z/OS Assembler Programming Part 2: Interfaces

z/OS Assembler Programming Part 2: Interfaces - Course Objectives

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

1. Follow classic z/OS conventions regarding save area chaining and the passing and receiving of parameters

2. Code or maintain Assembler programs that handle sequential files, using QSAM to read, write, and update records

3. Write programs to handle variable length records using QSAM

4. Debug most program ABENDs, using z/OS full dumps or symptom dumps to track down problems

5. Write mainline programs and subroutines; use the Program Binder to combine mainline and subroutine programs

6. Use the Binder to maintain load modules by replacing existing CSECTs with new versions of these CSECTs

7. Use the WTO, SNAP, and TIME macros

8. Use Dynamic Serial linkages (using LINK, LOAD, DELETE, XCTL) to invoke subroutines

9. Use various other system services (GETMAIN, FREEMAIN, STCKCONV, CONVTOD)

10. Create reentrant programs

11. Perform I/O against QSAM files while running in AMODE 31.

Note: this course focuses on AMODE 24 and AMODE 31 interfaces. It is a prerequisite to course code C510, "z/OS Assembler Programming: z/Architecture and z/OS" which covers the AMODE 64 interfaces (and lots more).

z/OS Assembler Programming Part 2: Interfaces - Topical Outline

Day One

z/OS Assembler Programming Part 2: Interfaces - Topical Outline, 2

Day Two

z/OS Assembler Programming Part 2: Interfaces - Topical Outline, 3

Day Three

Section Preview

- ❒ **Program Linkages**

    Control Sections

    Save Areas

    Addressability

    Return Codes

    Typical Linkages

    SAVE and RETURN Macros

    Getting the PARM from the EXEC Statement

Control Sections

- ❒ **Programs are organized into "chunks" of code (instructions and / or data areas called <u>Control Sections</u>, or <u>CSECT</u>s) that are the building blocks of the Linkage Editor and the Program Binder**

- ❒ **The beginning of a CSECT is indicated by the appearance of either a START or CSECT Assembler instruction:**

```
csectname  START  value
```

or

```
csectname  CSECT
```

Assembler Instructions

<u>Notes</u>

The *csectname* **must follow the rules for names in Assembler, with the furhter restriction that it may only be 8 characters long, maximum**

There may only be one START statement in a program; there may be any number of CSECT statements (although in this course we will normally have only one CSECT per program)

value **specifies a starting value for the Assembler's location counter (default: 0) in decimal or hex**

Each time a new CSECT statement is encountered, the Assembler sets that control section's location counter to 0 (zero)

END

❐ **A control section begins with a START or CSECT statement and continues until ...**

A new CSECT is begun

Or a DSECT is encountered

Or an END statement is encountered:

```
END   [starting-location]
```

Assembler Instruction

<u>Notes</u>

The END statement must be the last statement in your program: it denotes the end of the source module and any statements following it are discarded

starting-location **represents where in the program execution should begin when the program is actually run (the <u>Entry Point</u>)**

✗ The default ***starting-location*** is the first byte of the program

Save Areas

Source: Assembler Services Guide

- There is only one set of general purpose registers in a CPU, yet every program and subprogram needs to use these registers

- So, a convention has been established to allow any routine to use the registers when it needs to

    Each program provides a register save area (or just "save area")

    When a program is called by another program, the called program must save the registers of the calling program in the save area provided by the calling program

    Before the called program returns to the calling program, it must restore the calling program's registers

Register Save Area Layout

❒ **Save areas are 18 words (72 bytes), organized as folllows:**

+ 0	**Only used by PL/I**
+ 4	**↑Calling program's save area (backward pointer)**
+ 8	**↑Called program's save area (forward pointer)**
+12	**C(R14) - Return address to this program**
+16	**C(R15) - Entry point address of subroutine**
+20	**C(R0)**
+24	**C(R1) - Parameter list address**
+28	**C(R2)**
+32	**C(R3)**
+36	**C(R4)**
+40	**C(R5)**
+44	**C(R6)**
+48	**C(R7)**
+52	**C(R8)**
+56	**C(R9)**
+60	**C(R10)**
+64	**C(R11)**
+68	**C(R12)**

↑ means "points to" (that is, "contains the address of")

"C(R*nn*)" means "The contents of register '*nn*' "

Linkage Conventions

- **On entry to any program, the standard conventions expect the following general purpose register contents**

  - R1 - Address of list of parameter addresses (or zero if no parameters passed)
  - R13 - Address of register save area of calling program
  - R14 - Return address to calling program
  - R15 - Entry (starting) address of the called program

- **Similarly, when your program calls another program or routine, you are expected to set up the registers this way**

- **Note that these conventions work fine until you need to save all 64 bits of the general purpose registers**

  64-bit save area linkages are discussed in our course iwth course code C500, "z/OS Assembler Programming Part 4: z/Architecture and z/OS"

  But the vast majority of programs get along fine with these conventions, which assume 32-bit register values are all that's important

Return Codes

- When a subroutine returns, another convention is that the calling routine will find a return code in R15

  Traditionally, a value of 0 means all went well

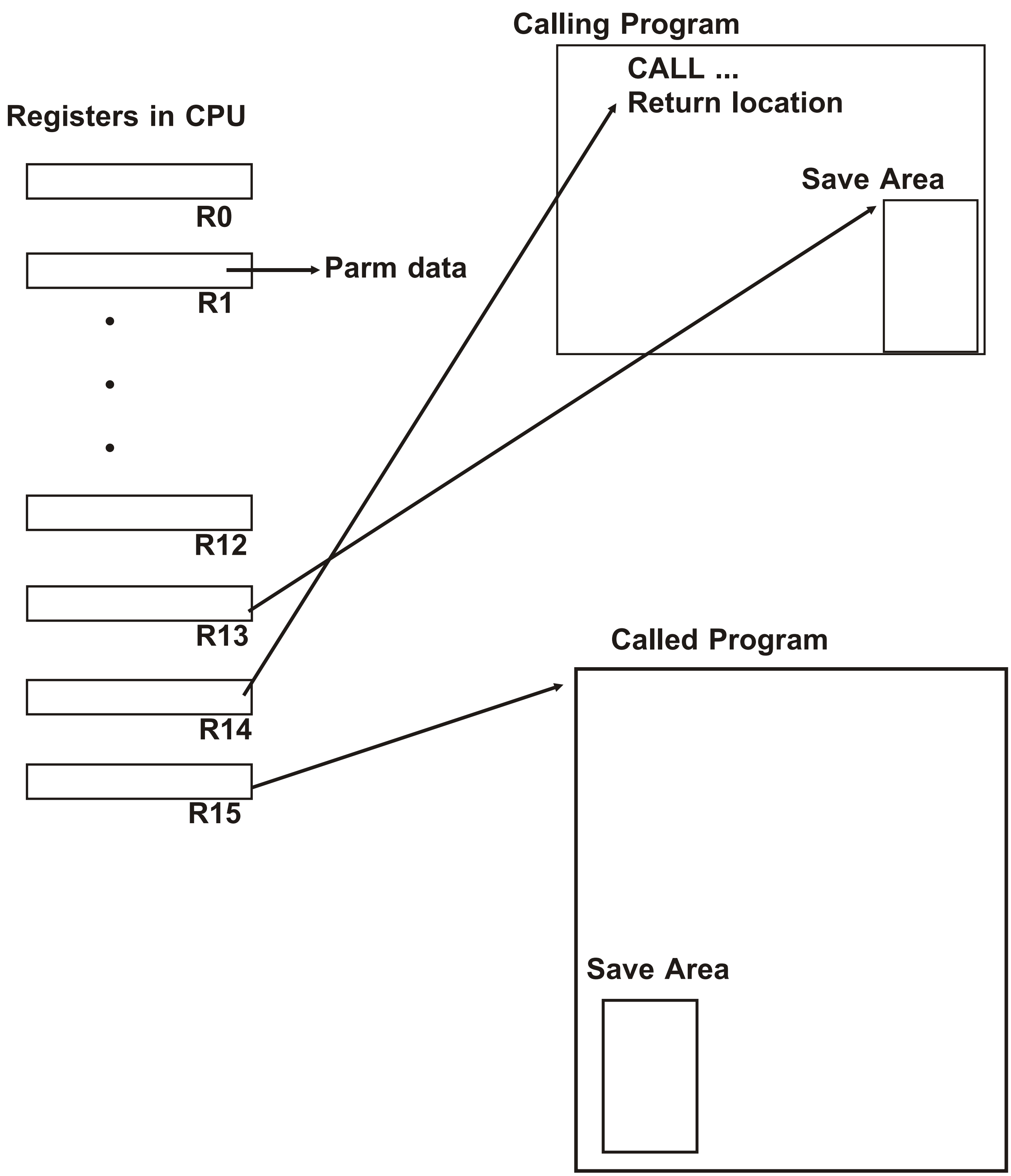
  Other values are often multiples of 4, with increasing severity of error meanings

  It doesn't have to be that way, however, and the meanings of return codes have to be agreed upon in advance by writers of the calling and called routine

- For a mainline program, the value in R15 is passed back so it may be tested by suceeding steps in the job, using the JCL COND parameter or the IF JCL statement

On Entry To A Called Program

❒ **Visually, the situation is this, just before a program gets control:**



Program Linkage On Entry

- ❒ Now, on entry to a program, the program must

  Save the calling program's registers in the calling program's save area

  Establish addressability

  Save the address of the calling program's save area in the called program's save area

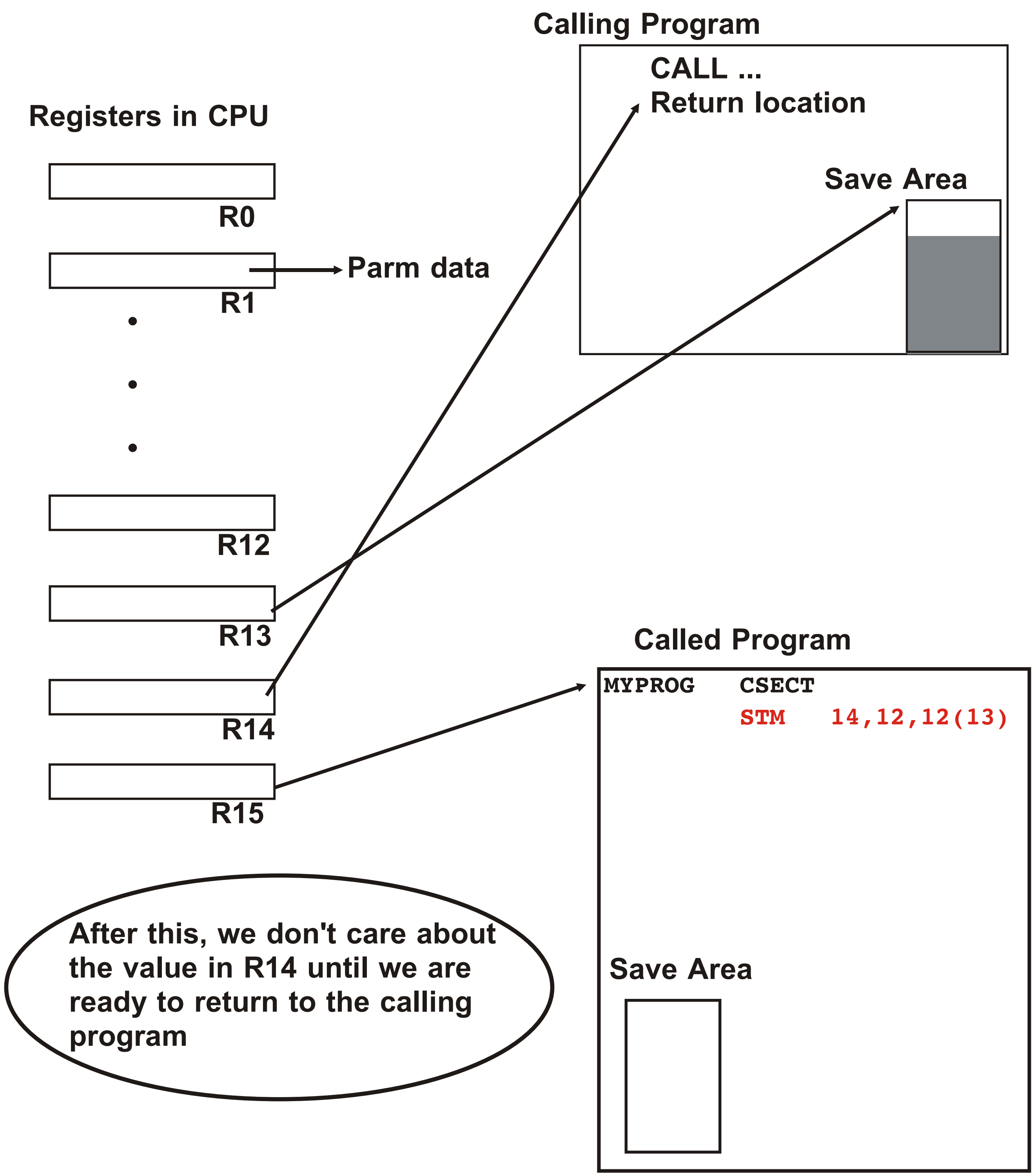
  Provide own save area, pointed at by R13

  Save address of program's save area in calling program's save area (Optional)

- ❒ Let's follow the process through ...

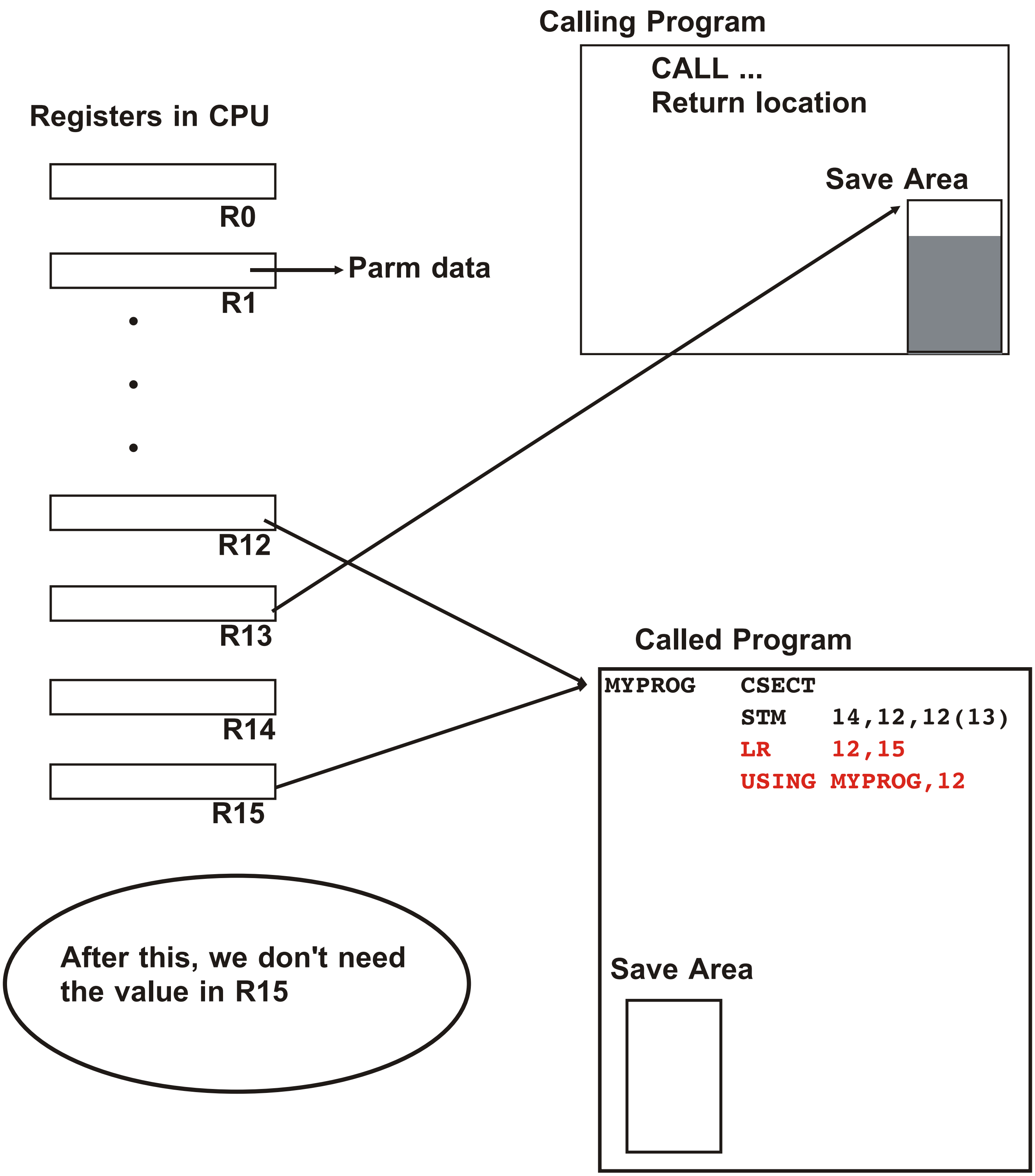
Save The Calling Program's Registers in the Calling Program's Save Area:

❒ STM sets the registers down in the correct order



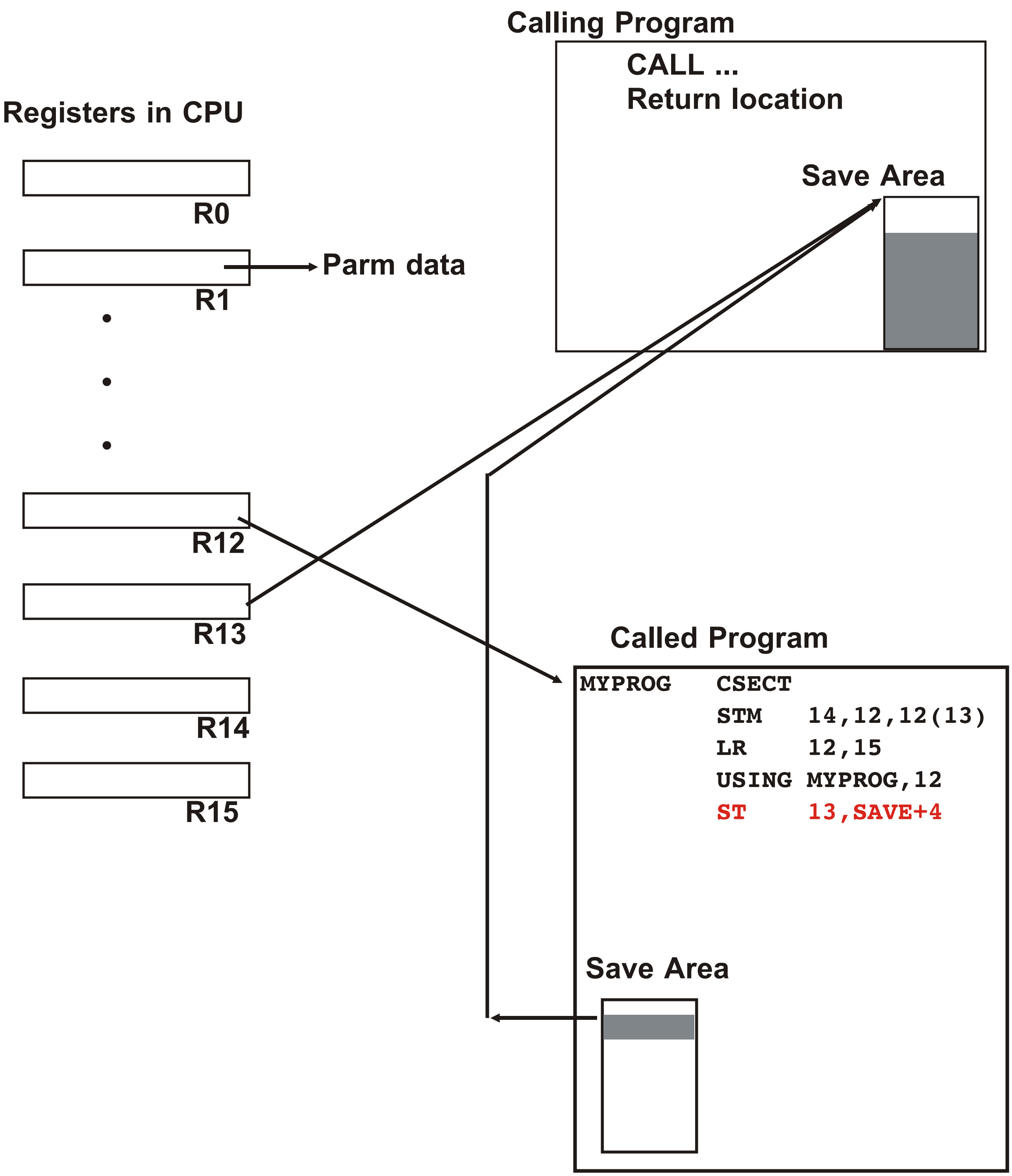
Establish Addressability:

- Use machine instruction (such as LR) and Assembler instruction (USING)



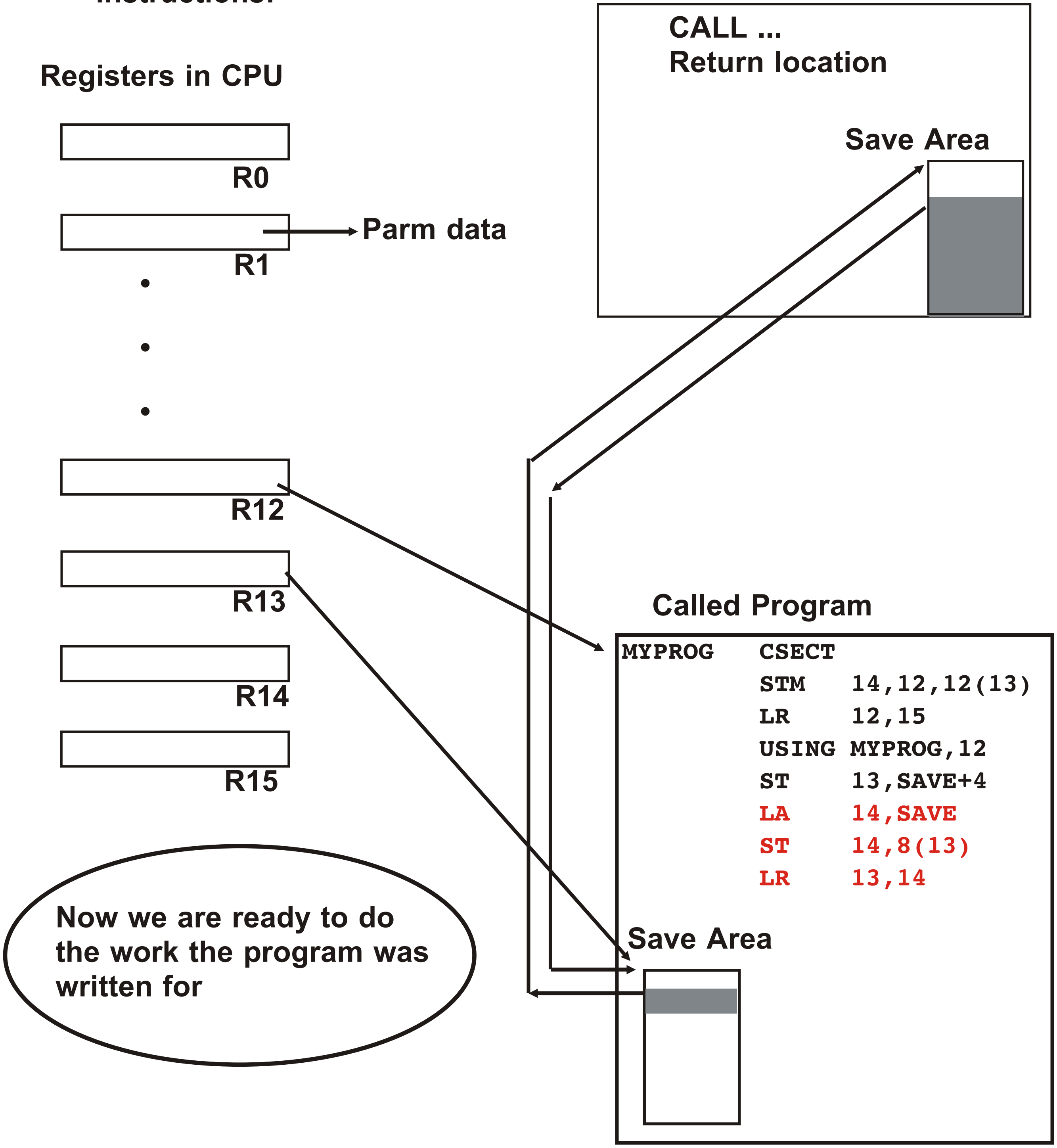
Save Pointer to Calling Program's Save Area:

❐ **The second word of our save area is available for that:**



Provide Own Save Area, Pointed at by R13 and Save Address of Program's Save Area in Calling Program's Save Area:

- One way to do this uses these instructions:



Program Linkage On Exit

- On exit, a program must

  Restore calling program's registers from calling program's save area

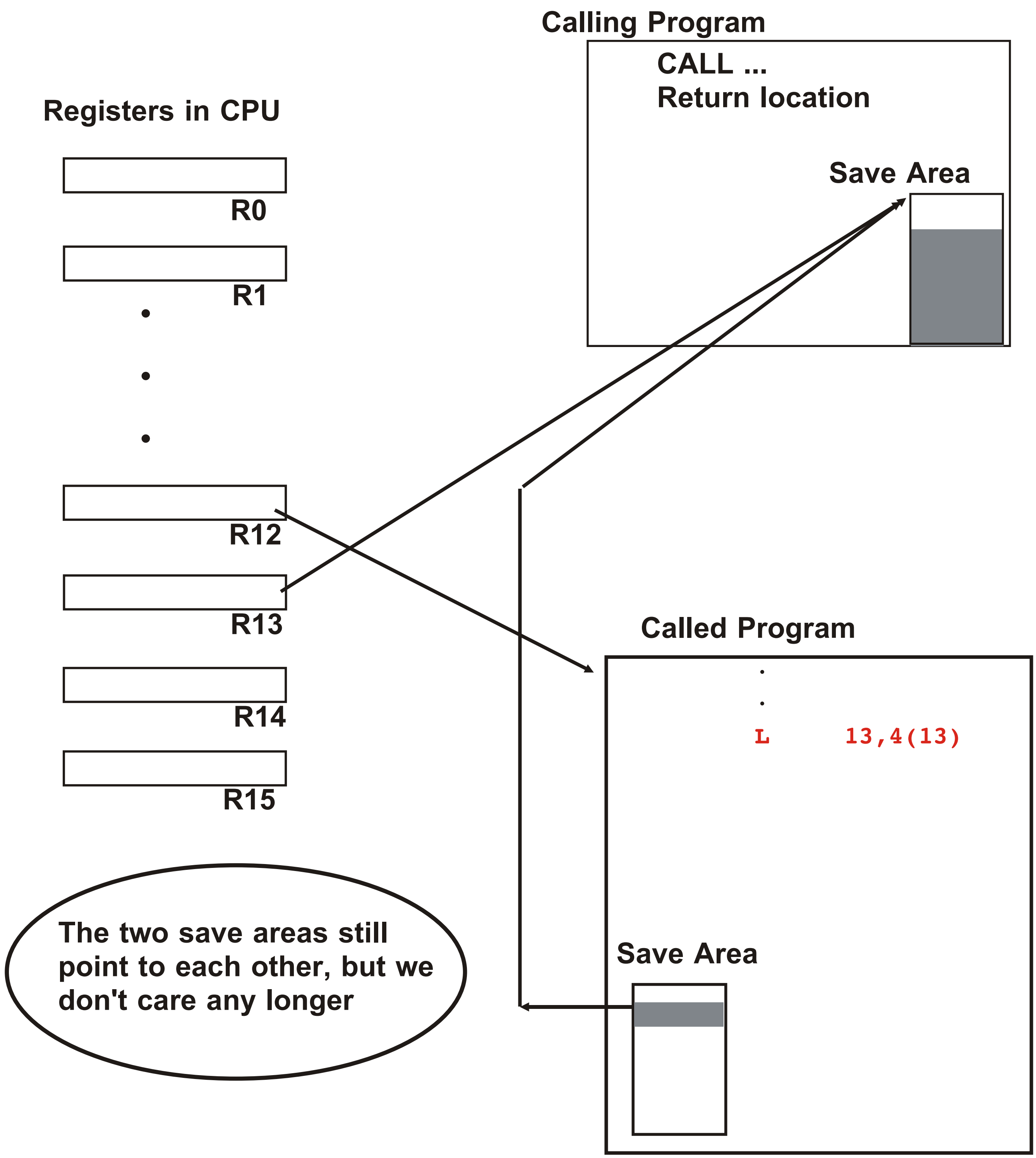
  Set a return code in R15 (optional)

  Branch to the address in R14

- Let's follow that process through, too ...

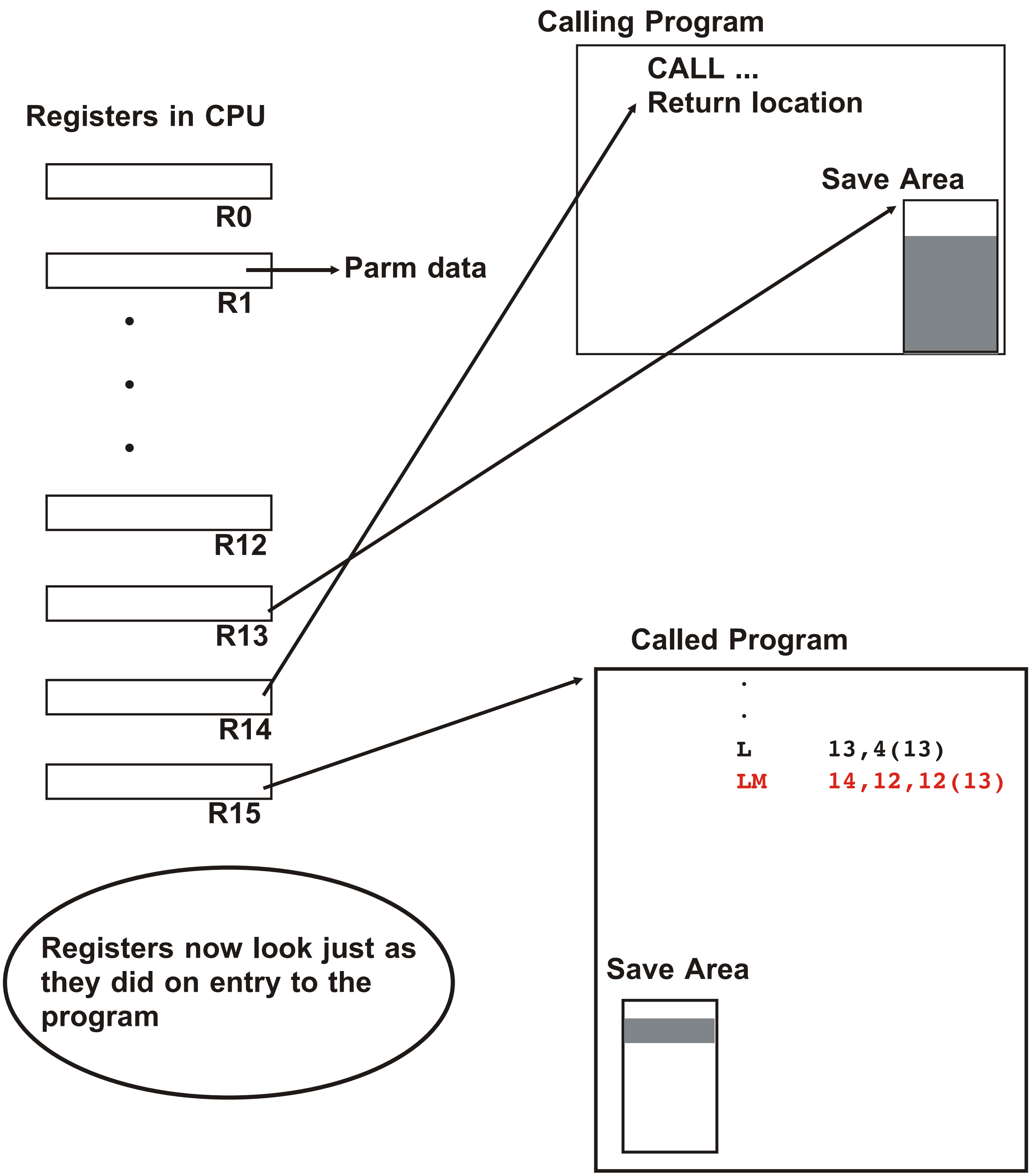
Pick Up Address of Calling Program's Save Area:

❒ This restores R13 to point to previous save area



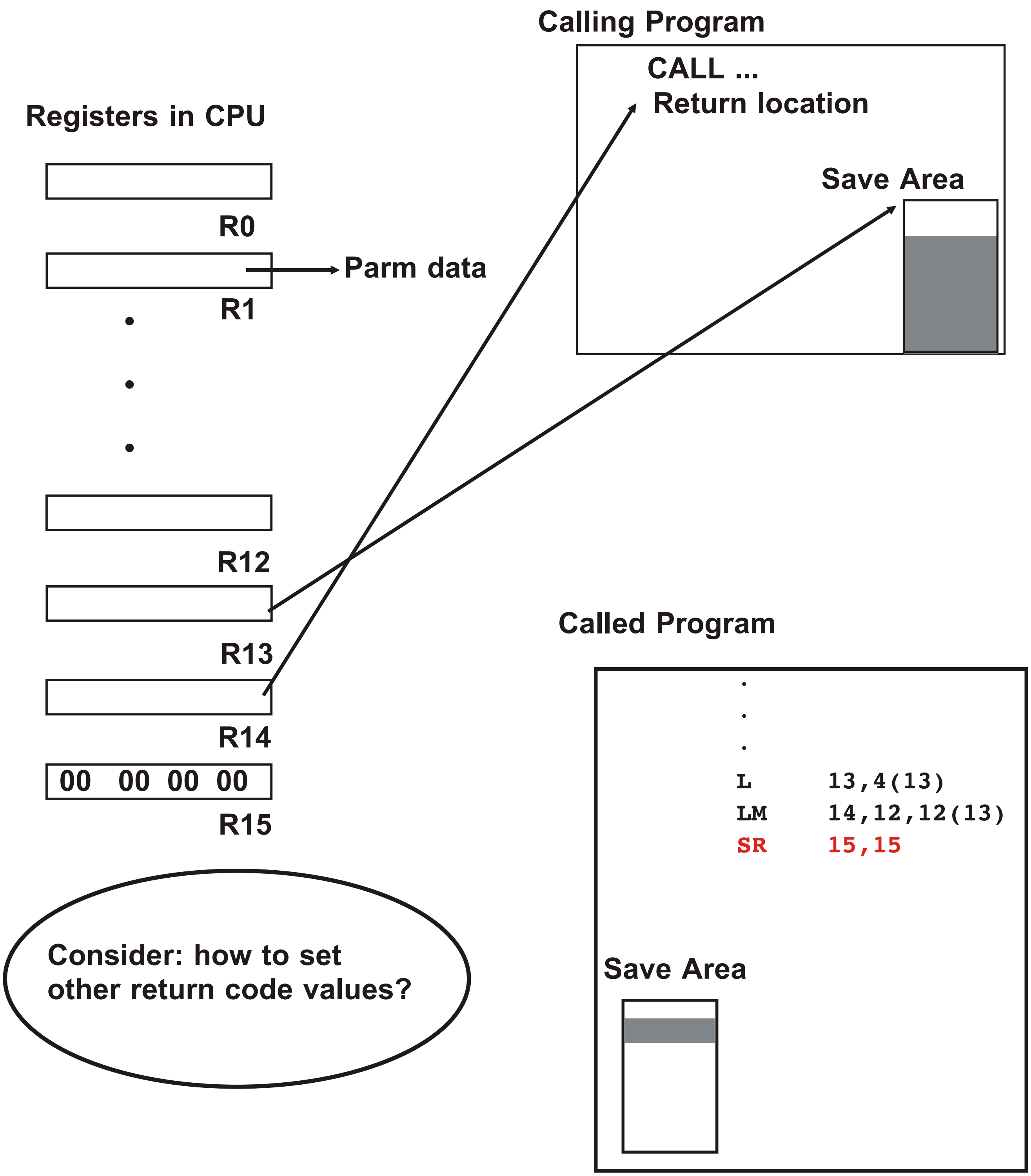
Restore Calling Program's Registers:

- ❐ Pick 'em up just the opposite way we put 'em down



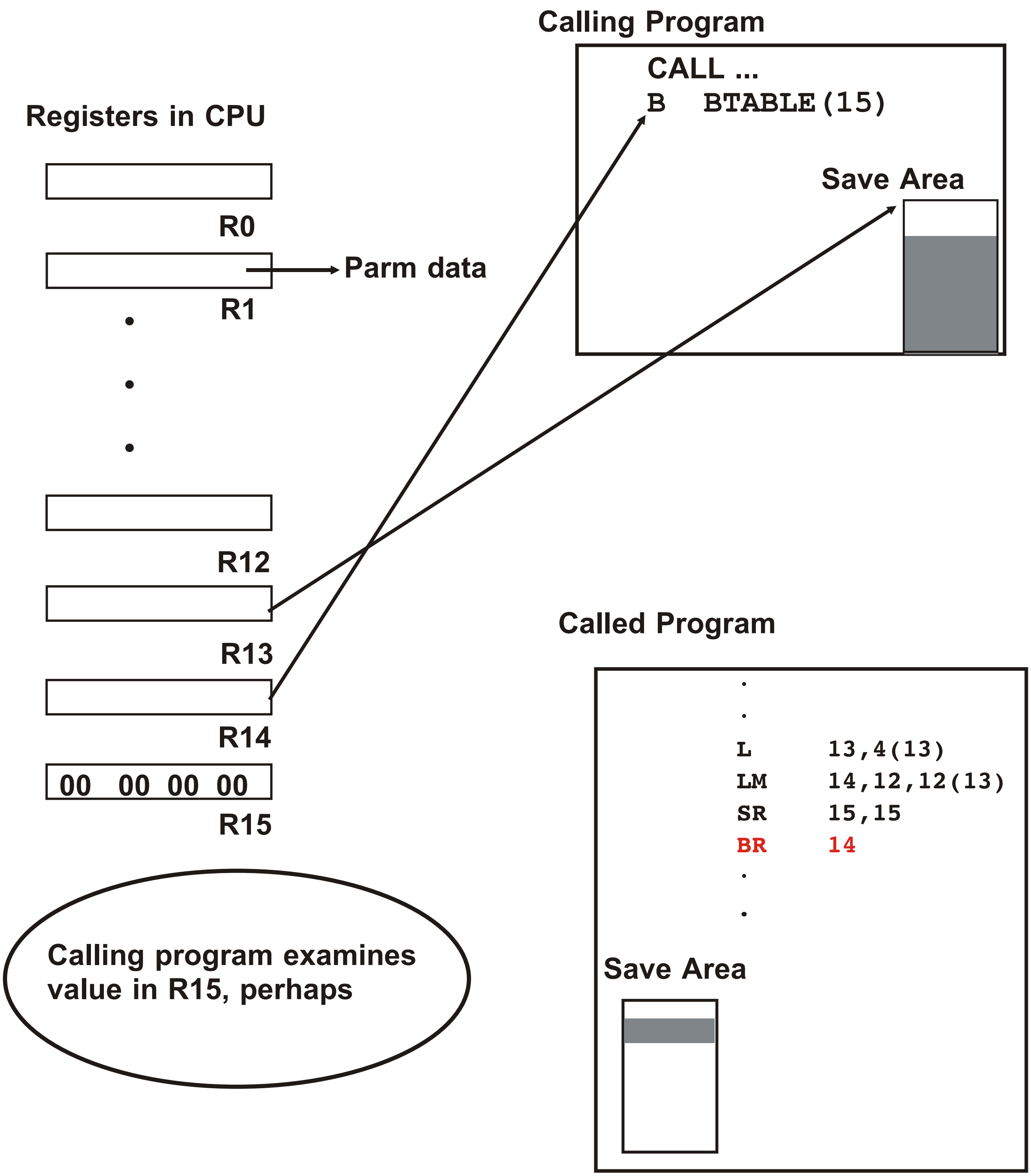
Set a Return Code in R15:

- In this example, we set a value of zero



Branch to Address in R14:

❒ This returns to the calling program



Typical Program Structure

❒ **The basic program linkages are illustrated here**

```
MYPROG   CSECT
         STM   14,12,12(13)   Save registers
         LR    12,15          Establish
         USING MYPROG,12         addressability
*  Save pointer to calling programs registers
         ST    13,SAVE+4     Store backward ptr
*  Point to own save area
         LA    14,SAVE
         ST    14,8(13)      Store foreward ptr
         LR    13,14         Establish own s.a.
**********************************************
         .
         .
         .
**********************************************
*  Pick up address of calling programs save area
         L     13,4(13)
         LM    14,12,12(13)   Restore registers
         SR    15,15          Return code = 0
         BR    14             Return to z/OS
**********************************************
*
*         Constants and data areas
*
***********************************************
SAVE     DC    18F'0'
         END   MYPROG
```

Services for Assembler Language Programs

- ❐ **IBM provides a large number of services that are available for application programs**

  A set of macros are provided to request some of these services from Assembler language programs

  A set of subroutines ("callable services") are provided to request the other services

- ❐ **These services are documented in these IBM publications:**

  MVS Programming: Assembler Services Reference, Volume 1 (ABE-HSP) and

  MVS Programming: Assembler Services Reference, Volume 2 (IAR-XCT)

  - ✗ These are the publications to use when looking up non-I/O-related services

- ❐ **Regarding macros, remember, continuation in Assembler requires:**

  Comma before column 72

  Non-blank character in column 72

  Continuation begins exactly in column 16

The SAVE Macro

Source: Assembler Services Reference, Vol. 2

Samples

```
SAVE  (14,12)

SAVE  (14,12),,'Entry to first routine'

SAVE  (14,12),,*
```

Working

Generates the STM instruction of standard linkage conventions

If third operand is specified, the macro generates a DC with the constant and a branch around the constant

- ✗ An asterisk (*) implies the constant to use is the name on the SAVE macro; if no name on the SAVE macro use the name of the current CSECT

The second operand is intended for non-standard register saving

- ✗ In particular, if you don't specify (14,12) in the first operand, coding a 'T' in the second operand ensures registers 14 and 15 are saved in the appropriate place in the save area; for example:

  ```
  SAVE    (3,7),T
  ```

- ✗ Not used much anymore, but you may see old code that uses this

The RETURN Macro

Samples

```
RETURN (14,12)
RETURN (14,12),,RC=n
RETURN (14,12),,RC=OK
RETURN (14,12),,RC=(15)
```

Working

Generates the LM and BR instructions

- ✗ But not the "L 13,4(13)"

If RC= operand specified, the macro generates the code to place return code in R15

- ✗ 'n' is an integer between 0 and 4095
- ✗ 'OK' is an example of using a symbol; 'OK' must be defined something like this:

```
OK          EQU    12
```

- ✗ If you code RC=(15), that says the return code is already in R15 and the RETURN macro generated code should not disturb it
  - ➢ Only Register 15 may be used in this way

Same remarks about the second operand as for SAVE

Standard Linkages Using SAVE and RETURN

❒ **Applying these new macros yields:**

```
MYPROG   CSECT
         SAVE   (14,12)       Save registers
         LR     12,15         Establish
         USING MYPROG,12         addressability
*  Save pointer to calling programs registers
         ST     13,SAVE+4    Store backward ptr
*  Point to own save area
         LA     14,SAVE
         ST     14,8(13)      Store foreward ptr
         LR     13,14        Establish own s.a.
**********************************************
         .
         .
         .
**********************************************
*  Pick up address of calling programs save area
*    and return to z/OS with a zero return code
         L      13,4(13)
         RETURN (14,12),,RC=0
**********************************************
*
*         Constants and data areas
*
***********************************************
SAVE     DC     18F'0'
         END    MYPROG
```

❒ **Most installations have their own home-grown linkage macros, usually named something like INIT, EXIT, ENTER, LEAVE, and so on**

Typically they also have options for establishing multiple base registers and other useful functions

Find out what your installation uses

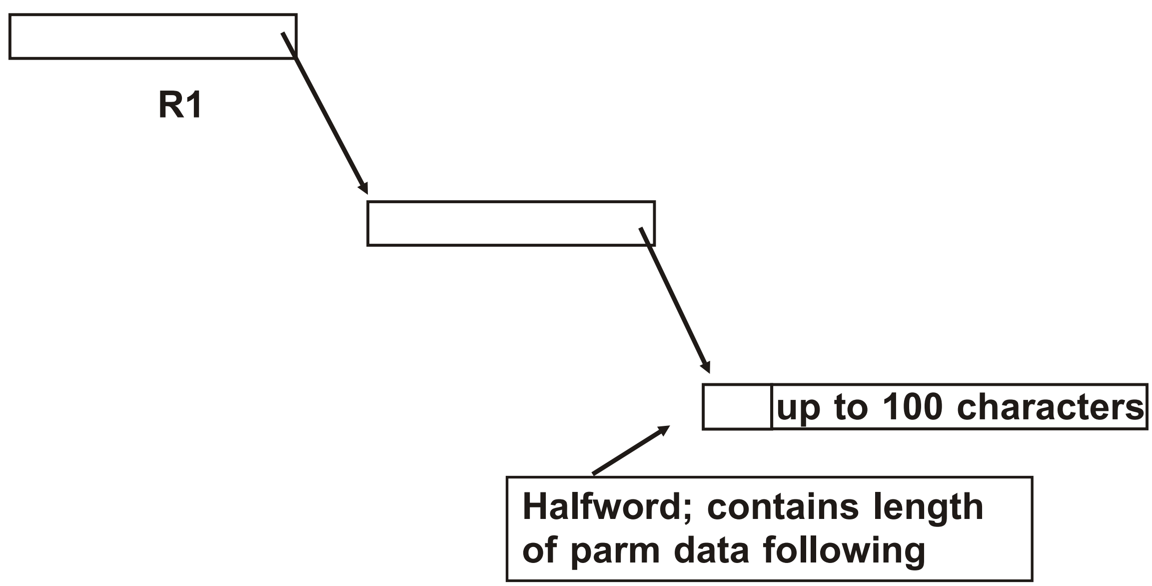
Gaining Access to the PARM Field

If program is invoked by:

```
//STEPX    EXEC  PGM=MYPROG,PARM='up to 100 characters'
```

At run time, program has access to the parm data

R1 points to a pointer to the data:



❐ **To get to the parm data, code something like:**

```
L     1,0(1)   Pick up addr of length
LH    2,0(1)   Pick up length
LA    3,2(1)   Pick up addr of data
```

Uses of the PARM Field

❒ **Once you have a pointer to the data, how can your program use it?**

This depends on the program design: you choose what the program expects to get

✗ Perhaps title information, processing switches, run-as dates, etc.

❒ **Techniques that might be useful in dealing with PARM data**

DSECTs

EX instruction (for working with variable length fields)

TRT instruction (to scan for particular characters)