl B WSA

A(CEETLOC)
dllfunc2_id

LE DLL Load Mechanism

DLL Appl A

```
main() {
...
dllfunc1();
...
dllfunc2();
}
```

DLL Appl B (separate program object with its own WSA)gets control and calls dllfunc2(), also resolved at Bind time via side deck.

DLL Appl B

```
applB() {
...
dllfunc2();
...
}
```

DLL Appl A WSA

A(dllfunc1)
A(DLL X WSA)
A(dllfunc2)
A(DLL X WSA)

DLL X

```
dllfunc1() { ... }
dllfunc2() { ... }
```

DLL X WSA

DLL Appl B WSA

A(CEETLOC)
dllfunc2_id

LE DLL Load Mechanism

DLL Appl A

```
main() {
...
dllfunc1();
...
dllfunc2();
}
```

What is LE doing?
Call LE "trigger DLL load on Call" routine.

DLL Appl B

```
applB() {
...
dllfunc2();
...
}
```

DLL Appl A WSA

A(dllfunc1)
A(DLL X WSA)
A(dllfunc2)
A(DLL X WSA)

DLL X

```
dllfunc1() { ... }
dllfunc2() { ... }
```

DLL X WSA

DLL Appl B WSA

A(CEETLOC)
dllfunc2_id

LE DLL Load Mechanism

DLL Appl A

```
main() {
...
dllfunc1();
...
dllfunc2();
}
```

What is LE doing?

Determine if the DLL is already loaded in this enclave.

DLL Appl B

```
applB() {
...
dllfunc2();
...
}
```

DLL Appl A WSA

A(dllfunc1)
A(DLL X WSA)
A(dllfunc2)
A(DLL X WSA)

DLL X

```
dllfunc1() { ... }
dllfunc2() { ... }
```

DLL X WSA

DLL Appl B WSA

A(CEETLOC)
dllfunc2_id

LE DLL Load Mechanism

DLL Appl A

```
main() {
...
dllfunc1();
...
dllfunc2();
}
```

What is LE doing?

Has DLL been loaded into this enclave before??

Yes -- use previously-obtained WSA.

DLL Appl B

```
applB() {
...
dllfunc2();
...
}
```

DLL Appl A WSA

A(dllfunc1)
A(DLL X WSA)
A(dllfunc2)
A(DLL X WSA)

DLL X

```
dllfunc1() { ... }
dllfunc2() { ... }
```

DLL X WSA

DLL Appl B WSA

A(CEETLOC)
dllfunc2_id

LE DLL Load Mechanism

DLL Appl A

```
main() {
...
dllfunc1();
...
dllfunc2();
}
```

What is LE doing?
Resolve all descriptors (and variable references) to DLL X (only) in DLL Appl B's WSA

DLL Appl B

```
applB() {
...
dllfunc2();
...
}
```

DLL Appl A WSA

A(dllfunc1)
A(DLL X WSA)
A(dllfunc2)
A(DLL X WSA)

DLL X

```
dllfunc1() { ... }
dllfunc2() { ... }
```

DLL X WSA

DLL Appl B WSA

A(dllfunc2)
A(DLL X WSA)

LE DLL Load Mechanism

DLL Appl A

```
main() {
...
dllfunc1();
...
dllfunc2();
}
```

What is LE doing?
Complete call to dllfunc2()
and then return directly to
DLL Appl B.

DLL Appl B

```
applB() {
...
dllfunc2();
...
}
```

DLL Appl A WSA

A(dllfunc1)
A(DLL X WSA)
A(dllfunc2)
A(DLL X WSA)

DLL X

```
dllfunc1() { ... }
dllfunc2() { ... }
```

DLL X WSA

DLL Appl B WSA

A(dllfunc2)
A(DLL X WSA)

Restrictions on DLLs

- DLL function entry point must be LE-style
- Implicitly loaded DLLs cannot be explicitly freed
- DLL and DLL application must have same AMODE
- C/C++ function main() cannot be exported
- Limited support for mixing DLLs and COBOL Dynamic Call (see COBOL Programming Ref)
- "Physical" loading of DLLs is not supported while C++ destructors are running ("Logical" load ok)
- DLLs must be linkedited as REENTRANT!!

Performance Considerations

- Since a reference to an exported data item can cause a DLL to be preloaded, consider packaging rarely-used (and possibly related) functions together in the DLL that exports no data.
  - This DLL won't be loaded until it's needed
- WSA is more costly to load than program text (in LPA)
  - Consider making strings and constants read-only to reduce WSA size
    - #pragma strings(readonly), or ROSTRING compiler option
    - ROCONST compiler option
- Don't use EXPORTALL when only a small subset of functions are needed
  - EXPORTALL could create large table to be searched

Shared Libraries (in OS/390 V2R9 and up)

- Think of it as "LPA for the HFS"
  - A single copy residing in a Shared Library Region in memory
  - Shared at the same memory location across address spaces *that loaded it*
  - This is for readonly program text only -- WSA is still allocated as before
  - Reduced memory usage, reduced I/O, fewer page faults
- Previously had to use sticky bit and LPA
  - Name conflicts, complicates install, 16M limit, reduces available storage in all address spaces

System Shared Libraries

- System Shared Libraries are...
  - marked with shared library extended attribute (extattr, le -E)
    - requires at least READ access to BPX.FILEATTR.SHARELIB FACILITY class profile
  - expected to be used "globally" across the system
  - mapped to addresses in system shared library region
    - high end of private storage
    - controlled by SHRLIBRGNSIZE() in BPXPRMxx PARMLIB
    - not charged against individual user's region utilization
    - shared on megabyte boundary (reduces ESQA usage)

User Shared Libraries

- User Shared Libraries are...
  - identified by ".so" filename suffix
  - meant for more "local" usage in a small set of address spaces
  - mapped to addresses from low end of private storage
    - charged against a user's region utilization
    - limited by SHRLIBMAXPAGES(), and also count towards total limited by MAXSHAREPAGES(), both in BPXPRMxx PARMLIB

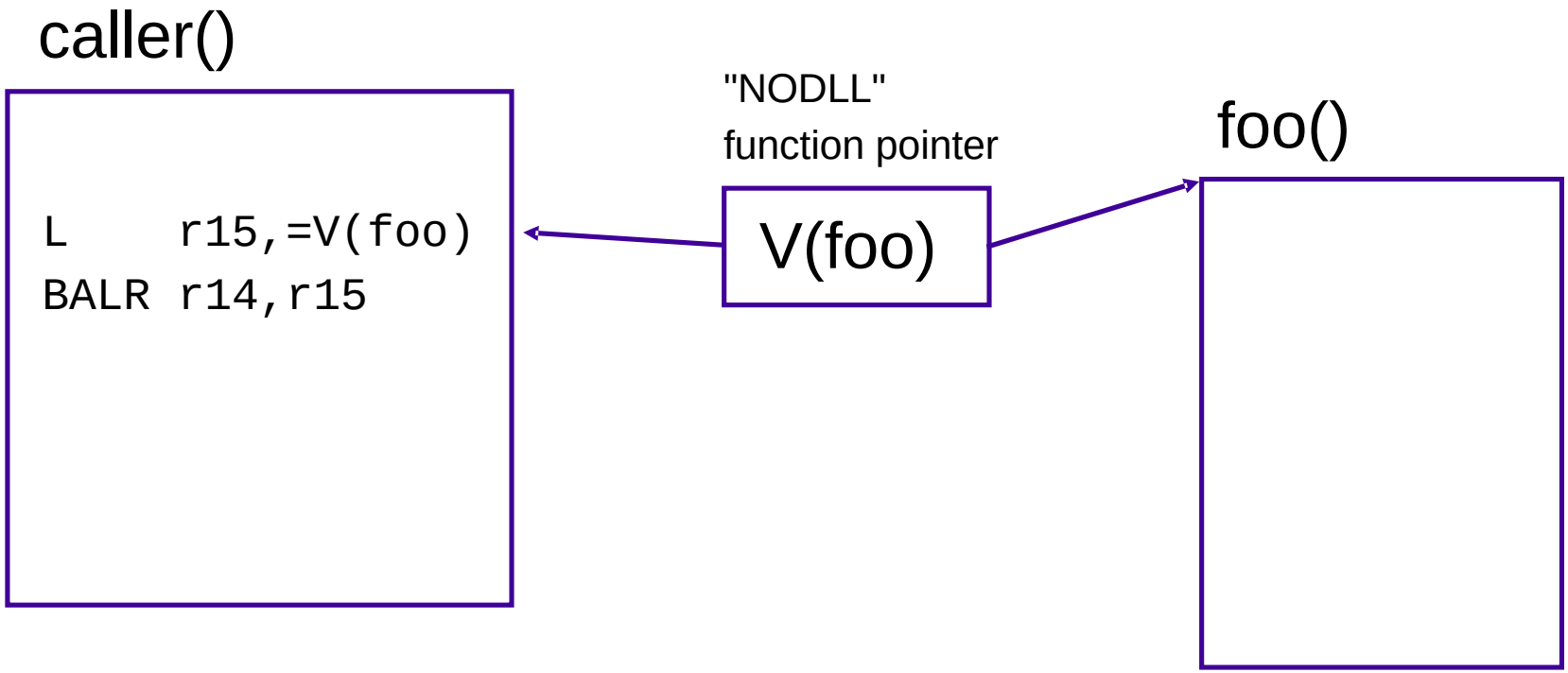
More on Shared Libraries

- Virtual address chosen for the executable on its first load will be used on all subsequent loads from all other address spaces, if possible
- If the load cannot be mapped at shareable address, then it will be treated as a non-shared private area load module, loaded into user private storage
  - Chance of address conflict much lower with System Shared Libraries

DLL considerations for XPLINK-compiled code

- All XPLINK-compiled C/C++ code is DLL-enabled
- Function Descriptors are unique, allowing pointer-to-int casting and comparison
- XPLINK and non-XPLINK compatibility is provided at the Program Object boundary (DLLs are a good way to do this)

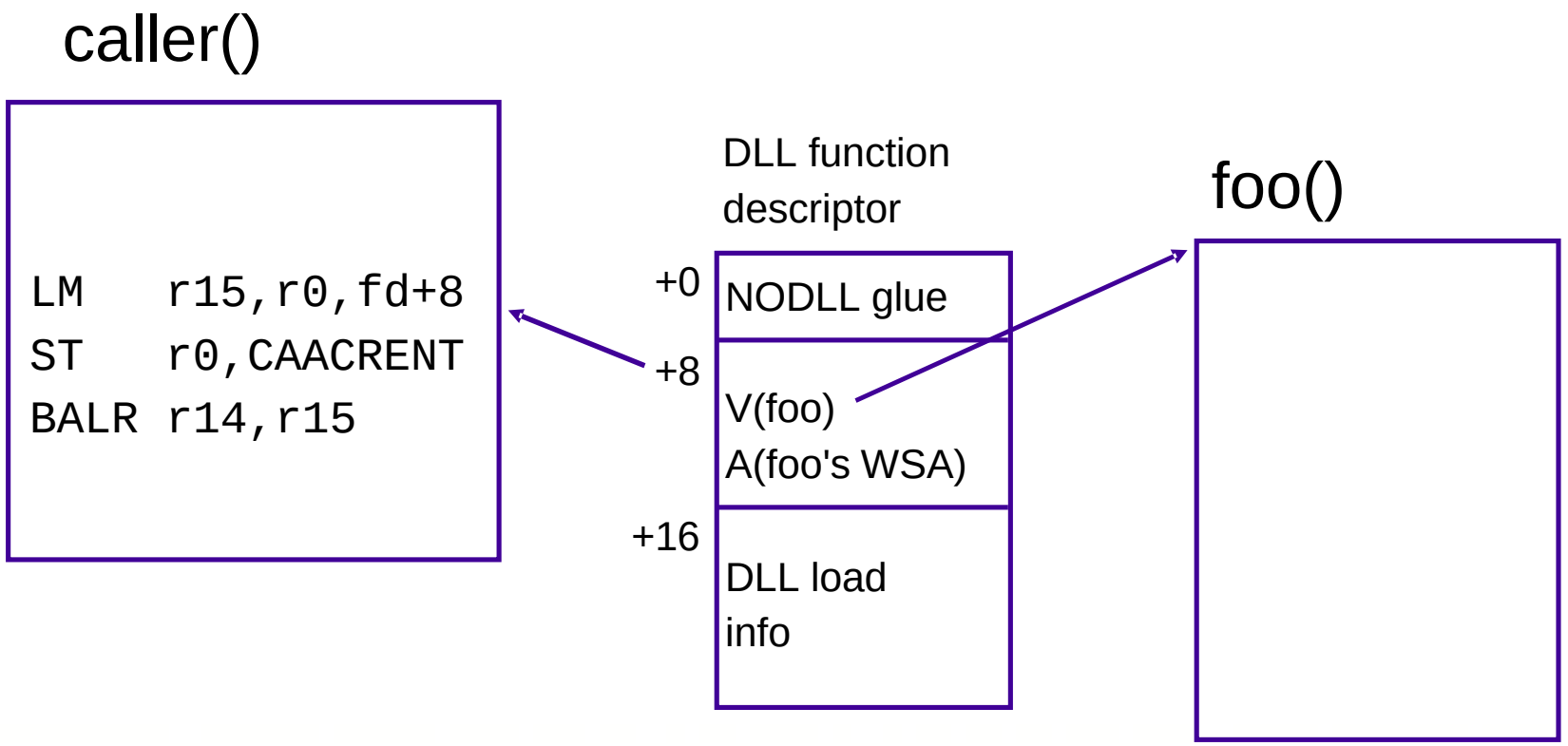
Non-DLL function linkage



Note that:

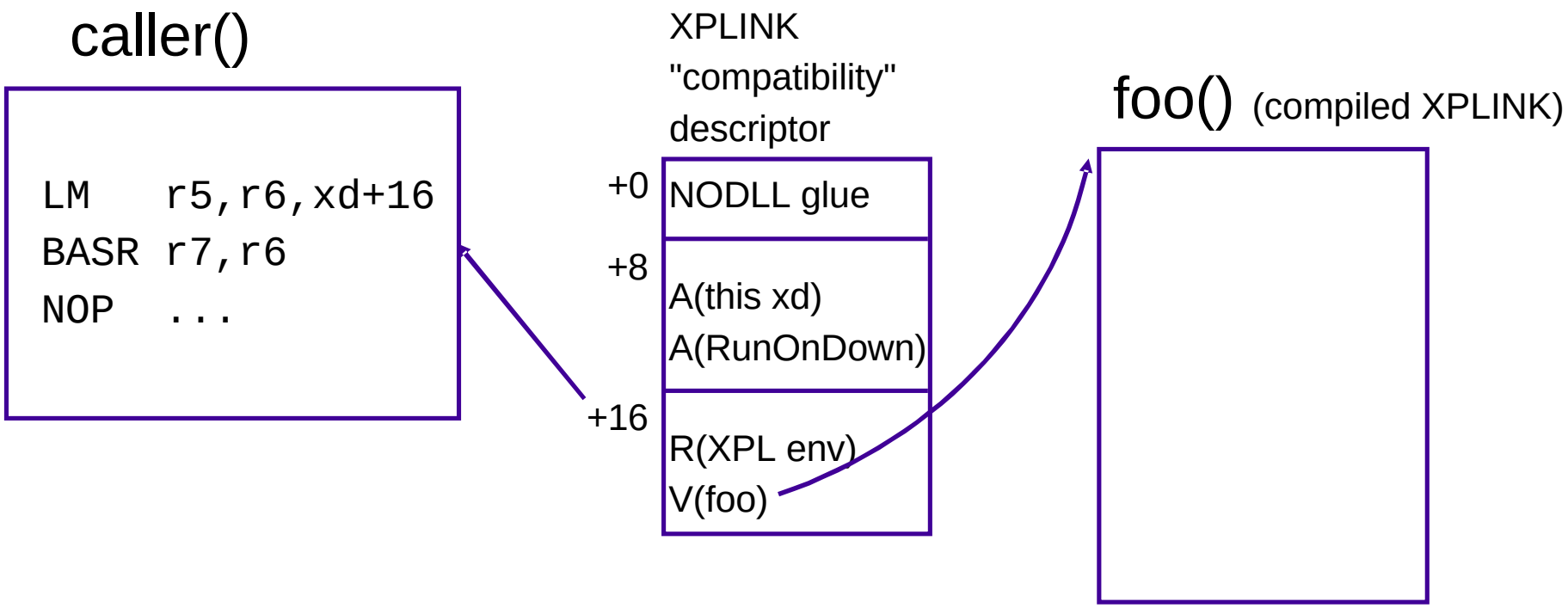
- caller() and foo() must be statically linked, or
- foo is entry point of LOADed module

DLL function linkage



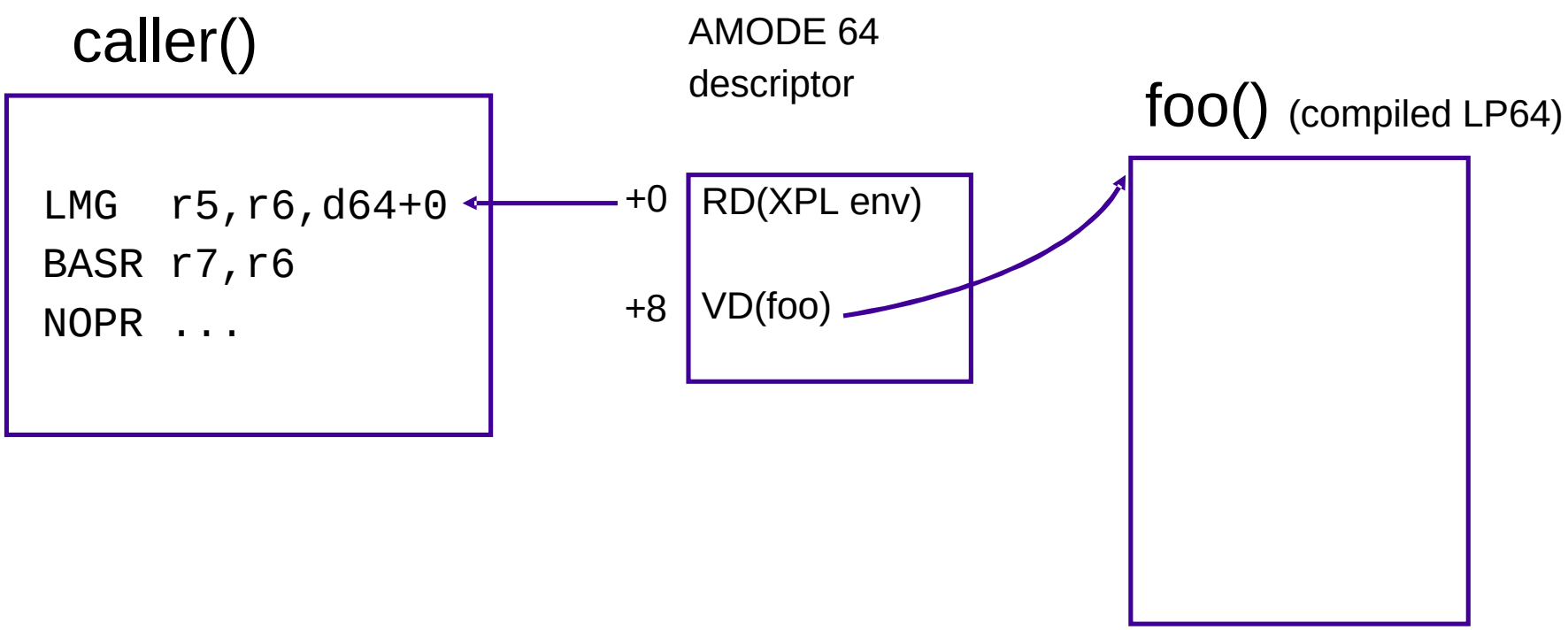
DLL linkage enables WSA-switching if caller() and foo() reside in different executables.

XPLINK function linkage



- All XPLINK function calls are "DLL-enabled"
- An XPLINK function's "environment" is contained within the WSA of the executable

AMODE 64 function linkage



- All AMODE 64 function calls are XPLINK and thus "DLL-enabled"
- An AMODE 64 function's "environment" is contained within the WSA of the executable
- An AMODE 64 descriptor has no compatibility glue

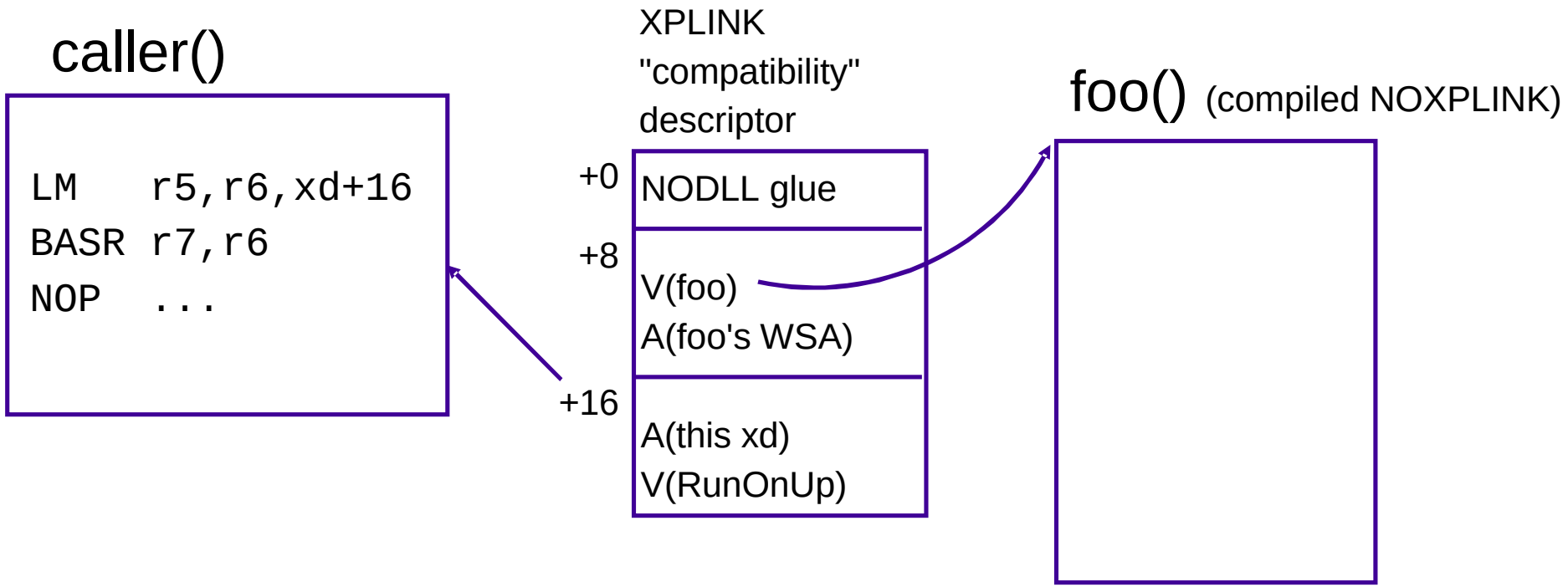
Compatibility Issues -- Calls By Pointer

- A function pointer may not be passed from "older" linkages to "newer" linkages without special action
  - ie. non-DLL -> DLL -> XPLINK
  - DLL(CALLBACKANY) suboption allows receiving non-DLL function pointers
    - all calls-by-pointer go through LE (expensive!)
  - XPLINK has a couple options
    - __bldxfd() CWI to convert to correct type of function descriptor
    - __callback type
    - XPLINK(CALLBACK) (similar to DLL(CBA))
- Function pointers may be passed from "new" to "old"

Compatibility Issues -- Function Pointer Comparison

- The following 3 cases only work in DLL-enabled code, which "dereferences" DLL function descriptors
  - comparing a non-DLL function pointer to a DLL function pointer
  - comparing a DLL function pointer to a constant function address
  - comparing DLL function pointers
- Uninitialized function pointers may not be used for comparisons in DLL-enabled code
  - Generated code will make sure containing DLL is loaded
- XPLINK and AMODE 64 function descriptors are unique, so their address can be used in comparisons.

Compatibility Issues -- Calling non-XPLINK from XPLINK



Compatibility between XPLINK and non-XPLINK functions is at the Program Object boundary, and implemented through the glue code.

Variable Descriptors

Prelinker

AS INITIALIZED

+0	0
+4	DLL index
+8	A(C sig CSECT)
+C	A(DLL appl's WSA)

AFTER DLL LOAD

+0	A(var) - A(DLL appl's WSA)
+4	unchanged
+8	unchanged
+C	unchanged

Binder

AS INITIALIZED

+0	0
+4	A(DLL entry in IEWBCIE)
+8	A(CEESTART)
+C	A(DLL appl's WSA)

AFTER DLL LOAD

+0	A(var) - A(DLL appl's WSA)
+4	unchanged
+8	unchanged
+C	unchanged

XPLINK

- no descriptor
- fullword w/real addr

DLL-compiled Function Descriptors

Prelinker

AS INITIALIZED

+0	LR	R0,R15
	L	R15,16(R15)
	BR	R15
+8	A(@@TRGLOC)	
+C	A(descriptor)	
+10	A(@@GETFN)	
+14	DLL index	
+18	A(C sig CSECT)	
+1C	A(DLL appl's WSA)	

AFTER DLL LOAD

+0	glue -- unchanged
+8	V(DLL function)
+C	A(DLL's WSA)
+10	unchanged
+14	unchanged
+18	unchanged
+1C	unchanged

Binder

AS INITIALIZED

+0	LR	R0,R15
	L	R15,16(R15)
	BR	R15
+8	A(CEETLOC)	
+C	A(descriptor)	
+10	A(CEETGTFN)	
+14	A(IEWBCIE entry)	
+18	A(CEESTART)	
+1C	A(DLL appl's WSA)	

AFTER DLL LOAD

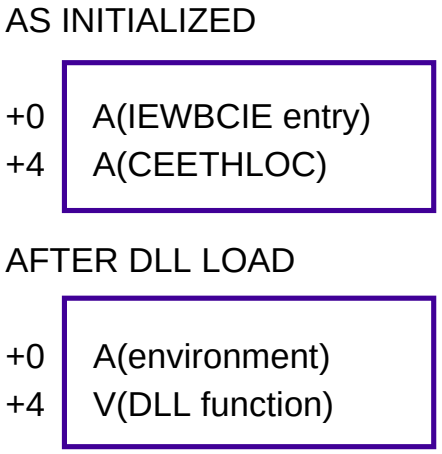
+0	glue -- unchanged
+8	V(DLL function)
+C	A(DLL's WSA)
+10	unchanged
+14	unchanged
+18	unchanged
+1C	unchanged

XPLINK-compiled Function Descriptors

- XPLINK distinguishes between "call by name" and "call by pointer".

- Under XPLINK, if a DLL contains at least 1 "by pointer" reference, then it will be preloaded by DLL initialization.
  - This means XPLINK has no DLL "trigger load on call" through a function pointer.
  - Preloaded when Program Object currently being loaded has an IMPORT from the DLL containing the “by pointer” reference.

XPLINK-compiled "By Name" Function Descriptors



Note: XPLINK Call-by-name function linkage is slightly different than the call-by-ptr linkage shown earlier:

```
LM   r5,r6,xd
BASR r7,r6
NOP  ...
```

No need to index +16 into the descriptor in this case to get around the "compatibility" stuff.

XPLINK-compiled "By Pointer" Function Descriptors

- Representing XPLINK-compiled function

+0	LR	R0,R15
	L	R15,12(,R15)
	BR	R15
+8	A(unique fd) (Note 1)	
+C	A(RunOnDown)	
+10	A(XPL fn environment)	
+14	A(XPL fn entry point)	

- Representing NOXPLINK-compiled function

+0	LR	R0,R15
	L	R15,900(,R12)
	BR	R15
+8	A(nonXPL fn entry point)	
+C	A(nonXPL fn WSA)	
+10	A(this fd)	
+14	A(RunOnUp) glue	

(Note 1) This may be the address of this descriptor.

Function Descriptor
Glue Code "Signature" Summary

- DLL-compiled function descriptors
  - LR R0,R15 180F
  - L R15,16(R15) 58FF0010
  - BR R15 07FF
- XPLINK-compiled function descriptors
  - Representing XPLINK-compiled functions
    - LR R0,R15 180F
    - L R15,12(,R15) 58F0F00C
    - BR R15 07FF
  - Representing non-XPLINK-compiled functions
    - LR R0,R15 180F
    - L R15,900(,R12) 58F0C384
    - BR R15 07FF