The Trainer's Friend, Inc.

6790 E. Cedar Ave., Suite 201
Denver, Colorado 80224
U.S.A.

Telephone: (800) 993-8716
(303) 393-8716
E-mail trainers@trainersfriend.com
Internet www.trainersfriend.com

Applications Assembler Programming for z

A Technical Discourse

Acknowledgements

I wish to thank the following people for their help and input on this paper.

Hunter Cobb, The Trainer's Friend - for raising the questions that lead to my writing the paper, and for assistance in checking the content.

Ed Jaffee, Phoenix Software - for posting code snippets that provided me with new insights and a deeper understanding

Abe Kornelis - for careful proofing and thoughtful suggestions for improvements

Lindy Mayfield - for careful reading and pointing out at least one embarrassing typo

Charles Mills - for causing me to think more deeply about the implications and details of reentrant code

Gerhard Postpischil - for spotting some typos and reporting them to me

Many others on usenets and listservs that I follow daily.

Section Preview

- **Applications Assembler Coding for z**
  - **Modern Applications Programming in Assembler**
  - **64 bit Registers**
  - **Importance of the Cache**
  - **The Goal of Baseless Programming**
  - **Relative Branching**
  - **The Long Displacement Facility**
  - **The Extended Immediate Facility**
  - **AMODE Implications**
  - **Changes in Program Linkages**
  - **Save Area Conventions**
  - **Coding for Reenterability**
  - **Establishing the Environment**
  - **Coding Style**
  - **Code Samples**

Modern Applications Programming in Assembler

This paper is an attempt to describe various techniques for Assembler coding when working in z/OS. The focus is on applications programming. Assembler programmers for ISVs (Independent Software Vendors) often have to use system hooks, exits, and facilities that applications people needn't worry about. Still, there are probably thoughts in here that will be of interest to ISV programmers too.

The issues we are looking at have to do with writing maintainable code that takes advantage of the advances in the Assembler itself and in the hardware instruction set that has been enhanced enormously in recent years. Some of the facilities we discuss predate the z Systems but are recent enough to warrant inclusion in the new styles of programming.

From a hardware perspective, then, we want to be aware of:

* Impact of the cache on performance
* Impact of 64-bit registers
* Impact of 64-bit addressing
* Availability of new instructions, specifically these features:
    + Relative branching
    + The Long Displacement facility
    + The Extended Immediate facility

From a software perspective, we want to take advantage of:

* Facilities for linkages in a 64-bit world
* Macro instructions that support 64-bit virtual
* Changes to existing macros
* Assembler facilities such as new extended mnemonics

Note that we don't actually discuss writing AMODE 64 code in this paper, but rather we discuss preparing to write AMODE 64 code. This paper got longer than expected, so coding AMODE 64 programs is discussed in a separate paper.

Modern Applications Programming in Assembler

These notes are not concerned with the latest "tricks", or with coding for special situations, but rather how to code basic linkages and when to use new instructions and capabilities for the common cases. We assume you do not have to write code that works in both newer and older processors, just the newer models of z series machines.

The focus here is writing new code, but some of the ideas can surely be used in maintaining existing code.

The major issues we are trying to meld are:

* 64-bit registers
* Importance of the cache
* The goal of baseless programming
* Importance of relative branching
* Long displacement facility
* Extended Immediate facility
* AMODE implications

Keep in mind there are many ways to accomplish our task, and many people have different approaches, all of which work. Our only criteria for program goodness, ultimately, are these:

* It must work correctly
* Performance must be acceptable

other useful criteria:

* Must be easy to read
* Must be easy to run / operate
* Must be easy to maintain / modify

64-bit Registers

All general purpose registers (GPRs) on z/Architecture systems are 64 bits in length. This does not mean you need to be in 64-bit addressing mode to use all 64 bits. For example, most instructions for doing binary arithmetic still work on 32-bit values. To do 64-bit integer arithmetic you must use the new instruction set, but you can do this while running in AMODE 24, 31, or 64.

When an address is calculated by the CPU, whether it is a branch address or the address of a data item, the CPU always creates a 64-bit intermediate address. Then the effective address is either used as is (if you are running AMODE 64) or truncated to 24-bits or 31-bits (if you are running AMODE 24 or AMODE 31, respectively).

When the contents of a register are referenced as data, which bits participate depend on the instruction used, not the current AMODE.

Importance of the Cache

Each CPU has an I-cache (Instruction fetch cache) and a D-cache (Data cache); cache is retrieved and restored to memory in 256 byte "lines". I-cache contents are assumed to be read only; D-cache contents are assumed to be modifiable (and thus may need to be reflected back in memory). If there are instructions and data in the same cache line and any of the data is changed, both caches need to be refreshed. This can cause a big performance hit. So the coding guidelines include:

separate instructions and data areas by [at least] 256 bytes
  note that even non-modifiable data in the I-cache takes
  up room that could be used to hold more instructions

code in a reentrant style
  reentrant code by its very nature separates instructions and data
  and never causes cache refresh due to storing into itself

Pipelines

You might be aware that the processor has multiple pipelines feeding it, anticipating and preparing instructions and data in advance of their use. If some instruction must wait for the result from a prior instruction, there can be a pipeline stall; when the processor takes a branch path that was not anticpated or expected, all or part of the pipeline can be flushed. But these behaviors are not influenced much by whether code is reentrant or baseless. Stalls are influenced by the sequence of instructions coded, and flushes can be minimized by thinking about the conditional branch choices you make. Neither of these concerns are addressed in this paper.

Think of the pipeline as anticipating the future, and the cache as saving the recent past, in case it comes in handy.

The Goal of Baseless Programming

One of the major difficulties in writing or maintaining Assembler code has been running out of registers to use as base registers for addressability to instructions or data.

A number of instructions and facilities have been introduced to help alleviate that problem. On the hardware side, relative branching, long displacement, and extended immediate facilities have had the largest impact. Collectively they have led to an informal movement to write what is called "Baseless programming".

This term is a bit of a misnomer. You need base registers for addressability to your data items. But with a little care you can remove or reduce the need for base registers for your instructions!

On the software side, the Assembler's support of labeled USINGs and dependent USINGs can reduce the number of registers you need for covering DSECTs.

While baseless programming is a worthwhile goal, we will find in some cases it is very difficult to accomplish. Do not let the wrong criteria drive your efforts. If you can make a program be "baseless", then good for you. But if the effort outweighs the results, don't waste time on it.

Relative Branching

The relative branching instructions do not branch to an address specified by a base+displacement or base+index+displacement, but rather the target address is specifed as a relative location from the instruction itself. That is, "branch to the address ***x*** bytes away from here". ***x*** is specified as an immediate halfword or fullword signed integer indicating how many halfwords away the target is. You typically code the immediate operand as a label and the Assembler computes the displacement for you. Halfword integers allow for branching to locations ±64KiB away; Fullword integers allow for branching to locations ±4GiB away.

Here is a summary of available branching instructions, and, for conditional branches, what mnemonics to use in order to generate relative branches.

BAL, **BALR** - first, these should be replaced with **BAS**, **BASR** respectively

but then, **BAS** can be replaced by one of these:

BRAS - Branch Relative and Save; halfword immediate; 4 byte instruction
BRASL - Branch Relative and Save Long; fullword immediate; 6 bytes

Note: you can code **JAS** and **JASL**, respectively for these instructions
Note: there is no replacement for **BASR**, since it only uses registers

BC - Branch on Condition has two counterparts:

BRC - Branch Relative on Condition; halfword immediate; 4 byte instruction
BRCL - Branch Relative on Condition Long; fullword immediate; 6 bytes

BCT - Branch on Count, has two slightly different counterparts:

BRCT- Branch Relative on Count; halfword immediate; 4 byte instruction; the count register (first operand) is a 32-bit register value
BRCTG - Branch Relative on Count Grande; halfword immediate; 4 byte instruction; the count register (first operand) is a 64-bit register value

Also, we have **BCTG** and **BCTGR** which are "Grande" versions of **BCT** and **BCTR**, but not "relative" versions

As with **BASR**, there are no "relative" counterparts for **BCR** or **BCTR**

Relative Branching Extended Mnemonics

Mnemonic	stands for this	can also use	Interpretation
J	BRC 15	BRU	Unconditional
JNOP	BRC 0	BRNOP	no-op
JH	BRC 2	BRH	High
JL	BRC 4	BRL	Low
JE	BRC 8	BRE	Equal
JNH	BRC 13	BRNH	Not High
JNL	BRC 11	BRNL	Not Low
JNE	BRC 7	BRNE	Not Equal
JP	BRC 2	BRP	Positive
JM	BRC 4	BRM	Minus
JZ	BRC 8	BRZ	Zero
JO	BRC 1	BRO	Overflow
JNP	BRC 13	BRNP	Not Positive
JNM	BRC 11	BRNM	Not Minus
JNZ	BRC 7	BRNZ	Not Zero
JNO	BRC 14	BRNO	No Overflow
JLU	BRCL 15	BRUL	Unconditional
JLNOP	BRCL 0	n/a	no-op
JLH	BRCL 2	BRHL	High
JLL	BRCL 4	BRLL	Low
JLE	BRCL 8	BREL	Equal
JLNH	BRCL 13	BRNHL	Not High
JLNL	BRCL 11	BRNLL	Not Low
JLNE	BRCL 7	BRNEL	Not Equal
JLP	BRCL 2	BRPL	Positive
JLM	BRCL 4	BRML	Minus
JLZ	BRCL 8	BRZL	Zero
JLO	BRCL 1	BROL	Overflow
JLNP	BRCL 13	BRNPL	Not Positive
JLNM	BRCL 11	BRNML	Not Minus
JLNZ	BRCL 7	BRNZL	Not Zero
JLNO	BRCL 14	BRNOL	No Overflow
JAS	BRAS		
JASR	BRASL		
JCT	BRCT		
JCTG	BRCTG		

More Relative Branching

The "index" branching instructions also have both Grande and relative counterparts:

BXH - Branch on Index High has
  BXHG - Branch on Index High Grande
  BRXH - Branch Relative on Index High (may use **JXH** mnemonic)
  BRXHG - Branch Relative on Index High Grande (may use **JXHG**)

BXLE - Branch on Index Low or Equal has
  BXLEG - Branch on Index Low or Equal Grande
  BRXLE - Brand Relative on Index Low or Equal (may use **JXLE** mnemonic)
  BRXLG - Branch Relative on Index Low or Equal Grande (may use **JXLEG**)

The remaining two branching instructions, **BASSM** and **BSM** do not have relative nor Grande counterparts: they simply work differently depending on the current addressing mode.

The first step in writing baseless code is to replace all non-relative branch instructions that have a storage address target with relative branch instructions (or equivalent mnemonics). This much generally works with no other changes. Except for macros that may have the older branch instructions embedded in them, including IBM supplied macros, although many have changes such as:

* Sensitivity to ARCHLVL in the SYSSTATE macro settings
* Options for you to specify which branch instructions you prefer

Also, IBM provides a copy book, IEABRC, that maps non-relative branch instructions to relative branches. You issue "**COPY IEABRC**" near the top of your code (at least before you issue any IBM macros) and this is pretty good magic. It doesn't work for all formats of branches, but it has saved us a lot of work from time to time.

The Long Displacement Facility

This facility introduced some new instructions, and enhanced some existing instructions, to support 20-bit signed displacements instead of 12 bit unsigned displacements. This allows an instruction to reference a storage location up to ±512KiB from the base register address, instead of the 12-bit range of 0-4095 bytes from the base register address. These instructions include:

OpCode	extends	name	OpCode	extends	name
AY	A	Add	AHY	AH	Add halfword
ALY	AL	Add Logical	CY	C	Compare
CHY	CH	Compare Halfword	CLY	CL	Compare logical
CLIY	CLI	CompareLogicalImm.	CLMY	CLM	Compare Logical Under Mask
CVBY	CVB	Convert to Binary	CVDY	CVD	Convert to Decimal
ICY	IC	Insert Character	ICMY	ICMY	Insert Chr. Under Mask
LY	L	Load	LAY	LA	Load Address
LB	n/a	Load Byte	LGB	n/a	Load Byte Grande
LHY	LH	Load Halfword	LMY	LM	Load Multiple
MVIY	MVI	Move Immediate	MSY	MS	Multiply Single
NY	N	aNd	NIY	NI	aNd Immediate
OY	O	Or	OIY	OI	Or Immediate
STY	ST	Store	STCY	STC	Store Character
STCMY	STCM	StoreCharUnderMask	STHY	STH	Store Halfword
STMY	STM	Store Multiple	SY	S	Subtract
SHY	SH	Subtract Halfword	SLY	SL	Subtract Logical
TMY	TM	TestUnderMask	XY	X	eXclusive or
XIY	XI	eXclusive or Immediate			

Generally speaking, the earlier instructions are four byte instructions while the long displacement versions are six byte instructions. So they take 50% more space in a program, possibly requiring additional base registers(!). On the other hand, these instructions can reference data more than 4KiB away from a base register, so you may need fewer base registers. In this case, you need to take some time and think about each particular program to decide if any long displacement instructions will help you.

The Extended Immediate Facility

This facility, introduced in models announced in September, 2005, provides instructions that have 32-bit immediate operands. This helps by not requiring defining data areas (no base/displacement space), and by including data fetch as part of the instruction fetch cycle. The instructions in this facility include:

AFI - Add Fullword Immediate
AGFI - Add Grande Fullword Immediate
ALFI - Add Logical Fullword Immediate
ALGFI - Add Logical Grande Fullword Immediate
CFI - Compare Fullword Immediate
CGFI - Compare Grande Fullword Immediate
CLFI - Compare Logical Fullword Immediate
CLGFI - Compare Logical Grande Fullword Immediate
IIHF - Insert Immediate High Fullword (leftmost word in 64-bit GPR)
IILF - Insert Immediate Low Fullword (rightmost word in 64-bit GPR)
LGFI - Load Grande Fullword Immediate (load fullword then sign extend)
LLIHF - Load Logical Immediate High Fullword (zero reg. then IIHF)
LLILF - Load Logical Immediate Low Fullword (zero reg. then IILF)
NIHF - aNd Immediate High Fullword
NILF - aNd Immediate Low Fullword
OIHF - Or Immediate High Fullword
OILF - Or Immediate Low Fullword
SLFI - Subtract Logical Fullword Immediate
SLGFI - Subtract Logical Grande Fullword Immediate
XIHF - eXclusive or Immediate High Fullword
XILF - eXclusive or Immediate Low Fullword

One extremely important instruction for baseless programming that does not belong to any facility (it's part of the original System z hardware) is:

LARL - Load Address Relative Long; it takes a signed fullword immediate operand as a relative displacement (in halfwords) from the instruction location itself and places the resulting address into the first operand (a register)

AMODE Implications

On z machines, a program can be initially started in 24-bit AMODE, 31-bit AMODE, or 64-bit AMODE. Then a program can switch AMODEs (note that a program running above the line can not run AMODE 24, but a program running below the line can run in any of the three modes). A program can call a subroutine and switch AMODEs at the point of call (static call; assumes subroutine is able to handle the specified AMODE). Dynamic calls use system-assisted linkages, and these can also involve changing AMODE based on the binder properties of the target module.

Some instructions operate differently in different AMODEs. Primarily this manifests itself in whether calculated addresses end up being 24-bit, 31-bit, or 64-bit addresses, and whether unused bits are set to zeros or left unchanged. All addresses are always first calculated as 64-bit "intermediate" addresses, and then the effective address is calculated depending on the current AMODE.

Switching AMODE can result in some surprises, primarily when the leftmost word of a register used for addressing contains non-zero values and you switch to or from AMODE 64.

An informal goal of this paper is to make suggestions for writing code that is not initially expected to run AMODE 64, but to make coding decisions that might make it easier to change to run in AMODE 64 if that becomes necessary.

Changes in Program Linkages

The tradition of having R15 point to the entry address of a program on entry is not maintained in AMODE 64. If an instruction has been branched to using BASSM, and the instruction is to get control in AMODE 64, the rightmost bit of the target address in the second operand must be set on. This makes the address an odd number, which is not valid.

A better way is to use the new LARL instruction (Load Address Relative Long). This instruction has two operands: a register and an immediate fullword of data. The second operand contains a signed integer indicating how many halfwords away, relative to the location of the instruction itself, to use in address calculation. So the current address + 2*(second_operand) is placed into the first operand.

Example:

```
LARL  12,MYPGM
```

instead of

```
LR    12,15
```

The nice thing is LARL works in all AMODEs, so you might as well get into the habit of using LARL in all routines, regardless of AMODE. One less decision to make in your life.

So you want to code in a way that handles all AMODEs as much as possible. Two other related issues:

* Save areas need to allow for 64-bit registers [old routines do not do this]
  NOTE: All save areas must be located below the bar
* Parameters may need to be passed as 64-bit addresses

Save Area Conventions

z/OS has made a number of changes in save area formats in support of 64-bit registers. In particular, the format called F4SA ("Format 4 Save Area") is 18 doublewords (144 bytes) and designed to be used when calling programs that start out in AMODE 64. The layout is:

+ 0	x'00000000' c'F4SA'
+ 8	c(GPR14)
+ 16	c(GPR15)
.	
.	
.	
+120	c(GPR12)
+128	a(previous_savearea)
+136	a(next_save_area)

Note that this changes the traditional forward chain pointer and backward chain pointer to the end of the save area instead of at the beginning

Room is provided for 64-bits for all registers

* A subroutine can check something like: **clc 4(4,13),=c'F4SA'** if it expects it might get invoked using this kind of save area
* The convention remains that R13 points to the save area on entry to the program

Save Area Conventions, continued

An alternative save area format is F5SA. This is a F4SA with 16 words (64 bytes) appended. The last 16 words are for saving the high order words of the passed registers. So the program that is called using a standard save area that needs to preserve the values in the high order word of the registers because it will be using at least some registers as 64-bit values, should use a F5SA for this purpose. (This will only occur in a subroutine, see next paragraph.)

From z/OS V1R3 on, the system always supplies a 144-byte save area (although it doesn't have the 'F4SA' string in it) to programs it invokes (that is from PGM= on EXEC JCL statement, from TSO CALL, or from dynamic call structures (LINK, XCTL, ATTACH, etc.)).

A passed parameter list is still set up with R1 pointing to a list of addresses. The list can be above the bar, or point to addresses above the bar, but in either of these cases the caller and the callee both have to know the format of the list. [Also, if the parameter list consists of 64-bit addresses, the tradition of turning on the leftmost bit of the last entry is not supported.]

New code should always provide a 144-byte save area; whether subroutines use it or not does not matter. Each program can continue to use classic save area linkages or it can save registers in the F4SA style: since the saving and returning is localized to your program, you can choose either approach with no impact on other programs.

Some programmers prefer to use BAKR / PR linkages, which use a system-provided stack to save GPRs and ARs. The author is not fond of this approach for several reasons: 1) save area chains are not available or meaningful in a dump; 2) it's possible to get a stack full exception; 3) the instruction is described in the chapter on Control Instructions, not the chapter on General Purpose Instructions, in the Principles of Operations; 4) performance can be quite slow.

Coding for Reenterability

For the application programmer, the main work in making a program reentrant is ensuring no byte of the load module is modified while the program is running. This is accomplished by:

* Dynamically obtaining storage from outside the load module
* Initializing this storage with values from the constants area of the load module
* Referencing the external storage version of items in instructions and macro operands for operations where the items might be changes

Obtaining storage may be done by invoking any one of several system services, including:

* GETMAIN - around the longest; fastest; not as many options / features
* STORAGE - more recent; more sophisticated options
* CPOOL - cell pools; most appropriate when working with many "cells" of the same size
* IARV64 - for obtaining large amounts of storage (>1MB) above the bar
* IARCP64 - for obtaining cellpools above the bar
* IARST64 - for obtaining small amounts of storage (<=64K) above the bar
* CEEGTST - LE callable routine

[For most application programs, we recommend GETMAIN as by far the best service to use. This macro and its service are not going to be enhanced, however, so if you have advanced or special needs look at STORAGE for below the bar requests or IARV64 or IARST64 for above the bar requests.]

Then create a DSECT that describes the items in the GETMAINed area. After issuing your GETMAIN you receive the address of the storage back in R1; copy this value into another register and establish the register as the base for the DSECT. Next copy initial values from your constant area into their respective DSECT locations.

This can be quite tedious and confusing at coding time, so here are some tips to help you…

Coding for Reenterability

For IBM-supplied macros, use List and Execute forms where possible. Copy the List form into your GETMAINed area and reference this location from your Execute form. We provide specific examples later.

Establish a naming convention for data items. Perhaps "src_" as a prefix for source values (constants) then copy to the data items in your DSECT from src_ with the same names.

If source values are in the same order and displacements in your constants area as in your DSECT, a single move can copy multiple fields at once, thus gaining performance. But be careful when doing maintenance.

You can use a DSECT area for multiple occurrences of a parm area. For example, a list form of a CALL macro should allow for the maximum number of parms in any of your calls, say:

```
plist    call  ,(0,0,0,0),mf=l
```

then every call in your code can reference this, like:

```
         call  rtn1,(string1,string2),vl,mf=(e,plist)
         call  rtn2,(string3)vl,mf=(e,plist)
         call  rtn3,(x,y,z,four),vl,mf=(e,plist)
```

Always have the Assembler calculate lengths for you, for example:

```
WTO      wto   text=,routcde=(11),mf=l
string1  dc    h'12',cl12'Twelve bytes'
string2  dc    h'26',cl26'This is the second string.'
string3  dc    h'258',258c'9'
size_tot equ   *-WTO
```

then you can use the equated symbol in instructions and constant definitions, for example:

```
         LA    3,size_tot
```

or

```
incr     dc    a(size_tot)
```

Coding for Reenterability

One approach that combines writing code that is both reentrant and "baseless" is to only provide base registers for data:

* One [or more] for constant area
* One [or more] for GETMAINed area (this should include the save area you establish, at the beginning of it)
* None for your CSECT *per se*

The major difficulty is macro expansions that include classic branches, like:

```
+               B     *+8
```

these kinds of branches require addressability (they resolve as base / displacement) within the CSECT.

We'll demonstrate ways to work around this for some specific IBM macros later, which may give you some insights for handling locally written macros with the same shortcomings. As we said before, it may be more trouble than it's worth in some cases.

* For a main program, being reentrant minimizes or eliminates cache refreshes, so there is some [potential] performance improvement
  + but there is no gain to being shared by multiple tasks, since a 'main' is, by definition, the top task
  + in fact, a main program may not be truly reentrant and you won't be able to tell when submitted from JCL or TSO CALL

* However, if a main program is put into the Link Pack Area, and you invoke it via JCL (use no STEPLIB nor JOBLIB), the program must be RENT and run from an authorized library

  If the program is marked RENT but run from a non-authorized library, it might run cleanly some times and abend other times

* Finally, if a program is to be run under the OMVS shell, it must be reentrant (even if it is not bound / linked as RENT)

Establishing the Environment

For standard macros to give expansions using new instructions or in a particular AMODE or style, you can issue the SYSSTATE macro specifying the desired environment; this sets some global symbol values, and various system services macros test these and generate appropriate instructions.

There are only two parameters the application programmer is concerned with, AMODE64 and ARCHLVL. AMODE64 can be YES or NO, with a default of NO; specify YES if you want macros to be aware of running AMODE64. ARCHLVL is 0, 1, or 2, with a default of 0; options 0 and 1 are for ESA/390 and ESA/390 with some retrofitted instructions. You want 2: z/Architecture. So you would normally code a SYSSTATE as:

```
[name]     SYSSTATE ARCHLVL=2,AMODE64=YES
```

or

```
[name]     SYSSTATE ARCHLVL=2,AMODE64=NO
```

Old code will work as before, without a SYSSTATE macro, because the defaults are set up for old code to Assemble as before.

You might have occasions to issue these at different points in your program, in particular changing AMODE64 if your program switches into and out of AMODE 64, so that macros in different ranges of your code generate the correct instruction sequences.

Remember: SYSSTATE is only used to set global symbols used by macro expansions. If you are not using macros impacted by these settings, you don't need to code this macro at all.

Establishing the Environment, 2

Some of the commonly used macros that test these two parameters are:

ARCHLVL	AMODE64
ABEND	
ATTACH	ATTACH
	ATTACHX
CALL	CALL
ENQ	ENQ
DSPSERV	DSPSERV
FREEMAIN	FREEMAIN
GETMAIN	GETMAIN
IARV64	IARV64
LINK	LINK
	LINKX
LOAD	
	RETURN
SAVE	SAVE
STORAGE	STORAGE
	TIME
	WTO
	WTOR
XCTL	XCTL
	XCTLX

Coding Style

There are a number of issues regarding coding style that we should address:

- **Case sensitivity - originally Assembler programs were written in all uppercase: after all, keypunches didn't have lowercase characters**

  - ✗ For a long time now, the Assembler has allowed mixed case code, even in macro invocations, although you should code
    *PROCESS COMPAT(NOCASE,MACROCASE)
    at the top of your code, to enable this

  - ✗ We have come to prefer mixed case code, some people despise it, and the rest are indifferent

  - ✗ Some characters are hard to differentiate when reading mixed case code; especially lowercase 'el', uppercase 'eye', and numeric 'one'; compare for readability:

```
ds    cl111
ds    CL111

cl    3,fielda
CL    3,fielda
```

  - ✗ Our examples will tend to be mostly lower case, but we will uppercase letters where it seems to help readability

Coding Style

- **<u>Long names</u> - originally names had to be a maximum of 8 characters and only alphanumeric and national characters**

  - ✗ For a long time now, the Assembler has allowed names up to 63 characters in length, and you can use an underscore (_) as a valid character in a name

- **<u>Program organization</u> - there are very few rules you must follow**

  - ✗ The major style approaches, historically, have been:
    - ➢ All instructions at top, data items at bottom
    - ➢ Linkage code, jump around data items, data items, bulk of program logic
    - ➢ Mix data near instructions where data used; this involves a lot of jumping around so you don't fall into data
  - ✗ We have, ourselves, gone through various swings in approach; our current thinking is this:
    - ➢ For non-reentrant programs where you provide one or more base registers for everything, the first approach makes sense
    - ➢ For reentrant programs where you provide a base register for instructions, the first and second approaches make sense
    - ➢ There is no case where the third approach makes sense
    - ➢ For reentrant, baseless programs, this organization seems best: linkage code, logic, constant data items, dsects for work areas (very much like the first option); however note that if you make use of certain attributes (*e.g.*, D', O', and T') you must define symbols above these kinds of references)

Coding Style

- **Use of literals - we are not a fan of literals, although many are; we always define a data item and reference that name**

- **Structured macros - there is a separate, priced, Assembler supplement called the HLASM toolkit which includes some pretty neat tools: structured macros, program analysis routines, and more**

  - ✗ We do not use the structured macros because: 1) not everybody has the package, and 2) we find the resulting code more difficult to read, not less; but, to each his own

- **Register equates - we use these sometimes, but not consistently, just a personal foible**

  - ✗ You can include equates for the GPRs by including the macro YREGS any place in your source code

  - ✗ There is another macro, ASMDREG, that includes equates for all the registers; our version was found in a data set named HLA.SASMMAC2

  - ✗ Many shops have their own macros that take care of this, perhaps in conjunction with setting up initial addressability

Code Samples

The rest of this paper consists of code fragments demonstrating various techniques to consider for use in your code. All our code samples, reentrant or not, baseless or not, are designed to be AMODE 31, RMODE 31. The base code contains two WTO macros, a CALL macro, and some simple I/O, to help us verify correct Assembly of at least a minimal level, when we tested the programs.

The programs we used for these experiments:

PGM1 - non-reentrant, not baseless; traditional

PGM2 - non-reentrant, but baseless

PGM3 - reentrant, not baseless; traditional reentrant style

PGM4 - reentrant and baseless; the ultimate goal

The complete programs are available in the ASM library of the base programmers toolkit from The Trainer's Friend.

Visit http://www.trainersfriend.com/TTFStore/index.html for more information about toolkits.

Code Samples

Entry / exit linkages - non-reentrant, not baseless

```
❶  *PROCESS COMPAT(NOCASE,MACROCASE)
❷           sysstate archlvl=2
   pgm1     csect
   pgm1     amode 31
   pgm1     rmode 31
            using pgm1,12
            save  (14,12)
❸           larl  12,pgm1
            st    13,save+4
            la    13,save
   .
   .
   .
            l     13,save+4
            return (14,12),,rc=0
   .
   .
   .
❹  save     dc    36f'0'
            end   pgm1
```

- **This is pretty classic / standard except for lines**

  ❶ - the *process to support mixed case
  ❷ - the sysstate macro to support z/Architecture
  ❸ - the LARL instruction to get addressability
  ❹ - the save area size of 36 fullwords instead of 18 fullwords
  (actually, it might be more clear to code the request as
  save dc 18D'0' to indicate a request
  for 18 doublewords)

Code Samples

Entry / exit linkages - non-reentrant, baseless

```
   *PROCESS COMPAT(NOCASE,MACROCASE)
             sysstate archlvl=2
   pgm2      csect
   pgm2      amode 31
   pgm2      rmode 31
❶            copy  ieabrc
             save  (14,12)
❷            using data,12
❷            larl  12,data
             st    13,save+4
             la    13,save
   ...
             l     13,save+4
             return (14,12),,rc=0
   ...
❸  data      ds    0f
   save      dc    36f'0'
             end   pgm2
```

Notes

❶ - we needed to insert the **copy ieabrc** to make the wto and call macro expansions work; for the reentrant case, this is not needed

Even so, we had to do extra work to make the call work; we discuss our findings in later examples

❷ - here is where we establish addressability to our data area

❸ - we needed to provide a label at the beginning of our data area to indicate where addressability should start (we could have used the existing **save** label, but felt it better to create a new label that only has this purpose)

Code Samples

Entry / exit linkages - reentrant, not baseless

```
   *PROCESS COMPAT(NOCASE,MACROCASE)
            sysstate archlvl=2
   pgm3     csect
   pgm3     amode 31
   pgm3     rmode 31
            using pgm3,12
            save  (14,12)
            larl  12,pgm3
❶           getmain r,lv=worksize
❷           st    13,4(1)
❸           lr    13,1
❹           using wareas,13
   ...
❺           lr    1,13
❻           l     13,4(13)
❼           freemain r,lv=worksize,a=(1)
            return (14,12),,rc=0
   ...
❽           ltorg
❾  wareas   dsect
   save     ds    36f'0'
   ...
❿  worksize equ   *-save
            end   pgm3
```

What's new

❶ - here we obtain storage for our save area and changeable variables
❷ - ❹ - here we link the save area and provide addressability
❺ - ❻ - save the address of gotten storage, point back to incoming SA
❼ - free gotten storage before leaving
❽ - although we don't use literals, macros sometimes do; this gathers literals together, out of the way of our DSECT addressability
❾ - gather description of external storage into a DSECT
❿ - note we have the Assembler calculate size of storage to get

We also had to change some of the logic for macro calls - discussed later

Code Samples

Entry / exit linkages - reentrant, baseless

```
   *PROCESS COMPAT(NOCASE,MACROCASE)
            sysstate archlvl=2
   pgm4     csect
   pgm4     amode 31
   pgm4     rmode 31
❶           using data,12
            save  (14,12)
❷           larl  12,data
            getmain r,lv=worksize
            st    13,4(1)
            lr    13,1
            using wareas,13
   ...
            lr    1,13
❸           l     0,fsize
            l     13,4(13)
            freemain r,lv=(0),a=(1)
            return (14,12),,rc=0
   ...
   data     ds    0f
❹  fsize    dc    a(worksize)
   ...
            ltorg
   wareas   dsect
   save     ds    36f'0'
   ...
   worksize equ   *-save
            end   pgm4
```

What's new

❶ and ❷ - addressability to data, not the CSECT

❸ - something about the FREEMAIN expansion requires this (but not the GETMAIN expansion)

❹ - use an adcon to generate an unsigned integer, since cannot code f(*equate_symbol*)

Code Samples

At this point we discuss some of the requirements / idiosyncracies of some commonly used IBM macros, in each of these four contexts.

Getmain

- **This macro has a lot of operands and options, but here are the ones we care about:**

```
GETMAIN R,LV=size
```

- *size* **may be specified as an integer or a symbol or as LV=(0), where you have pre-loaded the number of bytes into R0; the system rounds up to a multiple of 8, if necessary**
- **This form of the macro can only obtain storage below the line, which is desirable if you will be putting DCBs in your gotten storage, which is required if you are running AMODE 31**
- **The address of the gotten storage is returned in R1**

```
GETMAIN RU,LV=size
```

- **Same as above, except storage can be above the line, you can further extend the request by one of these:**

```
GETMAIN RU,LV=size,LOC=24
GETMAIN RU,LV=size,LOC=31
GETMAIN RU,LV=size,LOC=(24,31)
GETMAIN RU,LV=size,LOC=(31,31)
```

- **And so on; '24' means "below the bar", '31' means above the bar, and when you have two operands, the first is location for the virtual storage and the second is the location for the backing storage; there are some others, but usually the first two are sufficient for our needs**
- **'BELOW' and 'ANY' are synonyms for '24' and '31' respectively; they are still supported but the numeric values are recommended**

Code Samples

GETMAIN for data items is only required for our reentrant programs. But because we are running AMODE 31 / RMODE 31, we need to use GETMAIN for our DCBs in the non-reentrant programs.

You can ask for storage conditionally, in which case you should check R15 for a return code. But our feeling is for the code on entry, you should have no problem obtaining storage, and if you do, using the R form, you will be abended by the system.

You should make no assumptions about the state of gotten storage. z/OS promises to set it to binary zeros under a few conditions (8K or more requested from a pageable, private subpool; 4K or more requested from a pageable, private subpool and BNDRY=PAGE is also requested [this cannot be specified with the R option])

Freemain

If you get storage, at end of task the system frees it up for you. It is considered better practice to explicitly free it, however. The tricky part is that, in our cases, the gotten storage contains the address of our save area, which contains the address of our caller's save area. So we need to grab the backward chain pointer before we exit. So our code is:

```
lr    1,13
l     0,fsize   <-- only needed for baseless case
l     13,4(13)
freemain r,lv=(0),a=(1)
return (14,12),,rc=0
```

- **If you don't specify lv=(0), code it like getmain (lv=worksize), and then freemain generates a L instruction with a target of *-4; this requires addressability in the CSECT, which we have explicitly omitted in the baseless versions**

Code Samples

WTO - non-reentrant

- **This little service is invaluable for development and debugging in the non-LE world; it has several options, but we have found that the most useable is this format:**

```
WTO    TEXT=(r),ROUTCDE=(11)
```

  - ✗ In this case, ***r*** designates a register, (2)-(12), that contains the address of the message, which must be presented as a halfword-prefixed string; the string is the text to display, the halfword prefix contains the length of the text; maximum length is 126 bytes

  - ✗ The message text is written to the JCL listing (that's the ROUTCDE=(11) effect)

Code Samples

WTO - reentrant

- **If the service is to be requested in a reentrant program, you must provide a list and execute form of the macro; the list form will be copied into your gotten storage, and your execute forms will reference the list format; some examples**

```
❶           mvc   msg2(msg_size),src_msg2
            mvc   wto(wto_size),src_wto
   ...
            la    3,msg1
❷           wto   text=(3),mf=(e,wto)
            la    3,msg2
            wto   text=(3),mf=(e,wto)
   ...
   data     ds    0f
❸  msg1     dc    h'20',cl20'This is message one.'
❹  src_msg2 dc    al2(msg_size)
            dc    c'Today is: '
            dc    cl10' '
            dc    c'.'
❺  msg_size equ   *-src_msg2
❻  src_wto  wto   text=,routcde=(11),mf=l
❼  wto_size equ   *-src_wto
   *
            ltorg
   wareas   dsect
   save     ds    36f'0'
❽  wto      wto   text=,routcde=(11),mf=l
❾  msg2     ds    h
            ds    c'Today is: '
❿  date     ds    cl10' '
            ds    c'.'
   worksize equ   *-save
```

Notes on the next page …

Code Samples

WTO - reentrant, Notes

❶ - the two lines here populate the message we will be building dynamically and the list form of the WTO macro

❷ - the four lines here demonstrate invoking the WTO service twice; note that both invocations reference the same list form of the macro

❸ - this line defines a message that is not modified (thus it is in the constants area)

❹ - this line and the next three define the source for the message we will be building dynamically filling in the blank

❺ - here is where we have the Assembler calculate the size of the the source field

❻ - this is the source for the list form of the WTO macro

❼ - again, get the Assembler to calculate a size

❽ - this line reserves space in the gotten storage area for our list form WTO expansion

❾ - these four lines reserve space for the message text; notice the use of DS but putting values in, for documentation

❿ - this is the field we will be filling in before issuing the message

A return code is passed back in R15: 0 is good, anything else indicates a problem of some degree or other.

This code works the same in both the baseless and non-baseless programs that are coded as reentrant.

Code Samples

CALL

The CALL macro is how z/OS Assembler handles static calls. The macro identifies the program to invoke and passes a list of parameters. The called routine will be bound to the calling routine at program bind time.

Again, there are a few versions of interest to us:

```
        call  pgm,(parm,parm,...)
        call  pgm,(parm,parm,...),vl
        call  pgm,(parm,parm,...),mf=(e,plist)
        call  pgm,(parm,parm,...),vl,mf=(e,plist)
plist   call  ,(0,0,...),mf=l
```

Notes

- **The macro generates a Vcon for the called program, builds a list of addresses of the parameters, sets R1 to point to the list, loads the Vcon into R15, and issues BALR 14,15 to effect the actual transfer to the subroutine**
- **The programmer can load the subroutine address into R15 and issue the call as call (15),(parm,...)**
- **Technically, you can issue a CALL to any point in your program, but that creates unnecessary overhead**
- **Any number of parameters may be passed (including zero)**
- **If the parameter is the contents of a registers, enclose it in parentheses: call pgm,((4),area)**

Code Samples

CALL - Notes, continued

- **The VL option forces on the leftmost bit of the last address in the list of parameters, to signify end of list; optional**
- **Reentrant programs must use the list / execute forms; you can code the list form right in the DSECT for gotten storage, and you do not need to populate it: the execute form just uses the area and builds the parts it needs**
- **You can use one list form (we always name it "plist" for parameter list), for all execute form calls in your program, as long as the list form has at least as many parameter slots as the greatest number of parameters in any execute form**

Examples

From our non-reentrant versions:

```
         lhi   4,3
         call  appadate,((4),date)
```

From our reentrant versions:

```
         lhi   4,3,
         call  appadate,((4)),date),mf=(e,plist)
...
         ltorg
wareas   dsect
save     ds    36f'0'
wto      wto   text=,routcde=(11),mf=l
plist    call  ,(0,0,0,0),mf=l
msg2     ds    h
         ds    c'Today is: '
date     ds    cl10' '
         ds    c'.'
worksize equ   *-save
```

Code Samples

CALL - special features

There are two additional parameters you can code on CALL that are of interest to us:

- **LINKINST=***instruction_name* **- you can request an instruction different from BALR, such as BASR or BASSM to be used to do the actual transfer of control**

- **LINKOP=***val1* **for the operand(s) of the linkage instruction (default is '14,15'); if you have only a single value, it can be specified outside of apostrophes, otherwise you code: LINKOP='***val1***,***val2***'**

  - ✗ The key is, after substitution is the result a valid machine instruction?

  - ✗ Both of these operands may be specified on standard form or the execute form

    - ➢ They need not be specified on the list form, which is just for holding parameter addresses

Code Samples

CALL - difficulty

In the non-reentrant but baseless program (pgm2), we found out that the standard format of CALL generates a BAL around a constant; we had to put in addressability in for a short range:

```
         using  (*,end1),5
         larl   5,*
         call   appadate,((4),date)
end1     ds     0h
```

- **The problem goes away once you get reentrant and use the list and execute forms**

Code Samples

Doing I/O

I/O processing is a large, complex issue. In this paper we provide a brief introduction to the main issues for programs using the four styles we have been discussing. We hope this provides enough insight for you to successfully meet your specific I/O needs.

Keeping in mind we want to run AMODE 31 / RMODE 31, recall that we need to have DCBs reside below the line. The solution is code DCBEs containing 31-bit addresses we need and point to these DCBEs from the DCBs.

We address six macros: DCBE, DCB, OPEN, CLOSE, GET, PUT, in that order.

DCBE

- **A DCBE will be needed for each DCB you expect to OPEN; the contents of a DCBE may be updated, so for reentrant versions of code, you may need to copy your DCBE(s) into gotten storage**

- **Although there are many parameters here, we only care about a few:**

```
dcbe1    dcbe  eodad=eof,rmode31=buff,synad=err1
dcbe2    dcbe  rmode31=buff,synad=err2
```

  - ✗ 'eof' identifies the label in your program where control should pass on end of file (only makes sense for input files, of course)
  - ✗ 'rmode31=buff' says open should get buffers from above the line
  - ✗ synad= identifies the address of an error handling routine (if any) that should get control in the event of an uncorrectable I/O error

Code Samples

The synad routine can be very simple:

```
err1      abend 1
err2      abend 2
```

- **Note that this simple format of ABEND Assembles properly and works fine in reentrant, non-reentrant, baseless, non-baseless environments equally well**

Next, the DCBs; we go with the minimum for one input and one output file:

```
file1  dcb ddname=zinputx,dsorg=ps,macrf=(gm),dcbe=dcbe1
file2  dcb ddname=reprt,dsorg=ps,macrf=(pm),dcbe=dcbe2,  x
           recfm=f,lrecl=60
```

Now, even for our non-reentrant programs, we need to get storage below the line and copy the DCBs there. So, in anticipation of this, we actually set up our DCBs this way:

```
src_files ds 0h
file1  dcb ddname=zinputx,dsorg=ps,macrf=(gm),dcbe=dcbe1
fsize1 equ *-file1
file2  dcb ddname=reprt,dsorg=ps,macrf=(pm),dcbe=dcbe2,  x
           recfm=f,lrecl=60
fsize2 equ *-file2
filesize equ *-src_files
```

Code Samples

Then we need to include a DSECT that covers the area for the files, maybe:

```
          ltorg
fileDsect dsect
files     ds    0h
infile    ds    xl(fsize1)
outfile   ds    xl(fsize2)
```

Of course, for the reentrant programs we simply need to add **infile** and **outfile** as above into the existing DSECT Now we're ready to examine the actual I/O processing macros

For reentrant programs, we have included room for our DCBs in our original GETMAINs. For the non-reentrant programs, before we can do any I/O, we need to get storage below the line and copy the DCBs there:

```
          getmain r,lv=filesize
          lr    3,1
          using fileDsect,3
          mvc   infile(fsize1),file1
          mvc   outfile(fsize2),file2
```

Then, the OPEN:

```
          la    4,fsize1(,3)
          open  ((3),(input),(4),(output)),mode=31
```

Notes

- **The basic form of OPEN requires DCBname for the first operand of each pair of operands (*DCBname*,(*open_option*))**
- **The *DCBname* can't be a name in a DSECT unless you are using list and execute forms of OPEN**
- **For our code here, R3 contains the address of the copy of file1, the LA instruction sets the address of the copy of file2 into R4**

Code Samples

In the case of our reentrant programs, we must use the list and execute forms of OPEN, since the standard form of OPEN builds its parameter list inline. So we need to build a source for the list form, provide for space in our gotten storage DSECT, and copy the source for OPEN (and CLOSE) into the gotten storage. The pieces look like this:

```
         mvc   openwk(size_opn),opens
         .
         .
         .
         open  (infile,(input),outfile,(output)),   x
               mf=(e,openwk),mode=31
```

In our constants / data area:

```
opens    open  (,,,),mf=l,mode=31
size_opn equ   *-opens
closes   close (,,,),mf=l,mode=31
size_clo equ   *-closes
```

And in our DSECT:

```
openwk   open  (,,,),mf=l,mode=31
```

Note that the openwk area can be used for both OPEN and CLOSE, so no need to have separate areas.

You could check R15 after the OPEN: a value of zero means all files were successfully OPENed.

Now for doing the real I/O …

Code Samples

For all four programs, we can retrieve (GET) and write (PUT) records using the same syntax, no special concerns:

```
        get   infile,inrec
        .
        .
        .
        put   outfile,outrec
```

- **Where 'inrec' and 'outrec' are in your data areas or your gotten storage, depending on which style of coding you are using**
- **Note that error checking is done by the access method; if a non-recoverable I/O error is encountered, the access method will branch to your SYNAD routine, if you have one**

Finally, at end of file, you will be branched to your EODAD routine (**shutdown** in all our samples) where you should close your files:

```
        close ((3),,(4)),mode=31
```

for the non-reentrant programs; or

```
        close (infile,,outfile),mf=(e,openwk),mode=31
```

for the reentrant programs.

Again, you might check R15 after CLOSE for errors. In our samples, we checked R15 after both OPEN and CLOSE, and for a non-zero value jumped to an ABEND 0 macro. (Kinda' crazy have a User abend code of zero, but it works and keeps the style consistent.)

Some Observations on DCBEs

The Data Set Macros manual has inconsistent, sometimes misleading, comments about DCBEs. You may need to test each case where you use them. We believe this much is true:

* If you want your program to run AMODE 31, your DCBs must be below the line; but routines such as EODAD and SYNAD must have 31 bit addresses, regardless of where the routines are actually located

  In this case, you need a DCBE because the DCB itself only supports 24-bit addresses. When you specify DCBE=dcbename in your DCB, the macro expands the DCB so that the first word contains the 31-bit address of your DCBE (and two bits are also turned on indicating there is a DCBE)

  The question arises if the DCBE itself is ever modified (see page 38). The manual seems to indicate the DCBE is modified in certain cases, including when your are using "31-bit SAM" support, which would seem to include our small sample programs. But for our two reentrant programs (PGM3 and PGM4) we did nothing to copy the DCBEs into gotten storage, yet we Assembled and bound with the RENT option and the program ran with no errors - until I ran them under the OMVS shell; then they abended. Ah hah!

Some Observations on DCBEs

Suppose you encounter a situation where you abend because OPEN tries to modify a DCBE. In that case, you will need to include DSECT entries for the DCBE(s) affected, and copy the DCBE(s) from your constants section into those DSECT areas. AND (here's the secret) you need to put the address of the DCBEs in gotten storage into the the first words of the corresponding DCBs in gotten storage.

The same code works for both PGM3 and PGM4 type coding:

* After you move the DCBs to gotten storage, move the DCBEs, and set the DCBs in gotten storage to point to the DCBEs in gotten storage:

```
...
mvc     outfile(fsize2),file2
mvc     gdcbe1(szdcbes),dcbe1
la      3,gdcbe1
st      3,infile
la      3,gdcbe2
st      3,outfile
```

* In your constants area:

```
dcbe1     dcbe   eodad=shutdown,rmode31=buff,synad=err1
dcbe2     dcbe   rmode31=buff,synad=err2
szdcbes   equ    *-dcbe1
```

* In your DSECT area:

```
openwk    open   (,,,),mf=l,mode=31
gdcbe1    dcbe   eodad=shutdown,rmode31=buff,synad=err1
gdcbe2    dcbe   rmode31=buff,synad=err2
```

... and that's it. So we modified PGM3 and PGM4 to take this approach.

We hope you found this little paper helpful. We plan to write some further papers on related topics; tentatvely: "64 Bit Assembler Coding" and "Coding Assembler for LE" If you are interested in these, drop me a line or give me a call.

If you have suggestions for improvements, or you spot an error, or you have a question, feel free to contact me (see the front cover for contact information).