Chapter 20: Subroutine Linkage

We now begin discussion of subroutine linkage, which is the way in which data and control are passed from a calling program to a called subprogram. We start by considering simple subroutines without large argument blocks. We should mention that the term **"subprogram"** can be used generically to mean **"subroutine"**, **"procedure"**, or **"function"**.

In general, the term **function** refers to a subprogram that returns a value that can be used as a part of an assignment statement. Common functions include the trigonometric functions and the mathematical ones, such as square root. Here is a function call in the style of FORTRAN.

```
Y = SIN(X)
```

In general, the term **subroutine** refers to a subprogram that may or not return a value, but usually not a value that can be used in an assignment statement typical of function calls. Some programming languages use the term **"procedure"** or **"function returning void"** to replace the older term **"subroutine"**. A good example of such a subroutine would be the following.

```
QUADRATIC (A, B, C, X1, X2)
```

We might specify that the above subroutine would produce the roots of the quadratic equation written in the form $A \bullet X^2 + B \bullet X + C = 0$. The arguments **A**, **B**, and **C** would be passed to the subroutine, which would then calculate the two roots and return them in **X1** and **X2**.

Subroutines and functions are normally written to facilitate repetitive tasks. Indeed the concept arose in the late 1940's and was named the **Wheeler Jump"** (EDSAC, 1952) after David J. Wheeler, who developed the concept while working as a graduate student. It is worth noting that, while many early computing machines were developed as exercises in engineering, the EDSAC was developed solely to serve as a platform for research in computer programming. The development of the subroutine was one of many innovations associated with the EDSAC.

Early operating systems began as collection of subroutines to facilitate handling of input and output devices, mainly line printers. Each programmer had to write his own (yes, they were almost exclusively male) I/O procedures. As these were tedious to write correctly, programmers began to share code; these quickly evolved into a shared subroutine library.

When batch programming became common, the computer needed a **control program** to automate the processing of a sequence of jobs, read and processed one after another. This control program was merged with a set of subroutines for Input/Output handling and routines for standard mathematical functions to create the first operating system.

Subroutines and/or functions can be invoked for side effect only or in order to compute and return results. Many of the subroutines common in assembler programming are called for side effect – to manage print position on a page and issue a new page command when necessary.

Subroutine Linkage: Instructions

The language requires two instructions associated with subroutines: one to call the subroutine and one to affect a return from the subroutine. The instruction used is either **BAL** (Branch and Link) or **BALR** (Branch and Link, Register). Each instruction stores a return address in a designated register, which is accessed by the **BR** instruction used to return from the subroutine.

The BAL instruction is a type RX instruction, with opcode **X'45'**. This is a four–byte instruction of the form **BAL R1,D2(X2,B2)**. The object code has the format.

Type	Bytes	Operands	1	2	3	4
RX	4	R1,D2(X2,B2)	**X'45'**	R_1 X_2	B_2 D_2	D_2D_2

The first byte contains the 8–bit instruction code. The second byte contains two 4–bit fields, each of which encodes a register number. The first 4–bit field, denoted R_1, denotes the register to hold the return address. The second 4–bit field, denoted X_2, contains the optional index register. The next two bytes contain the base/displacement part of the address.

The format of the instruction is	**BAL Reg,Address**
An example of such an instruction is	**BAL 8,P10PAGE**

The first argument is a register number to be used for subroutine linkage. We explain this more fully in just a bit. The second argument is the address of the first instruction in the subroutine. There are other standards for this argument, but this is the one that IBM uses.

The BALR instruction is a type RR instruction, with opcode **X'05'**. This is a two–byte instruction of the form **BALR R1,R2**.

Type	Bytes	Operands	1	2
RR	2	R1,R2	**X'05'**	R_1 R_2

The first byte contains the 8–bit instruction code. The second byte contains two 4–bit fields, each of which encodes a register number. The first register, denoted R_1, is used as in the BAL instruction, to hold the return address. The second register, denoted R_2, is used to hold the address of the called routine. Later, we shall see that general–purpose registers R14 and R15 are favored for use by BALR in this context. The following two code sequences are equivalent.

```
BAL R8,P10PAGE

L R9,=A(P10PAGE)
BALR R8,R9
```

Each of the **BAL** and **BALR** instructions loads the address of the next instruction into the register denoted R_1, and then executes a branch to the indicated address. As we shall see, the BAL and BALR facilitate subprogram calling but are not necessary for doing so.

There is an important exception to the **BALR** instruction. If the second register is 0, the instruction just loads the first register with the address of the next instruction and then executes it. No branch is taken, as register 0 cannot hold an address. We shall see code such as:

```
      BALR  R12,0          Establish addressability by storing the
      USING *,R12          address START in R12 and using it as
START LA    R2,SAVEAREA    the base address.  No branch is taken.
```

We now present equivalent code sequences for calling a subroutine with the name **NUMOUT**. Technically speaking, it is the subroutine at the address associated with the label **NUMOUT**, but we may allow ourselves to speak loosely on occasion.

At present the code sequences written without the use of either **BAL** or **BALR** will serve only to illustrate the functioning of those two instructions. We shall use them in another context later.

We first investigate the standard instruction **BAL**, shown here in a fragment of code. The **BAL** instruction functions by storing the address **NUMRET** into general–purpose register R8 and then branching unconditionally to the address **NUMOUT**.

```
         MVC PRINT,BLANKS      CLEAR THE OUTPUT BUFFER
         BAL R8,NUMOUT         PRODUCE THE FORMATTED SUM
NUMRET   MVC DATAPR,THENUM     AND COPY TO THE PRINT AREA
```

Here is the equivalent code, written with a B (unconditional branch) instruction.

```
         MVC PRINT,BLANKS      CLEAR THE OUTPUT BUFFER
         LA  R8,NUMRET         LOAD THE ADDRESS OF THE INSTRUCTION
                               IMMEDIATELY FOLLOWING THE BRANCH
         B   NUMOUT            BRANCH DIRECTLY TO NUMOUT
NUMRET   MVC DATAPR,THENUM     AND COPY TO THE PRINT AREA
```

Here is the equivalent code, written with the more traditional **BALR** instruction.

```
         MVC  PRINT,BLANKS      CLEAR THE OUTPUT BUFFER
         LA   R9,NUMOUT         GET THE TARGET ADDRESS
         BALR R8,R9             PRODUCE THE FORMATTED SUM
NUMRET   MVC  DATAPR,THENUM     AND COPY TO THE PRINT AREA
```

Here is the equivalent code, written with a BR (unconditional branch) instruction.

```
         MVC PRINT,BLANKS      CLEAR THE OUTPUT BUFFER
         LA  R9,NUMOUT         GET THE TARGET ADDRESS
         LA  R8,NUMRET         LOAD THE ADDRESS OF THE INSTRUCTION
                               IMMEDIATELY FOLLOWING THE BRANCH
         BR  R9                BRANCH DIRECTLY TO NUMOUT
NUMRET   MVC DATAPR,THENUM     AND COPY TO THE PRINT AREA
```

Returning from the subroutine

This instruction is used to return from execution of the subroutine. It is an unconditional jump to an address contained in the register.

The format of the instruction is	**BR Reg**
An example of such an instruction is	**BR R8**

The first thing to note is that the register used in the BR will, under almost all circumstances, be that used in the **BAL** or **BALR** instruction calling the subroutine. A bit of reflection will show that this mechanism, by itself, will not allow the use of recursive subroutines. Actually, there is a workaround, but it involves the use of a separate register for each level of subroutine nesting.

A Standard Example

Here is an example of subroutine invocation in which the called program is in the same CSECT as the calling code. In such cases, the overhead of subroutine invocation is quite small. In particular, the subroutine has direct access to all data in the calling program.

```
         OPEN (FILEIN,(INPUT))
         OPEN (PRINTER,(OUTPUT))
         PUT PRINTER,PRHEAD
         GET FILEIN,RECORDIN
*
A10LOOP  BAL R8,B10DOIT          Call the subroutine
         GET FILEIN,RECORDIN     This address stored in R8.
         B A10LOOP
A90END   CLOSE FILEIN
         CLOSE PRINTER
         EXIT                    Macro to exit the program
*
B10DOIT  MVC DATAPR,RECORDIN
         PUT PRINTER,PRINT
         BR R8                   Return to address stored in
*                                R8 by the BAL instruction.
```

Functions and Subroutines

The assembly language programmer will generally speak of “subprograms”. The concepts of subroutines and functions are generally associated with higher level languages, though they are really just programming conventions placed on the use of the assembly language.

Given a programming language, such as the IBM Assembler, that provides support for subprogram linkage, all that is required to support functions is the designation of one or more general–purpose registers to return values from the function. This designation is simply a programming standard, followed by all programmers working on a project.

In CDC–7600 Assembly Language, the two designated registers were X6 and X7. The usage depended on the precision of the function result, which was usually a floating–point value.

Single–precision results	Register X7 would return the value.
Double–precision results	Register X7 would return the low–order 32 bits Register X6 would return the high–order 32 bits.

As a result, any subroutine could be called as a function. What one got as a result would be the value that the subroutine last placed in X7 or both X6 and X7. While this was chancy, it was predictable. It was the sort of thing programming geeks liked to do.

Control Section: CSECT

A **control section** (CSECT) is, by definition, a block of coding that can be relocated within the computer address space without affecting the operating logic of the program. A large system may comprise a number of control sections; each independently assembled.

Every program contains a large number of references, either to variables or addresses. In a large system, individual control sections may contain references to variables or addresses not found in the local assembly unit. These are **links** to other control sections.

A **link editor** edits the links; it searches each CSECT for references that cannot be resolved locally and attempts to find matches in other CSECTS in the system.

As an example, suppose your HLL program contains the following line of code.

```
Y = 100*SIN(X)
```

It is not likely that your program contains a function called SIN, presumably the sine. The linkage editor will find the appropriate function in the RTS library and resolve the reference by building the link from your program to the system routine.

Declaring External References

There are two possibilities if your code contains a reference that cannot be found within the local CSECT:

1. You forgot to declare the reference and have an error in your program.
2. The reference is to a variable or address in another CSECT.

In order to distinguish between the two cases, one must explicitly declare a reference as **external** if it is not declared within the local CSECT. In order to understand the processing of external labels, we must take a detour and review the output of the assembler.

The main job of the assembler is to take an assembler language program in text form and change it into a sequence of machine language statements that can be prepared for execution by the **link editor**. Were this the only function of the assembler, one could not support external symbols.

There are always at least two outputs of the assembler, which may be loosely called “data” and “metadata”. The data emitted (if one chooses to use this term) form the machine language code. The metadata emitted describe the machine code and add other information required for successful link editing. Part of these metadata is a listing of all external symbols, contained in the **ESD** (External Symbol Dictionary) [R_18, pages 538 – 544]. It is this metadata that allow the link editor to build a software system from a set of independently assembled modules.

There are two declarations used with regard to external symbols: **EXTRN** and **ENTRY**. For subprograms, these will be used in pairs. One module will declare a symbol as an **ENTRY** and other modules will declare their use of that symbol by declaring it as an **EXTRN**.

The format of the **EXTRN** statement is just a list of names, as in the following.

```
         EXTRN NAME1,NAME2,….
```

This directive informs the assembler that each of the names in the list is referenced within this assembly unit but is defined externally. This informs the linkage editor that an appropriate address to be associated with each name must be communicated to the containing program at link time, otherwise the symbol will become undefined.

The **ENTRY** instruction identifies symbols defined in the given source module as “external” so that they can be referenced by another source module. Symbols defined in an **ENTRY** statement are called **“entry symbols”**. The form of the **ENTRY** instruction is simple.

```
         ENTRY NAME1,NAME2,….
```

In essence, this is a list of symbols that the assembly unit will make available for use by other assembler units. Each entry symbol is entered into the **ESD** (External Symbol Dictionary) for access by the linkage editor. Any symbol used as the name entry of a **START** or **CSECT** instruction is automatically considered an entry symbol and is placed into the **ESD**, even if not explicitly identified by an **ENTRY** instruction [R_17, page 182].

An entry symbol may be referenced in the module in which it is declared.

An example of the expected use of this pair of assembler directives is shown below.

```
*         CALLING PROGRAM
          EXTRN   SUBA        SYMBOL IS USED HERE, DEFINED ELSEWHERE
PROG1
          L    R15,ADRSUBA    LOAD R15 WITH ADDRESS
          BALR R14,R15        BRANCH TO SUBROUTINE

ADRSUBA   DC   A(SUBA)        ADDRESS OF THIS EXTERNAL SYMBOL
          END PROG1

*         CALLED PROGRAM
          ENTRY SUBA
SUBA      Executable code associated with the symbol

          BR  R14             RETURN TO CALLING PROGRAM
          END
```

In this example, the symbol **SUBA** would be found twice in the ESD, once as an external symbol and once as an entry symbol.

Thus, the **EXTRN** and **ENTRY** statements cause their respective arguments to be placed in the ESD. As long as the linkage editor is provided with an entry symbol (defined by ENTRY) to match each external symbol (defined by EXTRN), all symbolic linkage will be handled properly by the linkage editor, and the executable system can be built.

The V Data Type

We return to the key issue associated with the use of an external symbol. Such a symbol would be declared as an entry symbol in the assembler unit in which it is defined. Most of what we have to deal with is how to declare such a symbol in an assembler unit in which that symbol is referenced but not defined. We have seen the use of the **EXTRN** declaration for this purpose.

An alternate way to handle this external declaration is to declare the symbol in the calling unit as a **V-type address constant**. This defines the symbol as an externally defined address. Recall that all “variable” references are really address references that access the contents of the address. Again, there are two ways to handle this. the first is to use a declaration of the form.

```
      Symbolic_Name   DC  V(Name)
```

This might be something like the following.

```
ADRSUBA  DC V(SUBA)
```

Where the symbol **SUBA** is declared elsewhere as an entry symbol.

The code to use this construct might appear as follows.

```
*           CALLING PROGRAM

            L    R15,ADDRSUBA
            BALR R14,R15

ADDRSUBA    DC V(SUBA)
            END
```

But this seems to be done rarely, if at all. Invocation of a subroutine found in a separate CSECT is achieved by use of the standard system macro, CALL. We give an example, and expand it. Note that the address of the called routine is presented as a literal of the proper type. The literal **=V(SUBA)** is called an external address constant. At assembly time, the constant is assigned the value **X'0000'**, which is changed by the linkage editor.

```
     CALL SUBA
+    L 15,=V(SUBA)      LOAD ADDRESS OF EXTERNAL REFERENCE
+    BALR 14,15         STORE RETURN ADDRESS INTO R14
```

The Save Areas
One key design feature when developing subroutines and functions is the minimization of side effects. One particular requirement concerns the general purpose registers. This design requirement is that after the return from a called subprogram, the contents of the general purpose registers be exactly the same as before the call.

The only exception to this rule is the use of a general purpose register for the return of a function value. In this case, the design of each of the calling routine and the called routine explicitly allows for the value of that one specified register to be changed.

Save and Restore
The strategy used is that each routine (including the main program) will save the contents of all but one of the general purpose registers on entry and restore those on exit. Recall that the main program is handled by the operating system as if it were a subprogram. This strategy has evolved from an earlier strategy in which a developer would save the contents of only those registers to be used explicitly in the subprogram. It was found to be better practice to have every subprogram start with a uniform set of assumptions rather than explicitly monitoring every register used by the subprogram.

In the statically allocated, non–recursive world of IBM assembler programs, each routine will declare an 18–fullword **"save area"**, used to save important data. There are three important system macros one should use when invoking a standard external subprogram. These are **CALL**, **SAVE**, and **RETURN**. These are described in **IBM z/VSE System Macros Reference** [R_23, pages 26, 338, and 339], Peter Abel's book [R_02, pages 346 – 353], and elsewhere.

The Save Area
Each assembler module should have a save area, which may be declared as follows.
The important feature is that eighteen full words (72 bytes) are allocated.

```
SAVEA    DS 18F
```

The structure of the save area is shown in the table below [R_02, page 346].

Word	Displacement	Contents of the Word
1	0	This is no longer used. It was once used by PL/1 programs.
2	4	The address of the save area of the calling program. Saved here to facilitate error processing.
3	8	The address of the save area of any subprogram called. The subprogram called will update this value.
4	12	The contents of general–purpose register 14, which contains the return address to the calling program.
5	16	The contents of general–purpose register 15, which contains the address of the entry point in this program.
6	20	The contents of general–purpose register 0.
7 – 18	24	These twelve fullwords contain the contents of general–purpose registers 1 through 12.

The Standard Sequence
Recalling that the user program is treated by the z/OS as a subprogram, we first sketch the sequence of events associated with a standard subprogram invocation. It is usually advisable to adhere to these standards in writing programs, as they avoid a number of bothersome problems.

1. Establish a save area, as defined above.
2. Before issuing the CALL macro, first load the address of this save area into R13.
   This is a standard use of that particular register.
3. Call the subprogram.
4. The called program should first invoke the SAVE macro to save the calling program's register contents into the area designated by R13.
5. On return to the calling program, the called program invokes the RETURN macro.

Here is a sketch of the code necessary. Here PROGA calls PROGB. Note the specific registers (12, 13, and 14; or R12, R13, and R14 if you wish). These should be used as is.

```
*          CALLING PROGRAM

PROGA
           LA   13,SAVEAREA
           CALL PROGB

SAVEAREA   DS 18F

*          CALLED PROGRAM

PROGB      SAVE (14,12)

           Miscellaneous code

           RETURN (14,12)
```

In other words, our user programs should have the same structure as **PROGB** just above.

The CALL Macro

This macro links a calling program to a called program. Presumably, the z/OS uses this macro, or a close variant of it, to invoke the user program. Again, we give the caution that this macro should really be viewed as the second statement of a pair, such as the following.

```
          LA   13,SAVEAREA
          CALL PROGB
```

The source code format of the CALL macro, as we shall use it, is as follows [R_23, page 26].

```
[LABEL]   CALL ENTRYPOINT [,(ADDRESS LIST)]
```

As is the case with almost all assembler language statements, this statement may have a label. The square brackets around the label indicate that it is optional. Read the above as follows: a valid call macro starts with an optional label, followed by the keyword CALL, followed by an entry point, followed by an optional address list. The comma is used only if the list is used.

CALL loads the address of the next sequential instruction into general–purpose register 14 to facilitate the return and then branches to the called program, using either **BAL** or **BALR**.

The operand **ENTRYPOINT** is the name of the first executable instruction of the called program. It may be the case that the subprogram has many entry points (though this is dubious coding practice). This is just where one wants to begin execution.

The **CALL** macro passes control to a control section at the specified entry point as follows: If a control section is not part of the object module that applies to the **CALL** macro, the linkage editor attempts to resolve this external reference by including the object module which contains the control section in which the entry point exists. Control is passed to that entry point.

The linkage relationship established by the **CALL** macro is the same as that created by a **BAL** instruction; that is, the issuing program expects control to be returned.

The operand **ENTRYPOINT** may be represented as **(15)**, indicating that general–purpose register 15 holds the address of the entry point in the called routine. The following two sequences appear to have the same effect. As we shall see later, the standard calls for R15 to hold the address of the entry point in the called routine.

```
SEQ1     CALL PROGA

SEQ2     L 15,=V(PROGA)
         CALL (15)
```

The entry **ADDRESS LIST** allows for passing one or more addresses, separated by commas, to the called program. To create the parameter list, the each address is expanded to a fullword on a fullword boundary in the specified order. If this is used, register R1 is loaded with the address of this parameter list; otherwise it is not altered. We shall discuss argument passing soon.

There are other options for the CALL macro as described in the manual **IBM z/VSE System Macros Reference.** These options appear to apply to variants of the assembler language that are more advanced than the System/370 assembler we are studying, and will not be discussed.

Here is an example of the use of the CALL macro with the address list.

```
CALLIT      CALL PROGB, (AP1,AP2)

AP1         A(PARAM1)
AP2         A(PARAM2)

PARAM1      DC H‘2’
PARAM2      DC H‘4’
```

The first address in the list of addresses has label **AP1**. It is this address that the macro will place into general–purpose register 1 before executing the **BAL** or **BALR**. What follows is a listing of a program written to use the **CALL** macro, along with the macro expansion.

```
                                     85 CALLIT    CALL  PROGB,(AP1,AP2)
000060                                87+         CNOP  0,4
000060 47F0 C062           00068      88+CALLIT   B     *+8
000064 00000000                       89+IHB0011B DC    V(PROGB)
000068 4110 C06A           00070      92+         LA    1,IHB0013
00006C 47F0 C072           00078      93+         B     IHB0013A
000070                                94+IHB0013  DS    0F
000070 00000178                       95+         DC    A(AP1)
000074 0000017C                       96+         DC    A(AP2)
                      00078           97+IHB0013A EQU   *
000078 58F0 C05E           00064      98+         L     15,IHB0011B
00007C 05EF                           99+         BALR  14,15
                                     100 NEXTONE  GET   FILEIN,RECORDIN

                                     221 *
000178 00000180                      222 AP1      DC A(P1)
00017C 00000182                      223 AP2      DC A(P2)
000180 0014                          224 P1       DC H'20'
000182 0028                          225 P2       DC H'40'
```

Line 85 contains the actual use of the macro. There are two address arguments.

Line 87 contains a conditional no–operation instruction, **CNOP**, which causes the following code to be aligned on the proper address boundaries.

Line 88 begins the actual expansion of the macro. Note that it branches around a declaration of the external address reference to be used as the entry point of the called subprogram.

Line 92 loads general–purpose register 1 with the address **IHB0013**, discussed below.

Line 93 branches around the lines associated with the address **IHB0013**, so that those data will not be executed. The label **IHB0013A** uses an equate to set it to the address for line 98.

The **DS 0F** on line 94 forces what follows to be aligned on fullword boundaries. What follows are two fullwords, each containing the address of an address parameter for the call.

Line 98 loads the external address, presumably now resolved by the link editor, into register R15 in preparation for the **BALR** on line 99.

Lines 222 and 223 declare the address constants, which refer to the constants defined in the next two lines. Note that the contents of AP1 are **X‘00000180’**, the address of P1.

The Return Macro

The return macro restores the contents of the registers that have been saved and returns to the calling program. To conform with the standard, it should be written as **`RETURN (14, 12)`**. Here is an actual expansion of the **`RETURN`** macro as used in a test program.

```
                                        123          RETURN (14,12)
0000AA 98EC D00C            0000C       125+         LM     14,12,12(13)
0000AE 07FE                             126+         BR     14
```

The SAVE Macro

Often the first executable instruction of a program or subprogram will be **`SAVE (14,12)`**. Here is the actual code from the second lab assignment, in which I chose to expand macros so that I could see the code generated in the macro expansions.

```
31  LAB02     CSECT
32            SAVE  (14,12)            SAVE CALLER'S REGISTERS
34+           DS    0H
35+           STM   14,12,12(13)
36            BALR  R12,0              ESTABLISH ADDRESSABILITY
37            USING *,R12
38            LA    R2,SAVEAREA        LINK THE SAVE AREAS
39            ST    R2,8(,R13)
40            ST    R13,SAVEAREA+4
41            LR    R13,R2
```

On entry, general–purpose register 13 contains the address of the supervisor's save area. The SAVE macro is written under this assumption. The user program always executes as a subprogram of the supervisor (Operating System).

Recall that the address designated **`12(13)`** corresponds to an offset of 12 from the value stored in register 13. If R13 contains address **`S0`**, then **`12(13)`** corresponds to address (**`S0 + 12`**).

As a reminder, we note that the line "**`USING *,R12`**" is a directive and does not translate into any generated machine code. From the viewpoint of the machine language generated the code reads as follows. Line 37 has been omitted from this list as it does not generate code.

```
35+           STM   14,12,12(13)
36            BALR  R12,0            ESTABLISH ADDRESSABILITY
38            LA    R2,SAVEAREA      LINK THE SAVE AREAS
```

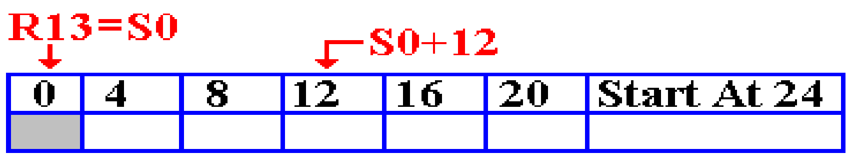
In particular, the machine language instruction following that for the **`BALR`** is the machine language instruction for the **`LA`**. It is that address that is loaded into register **`R12`**. Again, since the second register field in the **`BALR`** holds a 0, no branch is actually taken.

We now give an illustration of how these save areas are used to chain together various programs into what is often referred to as a **"call stack"**. This is especially useful when one wants to unwind the mess caused by an abnormal program termination.

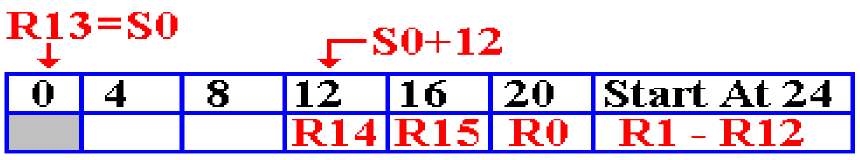
In this illustration, assume the following.

1. The save area of the supervisor begins at address S0.
2. The user main program is PA, with save area starting at address SA.
3. The user program calls program PB, which has save area starting at SB.

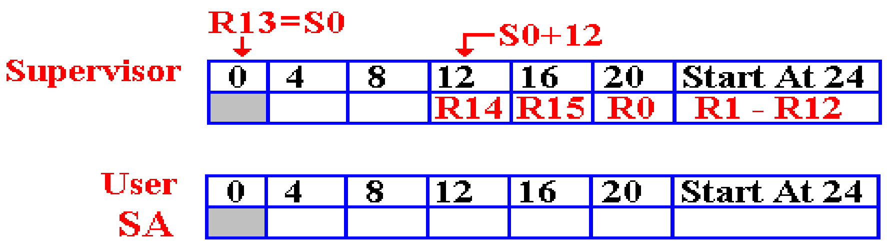
The User Program Saves the Registers
This is the case on entry to main program PA. Register 13 points to the start of the save area for the supervisor and `12(13)` points to the start of its register save area.



The instruction `STM 14,12,12(13)` is then executed. The contents of the supervisor's save area is now as follows.

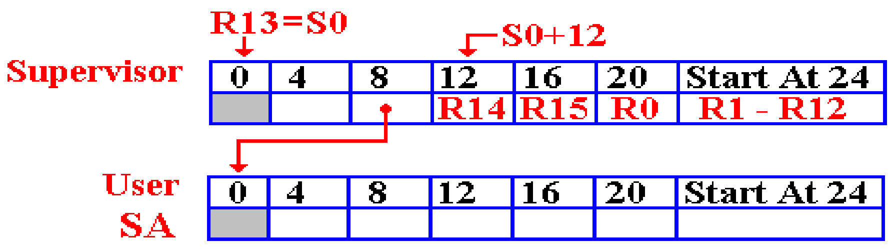


The User Program Links the Save Areas
The next step is to establish addressability in the user program, so that addresses such as that of the save area can be used. The user program then forward links the supervisor's save area. Before that happens, we just have two save areas.



Here is the code to do the forward link. The address of the start of the user save area is placed in the fullword at offset 8 from the start of the supervisor's save area.

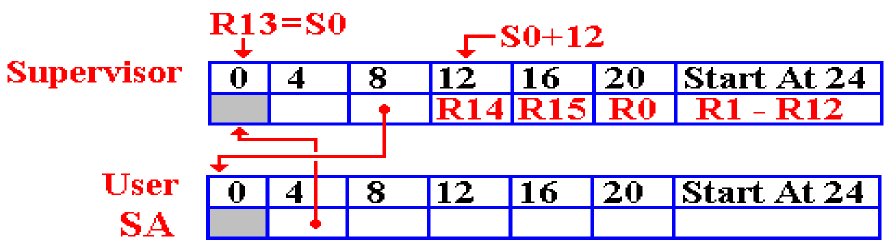
```
            LA     R2,SA       LINK THE SAVE AREAS
            ST     R2,8(,R13)  OFFSET 8 FROM S0
```



The program now executes the following code to establish the backward link from the user program to the supervisor's save area.

```
            ST     R13,SA+4
```

At this point, register R13 contains the address of the supervisor's save area. The result of this instruction is the completion of the two–way links.

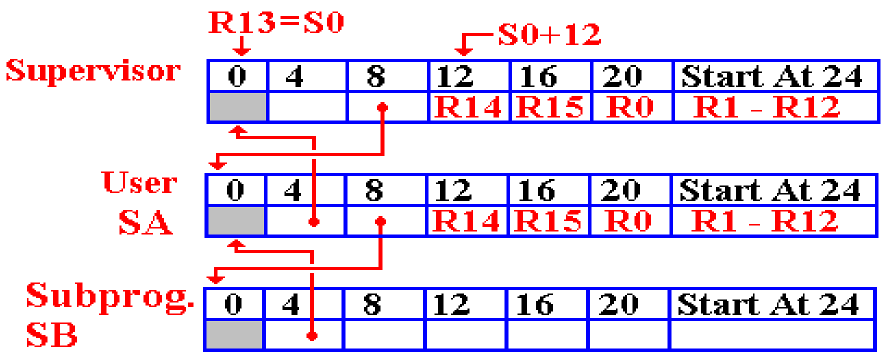


Now R2 holds the address of the user save area. Note that this could have been any of the registers in the range 2 – 12, excepting R12 which was used for addressability.

The next instruction established R13 as once again holding the address of the save area of the current program.

```
41            LR     R13,R2
```

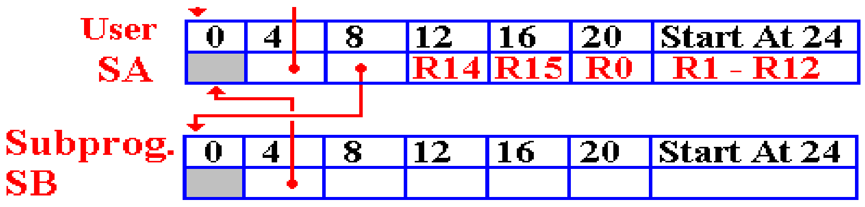
The Chain Continues
Suppose the user program calls a subprogram. The chain is continued.


R13=S0
S0+12
Supervisor
0 4 8 12 16 20 Start At 24
R14 R15 R0 R1 - R12
User SA
0 4 8 12 16 20 Start At 24
R14 R15 R0 R1 - R12
Subprog. SB
0 4 8 12 16 20 Start At 24

The Return Process
The return process restores the general–purpose registers to their values before the call of the subroutine.

R13 is first restored to point to the save area for the calling program. Then the other general–purpose registers are restored.

Here is the situation on return from Subprogram PB.



The code is as follows.

```
      L  R13,SB+4          GET ADDRESS OF CALLER'S SAVE AREA
      LM R14,R12,12(R13)   RELOAD THE REGISTERS.  R14 IS
                           LOADED WITH THE RETURN ADDRESS
      BR R14               RETURN TO THE CALLER
```

Entry/Exit Considerations

Consider how a program or subprogram starts and exits.

The start code is something like the following.

```
 SUB02    CSECT
          SAVE  (14,12)         SAVE CALLER'S REGISTERS
+         DS    0H
+         STM   14,12,12(13)
          BALR  R12,0           LOAD R12 WITH THE ADDRESS
          USING *,R12           OF THE NEXT INSTRUCTION
```

Here is the standard exit code. Assume the save area is at address **SB**.

```
          L  R13,SB+4          ADDRESS OF CALLERS SA
          LM R14,R12,12(R13)   CALLER'S REGISTER VALUES
          BR R14               RETURN
```

Note that the next to last instruction in the second section causes addressability in the called program to be lost. This is due to the fact that the value that had been loaded into R12 to serve as the base register for this code's addressability has been overwritten with the value that had been used in the calling program and will be used again.

Setting the Return Code

The IBM linkage standard calls for general–purpose register R15 to be loaded with a return code indicating success or failure. A value of 0 usually denotes a success.

It is common to set the return code just before the terminating BR instruction, just after the restoration of the calling routine's registers.

The following is typical code. Here, I am setting a return code of 3.

```
          L  R13,SB+4          ADDRESS OF CALLERS SA
          LM R14,R12,12(R13)   CALLER'S REGISTER VALUES
          LA R15,3             USE THE LOAD ADDRESS INSTRUCTION
                               TO LOAD THE CONSTANT VALUE 3.
          BR R14               RETURN
```

This uses the LA (Load Address) instruction in its common sense as a way to load a non–negative constant less than 4,096 into a general–purpose register. Note that an instruction such as `L R15,=H'3'` would require access to an element in the subprogram's literal pool. However, the value of the subprogram's base register has been overwritten, and addressability has been lost. Due to this, the subprogram's literal pool can no longer be accessed.

A normal return is usually indicated by a return code of 0, which is placed in R15. This may be done in at least two ways: either `LA R15,0` or `SR R15,R15`.

We now move on to a discussion of methods for passing argument values from a calling program to the called subprogram.

Call By Reference vs. Call By Value

Two of the more common methods for passing data to and from a subprogram are **call by reference** and **call by value**. Note that there are several other common methods.

In **call by value**, the calling program delivers the actual values of the data to the subroutine. This subroutine can alter the values of those data, but the altered values will not be returned to the calling routine. Some languages call these **"in parameters"**.

In **call by reference**, the calling program delivers the addresses of the data items to the subroutine. The values of these data can be changed by the called subroutine and those values propagated back to the calling program. These may be called **"in–out parameters"**.

Note that the return address for the subroutine can also be changed. This is an old trick used by malicious hackers in order to gain control of a computer. We may discuss this trick later.

The next question we must ask is how these values and/or addresses will be passed to the subroutine. We shall see that the System/370 assembler supports both mechanisms.

Mechanisms for Passing

By this title, we mean either the passing of values or the passing of addresses. Basically, there are three mechanisms for passing values to a subroutine. Each of these has been used.

1. Pass the values in registers specifically designated for the purpose.
2. Pass the values on the stack.
3. Pass the values in an area of memory and pass a pointer to that area by one of the other methods.

Remember that a **pointer** is really just an address in memory; it is an unsigned integer. In some high–level languages, a pointer may be manipulated as if it were an unsigned integer (which it is). In others, standard arithmetic cannot be applied to pointers. Languages, such as Java and C#, provide for pointers but do not allow direct arithmetic manipulation of these addresses. For that reason, the term "pointer" is not used.

Subroutines, Separate Assemblies and Linkage

The first example of a subroutine is the **B10DOIT** routine we have seen before.

```
B10DOIT  MVC DATAPR,RECORDIN
         PUT PRINTER,PRINT
         BR 8
```

While this is a valid subroutine, we note three facts of importance.

1. It is in the same assembly as the calling module.
2. It is in the same CSECT as the calling module.
3. The symbols (**DATAPR**, **RECORDIN**, **PRINTER**, and **PRINT**) used by the subroutine are in the scope of the calling module, but are visible to the subroutine.

The point of discussing subroutine **B10DOIT** is simple. There is an advantage to coding multiple subroutines within a given CSECT. These usually simplify the design of the code. However, they present very few issues for the passing of arguments or control.

In general, we would like to link modules that are not assembled at the same time. It might even be necessary to link assembled code with code written in a higher–level language and compiled. Thus COBOL programs might call assembler language routines.

We have already encountered the external declaration, which is used to declare that:

1. A label corresponds to an address declared in another module, and
2. The linking loader will have to resolve that address by loading the called module, assigning an address to this label, and updating that reference in the calling code.

Register Usage in Subprogram Calling

The following is the Standard Linkage Convention for the use of general–purpose registers. This convention is to be used by any subprogram that is to be run under z/OS. [R_19, page 146].

Register	Use
1	Address of the parameter list.
13	Address of calling routine's save area. This is an area set aside to save the register values from the calling program.
14	Address in the calling routine to which control is to be returned.
15	The address of the entry point in the called subprogram.

There is no direct standard for the other twelve general–purpose registers (0 and 2 – 12).

We have already seen an example of the use of registers 14 and 15. Consider the following code, which is written in the standard format preferred for S/370 assembler programs.

```
L 15,=V(PROGB)     LOAD ADDRESS OF EXTERNAL REFERENCE
BALR 14,15         STORE RETURN ADDRESS INTO R14
```

IBM has established a set of conventions for use by all modules when calling separately assembled modules. These insure that each module can work with any other module.

Implementation of Argument Passing Mechanisms

We shall now discuss in more detail the list of options presented for the passing of arguments to a subprogram. As of now, we shall assume that the subprogram is assembled separately from the main program. Assume that **PROGA** calls **PROGB**, or an entry within **PROGB**. In terms of the theory of programming languages, we assume that we have two different scopes; labels defined within **PROGA** have no meaning within **PROGB**, and vice–versa. The linkage is provided by the linkage editor, using external symbols and entry symbols. The reason for this focus is that the simpler situation presents no difficulties worth further discussion.

One simple way to pass an argument to a subprogram would use a register, which might contain either the value of the argument or the address of the argument. This might work for small systems designed and implemented by a single programmer, but the design does not scale well to large systems. For that reason, we shall mention this method only to be complete.

Another mechanism for passing arguments might be called the **argument block method**. In this method, one register is chosen to hold the address of a block used to contain the arguments, either values or addresses. One advantage of this method is that it does not limit the number of arguments to the number of registers available. We shall discuss the implementation of this idea as developed by IBM for the systems programs that ran on the System/360.

This method calls for general–purpose register **R1** to hold the address of a parameter block. In this parameter block we find a list of the **addresses** of the parameters, not the values. This is the mechanism that was discussed just above in the example of the **CALL** macro. Consider the following code fragment, taken from Yarmish & Yarmish [R_18, pages 544 & 545].

```
*        CALLING ROUTINE
S1       LA R1,PARAMLST      Load address of the parameter list
S2       L  R15,ADDSUB       Entry point to subroutine
         BALR R14,R15        Call the subroutine

ADDSUB   DC V(SUBA)          External reference
*
PARAMLST DC A(PARAM1)        Address of first parameter
         DC A(PARAM2)        Address of second parameter
         DC A(ANSWER)        Address of third parameter
*
PARAM1   DC F‘22’
PARAM2   DC F‘33’
ANSWER   DS F
*        CALLED ROUTINE
         ENTRY SUBA          Declare as an entry point
SUBA     L  R2,0(1)          Load R2 from the address at offset
                             0 from the address in R1.
S3       L  R3,4(1)          Load R3 from the address at offset
                             4 from the address in R1.
S4       L  R7,0(2)          Load R7 from the address in R2
S5       A  R7,0(3)          Add value at address in R3
S6       L  R8,8(1)          Get the third address passed
S7       ST R7,0(8)          Store the result into that address
         BR R14              Return
```

This code requires some comments. The line labeled **SUBA** has register **R2** loaded from an address that is offset 0 from the address stored in general–purpose register 1. The value there is **A(PARAM1)**, the address of the first parameter. The line labeled **S3** loads **R3** from an address that is offset 4 from the address in R1. The value here is **A(PARAM2)**.

Line **S4** loads **R7** from the address that is offset 0 from the value in **R2**. The value in **R2** is the address of **PARAM1**, so **R7** is now loaded with the value of **PARAM1**, which is 22. Line **S5** adds to **R7** the value that is at offset 0 from the address in **R3**. The address is that of **PARAM2**, so the value of **PARAM2** is added to **R7**, which now holds 55. Line **S6** loads **R8** from the value at offset 8 from the address in **R1**; this is A(ANSWER). Line **S7** stores the contents of R7 into the address held in **R8**, thus returning the answer to the calling program.

The Dummy Section (DSECT)
We now consider a logical extension to the argument block method of passing values and/or addresses to a separately assembled subprogram. In more modern terms, this might be considered as passing a record to a subprogram by specifying the structure of the record and the address of the first member of the record. Loosely speaking, a DSECT is the specification of the structure of a record without the allocation of memory for that structure.

Recall that parameters are passed by reference when it is desired to allow the called program to change the values as stored in the calling program. Call by reference is also the preferred method for passing reference to a data structure so large that making a copy for pass–by–value would be inefficient. In other words, the programmer may elect to use call–by–reference even in those cases in which the subprogram does not change any of the data passed to it.

For data represented by a large number of labels, it is convenient to use a dummy section. This process is best considered as passing a record structure to the called subprogram.

1. All the data items to be passed to the called program are grouped in a contiguous data block.
2. The address of this data block is passed to the called subprogram.
3. The called subprogram accesses items in the data block by offset from the address passed to it.

The DSECT is used in the called subprogram as a template for the assembler to generate the proper address offsets to be used in accessing the original data. Sharon Tuggle [R_09, pages 369 & 370] gives a very good introduction to the idea of a DSECT, which I quote here.

> "There are times in programming when the programmer needs to refer to whole blocks of data residing outside his own program (i.e., data passed between subroutines). Because of the quantity of data being referenced, it is impractical to use address constants and EXTRN statements to refer to each piece or pass each data item as an individual parameter. Instead, all the data items are collected in contiguous areas that make up a block of data; and one address, the address of the beginning of the block, is passed as a parameter between subroutines. This one address can be loaded into a register and that register used as the base register for a ***dummy control section*** (a DSECT)."

> "A DSECT is a convenient means that the programmer can use to describe the layout of an area of storage without actually reserving the storage. It is assumed that the storage is reserved elsewhere (most likely in another subroutine)."

The format of a DSECT instruction is as follows [R_17, page 177].

```
[LABEL]     DSECT
```

Following the optional label is the single statement declaring the start of a dummy section. The most common usages of the dummy section all require an ordinary symbol to label the DSECT. According to IBM [R_17, page 177] "The location counter for a dummy section is always set to an initial value of 0." In other words, items within the DSECT are given addresses relative to the start of the DSECT.

Here are some notes on the DSECT, taken from [R_17, pages 177 and 178].

1. The assembler language statements that appear in a dummy section are not assembled into object code.
2. When establishing the addressability of a dummy section, the symbol in the name field of the DSECT instruction, or any symbol defined in the dummy section, can be specified in a USING instruction.

The standard way to effect references to the storage area defined by a dummy section is to provide a USING statement that specifies both a general–purpose register that the assembler can use as a base register for the dummy section, and a value from the dummy section that the assembler may assume the register contains.

Example of Linkage Using a DSECT
Suppose that I want to pass customer data from PROGA (the calling program, in which the data are defined) to subprogram PROGB, in which they are used.

Here is a sketch of code and declarations in the calling program, which is PROGA.
This uses the calling convention that R1 holds the address of the parameters.

```
PROGA    CSECT
* SECTION TO CALL PROGB
         LA R1,=A(CREC)    LOAD RECORD ADDRESS
         L  R15,=V(PROGB)  LOAD ADDRESS OF PROGB
         BALR R14,R15      CALL THE SUBPROGRAM
         Next Instruction
*
CREC     DS 0CL96    THE RECORD WITH ITS SUBFIELDS
CNAME    DS CL30     OFFSET =  0
CADDR1   DS CL20     OFFSET = 30
CADDR2   DS CL20     OFFSET = 50
CCITY    DS CL15     OFFSET = 70
CSTATE   DS CL2      OFFSET = 85
CZIP     DS CL9      OFFSET = 87
```

The Dummy Section Itself
The dummy section will be declared in the called subprogram using a DSECT.

Here is the proper declaration for our case.

```
CRECB    DSECT
CNAME    DS CL30     OFFSET =  0
CADDR1   DS CL20     OFFSET = 30
CADDR2   DS CL20     OFFSET = 50
CCITY    DS CL15     OFFSET = 70
CSTATE   DS CL2      OFFSET = 85
CZIP     DS CL9      OFFSET = 87
```

The DSECT is a convenient means that can be used to describe the layout of a storage area without actually reserving storage.

In use of a DSECT, it is assumed that the storage has been reserved elsewhere and that the base address of that storage will be passed as a parameter.

The DSECT is just a mechanism to instruct the assembler on generation of offsets from the base address into this shared data area.

Use of the DSECT in PROGB
Here is a sketch of the use of the DSECT in PROGB, assembled independently.

Recall that general–purpose register R1 contains the address of the customer record.

```
PROGB    CSECT
         BALR R12,0         ESTABLISH ADDRESSABILITY
         USING *,R12
         LR R10,R1          GET THE PARAMETER ADDRESS
         USING CRECB,R10    ESTABLISH ADDRESSABILITY FOR
                            THE DUMMY SECTION
         MVC OUTC,CCITY     THIS ACCESSES THE DATA IN
                            THE CALLING PROGRAM
OUTC     DS CL15            DATA BELONGING TO PROGB

CRECB    DSECT
CNAME    DS CL30      OFFSET =  0
CADDR1   DS CL20      OFFSET = 30
CADDR2   DS CL20      OFFSET = 50
CCITY    DS CL15      OFFSET = 70
CSTATE   DS CL2       OFFSET = 85
CZIP     DS CL9       OFFSET = 87
```

A Detailed Look at Addresses
Here we recall two facts relative to the one instruction **`MVC OUTC,CCITY`**.

The label **`OUTC`** belongs to the called subprogram PROGB. It is accessed using base register R12, which is the basis for addressability in this subprogram. The label **`CCITY`** belongs to the dummy section. It is accessed using the base register explicitly associated with the DSECT; here it is R10. The **`LR R10,R1`** instruction copies the address of the parameter block in the calling program into R10 for use in the subprogram PROGB.

Suppose that **`OUTC`** has address **`X'100'`** (decimal 256) within **`PROGB`**, it is at that displacement from the value contained in the base register R12. From the DSECT it is clear that the label CCITY is at displacement 70 (**`X'46'`**) from the beginning of the data record. The field has length 15, or **`X'0F'`**

The instruction **`MVC OUTC,CCITY`** could have been written with explicit base register references as **`MVC 256(15,R12),70(R10)`**, with object code **`D2 0E C1 00 A0 46`**

`D2` is the operation code for **`MVC`**. **`0E`** encodes decimal 14, one less than the length.

`C1 00` represents an address at displacement **`X'100'`** from base register 12.

`A0 46` represents an address at displacement **`X'46'`** from base register 10.