#SHAREorg

SHARE

Saving Your Caller's Registers - Not Your Father's Save Area

Tom Marchant
Compuware
thomas.marchant@compuware.com

Tuesday, August 7, 2012
Session 11408

SHARE in Anaheim 2012

Acknowledgments

- Peter Relson, z/OS Core Technology Design, IBM
- The Assembler Services Guide, Release 12 or higher
  - SA22-7605-12 Release 12
  - SA22-7605-14 Release 13
- All of the people who have asked me to explain it to them

When I was first preparing to use the newer save area formats, I had some questions. I asked and Peter graciously answered. The answers lead to more questions. Peter patiently answered my questions. After several email exchanges, he made extensive revisions to the Linkage Conventions Chapter of the Assembler Services Guide. The z/OS Version 1 Release 12 edition of the manual is much clearer than earlier editions.

Many people have asked me to explain this material to them, and each time I did, my understanding grew.

I am grateful to Peter for his review of this presentation and for some suggestions that he made to improve the content.

Agenda

- Review the standard 72-byte save area
- F4SA used for programs that use the 64-bit registers
  - Chaining the save areas together
  - Using F4SA to identify the save area type
    - Following save area chains in a dump
- F7SA for AR mode programs
- F5SA and F8SA for 64-bit programs that are passed a 72-byte save area
- Using the Linkage Stack
- XP Link will ***not*** be covered

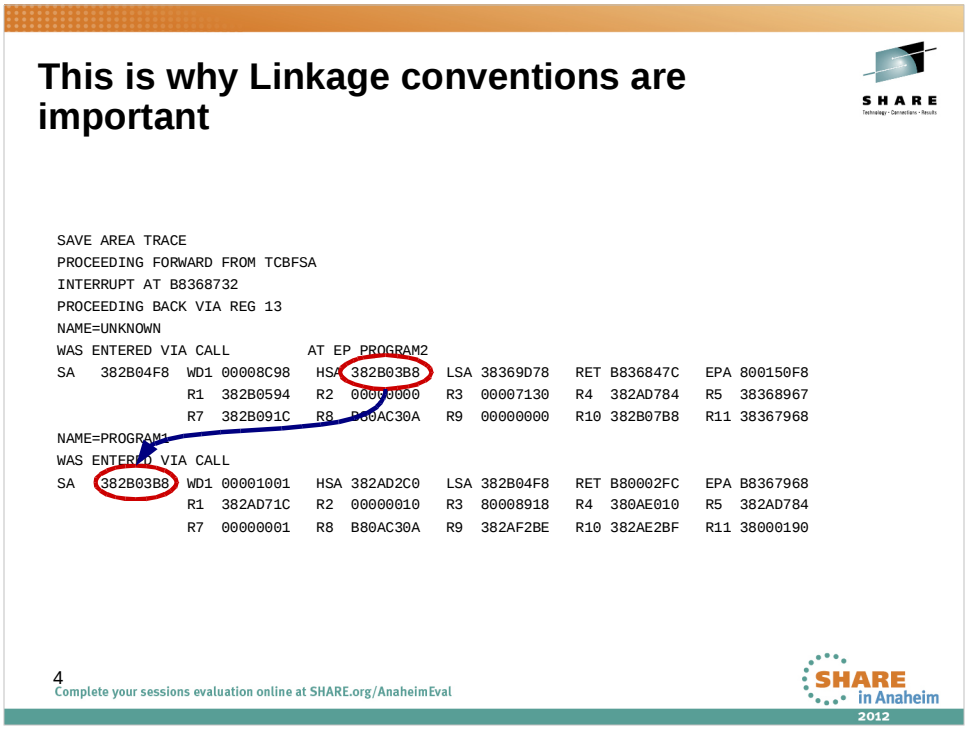
We will begin with a brief review of the 72-byte save area.

Then we will cover the newer save area formats, starting with F4SA. We will give special emphasis to the correct method of marking the save areas to identify the save area formats that are used.

Finally, we will cover the correct way to establish a save area for the programs that you call if you use the Linkage Stack to save your caller's registers.

XPLink formats used by some Language Environment programs will not be covered in this presentation.

This is why Linkage conventions are important



Standard save area formats are used so that the program flow can be followed when analyzing a dump. This shows part of a save area trace as it is shown in a SYSUDUMP or SYSABEND dump. The previous save area field that is in the second word of the save area is used to locate the previous save area.

SYSUDUMP and SYSABEND analyze the calling chain by first going forward from the first save area that is stored in the TCB (TCBFSA). This analysis is done only with standard 72-byte save areas, and then only for programs that maintain the forward chain. Not all programs do.

As we will see, with the newer save area formats, the only predictable way to follow the save area chain is going backward. In some cases, it is impossible to follow it going forward

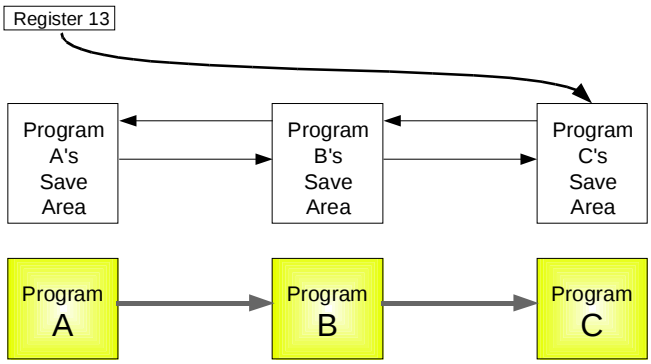
First some terminology

- These all refer to the same location
  - Offset 4
  - Word 1
  - The second word
- The Linkage Conventions chapter of the Assembler Services Guide uses all three of these ways to refer to various locations in the save areas.

Word 0 is the first word in the save area. It begins at offset 0.

Word 1 begins at offset 4.

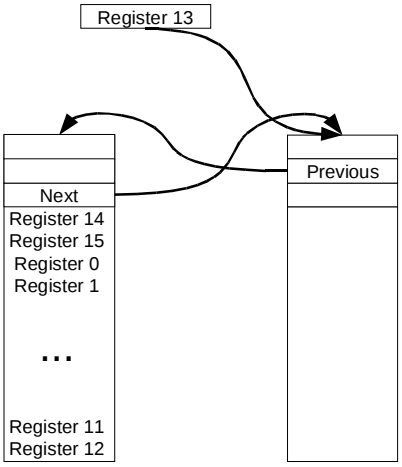
The basic form of these slides



We start out with Program A on the left side. Program A has a save area whose address is in register 13 when it calls program B. Program B uses this save area to save program A's registers and establishes its own save area, then calls program C which also creates a save area for any programs that it calls.

After this slide, the programs are not shown, only the save areas that are associated with them.

Standard 72-byte save area



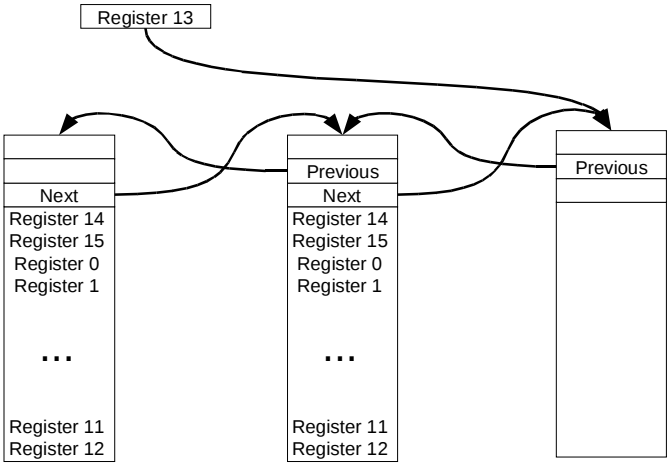

```
STM  14,12,12(13)
GETMAIN RU,LV=72
ST   13,4(,1)
ST   1,8(,13)
LR   13,1
```


When program B receives control, it saves registers 14-12 starting in offset 12 of the save area pointed to by register 13.

Then it allocates a new 72 byte (18 fullword) save area on a fullword boundary and chains the two save areas together. Offset 4 of the new save area points to the caller's save area. Offset 8 of the caller's save area is set to the address of the new save area. Finally, it updates register 13 to point to the new save area.

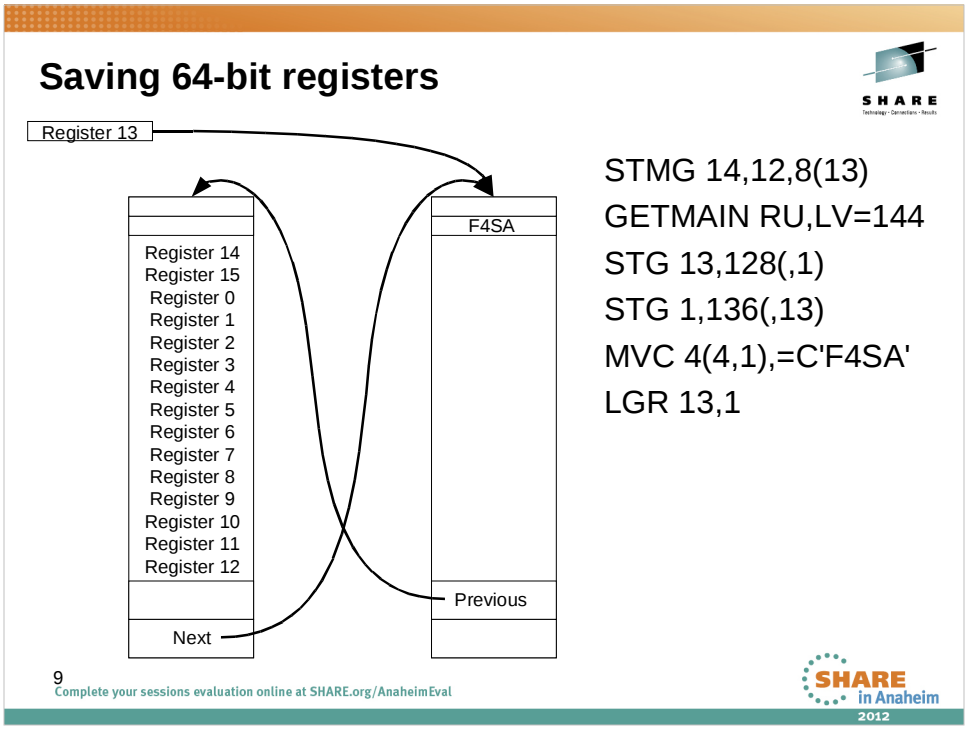
Register 13 is not saved in the caller's save area. It contains the address of the caller's save area and is stored in the new save area.a.

Standard 72-byte save area



When Program B calls program C, program C does the same thing.

Only the middle save area in this illustration has all of the fields filled in.

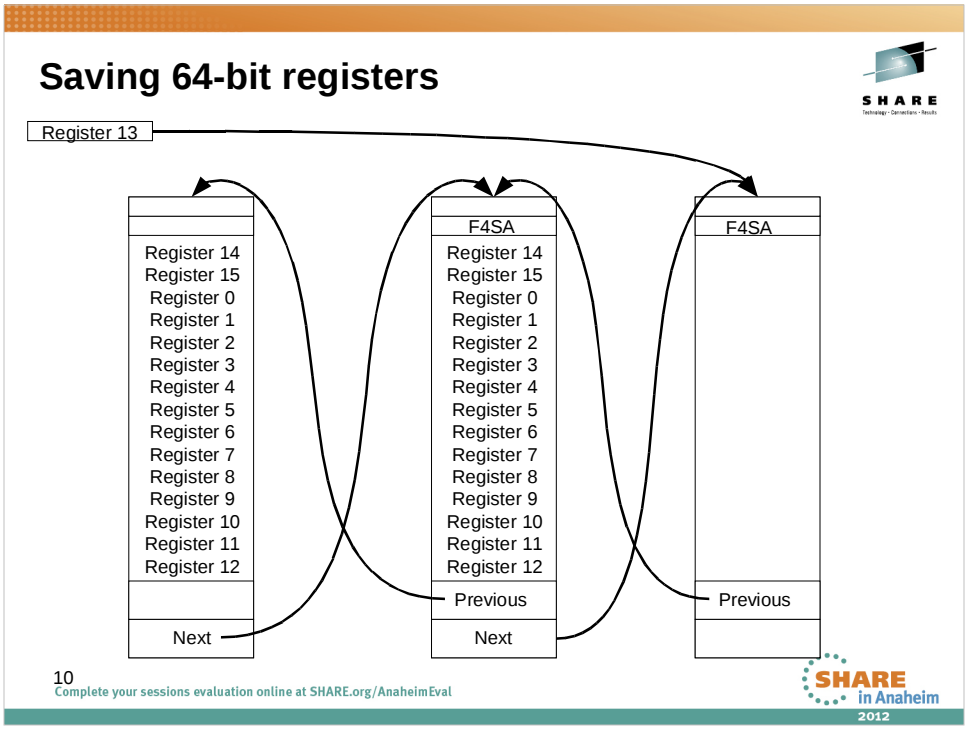


With z/Architecture, the registers are now 64 bits wide. Programs that use the high halves of the 64-bit registers need to save all 64 bits of the registers.

Program B saves program A's registers 14-12 starting at 0ffset 8 in program A's save area. A new 144 byte (18 doubleword) save area is allocated on a doubleword boundary. The address of the calling program's save area is stored in the doubleword at offset 128 (X'80') of the new save area. The address of the new save area is stored in the doubleword at offset 136 (X'88') of the calling program's save area.

“F4SA” is stored in offset 4 of the new save area. We do not know or care what is in offset 4 of program A's save area. Nor do we know how the new save area will be used.

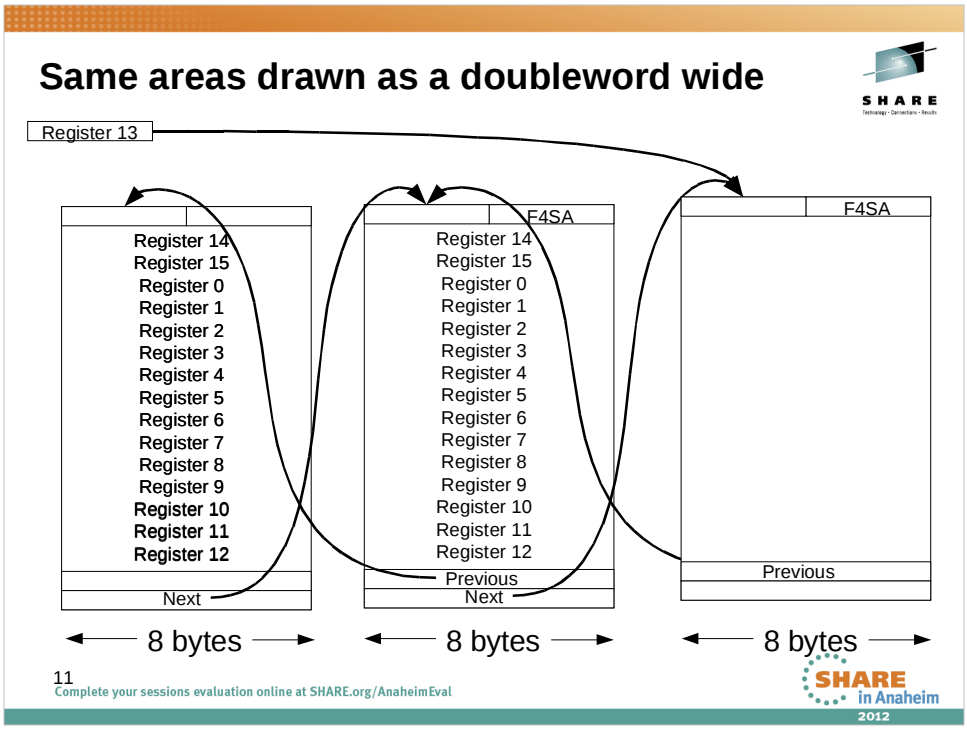
The instructions shown are for illustrative purposes. For better readability and maintainability, use the labels defined in IHASAVER rather than hard-coded offsets. The samples in the Assembler Services Guide use these labels to save and restore registers.



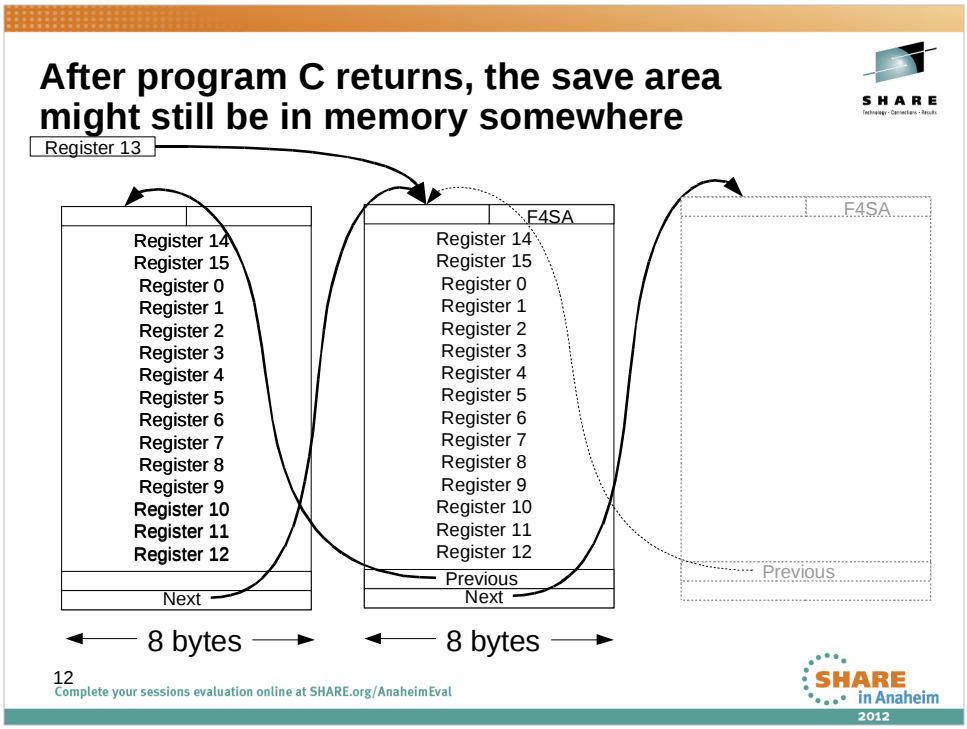
When program C is called, it does exactly the same thing. Here we can see the result when C is in control.

Because the save area addresses are stored as doublewords, a save area could be allocated above 2 GB, but doing so means that only AMODE(64) programs can be called by this program.

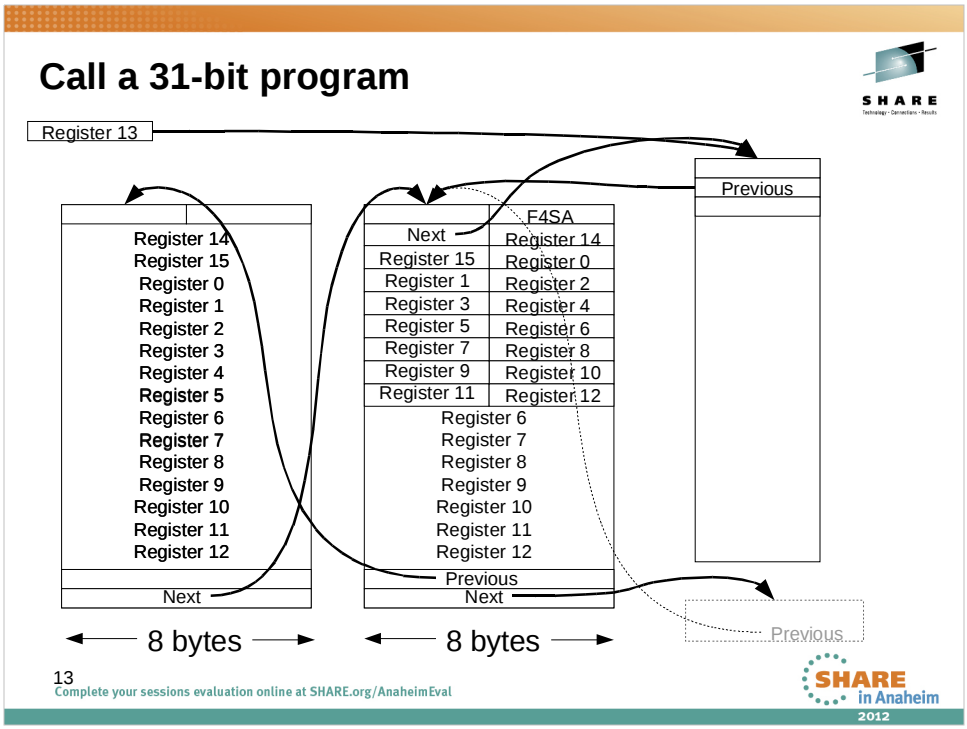
The Assembler Services Guide refers to AMODE 64 programs, but even if a program only uses the high halves to do arithmetic, the high halves must be preserved. This presentation refers to “64-bit programs” to signify programs that alter the high halves of some registers. “31-bit programs” refers to programs that do not alter the high halves of any registers.



This slide contains the same information as the previous illustration, but it is drawn differently. While the previous slide showed the save area as being 4 bytes wide, in this slide the save area is illustrated as a doubleword in width.

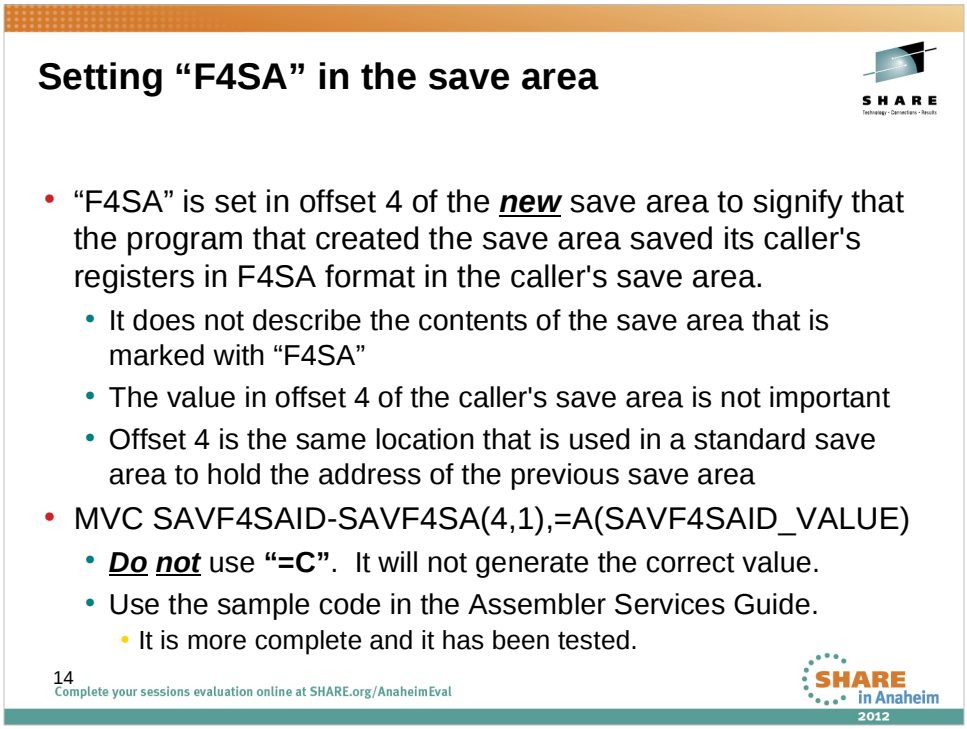


Now suppose that program C has freed its save area and returned to program B. Program C's save area might still exist somewhere in memory, and program B's save area still has the address where it was located. If it has not been reused, there is still a back chain pointer to B's save area.



Now program B calls a 31-bit program, C2. C2 saves program B's registers in standard format. This does not disturb anything important in program B's save area. Offset 4 still contains the value “F4SA” and the previous save area pointer is at offset 128 (X'80'). When C2 returns, program B will be able to restore program A's registers to return to it.

This illustrates why it is very difficult to follow the save area chain going forward. At this point in time, B's save area has a forward pointer at offset 8 and an old forward pointer at offset 136 (X'88').

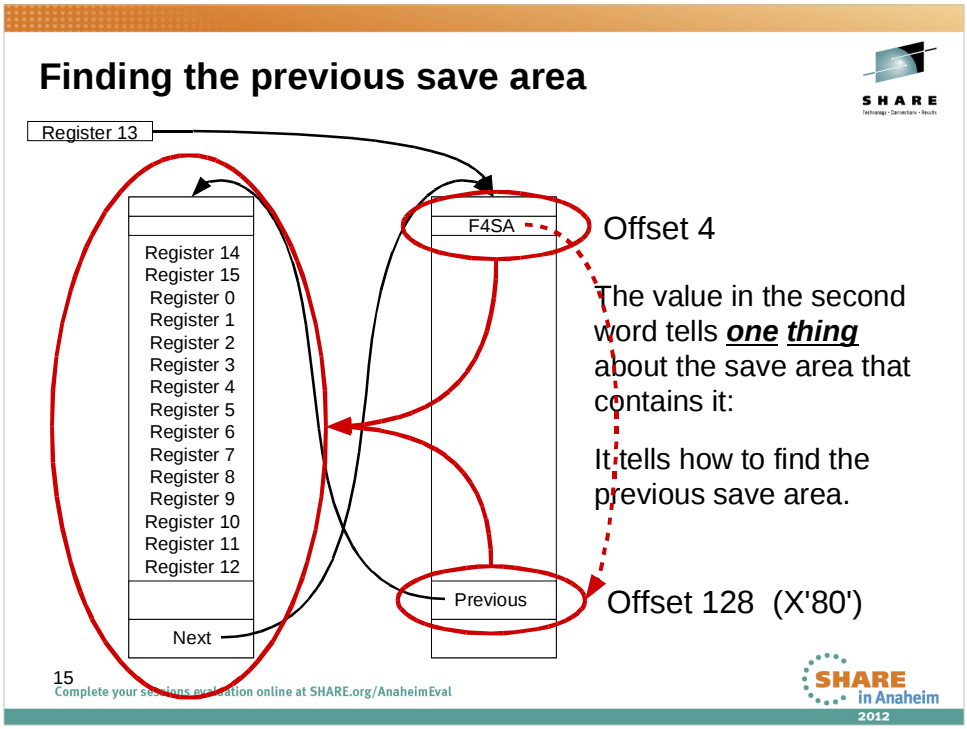


This is important.

The value in the second word of the save area does not describe the contents or the size of the save area that contains it. It tells how the program that allocated the save area saved its caller's registers in its caller's save area.

It might seem strange to use =A(SAVF4SAID_VALUE) when setting the value in offset 4. That is the correct way to set it.

The samples in the Assembler Services Guide are very good and they have all been tested and verified to work correctly. Use them.



When looking at a save area in a dump, examine the value at offset 4 to find the location of the previous save area.

If it is even, it is the address of a standard 72-byte save area or it is zero, indicating that there is no previous save area.

If it is “F4SA”, the previous save area is at offset 128 (X'80').

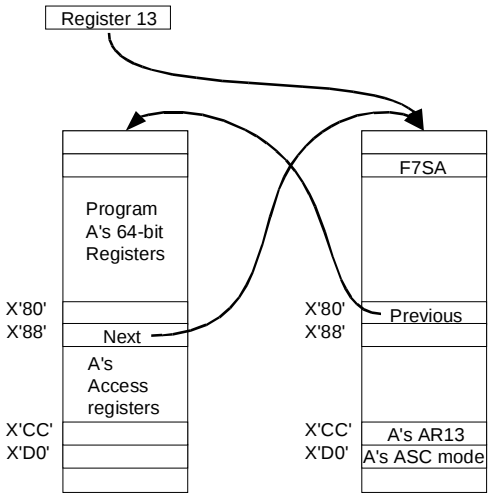
A question and Peter Relson's answer

Q: When the system gives control to a routine, is the 144-byte save area that is provided marked as F4SA?

A: The answer is "no". Because there is no caller of the "system" who allocated the storage for the 144-byte save area. So the routine (the system) who allocated the storage for the 144-byte area did not save its callers registers.

When the system passes control to a program with ATTACH, register 13 contains the address of a 144-byte save area on a doubleword boundary. The second word of the save area is zero, indicating that there is no previous save area.

Programs that alter Access Registers
216 byte save area – 27 Doublewords



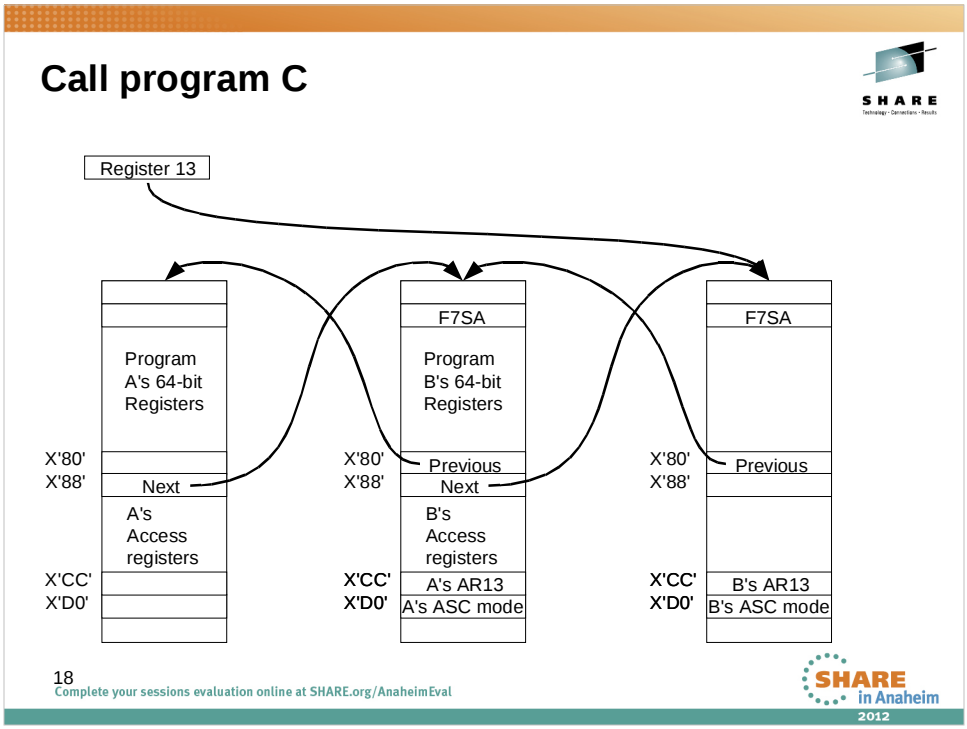
STMG 14,12,8(13)
STAM 14,12,144(13)
Allocate a 216-byte save area
Save the caller's ASC mode in offset 208 of the new save area
STAM 13,13,204(1)
STG 13,128(,1)
STG 1,136(,13)
MVC 4(4,1),=C'F7SA'
LGR 13,1

17
Complete your sessions evaluation online at SHARE.org/AnaheimEval

SHARE in Anaheim 2012

Next, we will look at the way an AR-mode program can save the Access Registers. For this, the program will require a 216 byte (27 doubleword) save area. The Access Registers are stored at offset 144 of the save area. The calling program's ASC mode is stored in offset 212 (X'D0') and Access Register 13 is stored at offset 208 (X'CC') of the new save area. This information is needed in order to be able to correctly locate the previous save area. For details about the code needed to save the ASC mode, see the Assembler Services Guide.

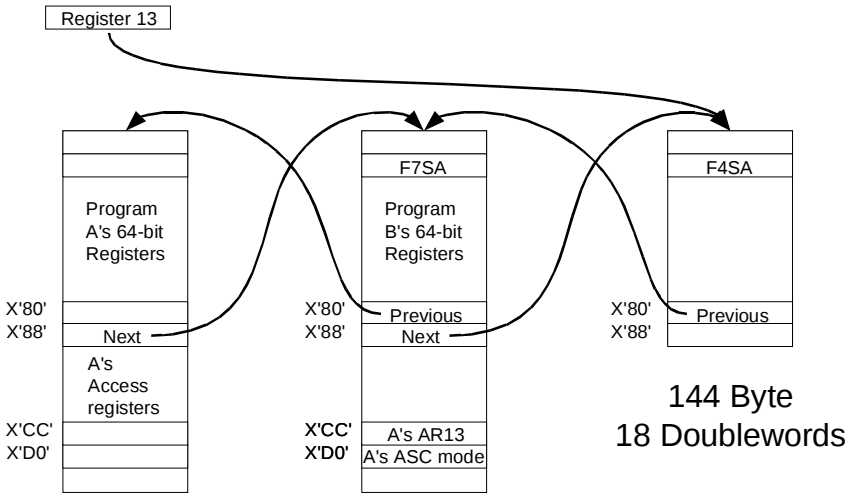
As with F4SA, the back chain pointer is at offset 128 (X'80') and the forward chain is at offset 136 (X'88'). The new save area is marked to show how we saved our caller's registers, this time with “F7SA”. Register 13 is updated with the address of our new save area.



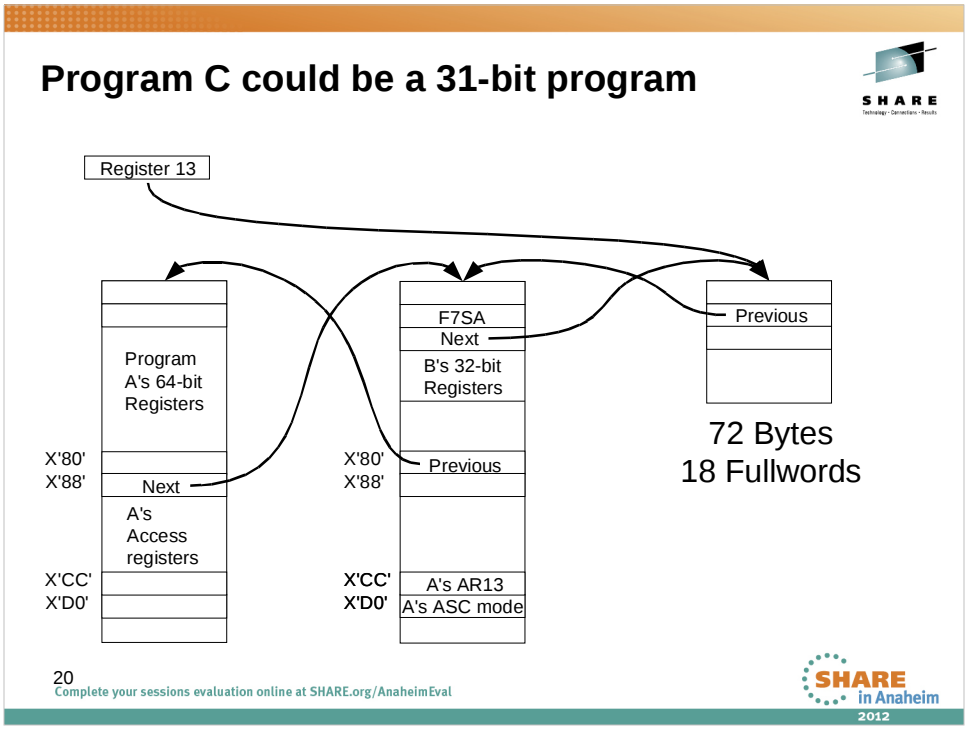
When program B calls AR mode program C, program C does the same thing.

The General Purpose Registers and the Access Registers are saved in the caller's save area. The ASC mode when we received control is stored in our save area, along with the ALET for the caller's save area that was in AR13.

Program C could be a 64-bit program that does not use Access Registers

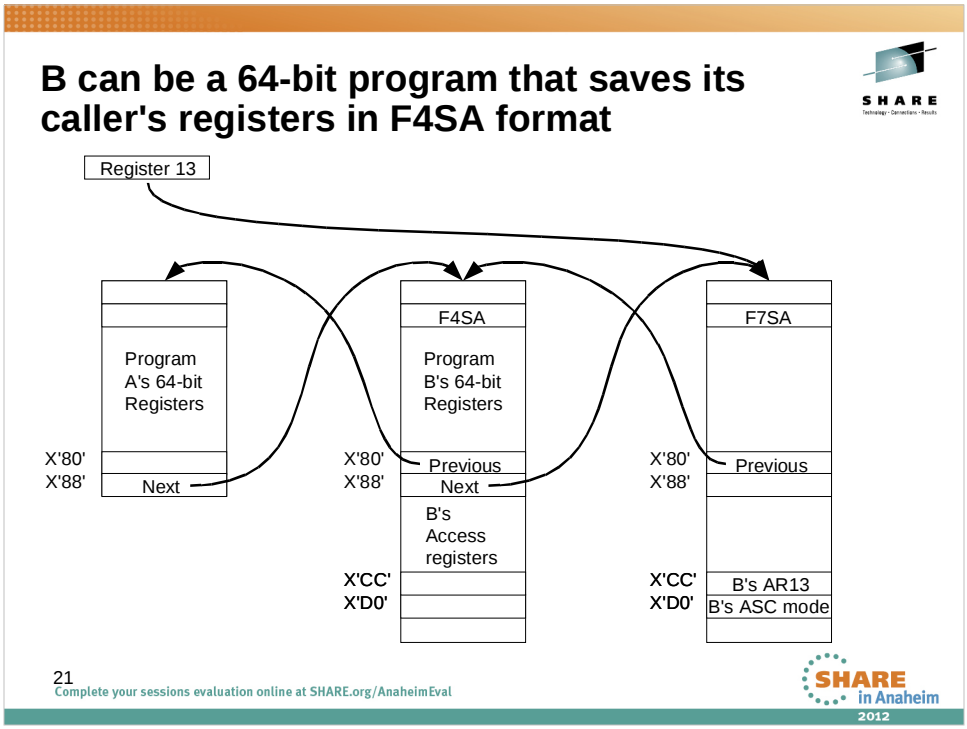


If program C is a 64-bit program that does not use the Access Registers, it will save program B's registers in F4SA format within the 216-byte save area that program B provided. No harm is done by this. Program C will allocate a save area that is in accordance with the specifications of any programs that it calls. It may be a 144-byte save area or a 216 byte save area. Regardless of the size of save area that C allocates, C will mark it as F4SA because that is the format that C used to save its caller's registers.



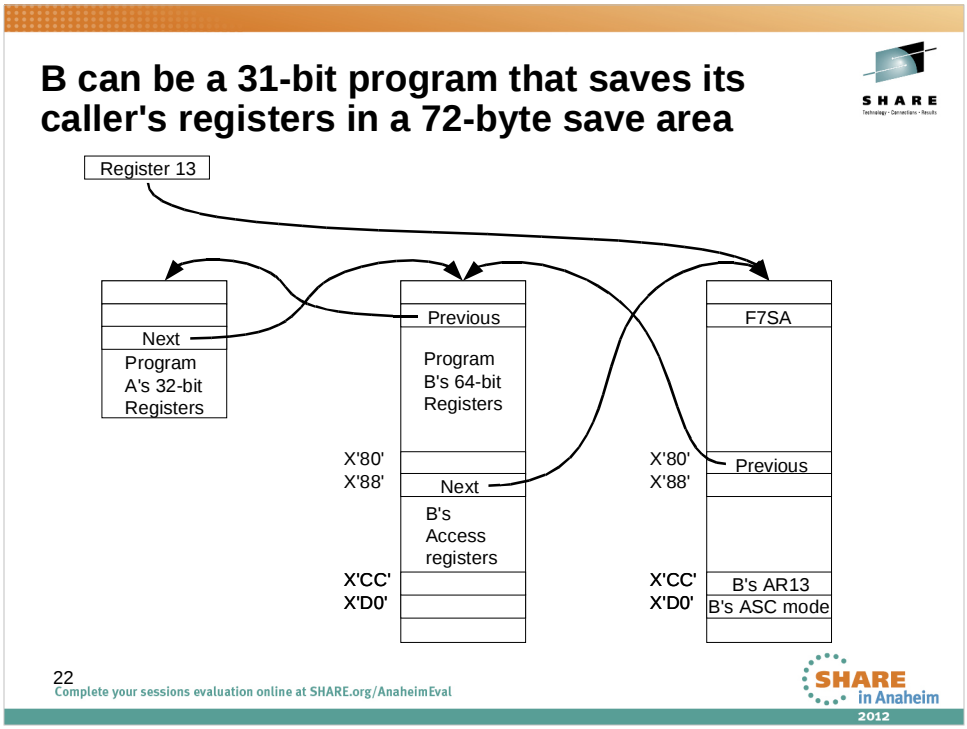
B can also call a 31-bit program. The 31-bit program will use the first 72 bytes of the 216-byte save are that was allocated by B. No harm is done by this. The 31-bit program saves its caller's registers in the usual way and the save areas are chained according to the rules for standard save areas. Offset 4 of the new save area has the address of the caller's save area. Offset 8 of the caller's save area is updated with the address of the new save area. The result is as is shown here.

With the mixing of save area formats that can occur, the only reliable way to analyze the save areas is going backward.



Here, program B is a 64-bit program that does not use Access Registers. It requires a 144-byte save area to save program A's registers in F4SA format. Program B will be calling program C, which does use the Access Registers. Program B knows that it must provide a 216-byte save area for C to save B's registers.

This is another illustration of how the value in offset 4 does not describe the save area that contains it.



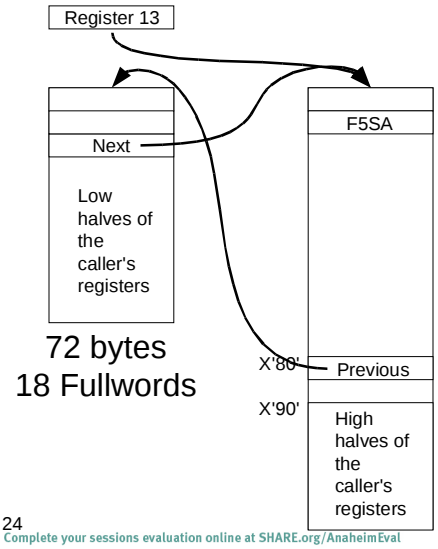
In this illustration, program B is a 31-bit program. It uses a 72-byte save area to save program A's registers in standard format. Program B will be calling program C, which requires a 216-byte save area for C to save B's registers. This might be unusual, but it is consistent with the rules for save areas.

A question and Peter Relson's answer

Q: Module C, called by B, would not be able to determine by looking at the save area that was passed to it whether or not it had enough space to store the access registers.

A: C does not "determine", C "requires". C's entry requirements define how big a save area its callers must provide. It is up to the callers to obey or face the consequences (such as an overlay of B's storage). For example, C might say "I need a 216 byte save area that I can use in F7SA format". If the user provided 144, then C was still going to store 216 bytes of data.

64-bit program that can be called by a 31-bit program

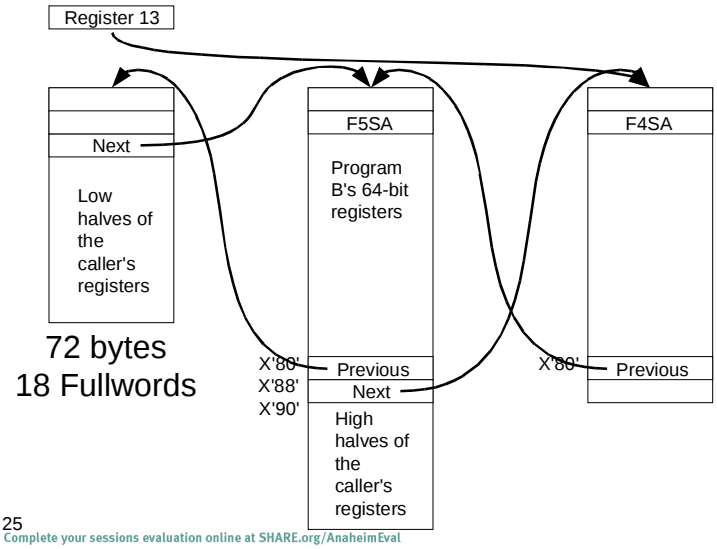



```
STM  14,12,12(13)
GETMAIN RU,LV=216
Save the high halves in offset 144 of the new save area
STG 13,128(,1)
ST   1,8(,13)
MVC 4(4,1),=C'F5SA'
LGR 13,1
```


When a 64-bit program is not able to control its callers, it can not require that a 144-byte save area be passed to it. In that case, the low halves of the registers are saved in the caller's save area. A 216 byte save area is allocated and the high halves are stored at offset 144 of the new save area.

This is a little more complicated than might be expected because the high halves of registers 15, 0 and 1 are preserved across the GETMAIN so that they can be saved in the save area. For details of how this is done, see the sample code in the Assembler Services Guide.

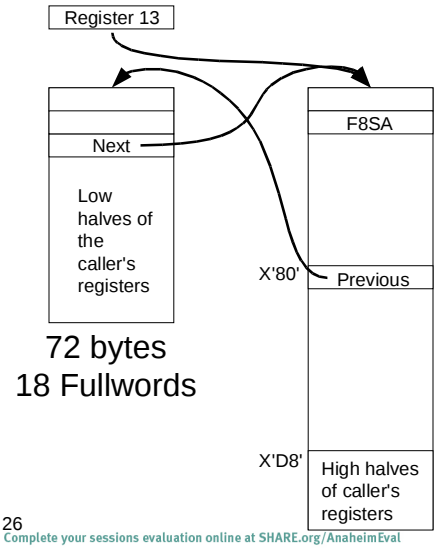
Program B calls 64-bit program C



When program B calls another 64-bit program, the first 144 bytes of program B's save area can be used to store the registers in F4SA format. As with F4SA, a 31-bit program that is called by B will save the 32-bit registers with no ill effects.

Another program that is coded to use F5SA format will store the low halves in the first 72 bytes. It will not know that 144 bytes are available.

64-bit program that can be called by a 31-bit program

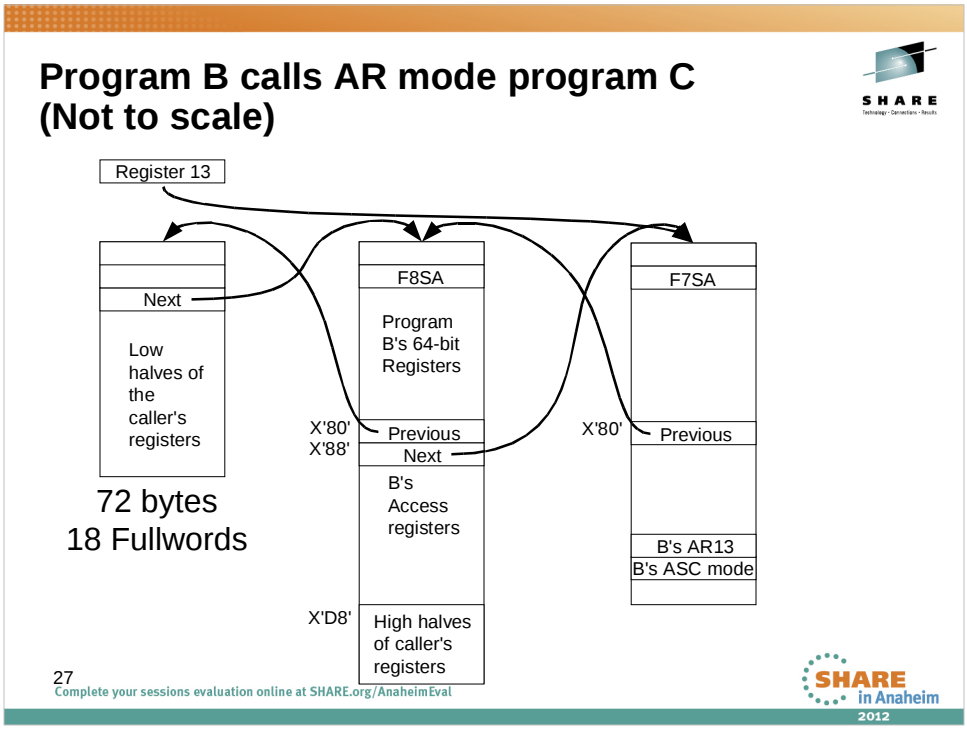



```
STM  14,12,12(13)
GETMAIN RU,LV=288
Save the high halves in
offset 216 of the new
save area
STG 13,128(,1)
ST   1,8(,13)
MVC 4(4,1),=C'F8SA'
LGR 13,1
```


F5SA is not adequate for a 64-bit program that will call an AR mode program that requires a 216-byte save area. In that situation, F8SA may be used.

F8SA is the same as F5SA except that the location that is used to save the high halves of the caller's registers is different. In an F5SA, the high halves of the caller's registers are stored at offset 144, allowing the program to call other programs that require a 144-byte save area.

In an F8SA, the high halves are stored at offset 216. This allows the 64-bit program to call AR mode programs that require a 216-byte save area.

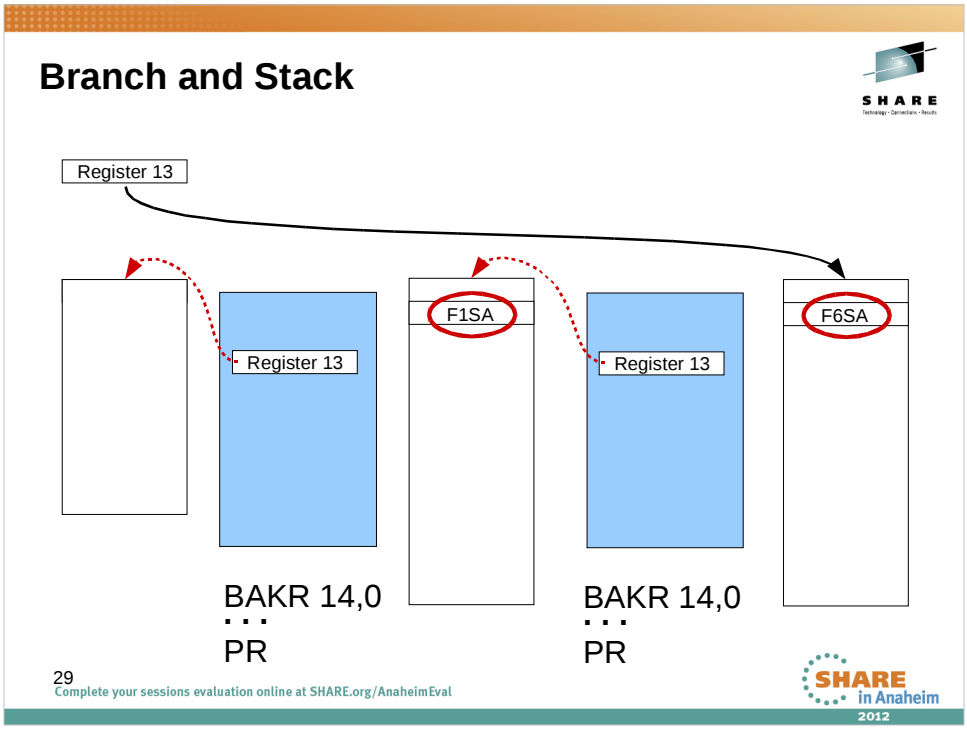


Program B can then call an AR mode program that requires a 216-byte save area. AR mode program C does not know or care that the high halves of program A's registers are stored at offset 216 of program B's save area.

A question and Peter Relson's answer

Q: I have another question about the example of using F5SA and F8SA. The documentation of GETMAIN says that it uses register 14 as a work area. Because of this, shouldn't the high half of reg 14 be saved prior to the GETMAIN in the same way that registers 0, 1 and 15 are saved?

A: GETMAIN won't clobber the high half of reg 14. Only the low half (which matches the linkage standard whereby a routine does not clobber high halves of regs 2-14). Otherwise, you're certainly right that it would be necessary to preserve that half of reg 14 before calling GETMAIN.

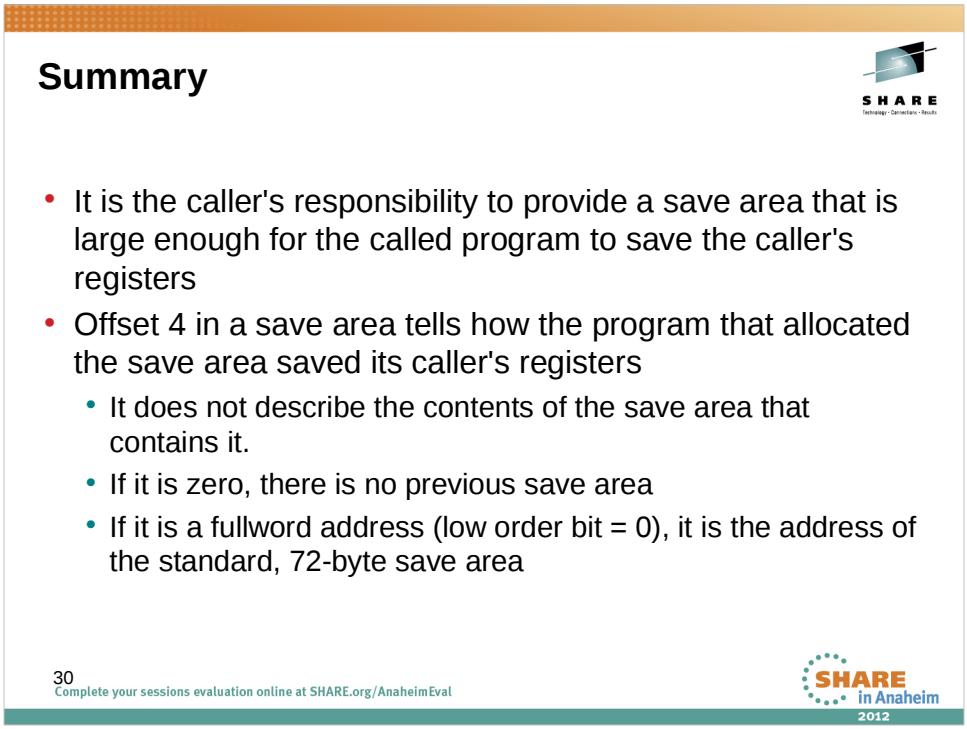


There is another way to save the caller's registers that does not use the save area that is passed in register 13. When a program issues a BAKR 14,0 instruction, the registers are saved on the Linkage Stack for the current task. The Return PSW in the stack entry contains the current PSW modified with the instruction address in Register 14.

The program then allocates a save area that is sufficient for any programs that it will call. Offset 4 is set to “F1SA” or “F6SA” to indicate that the caller's registers were saved on the Linkage Stack. There is no difference between F1SA and F6SA. Every program that uses the Linkage Stack should create an F1SA or F6SA.

Register 13 in the stack entry leads back to the save area that was passed to this program. This allows the calling chain to be followed backward when analyzing a dump. Finding the Linkage Stack entry once an F1SA or F6SA is found in a dump can be more difficult. It requires that the linkage stack entries be matched to save areas marked F1SA/F6SA. It is not possible to follow forward from a linkage stack entry to the F1SA or F6SA.

When the program is finished, it issues a PR instruction, restoring the registers, Access Registers, ASC mode and PSW from the current entry on Linkage Stack, making the previous entry the current one.



A called program does not examine the save area that was passed to it to decide how to save its caller's registers.

The second word of a save area does not describe the save area that contains it. It describes the way that the program that created the save area saved its caller's registers, either in the caller's save area or on the linkage stack.

When the program that created the save area saved its caller's registers in standard 72-byte format, it is the address of that save area. When the program saved its caller's registers in some other way, it is a code to indicate how those registers were saved.

Since all save areas must be on at least a fullword boundary, if the value of the word at offset 4 has the low order bit on, it can not be the address of a standard save area.

The EBCDIC character “A” is X'C1' so if the low order bit of the word at offset 4 is off, it can only be a standard 72-byte save area.

Summary

- Save area chains can only be followed predictably by going backward from register 13
- Standard save area
  - 72 bytes, 18 fullwords, on a fullword boundary
  - Back chain pointer is in offset 4
  - Forward chain pointer is in offset 8
- New save area format (ESA)
  - F1SA and F6SA
    - No difference between them
    - Must be large enough for any called programs
    - There is no back chain pointer

The backward chain is a necessary part of the save area. Every program must be able to find the save area where it saved its caller's registers. The forward chain is not required and there are two possible locations for it. As a result, the only predictable way to follow a calling chain is going backward.

A standard save area is 72 bytes or 18 fullwords. It must be allocated on a fullword boundary.

A save area marked as F1SA or F6SA must be on an appropriate boundary and be of at least the minimum size needed for the programs that it will call. It might be 18 words, 18 doublewords or 27 doublewords. If it is known that the program will *never* call another program, it can allocate an 8-byte area with “F1SA” or “F6SA” in the second word. Doing this will ensure that that someone doing analysis of a dump will know when to look for a linkage stack entry.

F1SA and F6SA do not have a back chain pointer; the previous registers were saved in the current entry in the linkage stack. Dump analysis requires matching F1SA/F6SA to the linkage stack entries.

Summary

- New 64-bit save area formats
  - Four formats
    - F4SA – 18 doublewords, 144 bytes
    - F5SA – 27 doublewords, 216 bytes
    - F7SA – 27 doublewords, 216 bytes
    - F8SA – 36 doublewords, 288 bytes
  - Allocated on a doubleword boundary
  - Back chain pointer at offset 128, X'80'
  - Forward chain pointer at offset 136, X'88'
- The Assembler Services guide has sample code for saving and restoring the registers using all of these formats

144-byte, 216-byte and 288-byte save areas are all allocated on a doubleword boundary.

Use the IHASAVER macro to map the fields in the save areas.

The sample code in Chapter 2 of the Assembler Services Guide has all been tested and verified to work correctly. Use it.

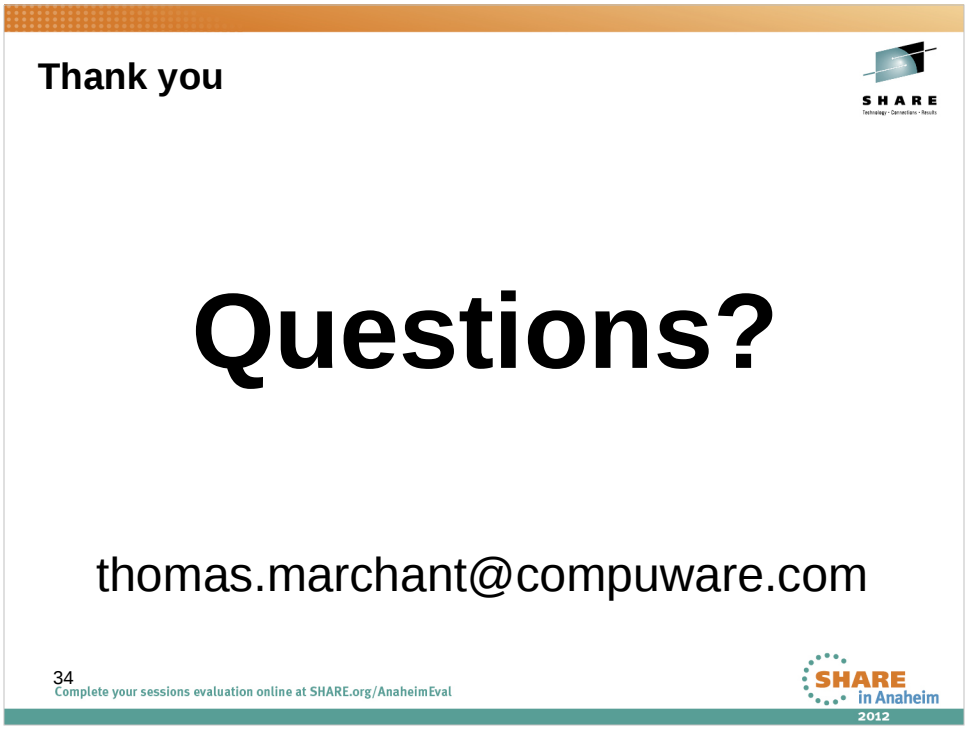
Additional information

- The Assembler Services Guide, Release 12 or higher
  - SA22-7605-12 Release 12
  - SA22-7605-14 Release 13
- z/Architecture Principles of Operation, SA22-7832
- IHASAVER macro

The Linkage Conventions chapter, Chapter 2, in the Assembler Services Guide has been extensively rewritten for the release 12 edition of the manual. It has sample code fragments showing how to save and restore the registers using all of the formats covered in this session.

The Principles of Operations manual describes the operation and contents of the Linkage stack. In z/OS, every task has its own Linkage Stack.

The IHASAVER macro maps all of the save area formats.



The best way to contact me with any questions is to send me email.