Using z/OS Macros in Metal C

Session 8739

Peter Relson
IBM Poughkeepsie
relson@us.ibm.com
2 March 2011

Trademarks

Abstract

A detailed exploration of using z/OS assembler macros in Metal C, focusing on symbolic substitution. The talk will concentrate on the IBM Health Checker for z/OS macros (to help you understand a key part of writing checks in Metal C). It will help you to figure out how to identify to Metal C the inputs and outputs so that you get the result you want.

Introduction

- History: a C/C++ program wanting to invoke assembler statements for which there was no high level language analog or service, let alone an assembler macro, had to call a separate routine, passing parameter data, and that separate routine, in assembler, would create the macro invocation
- Metal C lets you embed the assembler statements.
- But how do you tell Metal C how the assembler statements (including macros) are to access your C variables?
- We will concentrate on invoking macros, as most things that can be accomplished through simple assembler instructions can be accomplished within the high level language.

Agenda

- __ASM statement as it applies to macros, with recommendations
- Use of __ASM with the HZSxxxxx macros, showing reentrant forms throughout, identifying useful techniques
- Use of __ASM with simple instructions
- Use of our newly-learned techniques on a non-HZSxxxxx macro

Metal C Support: __asm

__asm statement when GENASM compiler option is in effect

The syntax of __asm (loosely) is

```
__asm ("assembler statement"
       : /* output definitions */
       : /* input definitions */
       : /* clobber list */
       );
```

```
Note: __ASM does not work, it must be __asm
```

Metal C Support: __asm example

```
__asm(" HZSFMSG REQUEST=CHECKMSG,"
"MGB=%3,"
"RETCODE=%0,"
"RSNCODE=%1,"
"MF=(E,%2,COMPLETE)"
: "=m"(theRetcode), "=m"(rsn), "=m"(PListFMsg)
: "m"(MGB)
: "r0","r1","r14","r15");
```

We will use this example in exploring the syntax. As we will see, this example uses symbols and identifies their input / output nature and what registers the compiler is to assume are clobbered.

Metal C Support (cont)

- “%n” refers to the nth (0-origin) output or input symbol definition
- The definitions identify the storage forms -- memory ("m") or register ("r") -- and also differentiate output from input+output. An input and/or output definition is of the form

  “ps”(variable) -- could be “ps”(expression)

  - “p” -- the “prefix” -- null, or “=” for “output” or “+” for “input and output”
  - “s” -- the storage classification -- “m” for memory or “r” for register
  - “variable” -- the Metal C variable to be used
- You can use \n to identify new line (i.e., in this case to delimit statements when you have more than one)

Metal C Support (cont)

Recommendations

- Always tell the truth to the compiler.
  - If the __asm statement(s) can result in updating your variable, then make sure that this item is identified as an "output". This includes return code and reason code.
  - If the __asm statement(s) require your variable as input and also update it, then make sure that this item is identified as an "output" and an "input".

The input / output nature of a macro parameter should be identified in the product documentation for that parameter.

Metal C Support: Output

Between the first and second colons are “output” things

```
__asm(" HZSFMSG REQUEST=CHECKMSG,"
"MGB=%3,"
"RETCODE=%0,"
"RSNCODE=%1,"
"MF=(E,%2,COMPLETE)"
: "=m"(theRetcode), "=m"(rsn), "=m"(PListFMsg)
: "m"(MGB)
: "r0","r1","r14","r15");
```

- Output things have a prefix of “=” or “+” (“=” means output only, “+” means output and input)
- “m” means to select register n, substitute with “0(n)” and set the variable's address into register n “before” doing the assembler statement(s)
- %0 is theRetcode, %1 is rsn, %2 is PlistFMsg, %3 is MGB

Metal C Support: Output

Example:

```
__asm(“ HZSFMSG ...RETCODE=%0”
        : “=m”(theRetcode)
        : /* … */
        : /* … */);
```

- output specification: "=m"(theRetcode)
- within the __asm macro invocation: RETCODE=%0
- Compiler picks a register (for example, 2) so the generated assembler has RETCODE=0(2)
- The compiler puts the address of theRetcode into register 2 "before"
- The macro expansion generates ST 15,retcode_operand -- ST 15,0(2) -- after calling the service

Metal C Support: Output

“r” means to select register n, substitute with “n” and store from register n into the variable “after”

Example:

```
__asm(“ HZSFMSG ...RETCODE=%0”
         : “=r”(theRetcode)
         : /* … */   : /* … */);
```

- output specification: "=r"(theRetcode)
- within the __asm macro invocation: RETCODE=(%0)
- Compiler picks a register (for example, 2) so the generated assembler has RETCODE=(2)
- The macro expansion generates
  LR   retcode_reg,15 (e.g., LR 2,15) after calling the service
- The compiler stores from register 2 into theRetcode

Metal C Support: Output (cont)

- Which is better? It depends on the complexity of the compiler's finding where to store. Probably the “=m” form is better in more cases. But the difference is miniscule.
- As we'll see, if we had used "m" in the "input" section instead of in the output section, we would have gotten the identical behavior. But the output form tells the truth.

Metal C Support: __asm example (same as before, for reference)

```
__asm(" HZSFMSG REQUEST=CHECKMSG,"
"MGB=%3,"
"RETCODE=%0,"
"RSNCODE=%1,"
"MF=(E,%2,COMPLETE)"
: "=m"(theRetcode), "=m"(rsn), "=m"(PListFMsg)
: "m"(MGB)
: "r0","r1","r14","r15");
```

Metal C Support: Input

After the second colon are “input” things

“m” means to select register n, substitute with “0(n)” and to set the variable's address into register n “before”

Example 1 (character variable):

```
__asm(“ HZSCHECK ...CHECKNAME=%0”
        : /* ... */
        : “m”(theCheckName)
        : /* … */);
```

- HZSCHECK has a CHAR 32 CHECKNAME parameter
- input specification: "m"(theCheckName)
- within the __asm macro invocation: CHECKNAME=%0
- Compiler picks a register (for example, 2) so the generated assembler has CHECKNAME=0(2)
- The compiler puts the address of CheckName into register 2

Metal C Support: Input

- The macro expansion knows that a specification of 0(2) is an RS-type expression locating the variable.
- In this case, the contents of theCheckname are moved into the parameter list (but it would have worked fine if this were a different macro that put the address of theCheckname into the parameter list).

Metal C Support: Input

Example 2 (integer variable):

```
__asm(“ DSPSERV ...BLOCKS=%0”
        : /* ... */
        : “m”(numBlocks)
        : /* … */);
```

- DSPSERV has a BLOCKS parameter
- input specification: "m"(numBlocks)
- within the __asm macro invocation: BLOCKS=%0
- Compiler picks a register (for example, 3) so the generated assembler has BLOCKS=0(3)
- The compiler puts the address of NumBlocks into register 3
- The macro expansion expects that the specification of 0(3) will locate the variable; it does
- This works

Metal C Support: Input

“r” means to select register n, substitute with “n” and to load into register n from the variable “before” (integer or pointer) or set the variable's address into register n “before” (character)

Example 1 (character variable):

```
__asm(“ HZSCHECK ...CHECKNAME=0(%0)”
        : /* ... */
        : “r”(theCheckName)
        : /* … */);
```

- HZSCHECK has a CHAR 32 CHECKNAME parameter
- input specification: "r"(theCheckName)
- within the __asm macro invocation: CHECKNAME=0(%0)
- Compiler picks a register (for example, 4) so the generated assembler has CHECKNAME=0(4)
- The compiler puts the address of CheckName into register 4

Metal C Support: Input

- The macro expansion knows that a specification of 0(4) is an RS-type expression locating the variable.
- In this case, the contents of theCheckname are moved into the parameter list (but it would have worked fine if this were a different macro that put the address of theCheckname into the parameter list).
- This “r” form has identical behavior to the “m” form shown earlier

Metal C Support: Input

Example 2 (character variable)

```
__asm(“ HZSCHECK ...CHECKNAME=(%0)”
        : /* ... */
        : “m”(theCheckName)
        : /* … */);
```

- within the __asm macro invocation: CHECKNAME=(%0)
- The macro expansion knows that a specification of (4) is a register expression locating the variable
- Same results as Example 1

Which is better? If the macro is going to do the "move", these are identical. If the macro is going to pass the address, the second form saves a "LA" instruction. Consider looking at the assembler macro expansion if you want to manage down to this level of detail.

Metal C Support: Input

Example 3 (integer variable):

```
__asm(“ DSPSERV ...BLOCKS=0(%0)”
        : /* ... */
        : “r”(numBlocks)
        : /* … */);
```

- DSPSERV has a BLOCKS parameter
- input specification: "r"(numBlocks)
- within the __asm macro invocation: BLOCKS=0(%0)
- Compiler picks a register (for example, 3) so the generated assembler has BLOCKS=0(3)
- The compiler puts the contents of NumBlocks into register 3
- The macro expansion expects that the specification of 0(3) will locate the variable, not be contents of the variable. WRONG!
- but "r"(&numBlocks) works

Metal C Support: Input

Example 4 (integer variable):

```
__asm(“ DSPSERV ...BLOCKS=(%0)”
        : /* ... */
        : “r(numBlocks)”
        : /* … */);
```

- input specification: "r"(numBlocks)
- within the __asm macro invocation: BLOCKS=(%0)
- The macro expansion expects that the register specification of (3) will locate the variable, not be the (contents of the) variable.
- Same as before: this does not work

Metal C Support: Input “m” or “r”?

The "m" form is more consistent -- it does not change behavior based on data type. And even if sometimes a macro requires that when you use the register form the "value" be in there, and sometimes the "address of the value" be in there, by using the "m" form we do not have to care (as with the “m” form we never end up using the register form). I strongly recommend the “m” form.

Metal C Support: Clobber List

After the (optional) third colon is the Clobber list

```
__asm(" HZSFMSG REQUEST=CHECKMSG,"
"MGB=%3,"
"RETCODE=%0,"
"RSNCODE=%1,"
"MF=(E,%2,COMPLETE)"
: "=m"(theRetcode), "=m"(rsn), "=m"(PListFMsg)
: "m"(MGB)
: "r0","r1","r14","r15");
```

- A comma-separated list from among "r0", "r1", ..., "r15" identifying the registers known not to be preserved within the __asm block. Among other things, the Clobber list tells the compiler regs it cannot use.
- Example: : “r0”, “r1”, “r14”, “r15”

Metal C Support: Recommendations

Recommendations

- Always use "m" form for macro for outputs and individual inputs
- Identify any registers that are clobbered using the "clobber list". For any z/OS service, unless explicitly documented otherwise, identify regs 0,1,14,15 as clobbered (and any others that are identified in the service's documentation). This not only makes sure the compiler knows that those registers will no longer have their previous values, but also keeps the compiler from assigning to those registers.

Metal C Support: Things to Notice

- Symbols are numbered in order of specification, the first being symbol 0.
- For multiple statements, use "\n" at the end of each.
- For "continuation" lines: the compiler will take care of this. Be sure, as with an assembler macro, to have the comma delimiters between keywords. For continuation, do not start with a blank – the compiler concatenates the individual lines together, so an extra blank would cause problems.
- For each statement / new line, start with a blank if you have no label.

Metal C Support: Things to Notice (cont)

- What if you know that it's an output, but you have no idea if the macro will
  - pass its address to the service (and in the service the output is set)
  - set a field in the parameter list, after the service the macro moves from the parameter list to your variable?
- The good news: you don't need to care. In both cases, it is fine to tell the macro a register specification such as thekey=(somereg) or an RS-type specification such as thekey=0(somereg)
- In many cases "m" form is just like "r" form, you just specify the "m" form without parens, and the "r" form with parens, and you don't specify the "&" on the “m” form.

Metal C Support: Assembler Mapping Macros

To insert non-executable assembler:

#pragma insert_asm(“[label] statement”)

For example,

#pragma insert_asm(" CVT   DSECT=YES,LIST=NO")

- The compiler places the inserted statement at an appropriate place (this turns out to be at the end of the module, where a DSECT will not cause difficulties).

Metal C Support: List and Execute forms

- List form defaults to 256 bytes, but you can use DS:nnn (e.g., DS:100) to define a length other than 256 (specified in decimal). This alternate form is necessary if the length is > 256. For cases where the length <= 256, this can save space (to the extent that you care).
- Allocate space for the list form in dynamic storage.

*

Metal C Support: List and Execute (cont)

- (For some macros) allocate space for the initialized list form in static storage, and copy from the static copy to the dynamic copy before using the execute form. The static list form should be in global scope so that it is part of the compiler-produced static data for the module.
- (For other macros) just use the execute form. Many macros that have no parameters specifiable on the list form support a "COMPLETE" option which indicates to specify everything on the execute form. When using COMPLETE, you do not need an initialized list form (copied from a static form) before using the execute form. IARV64 is one such macro. COMPLETE is the default for this and other such macros.

Assembler Macro Examples Intro

Assembler macros typically let you code

- RS form: KEYWORD=rs_type_expression such as KEYWORD=0(reg)
  The name comes from RS-type (base+displacement) instruction format
- KEYWORD=some_variable is also considered RS-form
- Some macros let you code RX form disp(index,base)
- register form: KEYWORD=(reg)
- (For positional parameters, the "KEYWORD=" would not be part of the specification)

Assembler Macro Examples Intro

- Because of the way that the compiler creates "internal" names based on the variable names you coded with, we will not use the KEYWORD=some_variable form as typically the assembler will not know of your metal C variable by its metal C name.
- We will mostly use the RS form.
- Notation to be used in the slides: register nX is the general register assigned by the compiler, associated with substitution specification X.

Assembler Macro Examples Intro

For a given macro parameter, you need to know the requirements of the parameter (its data type, its size) and what you specify.

- For example, consider this from macro HZSADDCK

```
,CHECKNAME=checkname
...
To code: Specify the RS-type address, or address in register
(2)-(12), of a 32-character field.
```

- Where an RS-type address is "base(displacement)" such as "0(4)" indicating a displacement of 0 bytes from the base in register 4

Assembler Macro Examples Intro

- As long as we can get the assembler specification to be "right", we will be in good shape.
- Suppose we have
  char thename[32];
  in our program.
- We cannot simply code our __ASM to have CHECKNAME=thename because the compiler may not surface "thename" to the assembler.
- But if we can get the address of "thename" into a register R and get CHECKNAME=0(R) within the __ASM, we will be all set.
- This is what C's symbol substitution lets you do -- the compiler gets to pick which register, and you get to refer to that register symbolically.

HZSxxxxx macros

The IBM Health Checker for z/OS provided mappings for its control structures and for its constants in z/OS 1.12 to use within metal C. No changes are made to the executable assembler macros, due to the presence of the metal C symbolic substitution support.

HZSFMSG

```
__asm(" HZSFMSG MF=(L,PListFMsg)": "DS"(PListFMsg));
__asm(" HZSFMSG REQUEST=CHECKMSG,"
"MGB=%3,"
"RETCODE=%0,"
"RSNCODE=%1,"
"MF=(E,%2,COMPLETE)"
: "=m"(rc), "=m"(rsn), "=m"(PListFMsg)
: "m"(MGB)
: "r0","r1","r14","r15");
```

HZSFMSG symbolic parameters

- parameter 0 -- output "=m"(rc). Put the address of rc into register n0 before the expansion. The macro sees RETCODE=0(n0)
- parameter 1 -- output "=m"(rsn). Put the address of rsn into register n1 before the expansion. The macro sees RSNCODE=0(n1)
- parameter 2 -- output "=m"(PListFMsg). Put the address of PListFMsg into register n2 before the expansion. The macro sees MF=(E,0(n2),COMPLETE)
- parameter 3 -- input "m"(MGB). Put the address of MGB into register n3 before the expansion. The macro sees MGB=0(n3)

HZSADDCK

- inputs: checkOwner, checkName, date, reasonlen, reason, parmslen, parms, entry code, exitrtn, checkrtn, msgtbl
- outputs: Handle
- Has so many parameters that the compiler cannot choose a unique register for each. We will explore an approach that does not require a unique register for each
  - Define a struct to hold the data so that we can access all the struct fields using one register (which will contain the address of the struct), There could be extra data movement. (You might have information in "one place" and then move it to the struct in order to use it. That is a limitation of the metal C support.)

HZSADDCK

```
typedef struct checkInfo_s {
int entry;             /* unique value for a check  */
char owner[16];        /* check owner               */
char name[32];         /* check name                */
char date[8];          /* YYYYMMDD                  */
char exitRtn[8];       /* exit routine name         */
char checkRtn[8];      /* check routine name        */
char *msgtbl;          /* the message table         */
int reasonLen;         /* 1 - 126                   */
char *reason;          /* Up to 126 char string     */
int parmsLen;          /* 1 - 256                   */
char *parms;           /* Up to 256 char string     */
} tCheckInfo;
tCheckInfo checkInfo;
```

HZSADDCK

- Note that in this example, the struct contains the address of the reason and the address of the parms, not the reason and parms themselves.
- To notice: for "continuation" lines: the compiler will take care of this. Be sure, as with an assembler macro, to have the comma delimiters between keywords. For continuation, do not start with a blank.

HZSADDCK

```
__asm(" HZSADDCK MF=(L,PListAddck)": "DS"(PListAddck));
__asm(" HZSADDCK ACTIVE,"
      "SEVERITY=LOW,"
      "INTERVAL=ONETIME,"
      "USS=NO,"
      "ENTRYCODE=0(%3)," /* Based on checkInfo struct */
      "CHECKOWNER=4(%3),"
      "CHECKNAME=20(%3),"
      "DATE=52(%3),"
      "REASONLEN=80(%3),"
      "REASON=(%4),"
      "PARMSLEN=88(%3),"
      "PARMS=(%5),"
      "EXITRTN=60(%3),"
      "CHECKROUTINE=68(%3),"
```

HZSADDCK

```
           "MSGTBL=76(%3),"
            "RETCODE=(%0),"
            "RSNCODE=(%1),"
            "MF=(E,(%2),COMPLETE)"
    : "=r"(rc), "=r"(rsn)
    : "r"(&PListAddck), "r"(&checkInfo), "r"(checkInfo.reason),
      "r"(checkInfo.parms)
    : "r0","r1","r14","r15");
```

HZSADDCK: symbolic parameters

- parameter 0 -- output "=r"(rc): Put the address of rc into register n0 before the expansion. The macro sees RETCODE=0(n0)
- parameter 1 -- output "=r"(rsn): Put the address of rc into register n1 before the expansion. The macro sees RSNCODE=0(n1)
- parameter 2 -- input "r"(&PListAddck)
  - put the address of PListAddck into register n2 before the expansion. The macro sees MF=(E,(n2),COMPLETE) so uses that register for the address of the parameter list. Note that even though PListAddck is written into, it is identified as an input. This is a drawback with use of the "r" form for parameter lists.
- parameter 3 -- input "r"(&checkInfo)
  - put the address of checkInfo into register n3 before the expansion. The macro sees ENTRYCODE=0(n3) and CHECKOWNER=4(n3), etc. These provide the RS-type specifications for various parameters which end up being moved into the parameter list.

HZSADDCK: symbolic parameters

- parameter 4 -- input "r"(checkInfo.reason)
  - since this is a pointer, put the value of checkInfo.reason into register n4 (hence the address of the reason) before the expansion. The macro sees REASON=(n4) which indicates that the address of the reason is in n4
- parameter 5 -- input "r"(checkInfo.parms)
  - since this is a pointer, put the value of checkInfo.parms into register n5 (hence the address of the parms) before the expansion. The macro sees PARMS=(n5) which indicates that the address of the parms is in n5

HZSADDCK

What don't we like about this?

- That you need different techniques for the character items than for the pointer items.
- That the parameter list must be identified as an input.
- That you must hardcode the offsets. This is not friendly but there is no linguistic alternative provided.

HZSADDCK alternatives

Are there other forms we could have used?

- Yes for the parameter list: "=m"(PListAddck) with MF=(E,%2,COMPLETE)
- Yes for reason: "m"(checkInfo.reason) with REASON=%4
- Yes for parms: "m"(checkInfo.parms) with PARMS=%5
- Yes for checkinfo (with a little trick): The "m" form always substitutes 0(n) and we need (for example) 4(n3). But we can accomplish this by coding 4+%3 which will result in 4+0(n3) which is equivalent to 4(n3).

HZSADDCK alternative

```
    __asm(" HZSADDCK ACTIVE,"
          "SEVERITY=LOW,"
          "INTERVAL=ONETIME,"
          "USS=NO,"
          "ENTRYCODE=0+%3," /* Based on checkInfo struct */
          "CHECKOWNER=4+%3,"
          "CHECKNAME=20+%3,"
          "DATE=52+%3,"
          "REASONLEN=80+%3,"
          "REASON=%4,"
          "PARMSLEN=88+%3,"
          "PARMS=%5,"
          "EXITRTN=60+%3,"
          "CHECKROUTINE=68+%3,"
```

HZSADDCK alternative

```
          "MSGTBL=76+%3,"
          "RETCODE=%0,"
          "RSNCODE=%1,"
          "MF=(E,%2,COMPLETE)"
    : "=m"(rc), "=m"(rsn), "=m"(PListAddck)
    : "m"(checkInfo), "m"(checkInfo.reason),
      "m"(checkInfo.parms)
    : "r0","r1","r14","r15");
```

Note the use of “offset+%3”

HZSADDCK (cont)

- If the full reason and/or the full parms string were placed within the structure, rather than the pointer to them, the technique used with parameter 3 could have been used for reason and parms.

HZSADDCK (cont)

```
typedef struct checkInfo_s2 {
int entry;          /* unique val for a check */
char owner[16];     /* check owner            */
char name[32];      /* check name             */
char date[8];       /* YYYYMMDD               */
char exitRtn[8];    /* exit routine name      */
char checkRtn[8]; /* check routine name     */
char *msgtbl;       /* the address of the
                       Message table          */
int reasonLen;      /* 1 - 126                */
char reason[126]; /* Up to 126 char string  */
char padding[2];    /* Round to word boundary */
int parmsLen;       /* 1 - 256                */
char parms[256];    /* Up to 256 char string  */
} tCheckInfo2;
tCheckInfo2 checkInfo2;
```

HZSADDCK

```
__asm(" HZSADDCK ACTIVE,"
      "SEVERITY=LOW,"
      "INTERVAL=ONETIME,"
      "USS=NO,"
      "ENTRYCODE=0+%3," /* Based on checkInfo struct */
      "CHECKOWNER=4+%3,"
      "CHECKNAME=20+%3,"
      "DATE=52+%3,"
      "REASONLEN=80+%3,"
      "REASON=84+%3,"
      "PARMSLEN=212+%3,"
      "PARMS=216+%3,"
      "EXITRTN=60+%3,"
      "CHECKROUTINE=68+%3,"
```

HZSADDCK

```
          "MSGTBL=76+%3,"
          "RETCODE=%0,"
          "RSNCODE=%1,"
          "MF=(E,%2,COMPLETE)"
: "=m"(rc), "=m"(rsn), "=m"(PListAddck)
: "m"(checkInfo2)
: "r0","r1","r14","r15");
```

What's different? Some of the offsets, and the fact that reason and parms now use register n3

HZSADDCK (use literals)

```
__asm(" HZSADDCK "
      "CHECKOWNER==CL16'IBMSAMPLE',"
      "CHECKNAME==CL32'HZS_SAMPLE_REMOTE_MC_HZSCPARS',"
      "ACTIVE,"
      "SEVERITY=LOW,"
      "REMOTE=YES,"
      "USS=NO,"
      "HANDLE=%4,"
      "PETOKEN=%3,"
      "INTERVAL=ONETIME,"
      "VERBOSE=NO,"
      "DATE==CL8'20090212',"
      "REASONLEN==A(41),"
  "REASON==CL41'Sample Metal C health check with HZSCPARS',"
```

HZSADDCK (use literals, cont)

```
"PARMSLEN==A(38),"
"PARMS==CL38'PARM1(1,999),PARM2(100),PARM3(CHOICE1)',"
"RETCODE=%0,"
"RSNCODE=%1,"
"MF=(E,%2,COMPLETE)"
: "=m"(rc), "=m"(rsn), "=m"(PListAddck)
: "m"(PEToken), "m"(checkHandle)
: "r0","r1","r14","r15");
```

- Metal C sets up addressability to the literal area using GR 3.

HZSADDCK: Recommendations

Recommendations

- Use "m" form for an input structure that you will use for several data items (with the "offset+" notation); use "r" form only if you strongly prefer using the "offset(" notation)

HZSCHECK

```
__asm(" HZSCHECK MF=(L,PListCheck)": "DS"(PListCheck));
__asm(" HZSCHECK REQUEST=RUN,"
"CHECKOWNER=%3,"
"CHECKNAME=%4,"
"RETCODE=%0,"
"RSNCODE=%1,"
"MF=(E,%2,COMPLETE)"
: "=m"(rc), "=m"(rsn), "=m"(PListCheck)
: "m"(ckOwner)
, "m"(ckName)
: "r0","r1","r14","r15");
```

*

HZSCHECK symbolic parameters

- parameter 0 -- output "=m"(rc). Put the address of rc into register n0 before the expansion. The macro sees RETCODE=0(n0)
- parameter 1 -- output "=m"(rsn). Put the address of rsn into register n1 before the expansion. The macro sees RSNCODE=0(n1)
- parameter 2 -- output "=m"(PListCheck). Put the address of PListCheck into register n2 before the expansion. The macro sees MF=(E,0(n2),COMPLETE)
- parameter 3 -- input "m"(ckOwner). Put the address of ckOwner into register n3 before the expansion. The macro sees CHECKOWNER=0(n3)
- parameter 4 -- input "m"(ckName). Put the address of ckName into register n4 before the expansion. The macro sees CHECKNAME=0(n4)

HZSPREAD

```
__asm(" HZSPREAD MF=(L,PListPRead)": "DS"(PListPRead));
__asm(" HZSPREAD CHECKOWNER=%5,"
"CHECKNAME=%6,"
"IPL=PRIOR,"
"INSTANCE=MOSTRECENT,"
"STARTBYTE=0,"
"BUFFER=%0,"
"DATALEN=%7,"
"BYTESAVAIL=%1,"
```

HZSPREAD (cont)

```
"RETCODE=%2,"
"RSNCODE=%3,"
"MF=(E,%4)"
: "=m"(PReadBuffer)
, "=m"(PReadBytes)
, "=m"(rc)
, "=m"(rsn)
, "=m"(PListPRead)
: "m"(ckOwner)
, "m"(ckName)
, "m"(persistentDataBytes)
: "r0","r1","r14","r15");
```

HZSPREAD symbolic parameters

- parameter 0 -- output "=m"(PReadBuffer). Put the address of PReadBuffer into register n0 before the expansion. The macro sees BUFFER=0(n0)
- parameter 1 -- output "=m"(PReadBytes). Put the address of PReadBytes into register n1 before the expansion. The macro sees BYTESAVAIL=0(n1)
- parameter 2 -- output "=m"(rc). Put the address of rc into register n2 before the expansion. The macro sees RETCODE=0(n2)
- parameter 3 -- output "=m"(rsn). Put the address of rsn into register n3 before the expansion. The macro sees RSNCODE=0(n3)
- parameter 4 -- output "=m"(PListPRead). Put the address of PListPRead into register n4 before the expansion. The macro sees MF=(E,0(n4))

HZSPREAD symbolic parameters (cont)

- parameter 5 -- input "m"(ckOwner). Put the address of ckOwner into register n5 before the expansion. The macro sees CKOWNER=0(n5)
- parameter 6 -- input "m"(ckName). Put the address of ckName into register n6 before the expansion. The macro sees CKNAME=0(n6)
- parameter 7 -- input "m"(persistentDataBytes). Put the address of persistentDataBytes into register n7 before the expansion. The macro sees DATALEN=0(n7)

HZSPWRIT

```
__asm(" HZSPWRIT MF=(L,PListPWrit)": "DS"(PListPWrit));
__asm(" HZSPWRIT BUFFER=%3,"
"DATALEN=%4,"
"RETCODE=%0,"
"RSNCODE=%1,"
"MF=(E,%2)"
: "=m"(rc)
, "=m"(rsn)
, "=m"(PListPWrit)
: "m"(PWritBuffer)
, "m"(persistentDataBytes)
: "r0","r1","r14","r15");
```

HZSPWRIT symbolic parameters

- parameter 0 -- output "=m"(rc). Put the address of rc into register n0 before the expansion. The macro sees RETCODE=0(n0)
- parameter 1 -- output "=m"(rsn). Put the address of rsn into register n1 before the expansion. The macro sees RSNCODE=0(n1)
- parameter 2 -- output "=m"(PListPWrit). Put the address of PListPWrit into register n2 before the expansion. The macro sees MF=(E,0(n2))
- parameter 3 -- input "m"(PWritBuffer). Put the address of PWritBuffer into register n3 before the expansion. The macro sees BUFFER=0(n3)
- parameter 4 -- input "m"(persistentDataBytes). Put the address of persistentDataBytes into register n4 before the expansion. The macro sees DATALEN=0(n4)

STORAGE macro Gotcha

Gotcha: Not all macros treat "(reg)" as containing the address of the variable (i.e., equivalent to "0(reg)"). Some treat "(reg)" as containing the value and "0(reg)" as locating the value (with the expansion doing the load). It may be worth your while trying the assembler expansion to see what it does for the 2 forms. And it is always worth trying to get the information from the doc (which is, unfortunately, often weak in this area).

STORAGE macro (cont)

Suppose we want to do a STORAGE RELEASE. We know that we need an area address, a length, and a subpool, all of them being inputs. We can safely use the "m" form for all three.

```
__asm("STORAGE RELEASE,ADDR=%0,LENGTH=%1,SP=%2"
:  /* no outputs */
: m(theAddr), m(theLength), m(theSubpool)
: "r0","r1","r14","r15");
```

STORAGE macro (cont)

The compiler will select 3 regs, put the address of theAddr in one, the address of theLength in another, the address of theSubpool in the 3rd. For example, producing

```
STORAGE RELEASE,ADDR=0(n0),LENGTH=0(n1),SP=0(n2)
```

- This will work. **Well, no it won't.** Because the STORAGE macro was poorly created and misleads. The "ADDR" keyword does not want an address, it wants an "area".
- Our previous attempt would attempt to "free the pointer" rather than "free the area".

STORAGE macro (cont)

If we have addressability to "theArea" and we also have the address of that area within a variable "theAddr", we have choices:

- We can specify theArea instead of theAddr and use the preceding form
- We can use theAddr and specify the "r" form.with register notation "r"(theAddr)
  - ➢the compiler will place the contents of theAddr into register n0
  - ➢...ADDR=(%0) resulting in ...ADDR=(n0)
  - ➢the macro knows that what is in register n0 is the address of the area
  - ➢or ... ADDR=0(%0) resulting in ...ADDR=0(n0)
  - ➢the macro knows that 0 past register n0 is the area

MVCDK: operands in architected registers

```
__asm(" L    0,%1\n"
      " L    1,%3\n"
      " MVCDK %0,%2"
: "=m"(target)
: "m"(lenMinusOne)
, "m"(source)
, "m"(destkey)
: "r0", "r1");
```

MVCDK (alternative)

```
__asm(" LR    0,%1\n"
      " LR    1,%3\n"
      " MVCDK %0,%2"
: "=m"(target)
: "r"(lenMinusOne)
, "m"(source)
, "r"(destkey)
: "r0","r1");
```

MVCDK (cont)

Note: If target is the simplest definition, something like

char target[n];

then you need to specify target[0] or *target (asterisk-target) instead of target in the output specification because the output is treated like an assignment (needing an "lvalue") which an array cannot be. Identifying target as in input would satisfy the semantic requirement but would fail to inform the compiler that target is changing.

To notice:

For multiple statements, use "\n" at the end of each

ALR (operands in registers)

Here's an example where the data needs to be placed into registers by the compiler

```
__asm( " ALR   %0,%1" :  "+r"(x) : "r"(y) );
```

Note the use of "+" for the first symbolic parameter x, since the target operand of Add-Logical-Register first must be read (input), then operated upon, then written/stored (output)

MVC instruction

Consider trying to move from a source to a target (ignoring that you can do this without resorting to assembler):

```
__asm(" MVC  0(16,%0),%1"
: "=r"(target)
: "m"(source)
: "r0" /* we do not clobber r0, but we prevent
          The compiler from choosing r0 since it
          would not work with r0 */
);
```

which might expand to

```
MVC   0(16,n0),0(n1).
```

This, unfortunately, does not work. An “r” form output does not set the register to the address of the variable before the assembler

*

MVC (cont)

This is a case where you must lie to the compiler in order to get things to work (!!).

```
__asm(" MVC  0(16,%0),%1"
: /* no outputs */
: "r"(target)   /* target is indicated as "input" */
, "m"(source)
: "r0"
);
```

Access Register mode

- It appears that the compiler isn't going to help much
- It lets you define a "far pointer" (this works well)
- But when you use the "m" form (or any other form it appears), only a GR is set, not the associated AR.
- Thus it is "up to you" to set the AR too.

Access Register mode MVC

Consider our MVC example with an AR-qualified source to an AR-qualified target. We can easily get the compiler to place the address of the target and source into registers but we need to go further to get the corresponding ALETs into those same registers.

Suppose that you had extracted the ALET for each of the target and the source, perhaps using the extractor

```
    unsigned int __get_far_ALET(void * __far p);
```

Access Register mode MVC

```
__asm(" SAR  %0,%1\n"
      " SAR  %2,%3\n"
      " MVC  0(16,%0),0(%2)"
: /* no outputs */
: "r"(target)
, "r"(targetALET)
, "r"(source)
, "r"(sourceALET)
: "r0"
);
```

AR Mode MVC (cont)

If you had used the "m" form instead of the "r" form for targetALET and sourceALET, you could use

`LAM    x,x,%n` substituted to `LAM    x,x,0(nY)`

instead of

`SAR    x,%n` substituted to `SAR    x,nY`

Requirements

- If your program is running in AR Address Space Control (ASC) mode and invoking an assembler macro, be sure that SYSSTATE ASCENV=AR
  is in effect (which can be done via __asm).
- If your program is running in AMODE 64 and invoking an assembler macro, be sure that
  SYSSTATE AMODE64=YES
  is in effect (which can be done via __asm).

ATTACH

What might an assembler use of ATTACH macro be?

```
          MVC   AttachDyn,AttachStatic
          ATTACH EPLOC=theEP,DCB=theDCB,SF=(E,AttachDyn)
          ST    R1,theTCB
          ST    R15,theRC
...
AttachDyn ATTACH SF=L
...
AttachStatic ATTACH SF=S,JSTCB=NO,SM=PROB,SVAREA=YES
```

Note that ATTACH has 2 parameter lists, and the one for the service is identified by SF, not MF. Before using theTCB we must, of course, check theRC.

ATTACH (cont)

What would we do in metal C?

In global scope

```
__asm("AttachStatic ATTACH SF=L,"
      "JSTCB=NO,SM=PROB,SVAREA=YES"
: "DS"(AttachStatic));
```

ATTACH (cont)

In local scope

```
__asm("AttachDyn ATTACH SF=L": "DS"(AttachDyn));
AttachDyn = AttachStatic;
__asm(" ATTACH EPLOC=%3,DCB=%4,"
      "SF=(E,%2)\n"
      " ST    1,%1\n"
      " ST    15,%0"
: "=m"(theRC), "=m"(theTCB), "=m"(AttachDyn)
: "m"(theEP), "m"(theDCB)
: "r0","r1","r14","r15");
```

Summary

- You really can invoke z/OS macros from within Metal C.
- Keep the compiler informed.
- Don't be afraid to check the generated assembler code to see that it did what you wanted.

*

Publications

- z/OS V1R12.0 Metal C Programming Guide and Reference
  SA23-2225-03
- z/OS V1R12.0 XL C/C++ Language Reference
  SC09-4815-10