Structured Programming in Assembler
Session 17871

Richard Cebula (riccebu@uk.ibm.com) IBM HLASM

Who am I?

- Richard Cebula – HLASM, IBM Hursley, UK
- riccebu@uk.ibm.com

- Develop and support the following products:
  - HLASM
  - SuperC
  - XREF
  - IDF
  - DISASM
  - Structured Programming Macros

HLASM Structured Programming

- Structured Programming for Assembler!?
- Control sections - C-, R-, DSECTs, COM and DXDs
- Subroutines, EXTRN / WXTRN and Conditional-Sequential RLDs
- The location counter and USING statements
- Type-checking in HLASM
- Program Objects
- Controlling Listings
- Structured Programming Macros (SPMs)

HLASM Structured Programming

- Structured Programming for Assembler!?
  - Yes!
  - HLASM is the *High Level* Assembler providing an extensive list of features for assembler programmers
  - HLASM can help programmers to:
    - Organise their assembler code better
    - Maintain their code better
    - Increase code reuse
- Remember, HLASM is available for all z Systems operating systems: z/OS, z/VM, z/VSE, z/Linux (inc. z/TPF)

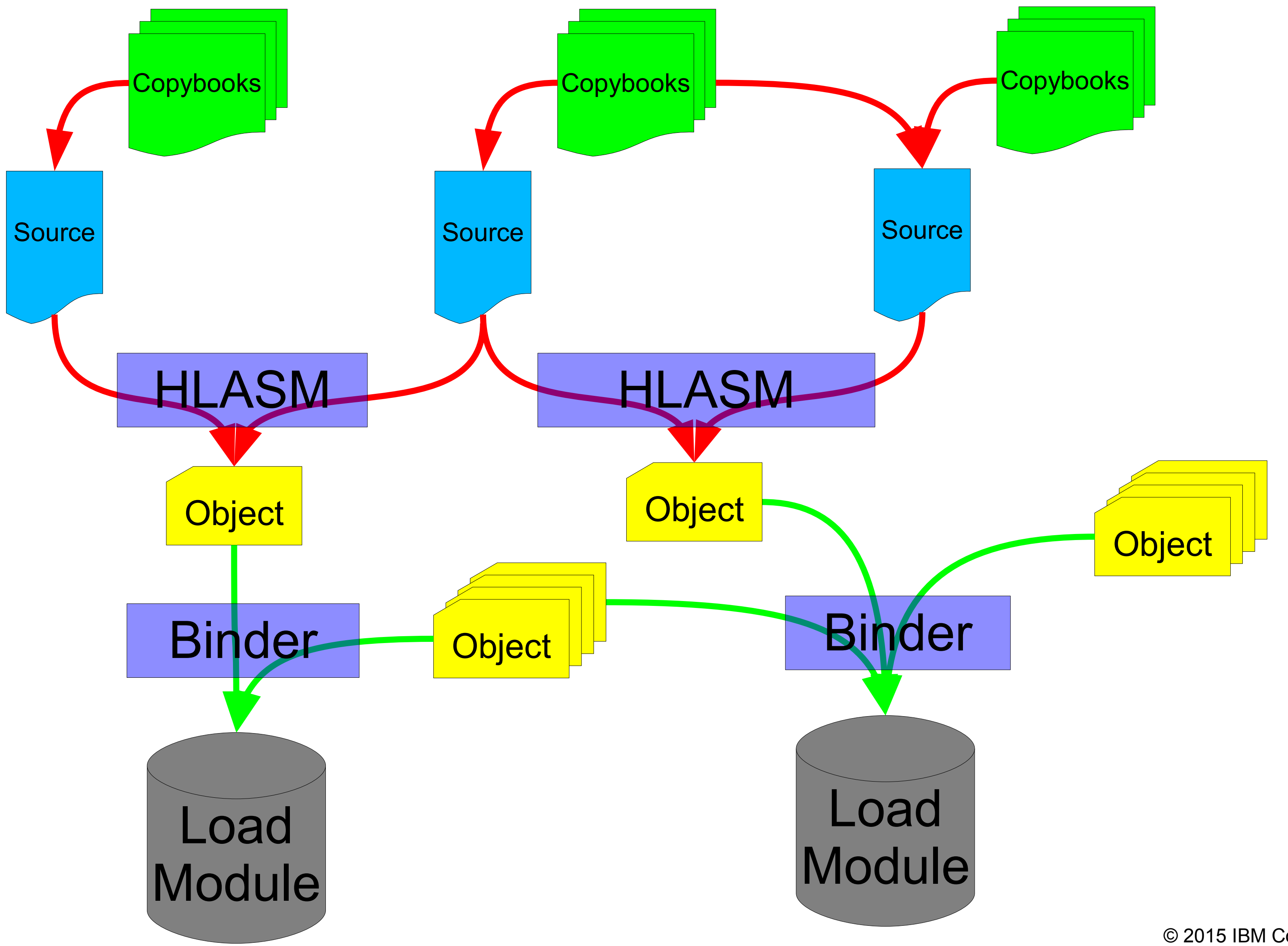
HLASM Structured Programming

Control sections

HLASM Structured Programming – Source and Object Modules

- An HLASM program consists of a number of sections of varying types
- The programmer should distinguish between:
  - Source Module
    - The source module is the division of code during assembly time
    - Each source module is assembled into a separate object module
    - Note that there is not always a 1-to-1 relationship between a source *file* and a source module
  - Object Module
    - The object module is the produced output from the assembler
    - The layout of the object module is determined by the type assembler options used
    - HLASM supports OBJ, GOFF and ELF object file formats

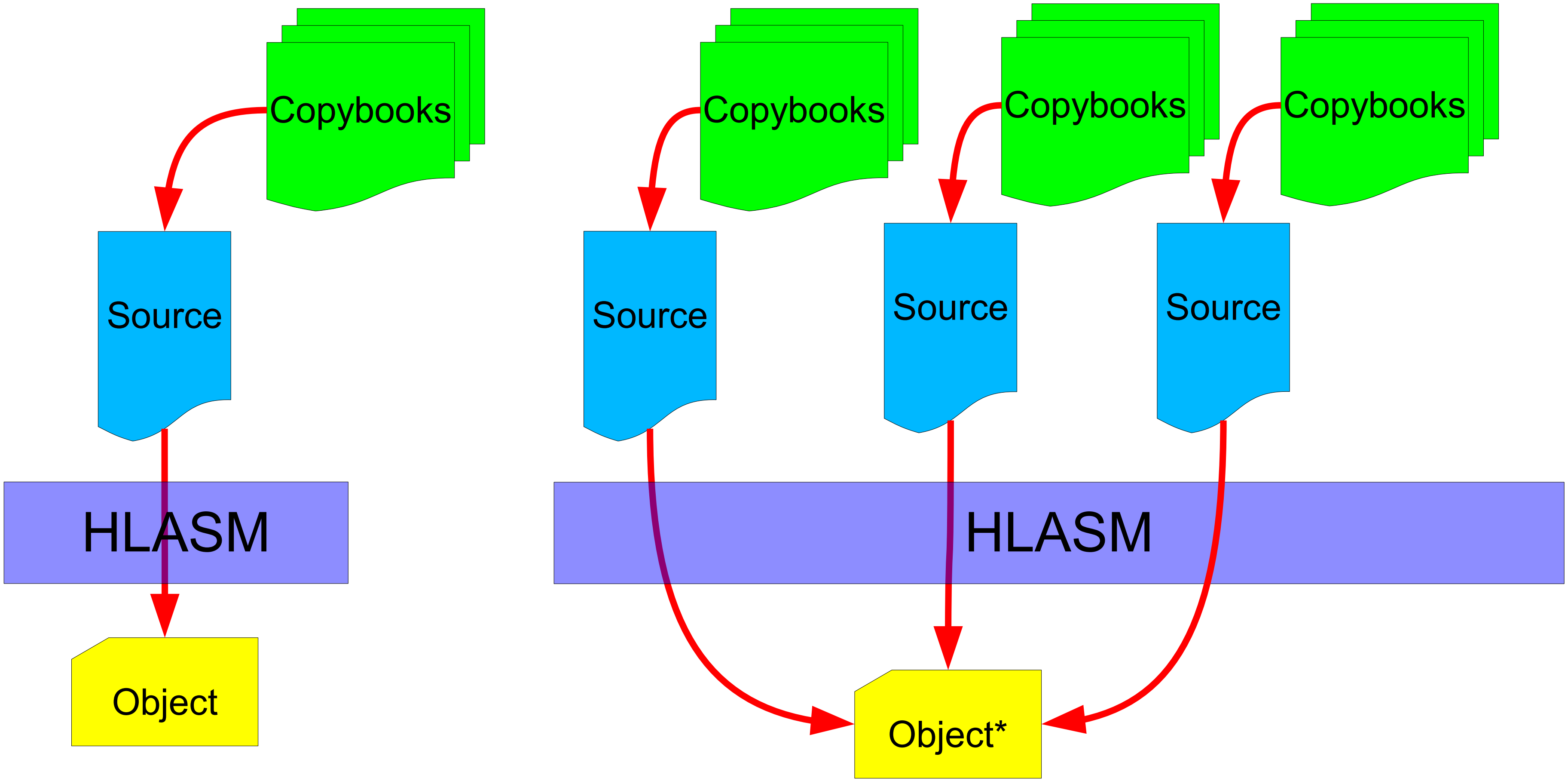
HLASM Structured Programming – Source and Object Modules



HLASM Structured Programming – Source Module – Sections

- Source module comprises 1 or more assembler statements
- Starts with any assembler statement except for MEXIT and MEND
- Ends with an END statement
- The BATCH option allows for more than a single source module to be specified in the same input *stream*
  - HLASM can take many source *files* on the same input *stream*, e.g. via use of the COPY statement
  - If HLASM encounters an END statement during processing then it completes assembling the source module.
  - If further input is found and the BATCH option is on (the default) then HLASM begins the assembly of a new source module

HLASM Structured Programming – Single VS Batch Assembly



In batch mode, all output is written to a single output object file.
*On z/Linux, use RPM asma90-1.6.0-27 or higher with the -R option to output to an object archive

HLASM Structured Programming – Object Module – Sections

- In the load module model, a *control section* is the smallest subdivision of a program that can be located as a unit
- Each assembled control section contains the object code for machine instructions and data
- Each source module (consisting of 1 or more source files) is assembled into 1 relocatable object module.
- The binder combines the object code of 1 or more sections into a load module (or program object)
  - The binder also calculates any addresses and space needed for common sections and external dummy sections from different object module
- The program loader loads the load module into virtual storage and converts any relocatable addresses into fixed locations

HLASM Structured Programming – Creating Sections

- The first section of a program may be initiated via the START, CSECT or RSECT assembler instructions
  - START – initiates the first or only section of a source module
  - CSECT – can be used anywhere in the source module to initiate or continue a control section
  - RSECT – similar to CSECT but causes the assembler to check for possible violations of reenterability
- Unnamed sections are those that do not have a name – although this is valid, it is not recommended.

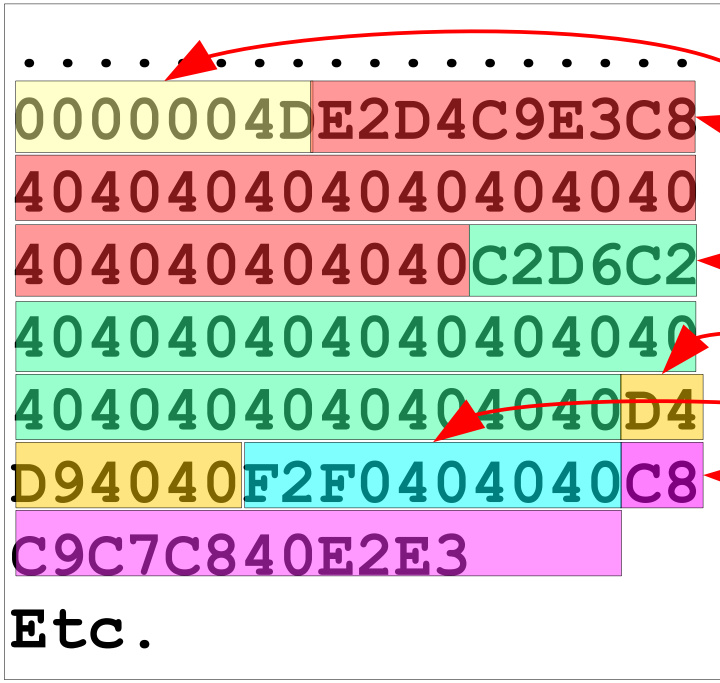
HLASM Structured Programming – Reference Control Sections

- Reference control sections are used for referencing storage areas or to describe data and are not assembled into object code
  - Started by the DSECT, COM and DXD statements
  - As with other forms of control sections, they continue until interrupted by either another control section or an END statement
- DSECT – Dummy control section
  - Reference control section that describes the layout of data in storage without reserving any virtual storage
  - There is no object code nor space in the object module reserved. A DSECT will cause the assembler to assign location values for the DSECT's symbols relative to its beginning
  - Data can be referred to symbolically by using the symbols defined in the dummy section

HLASM Structured Programming – Reference Control Sections

```
. . . . . . . . . . . . . . . . . .
0000004DE2D4C9E3C8
404040404040404040
40404040C2D6C2
404040404040404040
40404040404040D4
D94040F2F040404040C8
C9C7C840E2E3
Etc.
```

HLASM Structured Programming – Reference Control Sections



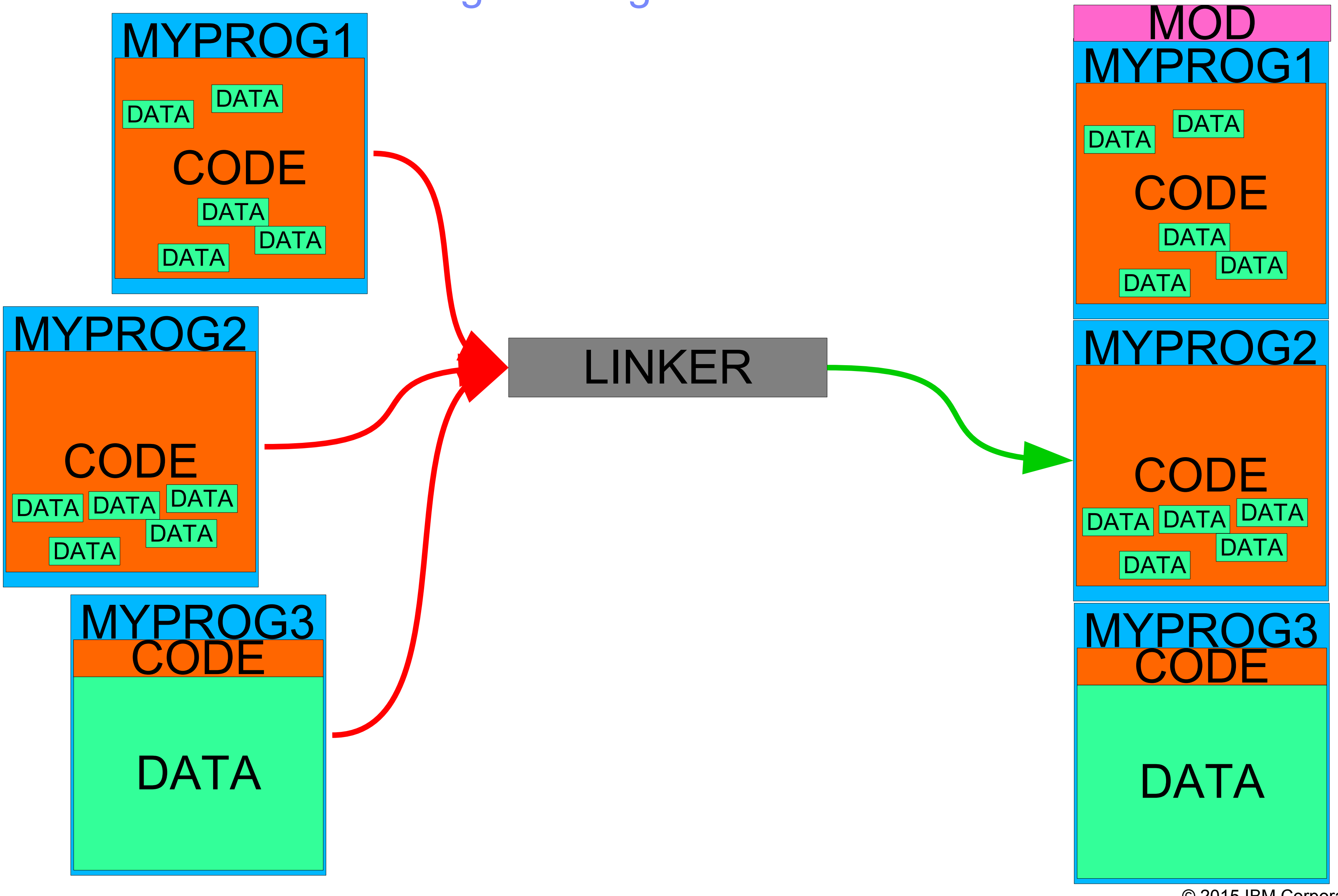

```
CUSTOMER      DSECT
cust_number dc  f'0'
cust_name   dc  cl20' '
cust_cname  dc  cl20' '
cust_title  dc  Cl4' '
house_num   dc  Cl5' '
addr_line1  dc  cl40' '
addr_line2  dc  cl40' '
city        dc  cl40' '
```

HLASM Structured Programming – Reference Control Sections

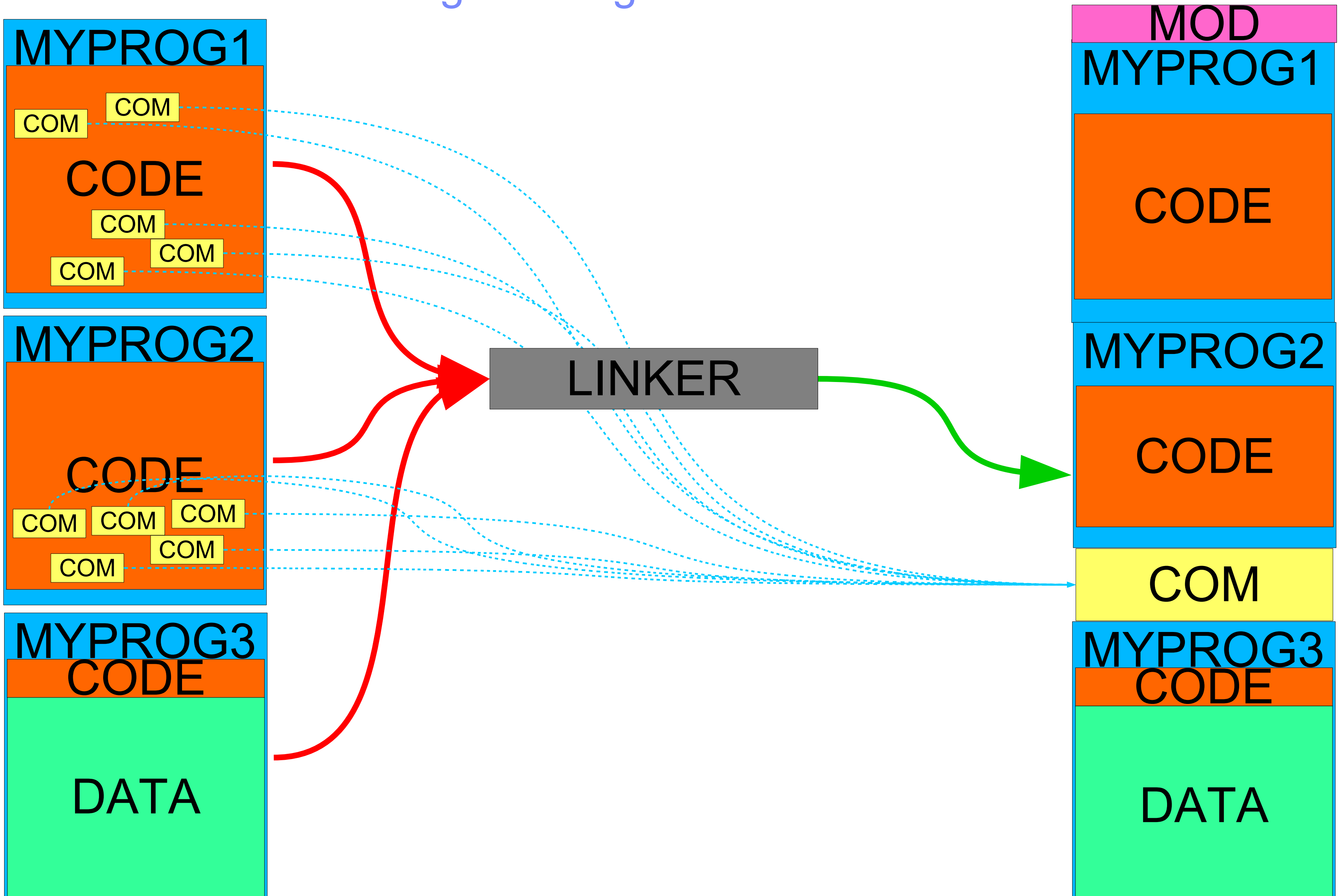
- COM – Common control section
  - Allows for the definition of a common storage area in one or more source modules
  - At link time, only one copy of a COM section is created for all the object modules being linked as a single program
  - Only the storage area is provided – the data must be provided at execution time, i.e. COM is *uninitialised!*

- External dummy sections
  - Created via DXD, DSECT and CXD instructions or via the Q-type address constant
  - To use:
    - Use a DXD instruction to define the external dummy section
    - Provide a Q-type constant to address the external dummy section
    - Use the CXD instruction to obtain the *total length* of all external dummy sections
    - Allocate the storage required as calculated by the CXD
    - Address the allocated storage (plus any offset into the areas as required)

HLASM Structured Programming – Reference Control Sections



HLASM Structured Programming – Reference Control Sections



HLASM Structured Programming – Reference Control Sections

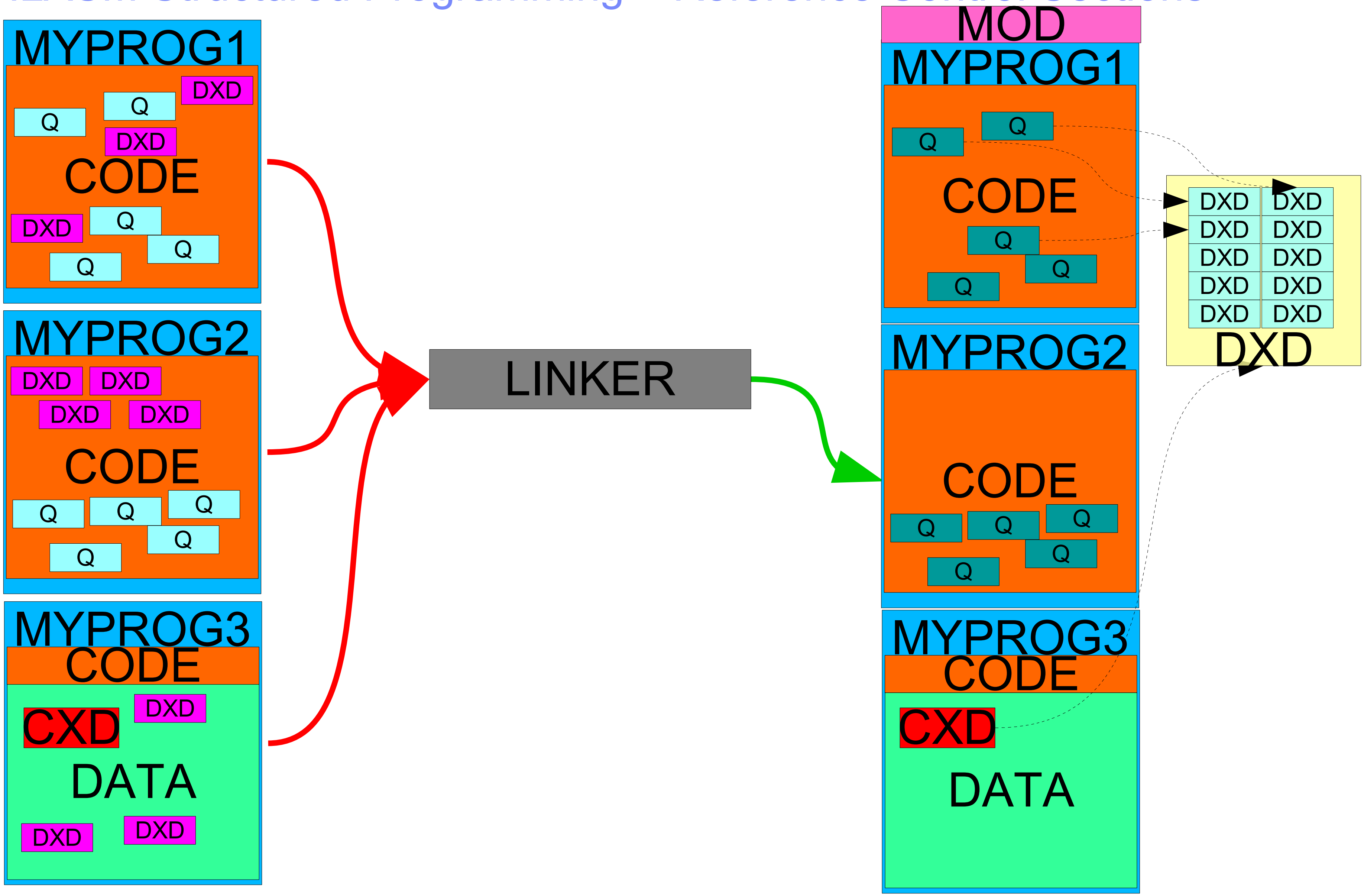
- What happens with my DXDs at *link time*?
  - All DXDs are not allocated as part of the module – instead just the references to the DXD items in the linked module are retained.
  - The linker collects all the DXDs in all the modules it is linking together and creates the complete External Dummy Section.
  - All DXD Q-type references throughout the various modules are replaced by offsets from the start of the completed External Dummy Section.
  - The total length of the External Dummy Section is placed into every CXD.
- To make use of the DXDs, the program has to allocate the correct amount of storage as per the CXD and then reference each member of the DXD via Q-type offsets, e.g.:

```
my_prog3:
...code goes here...
    L r0,DXD_TOTAL
    GETMAIN R,LV=(0)
...code goes here...
DXD_TOTAL CXD
```

```
my_prog1:
    L r7,=Q(MY_DXD_SYM)
    AR r7,r1
    L  r7,0(,r7)
```

```
my_prog2:
    L r7,=Q(MY_OTHER_DXD)
    AR r7,r1
    L  r7,0(,r7)
```

HLASM Structured Programming – Reference Control Sections



HLASM Structured Programming

Subroutines

HLASM Structured Programming – Subroutines

- Breaking sections of code out into subroutines is one of the easiest ways to structure programs.
- If the subroutine is close to the code that calls it then the following code should suffice:

```
bras     r14,mysubroutine
```

- However, if the subroutine is beyond the range of an active using then an A-type address constant should be used instead:

```
l        r15,=A(mysubroutine)
basr     r14,r15
```

- This technique only works for addresses which can be resolved during assembly time.
- It is much more common to use BRAS/BRASL. BRASL also works with for external entry points

HLASM Structured Programming – External Subroutines

- An *external reference* is a symbol that is unresolvable at assembly time. Instead, the symbol is resolved by the binder when the object files are linked together.
- To make a symbol externally available, the module in which the symbol is declared must either define it as the name of a control section or as an ENTRY, e.g.:

```
         MYSUB ENTRY
```

- Note that START, CSECT, RSECT are automatically considered as ENTRY symbols
- In the module that references the external symbol, the external symbol must be declared as EXTRN before being used:

```
         EXTRN MYSUB
      ...some code...
      subaddr    dc   a(MYSUB)
```

- The WXTRN instruction can also be used instead of EXTRN. However
  - For EXTRN, if the symbol is not found from the list of modules being linked, then a library search is performed to try and find the symbol
  - For WXTRN, no library search is performed

HLASM Structured Programming – External Subroutines

- Alternatively, the V-type constant automatically declares a symbol as EXTRN:

```
mod1     CSECT
mod1     AMODE 31
mod1     RMODE 24
         Using *,15
         extrn mod2
         l     15,=a(mod2)
         basr  14,15
         br    14
```

```
mod2     CSECT
mod2     AMODE 31
mod2     RMODE 24
         la    15,2
         br    14
```

```
mod1     CSECT
mod1     AMODE 31
mod1     RMODE 24
         using *,15
         l     15,=v(mod2)
         basr  14,15
         br    14
```

```
mod2     CSECT
mod2     AMODE 31
mod2     RMODE 24
         la    15,2
         br    14
```

HLASM Structured Programming – Using CSRs – z/OS only

- For z/OS 1.13 and above, with HLASM APAR PM74898 applied and for GOFF only...
- Conditional Sequential RLDs, (CSRs) allow an assembler programmer to specify multiple external references in a single V-type address constant separated by colons, e.g.:

```
my_modules    dc v(mod1:mod2:mod3)
```

- The assembler program continues to use the V-type as normal but:
  - When the binder goes to resolve the external symbol, it will resolve it to the first available external symbol
  - If the first symbol isn't present, then the 2^{nd} available symbol is chosen and so on...
- For a more detailed look at how to use CSRs see:
  http://www.ibm.com/support/docview.wss?uid=swg21598283

The location counter and USINGs

HLASM Structured Programming – Using the location counter

- The location counter is an internal count from the start of the source module in bytes and is represented by the symbol *
- The LOCTR instruction may be used to specify multiple location counters within a control section.
- Each location counter is given consecutive addresses and the order of location counters produced is dependent on the order in which they are defined
- A location counter continues until it is interrupted by the START, CSECT, DSECT, RSECT, CATTR or LOCTR instructions
- Specifying LOCTR to an already defined location counter will resume that counter

HLASM Structured Programming – The USING statement

- The USING specifies a base register for a series of location counter displacements to a particular symbol within a control section
- There have been improvements to HLASM's USING statement including 'Labeled USINGs' and 'Dependent USINGs'
  - Labeled USINGs
    - Allow simultaneous references to multiple instances of an object
    - Allow one object to be referenced per register
  - Dependent USINGS
    - Address multiple objects with a single register
    - Allow for a program to require fewer base registers
    - Allow for dynamic structure remapping during execution

HLASM Structured Programming – Labeled USINGs

- The labeled USING statement has the syntax in the form:
  - *qualifier* USING *base,register*

- A symbol can be referenced by prefixing it with a qualifier and therefore the same symbol may be addressed by 2 or more base registers at the same time, e.g.

```
* COPY THE CUSTOMER DETAILS TO THE NEW CUSTOMER RECORD
CSTMR1      USING       CUST_DATA,R4
CSTMR2      USING       CUST_DATA,R5
            MVC CSTMR2.CUST_DETAILS,CSTMR1.CUST_DETAILS
            L           R3,NEW_CUST_NUMBER
            ST          R3,CSTMR2.CUST_NUM
```

- Without labeled USINGs, the above MVC would have to be written using manually calculated offsets, e.g.

```
* COPY THE CUSTOMER DETAILS TO THE NEW CUSTOMER RECORD
    USING       CUST_DATA,R5
    MVC         CUST_DETAILS,CUST_DETAILS-CUST_DATA(R4)
```

HLASM Structured Programming – Dependent USINGs

- A dependent USING is one that specifies an *anchor location* rather than a register as its operand
- Allows the programmer to address more than one DSECT at the same time using the same base register

```
* ADDRESS THE DATA IN BOTH XPL_DATA AND REQI_DATA
           USING      XPL_DATA,R10
           USING      REQI_DATA,L_XPL_DATA+XPL
           LA         R1,XPL_INPUT
           LA         R2,REQI_INPUT
           . . .
```

- HLASM will add the offset from REQI_DATA to XPL_DATA in order to generate the offsets for the fields in the REQI_DATA DSECT based off register 10
- The displacement limit for a dependent using is still limited to 4095 (as usual)

HLASM Structured Programming – Labeled Dependent USINGs

- Dependent USINGs can also have a label allowing for some complex USING issues to be easily resolved using a single a base register

```
         USING E,7                             1 Top level
*                                           |
D1E      USING D,D1           1             | 2 Map D1 into E at D1
D1F1     USING F,D1E.F1           2         | | 3 Map F1 into D1 at F1
D1F2     USING F,D1E.F2           2         | | 3 Map F2 into D1 at F2
D1F3     USING F,D1E.F3           2         | | 3 Map F3 into D1 at F3
*                                           | 2 Middle level
D2E      USING D,D2           1             | | Map D2 into E at D2
D2F1     USING F,D2E.F1           3         | | 3 Map F1 into D2 at F1
D2F2     USING F,D2E.F2           3         | | 3 Map F2 into D2 at F2
D2F3     USING F,D2E.F3           3         | | 3 Map F3 into D2 at F3
*                                           | 2 Middle level
D3E      USING D,D3           1             | | Map D3 into E at D3
D3F1     USING F,D3E.F1           4         | | 3 Map F1 into D3 at F1
D3F2     USING F,D3E.F2           4         | | 3 Map F2 into D3 at F2
D3F3     USING F,D3E.F3           4         | | 3 Map F3 into D3 at F3
```

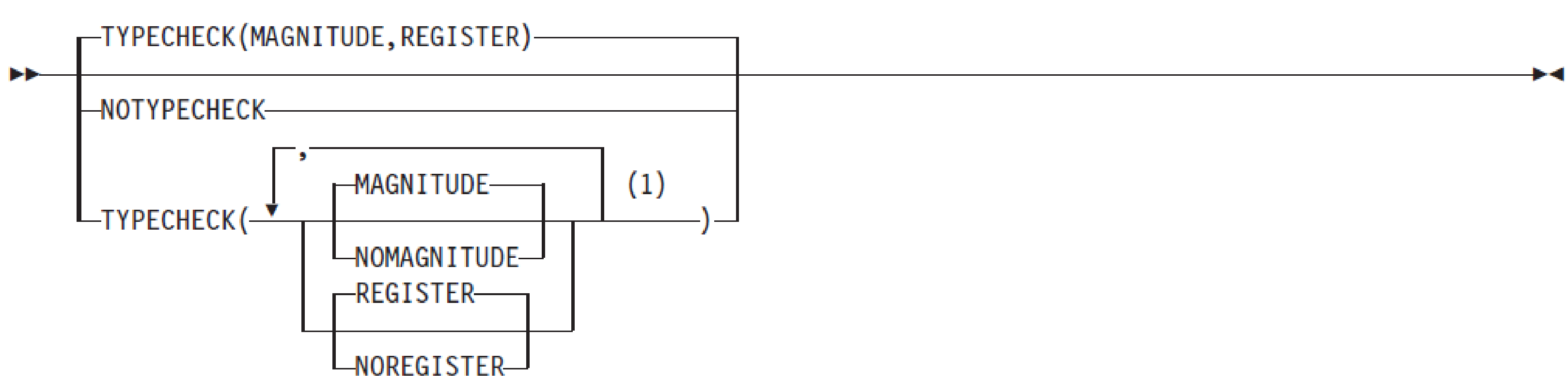
HLASM Structured Programming – Labeled Dependent USINGs

```
* Move fields named X within DSECTs described by F
    MVC D1F1.X1,D1F1.X2     Within bottom-level DSECT D1F1
    MVC D1F3.X2,D1F1.X1     Across bottom-level DSECTs in D1
    MVC D3F2.X2,D3F3.X2     Across bottom-level DSECTs in D3
    MVC D2F1.X1,D3F2.X2     Across bottom-level DSECTs in D2 & D3

* Move DSECTs named F within DSECTs described by D
    MVC D3E.F1,D3E.F3       Within mid-level DSECT D3E
    MVC D1E.F3,D2E.F1       Across mid-level DSECTs D1E, D2E

* Move DSECTs named D within E
    MVC D1,D2               Across top-level DSECTs D1, D2
```

HLASM Type Checking

HLASM Structured Programming – Type checking - TYPECHECK

- HLASM provides a number of type-checking facilities to assist make programs safer to change
- HLASM type-checking is provided by the TYPECHECK option:



Notes:

1 Choose at least one option.

HLASM Structured Programming – Type checking - TYPECHECK

- MAGNITUDE
  - Causes HLASM to perform magnitude validation of signed immediate data fields, e.g.:

```
000000 A718 FFFF            0FFFF     7          lhi   1,x'ffff'
 ** ASMA320W Immediate field operand may have incorrect sign or magnitude
```

- REGISTER
  - Causes HLASM to perform type checking of register fields of machine instruction operands, e.g.:

```
                      00001           7 fpr1     equ   1,,,,fpr
000000 A718 FFFF            0FFFF     8          lhi   fpr1,x'ffff'
** ASMA323W Symbol fpr1 has incompatible type with general register field
```

- NOTYPECHECK
  - Turns off all type-checking

HLASM Structured Programming – Type checking – User types

- The DC and DS statements allow users to create their own “Program Types” using the P parameter, e.g.:

```
my_data dc cap(c'read')l3'this is my data'
```

- The value of the *program type* is returned via the SYSATTRP macro function
- The value of the *assembler type* is returned via the SYSATTRA macro function
- The value of the type *attribute* is returned via the macro T' operator

```
 &d_type   setc  sysattrp('my_data1')
           mnote *,'SYSATTRP is --> &d_type'
+*,SYSATTRP is --> read
 &d_type   setc  sysattra('my_data1')
           mnote *,'SYSATTRA is --> &d_type'
+*,SYSATTRA is --> CA
 &d_type   setc  t'my_data1
           mnote *,'SYSATTRA is --> &d_type'
+*,SYSATTRA is --> C
 my_data1 dc    cap(c'read')l3'mr smith'
```

HLASM Structured Programming – Type checking – EQU

- The EQU statement can be used to:
  - Specify a 32-bit value used as the program type (4th operand of EQU)
  - Specify a register type (5th operand of EQU)
- The program type for an EQU symbol will be returned via the SYSATTRP function as for macro symbols
- The register types that are available to use are:
  - AR – Access registers
  - CR, CR32, CR64 – Control registers
  - FPR – Floating point registers
  - GR, GR32, GR64 – General purpose registers
  - VR – z13 vector registers
- Note that there is no vector register type-checking on pre-z/Architecture vector instructions
  - The hardware hasn't been available to use for a while now...don't use them!!

HLASM Structured Programming

Program Objects

HLASM Structured Programming – Program Objects

- *Program Objects* are a newer form of the z/OS *load module* which reside in PDSEs.
- There are 3 ways of creating program objects:
  - Use the GOFF assembler option. This will generate default classes causing the binder to store the linked program object into a PDSE.
  - Use the GOFF assembler option and create your own classes.
  - Store the output of binding a program into a PDSE. The binder will create default classes and create the program object for you.
- Program objects have advantages over load modules such as:
  - Can be stored in PDSE and zFS / HFS
  - Multiple dimensions and multiple A/RMODEs in the same module
  - Limited to 1GB in size – load modules are limited to 16MB in size
  - External symbols can have up to 32K characters rather than 8
  - Open-ended architecture – use the Binder interfaces to get to the data

HLASM Structured Programming – Program Objects

- A load module consists of a number of control sections whereas a PO is a collection of *classes* and *sections*.
  - A *class* is an independently loadable module of data. The class itself also has various attributes to determine whether the data is to be loaded, reusable, etc.
  - A *section* is a collection of data from one or more classes.
  - An *element* is an intersection between a class and a section. The attributes of an element are defined by the class for that element's particular section.

MYPROG	Class X	Class Y	Class Z
Section A	Element	Element	Element
Section B	Element	Element	Element
Section C	Element	Element	Element

HLASM Structured Programming – Program Objects

- The CSECT instruction is now used to create a section to which a number of (default) classes belong. The classes which are created by HLASM are:
  - B_IDRL – used to contain IDR (identification data) for each section
  - B_PRV – used to contain any external dummy section data for a section
  - B_TEXT – used to contain a section's object code
- Each of these classes has an *element definition* (ESD symbol type ED) whose owning ID is the ESDID of the section.
- The CSECT instruction will also generate a *label definition* (ESD symbol LD) to be an entry point for the section and will refer to the B_TEXT class for that section.

IBM

HLASM Structured Programming – Program Objects

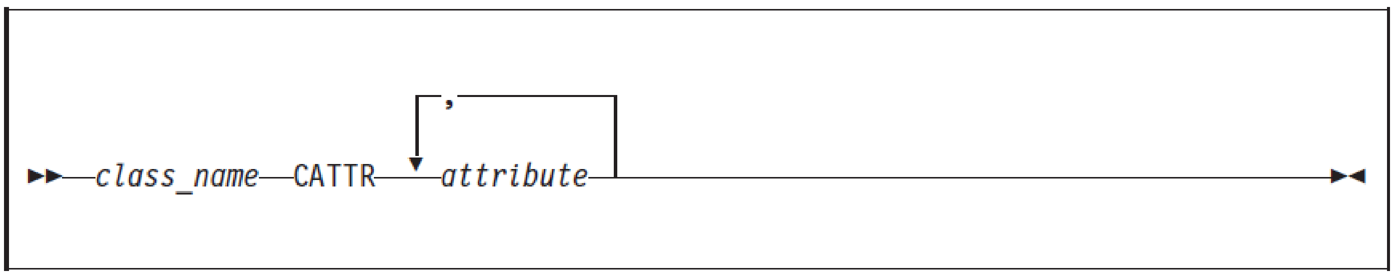
```
                              External Symbol Dictionary

Symbol    Type    Id      Address   Length    Owner Id Flags Alias-of
LISTINGB   SD 00000001
B_IDRL     ED 00000002                         00000001
B_PRV      ED 00000003                         00000001
B_TEXT     ED 00000004 00000000 00000084   00000001   08
LISTINGB   LD 00000005 00000000             00000004   08
EXTERNAL_FUNCTION
           ER 00000006                         00000001
FUNCY      ER 00000007                         00000001
listme     ER 00000008                         00000001       LISTINGZ
COMMON_DATA
           SD 00000009
B_IDRL     ED 0000000A                         00000009
B_PRV      ED 0000000B                         00000009
B_TEXT     ED 0000000C 00000000 00000018   00000009   00
COMMON_DATA
           CM 0000000D 00000000             0000000C   00
```

HLASM Structured Programming – Program Objects

```
                              External Symbol Dictionary

Symbol    Type    Id       Address  Length    Owner Id Flags Alias-of
LISTINGB   SD 00000001
B_IDRL     ED 00000002                        00000001
B_PRV      ED 00000003                        00000001
B_TEXT     ED 00000004 00000000 00000084   00000001    08
LISTINGB   LD 00000005 00000000            00000004    08
EXTERNAL_FUNCTION
           ER 00000006                        00000001
FUNCY      ER 00000007                        00000001
listme     ER 00000008                        00000001       LISTINGZ
COMMON_DATA
           SD 00000009
B_IDRL     ED 0000000A                        00000009
B_PRV      ED 0000000B                        00000009
B_TEXT     ED 0000000C 00000000 00000018   00000009    00
COMMON_DATA
           CM 0000000D 00000000            0000000C    00
```

HLASM Structured Programming – Program Objects

- Classes can be created by using the CATTR instruction.



- Some of the attributes that can be specified are:
  - ALIGN(*n*) – align the class on a 2^n boundary (values of *n* are 0, 1, 2, 3, 4, 12)
  - READONLY – text is storage protected
  - REFR, RENT, REUS – same as for load modules but for just a class
  - RMODE(24|31|ANY)
  - PART(name) – a named subdivision of an element that can have an internal structure. The ESD type for a part is PD.

- The default attributes for a class are:
  - ALIGN(3), EXECUTABLE, NOTREUS, RMODE(24)

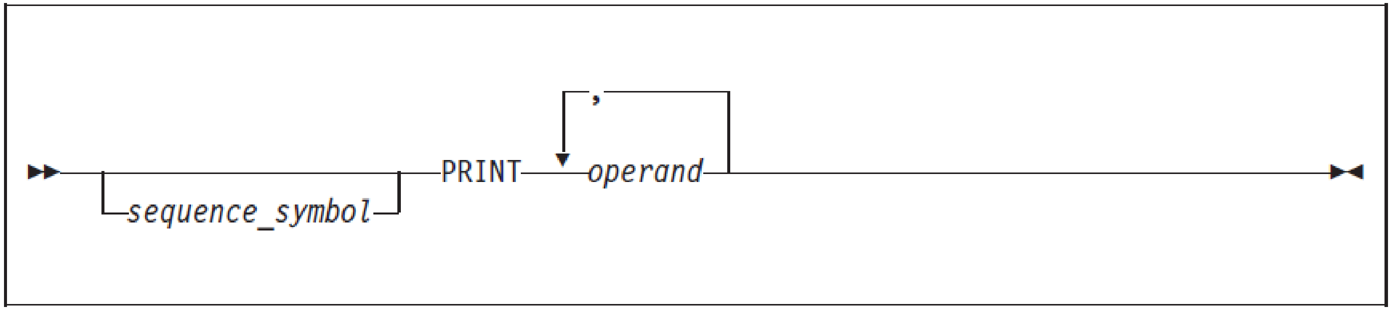
Controlling Listings

HLASM Structured Programming – Controlling Listings

- HLASM listings contain a large amount of very useful information but may easily become daunting to look at. Fortunately, HLASM has some useful instructions to help manage the data is presented in the listing.
- TITLE – Specifies headings for each page of the source and object section of the listing. Using the TITLE instruction also causes the page header to be printed.
- EJECT – Causes the next line of assembler listing to be printed at the top of a new page.
- SPACE – Inserts one or more blank lines into the listing on the current page.

HLASM Structured Programming – Controlling Listings

- PRINT – Controls the amount of detail printed in the listing.



  - ON|OFF – Print or stop printing the source and object section of the listing
  - GEN|NOGEN – Print or stop printing macro expansion. Note that the MNOTE instruction always causes a message to be printed.
  - NODATA|DATA – Print only the first 8-bytes of a data constant in the listing. If PRINT DATA is used then the entire constant is printed in the object code.
  - NOMCALL|MCALL – Control whether or not nested macro calls are printed
  - MSOURCE|NOMSOURCE – Control whether or not the printing of source statements produced from macro processing are printed
  - UHEAD|NOUHEAD – Control whether or not to print the ACTIVE USINGS at the start of each page of listing.

HLASM Structured Programming

Structured Programming Macros

HLASM Structured Programming – Structured Programming Macros

- HLASM provides a set of *Structured Programming Macros* (SPMs) as part of the HLASM Toolkit feature
- The SPMs provide
  - Branching structures (if) – IF, ELSEIF, ELSE, ENDIF
  - Looping structures (Do) – DO, ITERATE, DOEXIT, ASMLEAVE, ENDDO
  - Searching – STRTSRCH, ORELSE, ENDLOOP, ENDSRCH, EXITIF
  - N-way branching – CASENTRY, CASE, ENDCASE
  - Selection on general sequential cases – SELECT, WHEN, NEXTWHEN, OTHERWISE, ENDSEL
- Why use SPMs?
  - Improve code readability, maintainability, understandability
  - Faster application development
  - Cleaner code
  - Eliminates extraneous labels – makes it easier to revise

IBM

HLASM Structured Programming – Structured Programming Macros

- Ever seen some code like this?

```
CHK1     ICM  2,15,RETCODE   RETURN CODE 0?
         BC   8,RETOK        BRANCH TO RETURN OK
         SLL  2,2            MAKE BRANCH OFFSET
         ICM  3,15,RESN      RETURN NOT 0 AND REASON PRESENT?
         BC   7,RESTAB(2)    BRANCH INTO REASON TABLE
CHK2     L    15,CHKRES      CALL CHECKRES SUBROUTINE
         BALR 14,15
         LTR  15,15          RETURN OK?
         BC   8,CHK1         GOTO REPEAT CHECKS
         BC   1,RETOK        RETURN WILL DO
         BC   4,CHK1         LOOKUP REASON
RETOK    C    2,=F'3'        WAS RETURN LESS THAN 3?
         BC   4,CHKERRL4     YES
         LA   3,0
         BC   15,CONT242     CONTINUE CODE
CHKERRL4 L    15,OMINERR     OUTPUT MINOR ERROR
CONT242  . . .
```

HLASM Structured Programming – Structured Programming Macros

- Ever seen someone make a fix like this?

```
CHK1     ICM  2,15,RETCODE   RETURN CODE 0?
         BC   8,RETOK        BRANCH TO RETURN OK
         SLL  2,2            MAKE BRANCH OFFSET
CHK1B    ICM  3,15,RESN      RETURN NOT 0 AND REASON PRESENT?
         BC   7,RESTAB(2)    BRANCH INTO REASON TABLE
CHK2     L    15,CHKRES      CALL CHECKRES SUBROUTINE
         BALR 14,15
         LTR  15,15          RETURN OK?
         BC   8,CHK1         GOTO REPEAT CHECKS
         BC   1,RETOK        RETURN WILL DO
         BC   4,CHK1B        LOOKUP REASON
RETOK    C    2,=F'3'        WAS RETURN LESS THAN 3?
         BC   4,CHKERRL4     YES
         LA   3,0
         BC   15,CONT242     CONTINUE CODE
CHKERRL4 L    15,OMINERR     OUTPUT MINOR ERROR
CONT242  . . .
```

HLASM Structured Programming – Structured Programming Macros

- Maybe the previous slides were a contrived example...you'd be surprised...
  - CHK1B might be an easy fix for now but what will the code be like when we reach CHK9C ... We've all seen it happen...
- Why does assembler code have a reputation for always being an unmaintainable mess?
  - Code gets maintained rather than developed...
  - Documentation gets lost...
  - Programmer's make bad choices...
- What can we do to fix this?
  - Structure your code
  - Don't call everything tmp1, tmp2, temp1, temp2, etc.
  - Use HLASM 's facilities to help you

HLASM Structured Programming – Structured Programming Macros

- To use – just copy in the ASMMSP copybook
  - By default, the SPMs produce based branch on condition instructions. To cause the SPMs to produce relative branch instructions, use the ASMMREL macro:

```
ASMMREL ON
```

- Global variables used by the macros begin with &ASMA_
- User-visible macros have meaningful mnemonics
  - Internal non-user macros begin with ASMM

HLASM Structured Programming – SPMs - IF...

- Provides simple selection for a given condition or instruction
- Multiple forms of IF statement:
  - IF (*condition*)
  - IF (*instruction, parm1,parm2,condition*)
  - IF (*compare instruction,parm1,condition,parm2*)
  - IF CC=*condition_code*
- Conditions may be joined together with Boolean operators
  - AND
  - OR
  - ANDIF
    - A OR B AND C → A OR (B AND C) *but* A OR B ANDIF C → (A OR B) AND C
  - ORIF
    - A AND B ORIF C OR D → (A AND B) OR (C OR D)
  - NOT

HLASM Structured Programming – SPMs - IF...

- Provides simple selection for a given condition or instruction
- Multiple forms of IF statement:
  - IF (*condition*)

```
...
LT R1,CUSTOMER_NUMBER
IF (Z) THEN
    Code to assign a new customer number
ELSEIF (N) THEN
    Code for negative customer number - error
ELSE
    Look up the customer and do something useful
ENDIF
...
```

HLASM Structured Programming – SPMs - IF...

- Provides simple selection for a given condition or instruction
- Multiple forms of IF statement:
  - IF (*condition*)

```
...
LT R1,CUSTOMER_NUMBER
IF (Z) THEN
    Code to assign a new customer number
ELSEIF (N) THEN
    Code for negative customer number - error
ELSE
    Look up the customer and do something useful
ENDIF
...
```

THEN is actually a comment – not needed

HLASM Structured Programming – SPMs - IF...

- Provides simple selection for a given condition or instruction
- Multiple forms of IF statement:
  - IF (*instruction, parm1,parm2,condition*)

```
...
IF (LT,R1,CUSTOMER_NUMBER,Z) THEN
    Code to assign a new customer number
ELSEIF (N) THEN
    Code for negative customer number - error
ELSE
    Look up the customer and do something useful
ENDIF
...
```

HLASM Structured Programming – SPMs - IF...

- Provides simple selection for a given condition or instruction
- Multiple forms of IF statement:
  - IF (*compare instruction,parm1,condition,parm2*)

```
...
if     (cli,x_val,eq,X'55'),OR,  Value is x55?          X
       (cli,x_val,eq,X'5D'),OR,  Value is x5D?          X
       (cli,x_val,eq,X'51'),OR,  Value is x51?          X
       (cli,x_val,eq,X'59')      Value is x59?
    Code goes here...
endif
...
```

HLASM Structured Programming – SPMs - IF...

- Provides simple selection for a given condition or instruction
- Multiple forms of IF statement:
  - IF (*compare instruction,parm1,condition,parm2*)

```
...
if     (cli,x_val,eq,X'55'),OR,  Value is x55?        X
       (cli,x_val,eq,X'5D'),OR,  Value is x5D?        X
       (cli,x_val,eq,X'51'),OR,  Value is x51?        X
       (cli,x_val,eq,X'59')      Value is x59?
    Code goes here...
endif
...
```

Remember to continue statements

HLASM Structured Programming – SPMs - IF...

- The following are the condition mnemonics permitted by the SPMs

Case	Condition Mnemonics	Meaning	Complements
After compare instructions	H, GT L, LT E, EQ	Higher, greater Less than Equal	NH, LE NL, GE NE
After arithmetic instructions	P M Z O	Plus Minus Zero Overflow	NP NM NZ NO
After test under mask instructions	O M Z	Ones Mixed Zero	NO NM NZ

HLASM Structured Programming – SPMs - IF...

- Provides simple selection for a given condition or instruction
- Multiple forms of IF statement:
  - IF CC=*condition code*

```
...
L      R15,TRREME
TRT    0(1,R6),0(R15)
IF     CC=0,OR,CC=2          Did the operation complete?
    Yes - let's examine register 2 for TRT result...
ELSE
    Data was examined but there is still more to go...
    Examine the result and re-drive the instruction
ENDIF
...
```

HLASM Structured Programming – SPMs – DO...

- Provides iterative execution
- Multiple forms of DO statement:
  - DO ONCE/INF...
  - DO FROM=(Rx,i),TO=(Ry+1,j),BY=(Ry,k)...
  - DO WHILE=(*condition*)...
  - DO UNTIL=(*condition*)...
- Loop control keywords
  - DOEXIT – uses IF-macro style syntax to exit the current DO or any containing labeled DO
  - ASMLEAVE – unconditionally exit the current DO or any containing labeled DO
  - ITERATE – requests immediate execution of the next loop iteration for the current or any containing labeled DO

HLASM Structured Programming – SPMs - DO...INF

- Provides iterative execution
- Multiple forms of DO statement:
  - DO INF → infinite loop

```
...
* Count the number of customers
   XGR     R2,R2                         Clear the counter
   DO      INF
      LT      R3,CUSTOMER_NUMBER(R2) Get the next customer
      DOEXIT (Z)                      Exit if at end
      ALGFI  R2,L'CUSTOMER_REC        Increment counter
   ENDDO
...
```

HLASM Structured Programming – SPMs - DO...FROM

- Provides iterative execution
- Multiple forms of DO statement:
  - DO FROM → Counted loop – note that this generates a BCT

```
...
* Output only top 10 customers
    DO      FROM=(R1,10)
        L       R3,CUSTOMER_NUMBER(R1) Get customer
        OUTPUT_CUSTOMER CUST=R3        Call subroutine
    ENDDO
...
```

HLASM Structured Programming – SPMs - DO...

- Provides iterative execution
- Multiple forms of DO statement:
  - DO FROM → Counted loop – note that this generates a BXH

```
...
XC   FLAG,FLAG                                    Clear process flag
IF   (CLI,INPUT_RECORD,NE,C'*')                   Not a comment?
     DO       FROM=(R1,1),TO=(R5,72),BY=(R4,1)
          LA   R2,INPUT_RECORD(R1)                Get next character
          ...
          IF   (CLC,0(5,R2),EQ,=C' END '),AND,                  X
               (CLI,FLAG,EQ,2)
               ...                             Process the END record
               ASMLEAVE                        Leave the process loop
          ENDIF
     ENDDO
ENDIF
...
```

HLASM Structured Programming – SPMs – DO...Backwards...

- Provides iterative execution
- Multiple forms of DO statement:
  - DO FROM → Counted loop – note that this generates a BXH

```
...
XC   FLAG,FLAG                                      Clear process flag
IF   (CLI,INPUT_RECORD,NE,C'*')                     Not a comment?
     DO      FROM=(R1,1),TO=(R5,72),BY=(R4,-1)
          LA   R2,INPUT_RECORD(R1)                  Get next character
          ...
          IF   (CLC,0(5,R2),EQ,=C' END '),AND,                         X
               (CLI,FLAG,EQ,2)
               ...                               Process the END record
               ASMLEAVE                          Leave the process loop
          ENDIF
     ENDDO
ENDIF
...
```

HLASM Structured Programming – SPMs – SELECT...

- Provides selective execution similar to an IF

```
...
    SELECT
        WHEN  (CLI,PREFIX,EQ,C'+')
            MVC OUT_BUFFER(L'STMT_MACROEXP),STMT_MACROEXP
        WHEN  (CLI,PREFIX,EQ,C'=')
            MVC OUT_BUFFER(L'STMT_COPYBOOK),STMT_COPYBOOK
        WHEN  (CLI,PREFXI,EQ,C' ')
            MVC OUT_BUFFER(L'STMT_NORMAL),STMT_NORMAL
        OTHERWISE
            BAL R14,ERROR_ROUTINE  Listing isn't correct...
            J   EXIT_ROUTINE
    ENDSEL
* Output the type of statement encountered
...
```

Summary

- Control sections – C-, R-, DSECTs, External dummy sections
  - CSECT, RSECT, DSECT, DXD, COM
- Subroutines
  - EXTRN / WXTRN
  - Conditional Sequential RLDs (CSRs)
- The location counter and USING statements
  - LOCTR
  - Labeled USINGs
  - Dependent USINGs
- Program Objects
- Controlling Listings
- Structured Programming Macros (SPMs)
  - IF...ELSEIF...ELSE...ENDIF
  - DO WHILE / UNTIL / FROM...ENDDO
  - SELECT...WHEN...OTHERWISE...ENDSEL
  - ASMLEAVE, DOEXIT

Where can I get help?

- z/OS V2R1 Elements and Features
  http://www.ibm.com/systems/z/os/zos/bkserv/v2r1pdf/#IEA
- HLASM Publications
  http://www.ibm.com/systems/z/os/zos/library/bkserv/v2r1pdf/#ASM
- HLASM Programmer's Guide (SC26-4941-06)
  http://publibz.boulder.ibm.com/epubs/pdf/asmp1021.pdf
- HLASM Language Reference (SC26-4940-06)
  http://publibz.boulder.ibm.com/epubs/pdf/asmr1021.pdf
- HLASM Toolkit Features User's Guide (GC26-8710-10)
  http://publibz.boulder.ibm.com/epubs/pdf/asmtug21.pdf
- z/Architecture Principles of Operation
  http://www.ibm.com/support/docview.wss?uid=isg2b9de5f05a9d57819852571c500428f9a