Chapter 18: Writing Macros

This lecture will focus on writing **macros**, and use stack handling as an example of macro use. Macros differ from standard subroutines and functions. Functions and subroutines represent separate blocks of code to which control can be transferred. Linkage is achieved by management of a **return address**, which is managed in various ways.

A macro represents code that is automatically generated by the assembler and inserted into the source code. Macros are less efficient in terms of code space; each invocation of the macro will generate a copy of the code. Macros are more efficient in terms of run time; they lack the overhead associated with subroutine call and return. There is an important definition that is key to understanding what a macro is and what it does.

**Definition:** A macro definition is a pattern for a **character–by–character textual substitution** without interpretation, and a macro invocation causes the assembler to effect that substitution exactly as written.

### Dynamic Memory: Stacks and Heaps

Before discussing macros, let’s discuss an application. The first thing to note in our discussion of dynamic memory, especially stacks and heaps, is that these features are not supported by our version of the System/370 assembler.

A **stack** is a LIFO (Last–In / First–Out) data structure with three basic operations:

- **PUSH** places an item onto the stack,
- **POP** removes an item from the stack
- **INIT** initializes the stack.

A heap is a dynamic structure used by a RTS (Run–Time System) to allocate memory in response to object creators, such as New. A modern RTS will allocate an area of memory for use by both the stack and the heap. By convention in system design:

1. The stack starts at high memory addresses and moves toward lower addresses.
2. The heap starts at low memory addresses and moves toward higher addresses.

### Division of the Dynamic Memory Space

This shows how the available space is divided between the stack and the heap. There is no fixed allocation to either, just a limit on the total space used.

A stack is often managed using a stack pointer, SP, that locates its top.
Our Stack Implementation
The first caution in our implementation regards the selection of names for our macros. IBM has macros called “PUSH” and “POP”, associated with handling print output. We must pick other names for our stack macros. Our goal in this lecture is to examine the basic stack structure, and its implementation using macros.

Our implementation will use a fixed–size array to hold the stack. The design will be atypical in that the stack will grow towards higher addresses. The stack pointer will point to the location into which the next item will be pushed. The two basic stack operations, as we implement them, are illustrated in the figures below.

**PUSH**

```
STACK[SP] = ITEM
SP = SP + 1      // Moves toward higher addresses
```

**POP**

```
SP = SP – 1
ITEM = STACK[SP]
```

This non–standard approach is easier for me to code.

A Stack Example
Here we push four integers, one after the other. We then pop the values.

Push onto the stack

```
Empty
Push 1
Push 2
Push 3
Push 4
```

Pop from the stack: note the order is reversed.

```
Pop
Pop
Pop
Pop
Empty
```
Our Stack Implementation: Macro or Subroutine?
We have a choice of implementation method to use for our stack handler.
I have chosen to use an approach using macros for two reasons.
1. I wanted to discuss macros.
2. I wanted to use a stack to illustrate the subroutine call mechanism.
   That makes it difficult to use a subroutine for the stack.

We shall write three macros for the stack.

STKINIT This is a macro without parameters.
   It will initialize the stack count and also the stack pointer.

STKPUSH This is a macro with a single parameter.
   It pushes the 32–bit contents of a register onto the stack.

STKPOP This is a macro with a single parameter.
   It pops the contents of the stack top into a 32–bit register.

AGAIN: These names are chosen to avoid name conflicts with existing macros.

Mechanics of Writing Macros
The MACRO definitions should occur very early in the source code of the assembler
program. Only comments and assembler control directives may precede a MACRO
definition. This commonly includes the PRINT directive.

A MACRO begins with the key word MACRO, includes a prototype and a macro body, and
ends with the trailer keyword MEND.

Parameters to a MACRO are prefixed by the ampersand “&”.

Here is an example.

Header MACRO
Prototype DIVID &QUOT,&DIVIDEND,&DIVISOR
Model Statements ZAP &QUOT,&DIVIDEND
                     DP &QUOT,&DIVISOR
Trailer MEND

Note that the header and trailer must appear exactly as shown above. Each of the terms
“MACRO” and “MEND” begin in column 10. Nothing else is allowed on either line.

The basic idea of a macro is to replace multiple copies of repeated code with a single macro
 invocation. Here, the savings are minimal, as we are replacing two lines of code with one
line of code. Again, the reader is cautioned the some teaching examples are quite small.

With the above macro definition, based on packed decimal arithmetic, the idea is to replace
the following two lines of code with the line that follows them.

Replace      ZAP X,Y
             DP X,Z
With         DIVID X,Y,Z
Example of Macro Expansion
In assembly language, a macro is a single statement that causes the assembler to emit a sequence of other statements specified by the macro definition. Consider the above example, with prototype

```
DIVID &QUOT,&DIVIDEND,&DIVISOR.
```

The macro body is

```
ZAP &QUOT,&DIVIDEND
DP &QUOT,&DIVISOR
```

Here is an example of the macro expansion. We assume that the labels used as “parameters” have been properly defined by DS or DC statements.

```
DIVID MPG,MILES,GALS MACRO INSTRUCTION
+ ZAP MPG,MILES ITS EXPANSION
+ DP MPG,GALS
```

What Do We Mean by “Expansion”?
Consider the following code fragment, written to include a call to a macro.

```
PACK MILES,CARDIN+10(4) COLUMNS 10 – 13
PACK GALS,CARDIN+14(3) COLUMNS 14 – 16
DIVID MPG,MILES,GALS INVOKE THE MACRO
MVC MPGPR,=X’40202020’ MOVE THE EDIT MASK
ED MPGPR,MPG EDIT FOR PRINTING
```

Here is the code that is actually generated. I have inserted line numbers. Note that the macro invocation itself is not an executable instruction.

```
51 PACK MILES,CARDIN+10(4) COLUMNS 10 – 13
52 PACK GALS,CARDIN+14(3) COLUMNS 14 – 16
54 ZAP MPG,MILES ITS EXPANSION INTO
55 DP MPG,GALS TWO LINES OF CODE
56 MVC MPGPR,=X’40202020’ MOVE THE EDIT MASK
57 ED MPGPR,MPG EDIT FOR PRINTING
```

Symbolic Parameters
The macro prototype contains a list of symbolic parameters. Each symbolic parameter is written as follows:

1. The name begins with an ampersand (&).
2. The ampersand is followed by one to seven alphanumeric characters, the first of which must be a letter. The total length must be between 2 and 8 characters: first an “&”, then a letter, then zero to six alphanumeric characters.
3. Symbolic parameters have a local scope; that is, the name and value they are assigned only applies to the macro definition in which they have been declared. [Page 251, R_17]
Keyword Macros
A standard invocation of the above macro might appear as follows:

```
DIVID  MPG,MILES,GALS
```

In the above macro invocation, the arguments are passed by position. A macro invoked this way is called a **positional macro**. Another use, called a **keyword macro**, allows arguments to be passed in any order because each argument is tagged with an explicit symbolic parameter. Keyword macros also allow default values for each or all of the parameters.

The definition of a keyword macro differs from that of a positional macro only in the form of the prototype. Each symbolic parameter must be of the form `&PARAM=[DEFAULT]`. What this says is that the symbolic parameter is followed immediately by an “=”, and is optionally followed by a default value. As a keyword macro, the above example can be written as:

```
MACRO
Prototype
  DIVID2  &QUOT=, &DIVIDEND=, &DIVISOR=
Model Statements
  ZAP    &QOUT, &DIVIDEND
  DP     &QUOT, &DIVISOR
Trailer
  MEND
```

Here are a number of equivalent invocations of this macro, written in the keyword style. Note that this definition has not listed any default values.

```
DIVID2  &QUOT=MPG, &DIVIDEND=MILES, &DIVISOR=GALS
DIVID2  &DIVIDEND=MILES, &DIVISOR=GALS, &QUOT=MPG
DIVID2  &QUOT=MPG, &DIVISOR=GALS, &DIVIDEND=MILES
```

It is possible to use labels defined in the body of the program as default values.

```
MACRO
  DIVID2  &QUOT=MPG, &DIVIDEND=, &DIVISOR=
  ZAP    &QOUT, &DIVIDEND
  DP     &QUOT, &DIVISOR
  MEND
```

With this definition, the two invocations are exactly equivalent.

```
DIVID  MPG,MILES,GALS
DIVID2 &DIVIDEND=MILES, &DIVISOR=GALS
```

The invocation of the macro DIVID2 will expand as follows:

```
ZAP  MPG,MILES
DP   MPG,GALS
```

It is interesting to note that a keyword macro cannot be invoked as if it were a positional macro. The student should consult the following listing to see what happens.

From the listing of the macro invocations, we can infer that the statement

```
DIVID2  MPG,MILES,GALS
```

is treated as if there were no arguments present.
One may specify default constants in the keyword macro, being careful to observe the correct syntax. For example, one might be tempted to specify $&DIVISOR=10$, but the number by itself will name a register. The only way to do this would be set $&DIVISOR$ to $= \text{P}'10'$, by using the construct required to pass literals to a keyword macro.

```assembly
MACRO
DIVID3 $&QUOT=\text{MPG}, &DIVIDEND=, &DIVISOR==\text{P}'10'$
ZAP $&QUOT, &DIVIDEND$
DP $&QUOT, &DIVISOR$
MEND
```

The above usage is explained simply “If the value of a keyword operand is a literal, two equal signs must be specified.” [R_17, page 300]. A more complete explanation of the above can be seen by considering the macro `DIVID2`. The student will note the shortening of the keywords in what follows, in an attempt to fit the listings on the page.

Here is the prototype `DIVID2 $&QUOT=\text{MPG}, &DVD=, &DVS=\text{P}'20'$`

Here is a correct invocation `DIVID2 \text{QUOT=}ARG1, \text{DVD=}ARG2, \text{DVS=}\text{P}'20'$`

The key here is to remove the text fragments “\text{QUOT=}”, “\text{DVD=}”, and “\text{DVS=}”, and see what remains. Let’s do that. Consider `\text{QUOT=}\text{ARG1}, \text{DVD=}\text{ARG2}, \text{DVS=}\text{P}'20'$`

What remains is “\text{ARG1}”, “\text{ARG2}”, and “\text{P}'20'$”, each of which is a correct argument. The third argument is a literal value for the packed decimal with value 20. Had we invoked the macro with the third argument as `\text{DVS=}\text{P}'20'$`, the third argument would have been just “\text{P}'20'$”, which is meaningless to the assembler.

**Sample Expansion Listings for Macros**

Here is some assembly output from a program that I wrote to test these ideas.

```assembly
31 *
32 * MACRO DEFINITIONS
33 *
34 MACRO
35 DIVID $&QUOT, &DVD, &DVS$
36 ZAP $&QUOT, &DVD$
37 DP $&QUOT, &DVS$
38 MEND
39 *
40 MACRO
41 DIVID2 $&QUOT=, &DVD=, &DVS=$
42 ZAP $&QUOT, &DVD$
43 DP $&QUOT, &DVS$
44 MEND
45 *
46 MACRO
47 DIVID3 $&QUOT=, &DVD=, &DVS==\text{P}'10'$
48 ZAP $&QUOT, &DVD$
49 DP $&QUOT, &DVS$
50 MEND
51 *
```
Here is the listing for the expansions of the macros. Note the use of a literal argument in lines 100 and 109. In the positional macro, the literal has a single equals sign, while in the keyword macro it has two equals signs.

Note the errors in the first expansion of `DIVID2`. Consider line 104 in particular. The macro definition indicates that the text “ZAP” is to be followed by a text string for the first argument, followed by a comma, followed by a text string for the second argument. However, neither text string has been provided properly, so it attempts to generate the string “ZAP , ”, which has no meaning.

```
95 * SOME MACRO INVOCATIONS
96 *
97 DIVID ARG1,ARG2,ARG3
00004A F831 C0B2 C0B6 000B8 000BC 98+ ZAP ARG1,ARG2
000050 FD31 C0B2 C0B8 000B8 000BE 99+ DP ARG1,ARG3
100 DIVID ARG1,ARG2,=P'30'
000056 F831 C0B2 C0B6 000B8 000BC 101+ ZAP ARG1,ARG2
00005C FD31 C0B2 C322 000B8 00328 102+ DP ARG1,=P'30'
103 DIVID2 ARG1,ARG2,ARG3
000062 0000 0000 0000 00000 00000 104+ ZAP ,
** ASMA074E Illegal syntax in expression - ,
000068 0000 0000 0000 00000 00000 105+ DP ,
** ASMA074E Illegal syntax in expression - ,
106 DIVID2 DVD=ARG2,DVS=ARG3,QUOT=ARG1
00006E F831 C0B2 C0B6 000B8 000BC 107+ ZAP ARG1,ARG2
000074 FD31 C0B2 C0B8 000B8 000BE 108+ DP ARG1,ARG3
109 DIVID2 DVD=ARG2,DVS==P'20',QUOT=ARG1
00007A F831 C0B2 C0B6 000B8 000BC 110+ ZAP ARG1,ARG2
000080 FD31 C0B2 C324 000B8 0032A 111+ DP ARG1,=P'20'
112 DIVID3 DVD=ARG2,QUOT=ARG1
000086 F831 C0B2 C0B6 000B8 000BC 113+ ZAP ARG1,ARG2
00008C FD31 C0B2 C326 000B8 0032C 114+ DP ARG1,=P'10'
115 *
```
A Potential Problem with Macros.
It might appear that a macro invocation cannot be the target of a branch instruction. Here is some of my early code. I had defined a macro, \texttt{STKPOP}, in the proper place. It was used by a routine, called \texttt{DOFACT}, to be discussed later. As we shall see, \texttt{DOFACT} computes the factorial of a small integer, hence the name.

At the time, I was working with non-standard ways to invoke subroutines. I tried the following code:

\begin{verbatim}
B DOFACT CALL THE FACTORIAL CODE
\end{verbatim}

Here is the branch target.

\begin{verbatim}
DOFACT STKPOP 4 POP THE ARGUMENT INTO R4
   STKPOP 8 POP THE RETURN ADDRESS
   BR 8 BRANCH TO RETURN ADDRESS
\end{verbatim}

That did not assemble. The complaint was that the symbol \texttt{DOFACT} was not defined. What happened? The label was clearly there in the source code. Where did the label go?

Here is What Happened.
Consider the following expansion from a macro call. It has been edited for clarity. At present, the reader should not worry about lines 134 – 136 of the listing, but just focus on line 137 (the macro invocation) and its expansion.

\begin{verbatim}
0000BA 4840 C4AE 134 A92PO LH 4,STKCOUNT
0000BE 4940 C5B4 135 CH 4,=H'0'
0000C2 47D0 C0FE 136 BNP A98DONE
   137 STKPOP 4
0000C6 4830 C4AE 138+ LH 3,STKCOUNT
0000CA 4B30 C5B2 139+ SH 3,=H'1'
0000CE 4030 C4AE 140+ STH 3,STKCOUNT
0000D2 8B30 0002 141+ SLA 3,2
0000D6 4120 C4B2 142+ LA 2,THESTACK
0000DA 5843 2000 143+ L 4,0(3,2)
0000DE The next instruction
\end{verbatim}

Note that the \texttt{STKPOP} instruction on line 137 is not assigned an object code address.

The instruction on line 136 is at address C2 and has length 4. The next instruction will be at address C6. Only the expanded code is “real”. Line 137 is basically a comment.

In other words, we note two facts:
1. The expansion code is what counts for code accuracy.
2. The label \texttt{DOFACT} does not “make it” into the expanded code.

In my early work on the subject I had concluded that a macro invocation could not also be a branch target. Then I did something almost radical, I actually read the relevant portion of the IBM Assembler Language Manual [R_17]. I found the solution.
The Solution to the Branch Target Problem

In order to solve the above problem, we need to focus on a more precise statement of the form of a macro definition. We must focus on the prototype and body.

The general form of a prototype statement is as follows.

Symbolic Name   Name of macro   Zero or more symbolic parameters

If the symbolic name is to be used, it has the form of a symbolic parameter.
If the symbolic name is to be used, it must be duplicated on the first line of the body.

Here is an example, using the DIVID macro.

```
MACRO
&LABEL DIVID &QUOT,&DIVIDEND,&DIVISOR
&LABEL ZAP &QUOT,&DIVIDEND
&LABEL DP &QUOT,&DIVISOR
MEND
```

Note that the symbolic parameter “&LABEL” is treated as any other such parameter. In particular, it has local scope; thus the parameter has meaning only within the macro. The most important point is that the label, first seen in the prototype is repeated in the first model statement. It is that repetition that allows the label to be present in the expanded code.

Consider the prototype &LABEL DIVID &QUOT,&DIVIDEND,&DIVISOR matched against the invocation B10DIV DIVID X,Y,Z

This forces the following substitutions in the model statements of the macro body.

&LABEL is replaced by B10DIV, &QUOT is replaced by X, etc. This positional replacement mimics that seen in arguments to functions as used in high–level languages.

Code Example to Illustrate the Solution

```
MACRO
&LABEL DIVID &QUOT,&DIVIDEND,&DIVISOR
&LABEL ZAP &QUOT,&DIVIDEND
&LABEL DP &QUOT,&DIVISOR
MEND
```

Note that each of the labels B10DIV and B20DIV now appears in the expanded code and can be used as a branch target address.
Concatenation: Building Operations
In a model statement, it is possible to concatenate two strings of characters.
Consider the macro prototype to load a register from one of several sources.
Note the use of the string "&NAME" to allow this to be a branch target.

MACRO
&NAME LOAD &REG,&TYPE,&ARG
&NAME L&TYPE &REG,&ARG
MEND

Consider a number of invocations.

LOAD R7,R,R6 becomes LR R7,R6
LOAD R7,H,HW becomes LH R7,HW
LOAD R7,,FW becomes LR7,FW

Note that the second argument in the third example is empty. The empty string is concatenated to "L" to produce the single character "L".

Our Stack Data Structure
The stack is implemented as an array of full words, with two auxiliary counters.
There is a halfword that counts the number of items on the stack.
There is a halfword constant that gives the maximum stack capacity. This is not changed by the code. There is the fixed–size array that holds the stack elements.

Here is the declaration of the stack.

STKCOUNT DC H’0’ THE NUMBER OF ITEMS STORED ON STACK
STKSIZE DC H’64’ THE MAXIMUM STACK CAPACITY
THESTACK DC 64F’0’ THE STACK IS ACTUALLY AN ARRAY OF 64 FULLWORDS, REQUIRING 256 BYTES OF STORAGE.

Note that the elements are full–words while the addresses are byte addresses. The elements of the stack will be stored at the following addresses.

THESTACK, THESTACK + 4, THESTACK + 8, THESTACK + 12 up to a full word starting at THESTACK + 252.

Initialize the Stack
Here is the macro that initializes the stack.

*STKINIT
MACRO
&L1 STKINIT
&L1 SR 4,4 CLEAR R4 – SUBTRACT FROM SELF
STH 4,STKCOUNT STORE AS THE STACK COUNT
MEND

*Note the standard trick of clearing a register by subtracting it from itself. The register exists only for the purpose of placing a 0 into the stack count. Following standard practice, the contents of the stack are not changed, because the elements of interest will be overwritten before they are used. Note that this macro does not have any symbolic parameters.
PUSH: Placing Items Onto the Stack
Here is the macro STKPUSH

*STKPUSH
MACRO
&L2    STKPUSH &R
&L2    LH 3,STKCOUNT
*      SLA 3,2
      LA 2,THESTACK
      ST &R,0(3,2)
      LH 3,STKCOUNT
      AH 3,=H'1'
      STH 3,STKCOUNT
      MEND

This macro has one symbolic parameter: &R. It is to be a register number.
When called as STKPUSH 4, the operative statement is changed by the assembler to
ST 4,0(3,2) and executed as such at run time.

POP: Removing Items From the Stack
Here is the macro STKPOP

*STKPOP
MACRO
&L3    STKPOP &R
&L3    LH 3,STKCOUNT
      SH 3,=H'1'
      STH 3,STKCOUNT
      SLA 3,2
      LA 2,THESTACK
      L &R,0(3,2)
      MEND

* Again, this macro has one symbolic parameter: &R. Again, a register number.
When called as STKPOP 6, this is assembled with the last statement as
L 6,0(3,2).
NOTE:When invoked as STKPOP MYDOG, this will
assemble as L MYDOG,0(3,2); the assembler takes anything.
Needless to say, this last invocation will generate nonsense code if it assembles at all. If the code does assemble, it will likely generate a run time error. The only way in which this bit of doggerel (pardon the pun) would assemble is if the symbol MYDOG were equated (with EQU) to an integer that could be interpreted as a general purpose register.
Using the Macros
Here is the part of the unexpanded source code that uses the macros. Here, it is obvious that I have retained register R4 for communicating results with macros and subroutines. That is an arbitrary choice.

```
STARTUP
OPEN (FILEIN, (INPUT))       OPEN THE STANDARD INPUT
OPEN (PRINTOUT, (OUTPUT))    OPEN THE STANDARD OUTPUT
PUT PRINTER, PRHEAD          PRINT HEADER
STKINIT                      INITIALIZE THE STACK
GET FILEIN, RECORDIN         GET THE FIRST RECORD, IF THERE

* A10LOOP
MVC DATAPR, RECORDIN          MOVE INPUT RECORD
PUT PRINTER, PRINT            PRINT THE RECORD
PACK PACKIN, FIELD01          CONVERT DIGITS INPUT TO PACKED
CVB R4, PACKIN                CONVERT THE NUMBER TO BINARY
STKPUSH 4                     PUSH THE NUMBER ONTO THE STACK
GET FILEIN, RECORDIN          GET THE NEXT RECORD
B A10LOOP                     GO BACK AND PROCESS

* A90END
CLOSE FILEIN
PUT PRINTER, ENDNOTE          ANNOUNCE THE END OF INPUT DATA
A92POP
LA 4, STKCOUNT                GET THE STACK COUNT
CH 4, =H'0'                   IS THE COUNT POSITIVE?
BNP A98DONE                   NO, WE ARE DONE
STKP 4                        GET NEXT NUMBER INTO R4
MVC PRINT, BLANKS             CLEAR THE OUTPUT BUFFER
BAL 8, NUMOUT                 PRODUCE THE FORMATTED SUM
MVC DATAPR, THENUM            AND COPY TO THE PRINT AREA
PUT PRINTER, PRINT            PRINT THE RESULT
B A92POP                      GO AND GET ANOTHER OUTPUT
A98DONE                       CLOSE PRINTER

Expansion of the Stack Pop
Here is the expanded code, edited from the assembler listing.

```

```
136 A92POP
LA 4, STKCOUNT
137 CH 4, =H'0'
138 BNP A98DONE
139 STKP 4
140+ LH 3, STKCOUNT
141+ CH 3, =H'0'
142+ SH 3, =H'1'
143+ STH 3, STKCOUNT
144+ SLA 3, 2
145+ LA 2, THESTACK
146+ L 4, 0 (3, 2)
147 MVC PRINT, BLANKS
148 BAL 8, NUMOUT
149 MVC DATAPR, THENUM
150 PUT PRINTER, PRINT

Note: There is no RETURN statement or the like. The code is inserted in line.
```
A Problem with the Macros
There is a problem with each of the macros STKPUSH and STKPOP. We show it for STKPOP, because it is easier to see in this macro. Suppose we have code with the following two macro calls, one immediately following the other.

```
STKINIT
STKPOP 6       NOTE: WE HAVE NOT PUSHED AN ITEM
```

The macro STKINIT will set the value at location STKCOUNT to 0. Now look at the code in the expansion of macro STKPOP.

```
139       STKPOP 4
140+      LH 3,STKCOUNT
141+      CH 3,='0'        NOTE: WE HAVE NOT PUSHED AN ITEM
142+      SH 3,='1'
143+      STH 3,STKCOUNT
```

STKCOUNT will be set to −1, and the pop will reference the full word just before the stack. This is the pair STKCOUNT, STKSIZE: an error. After line 143, the values will be.

```
STKCOUNT DC X'FFFF'     MINUS ONE
STKSIZE DC X'0040'     HEXADECIMAL REPRESENTATION OF 64.
```

Register 6 would be loaded with X'FFFF0040', which is a negative number. A bit of arithmetic reveals this to be the negative of the number represented in hexadecimal as X'0000FFC0' or as 65,472 in decimal.

Avoiding the Problem: A Flawed Solution
The obvious solution is to test the value of STKCOUNT and avoid popping a value if the stack is empty. Here is some code that appears to do just that.

```
*STKPOP
MACRO
STKPOP &R
  LH 3,STKCOUNT GET THE STACK SIZE
  CH 3,='0'
  BNP NOPOP
  SH 3,='1' SUBTRACT 1 WORD OFFSET OF LAST
  STH 3,STKCOUNT WORD AND STORE AS NEW SIZE
  SLA 3,2 BYTE OFFSET OF STACK TOP
  LA 2,THESTACK ADDRESS OF STACK START
  L &R,0(3,2) LOAD ITEM INTO R4
  NOPOP NOP A DO NOTHING TARGET FOR BNP
MEND
*
```

If the macro is written this way, the code will assemble and run correctly. Actually, it runs correctly due only to a quirk in the code. It is a general principle that erroneous code might run on occasion, but it will not run always.

We shall hold out for code that is correct in that it will always assemble, always run, and always produce the correct result.
What Is the Flaw?
The macro definition given above works ONLY because the macro is invoked only one time. If the macro is invoked twice, trouble appears. In this modification of running code, the macro is called twice in a row.

```
A90END  CLOSE FILEIN   NO MORE INPUT TO PROCESS
         PUT PRINTER,ENDNOTE  NOTE THE END OF DATA INPUT
A92POP  LH  4,STKCOUNT  GET THE STACK COUNT
        CH  4,=H'0'       IS IT POSITIVE
        BNP A98DONE       NO - WE ARE DONE HERE
STKPOP  4         GET NEXT NUMBER INTO R4
STKPOP  5         **** BAD CALL
        MVC PRINT,BLANKS  CLEAR THE OUTPUT AREA
        BAL 8,NUMOUT     PRODUCE THE FORMATTED SUM
        MVC DATAPR,THENUM AND MOVE TO PRINT AREA
        PUT PRINTER,PRINT PRINT THE NUMBER
        B  A92POP        GO GET ANOTHER
A98DONE  CLOSE PRINTER
```

Listing for Double Use of the Macro
Notice in the listing below that the first macro expansion produces no problems. It is the second expansion that gives rise to the assembler error. The symbol **NOPOP** has already been used when it is redefined in the second expansion. This is not allowed.

Note that this would not be a problem for a symbolic parameter, which has scope local to the particular expansion of the macro.

```
139  STKPOP  4
140+  LH  3,STKCOUNT
141+  CH  3,=H'0'
142+  BNP NOPOP
143+  SH  3,=H'1'
144+  STH 3,STKCOUNT
145+  SLA 3,2
146+  LA  2,THESTACK
147+  L  4,0(3,2)
148+NOPOP NOP
148  STKPOP  5
149+  LH  3,STKCOUNT
150+  CH  3,=H'0'
151+  BNP NOPOP
152+  SH  3,=H'1'
153+  STH 3,STKCOUNT
154+  SLA 3,2
155+  LA  2,THESTACK
156+  L  4,0(3,2)
157+NOPOP NOP
** ASMA043E Previously defined symbol - NOPOP**
Avoiding the Problem: A Correct Solution
Here is a solution to the problem. It works, but it complex to write. The solution is based on the current location operator, $. It is a jump to a relative address in bytes. The complexity in writing this is due to counting the bytes in each instruction beginning with the branch instruction and ending just before the branch target. It is easy to miscount.

*STKPOP

MACRO
STKPOP &R
LH 3,STKCOUNT       GET THE STACK SIZE
SH 3,=H'1'           SUBTRACT 1 TO GET WORD OFFSET
*  
CH 3,=H'0'           IS THE NEW SIZE NEGATIVE?
BM  *+20             YES, SO CANNOT POP AN ITEM
STH 3,STKCOUNT       WORD AND STORE AS NEW SIZE
SLA 3,2              BYTE OFFSET OF STACK TOP
LA 2,THESTACK        ADDRESS OF STACK START
L &R,0(3,2)          LOAD ITEM INTO R4
SLA 3,0              A NO-OP TO SERVE AS A TARGET
MEND

Observations on the First Solution
The complexity of the above instruction is based on the necessity of counting bytes in the object code, not instructions in the source code. The above example is simple, because all instructions to be skipped have the same length. Let’s look at this again.

CH 3,=H'0'             IS THE NEW SIZE NEGATIVE?
BM  *+20               RX 4      A type RX instruction, length 4 bytes
STH 3,STKCOUNT         RX 4      This instruction is at address *+4
SLA 3,2                RS 4      A type RS instruction at address *+8
LA 2,THESTACK          RX 4      This is at address *+12
L &R,0(3,2)            RX 4      Another 4-byte instruction at *+16
SLA 3,0                The branch target at address *+20

The Preferred Solution
What we need is a way to generate a branch target that would be unique to each expansion of the macro. As should be expected, the System/370 assembler provides a method, which is based on concatenation of system variable symbols. We describe this process in two stages, first reviewing the idea of using concatenation to build symbols and operations. In our earlier discussion we used concatenation to build load operators for various types.

MACRO
&NAME  LOAD &REG,&TYPE,&ARG
&NAME  L&TYPE &REG,&ARG
MEND

Consider a number of invocations, each of which constructs a load operator.
LOAD R7,R,R6 becomes LR R7,R6
LOAD R7,H,HW becomes LH R7,HW
LOAD R7,,FW becomes L R7,FW
System Variable Symbols
The System/370 assembler provides a large number of special predefined symbols called “system variable symbols”. There are a number of these symbols. I mention three.

&SYSDATE
The system date, in the 8 character form “MM/DD/YY”.
Use in the form of a declaration of initialized storage, as in
TODAY DC C’&SYSDATE’

&SYSTIME
The system time of day, in the five character form “HH.MM”.
Also used in the form of a declaration, as in
NOW DC C’&SYSTIME’

&SYSNDX
The macro expansion index. For the first macro expansion, the
Assembler initializes &SYSNDX to the string “0001”. Each
expansion of any macro invocation increases the value represented
by 1, giving rise to the sequence “0001”, “0002”, “0003”, etc.

The &SYSNDX system variable symbol can prevent a macro from generating duplicate labels.
The system symbol is concatenated to a leading character, which begins the label and must
be unique within the macro definition. In what follows, we use the letter “L”. Consider the
following string, used as a label within the body of a macro definition.
L&SYSNDX L R4,STKSAV4
Note that the string “L&SYSNDX”, as written, contains eight characters: the initial character
“L” followed by the 7 character sequence “&SYSNDX”. On expansion, this will be converted
to labels such as “L0001”, “L0002”, etc. As the string “&SYSNDX” already takes seven
caracters, it is better to make the prefix a single letter, though multiple letters are allowed.
In actual fact, the requirement for the leading characters, to which the &SYSNDX is to be
appended can be any sequence of one to four characters, provided only that the first character
is a letter. Thus the following are valid, but they disrupt the flow of the listing.
A12&SYSNDX ... This label might become A120003.
WXYZ&SYSNDX ... This might become WXYZ0117.

A Simple Example of Label Generation
Consider the simple macro used for packed division in the previous lecture.
We adapt it to prevent division by zero.

MACRO
&LABEL DIVID &QUOT,&DIVIDEND,&DIVISOR
&LABEL ZAP &QUOT,&DIVIDEND
CP &DIVISOR,=P’0’ IS IT ZERO
BNE A&SYSNDX NO, DIVISION IS OK
ZAP &QUOT,=P ’0’ YES, SET QUOTIENT TO 0
B B&SYSNDX
A&SYSNDX DP &QUOT,&DIVISOR
B&SYSNDX NOPR R3 DO NOTHING
MEND

Note that the format of the NOPR instruction requires a register number
(here R3), even though the instruction does nothing.
Sample Expansion of the Macro
With the above definition, consider the following expansions.

```
A10START DIVID X,Y,Z
  +A10START ZAP X,Y
  +      CP Z,=P'0'  IS IT ZERO
  +      BNE A0001  NO, DIVISION IS OK
  +      ZAP X,=P'0'  YES, SET QUOTIENT TO 0
  +      B   B0001
  +A0001  DP X,Z
  +B0001  NOPR R3  DO NOTHING

A20DOIT DIVID A,B,C
  +A20DOIT ZAP A,B
  +      CP C,=P'0'  IS IT ZERO
  +      BNE A0002  NO, DIVISION IS OK
  +      ZAP X,=P'0'  YES, SET QUOTIENT TO 0
  +      B   B0002
  +A0002  DP A,C
  +B0002  NOPR R3  DO NOTHING
```

Note that each invocation has distinct labels. This removes the name clashes.

For the first expansion of the macro DIVID, the label &SYSNDX is replaced by the string “0001” and on the second expansion, the label is replaced by “0002”.

It is important to note that the &SYSNDX is incremented due to the expansion of any macro. Were there another macro expansion between the two invocations of the macro DIVID, the second invocation of that macro would be associated with the replacement of the label &SYSNDX by the string “0003”. The string “0002” would be associated with the intermediate macro expansion, assuming that it used the system symbol &SYSNDX.

The Preferred Solution Applied to STKPOP
Here is a revision of the code that will avoid the problem of duplicate labels.

```
*STKPOP
  MACRO
  STKPOP &R
  LH 3,STKCOUNT GET THE STACK SIZE
  CH 3,=H'0'
  BNP &SYSNDX
  SH 3,=H'1'  SUBTRACT 1 WORD OFFSET OF LAST
  STH 3,STKCOUNT WORD AND STORE AS NEW SIZE
  SLA 3,2  BYTE OFFSET OF STACK TOP
  LA 2,THESTACK ADDRESS OF STACK START
  L &R,0(3,2) LOAD ITEM INTO R4
  &SYSNDX NOP  A DO NOTHING TARGET FOR BNP
  MEND
```
STKPOP: Preferred Solution with Two Invocations
The following listing was produced when the revised macro definition above was implemented in the source code.

```
139     STKPOP 4
140+    LH  3,STKCOUNT
141+    CH  3,=H'0'
142+    BNP L0001
143+    SH  3,=H'1'
144+    STH 3,STKCOUNT
145+    SLA 3,2
146+    LA  2,THESTACK
147+    L  4,0(3,2)
148+L0001 NOP
148     STKPOP 5
149+    LH  3,STKCOUNT
150+    CH  3,=H'0'
151+    BNP L0002
152+    SH  3,=H'1'
153+    STH 3,STKCOUNT
154+    SLA 3,2
155+    LA  2,THESTACK
156+    L  4,0(3,2)
157+L0002 NOP
```

Pushing from Various Sources
We look first at the handling of our STKPUSH. The only restriction on the stack is that every value pushed be treated as a 32–bit fullword. As a result, a 16–bit halfword will be sign–extended to a 32–bit fullword before being pushed onto the stack. This is similar to the function of the LH instruction, which loads a register from a halfword.

The key instruction in the original STKPUSH macro is the following.

```
ST &R,0(3,2) STORE THE ITEM INTO THE STACK
```

In this case, the item to be placed on the stack is found in the register indicated by the symbolic parameter &R.

The way to extend this instruction to all data types is as follows.

1. Select a register to be a fixed source for the word on the stack, and
2. Construct instructions to load that fixed register from the source.

What Shall Be Stored on the Stack?
At this point, we have a decision to make. What data types to store? The size restriction on the stack limits the simple choices to addresses and the contents of registers, halfwords, and fullwords. We must select a working register for the new macro. I select R4.

The “key code” becomes as follows.

Stacking an address `LA R4,&ARG` Load address into R4.
Stacking a halfword `LH R4,&ARG` Load halfword into R4.
Stacking a fullword `L R4,&ARG` Load fullword into R4.
Stacking a register `LR R4,&ARG` Load value from source register
Passing the Type in a Macro Invocation

The solution adopted to the problem above is to pass the type in the macro call and use concatenation to build the load operator. Here is some code taken from a macro definition that has been run and tested.

First, we show the macro prototype.

\begin{verbatim}
&L2   STKPUSH &ARG, &TYP
\end{verbatim}

Next we show the “key instruction” in the macro body.

\begin{verbatim}
L&TYP R4, &ARG
\end{verbatim}

Here are four typical invocations of the macro.

\begin{verbatim}
STKPUSH R7, R     PUSH VALUE IN REGISTER.
STKPUSH HHW, H    PUSH A HALFWORD VALUE.
STKPUSH FFW, A    PUSH AN ADDRESS.
STKPUSH FFW        PUSH A FULLWORD.
\end{verbatim}

Note that the last invocation lacks a second argument. In the expansion, this causes \&TYP to be set to ‘ ‘, a blank; “L&TYP” becomes “L ”.

The Macro Definition

Here is the definition for the macro at this stage of its development.

\begin{verbatim}
MACRO
&L2   STKPUSH &ARG, &TYP
&L2   LH  R3, STKCOUNT
SLA  R3, 2
LA   R2, THESTACK
L&TYP R4, &ARG
ST   R4, 0 (3, 2)
LH   R3, STKCOUNT
AH   R3, =H'1'
STH  3, STKCOUNT
MEND
\end{verbatim}

Again, the “&L2” allows the macro invocation to be a branch target. This is a practice that your author has decided to employ, even absent a present need to use any invocation of the macro as a branch target. This is a flexibility option only; one that is easy to implement.

At this point, the code fixes on general–purpose registers R3 and R4 for use. There is no particular logic to these choices; it is just that two registers had to be chosen. The point here is to focus on the construction of the operator using the concatenation “L&TYP”.

This macro will be invoked with four distinct values for the second parameter, &TYP. Again, the value is “” for push fullword, “H” for push a sign–extended halfword, “A” for an address, and “R” for register. As always, there is insufficient error checking code. It is assumed that the macro will always be invoked with the correct type.
Some Invocations of this Macro

91 STK PUSH R7, R
92+ LH R3, STK COUNT
93+ SLA R3, 2
94+ LA R2, THE STACK
95+ LR R4, R7
96+ ST R4, O (3, 2)
97+ LH R3, STK COUNT
98+ AH R3, =H'1'
99+ STH 3, STK COUNT

100 STK PUSH HHW, H
101+ LH R3, STK COUNT
102+ SLA R3, 2
103+ LA R2, THE STACK
104+ LH R4, HHW
105+ ST R4, O (3, 2)
106+ LH R3, STK COUNT
107+ AH R3, =H'1'
108+ STH 3, STK COUNT

More Invocations of this Macro

109 STK PUSH FFW
110+ LH R3, STK COUNT
111+ SLA R3, 2
112+ LA R2, THE STACK
113+ L R4, FFW
114+ ST R4, O (3, 2)
115+ LH R3, STK COUNT
116+ AH R3, =H'1'
117+ STH 3, STK COUNT

118 STK PUSH FFW, A
119+ LH R3, STK COUNT
120+ SLA R3, 2
121+ LA R2, THE STACK
122+ LA R4, FFW
123+ ST R4, O (3, 2)
124+ LH R3, STK COUNT
125+ AH R3, =H'1'
126+ STH 3, STK COUNT

NOTE: The originals of the program listing are found at the end of the chapter.
Saving the Work Registers
As written, this macro has the side effect of changing the values of three registers: R2, R3, and R4. The value of R4 is preserved only if it is being pushed. We should write macros so that they operate without side effects. The only way to do this is to save and restore the values of the work registers. There are many ways to do this. The simplest is to alter the stack data structure. Here is the new version.

```
STKCOUNT DC H'0'       NUMBER OF ITEMS STORED ON STACK
STKSIZE  DC H '64'     MAXIMUM STACK CAPACITY
STKSAV2  DC F'0'       SAVES CONTENTS OF R2
STKSAV3  DC F '0'      SAVES CONTENTS OF R3
STKSAV4  DC F '0'      SAVES CONTENTS OF R4
THESTACK DC 64F '0'    THE STACK HOLDS 64 FULLWORDS
```

This new definition does not alter the STKINIT macro. It does affect the other two macros: STKPOP and STKPUSH. We illustrate the latter.

The First Revision of STKPUSH
Here is the revision that allows the work registers to be saved.

```
MACRO
   STKPUSH &ARG, &TYP
   &L2 ST R2, STKSAV2
   &L2 ST R3, STKSAV3
   ST R4, STKSAV4
   LH R3, STKCOUNT
   SLA R3, 2
   LA R2, THESTACK
   L&TYP R4, &ARG
   ST R4, 0(3, 2)
   LH R3, STKCOUNT
   AH R3, =H'1'
   STH R3, STKCOUNT
   L R4, STKSAV4
   L R3, STKSAV3
   L R2, STKSAV2
MEND
```

The Status of the Macros at This Point
There are a few issues to be addressed at this point.

The only macro that will not change is the initialization macro, STKINIT.

1. We have not yet dealt with generalizing the STKPOP macro.
2. We have not yet dealt with either the stack empty problem or that of the stack being full. Each has to be addressed.

Each of these issues requires some additional code. We now move towards the final versions of each of the macros.
The First Revision of STKINIT
Here is a revision of the STKINIT code that allows initialization of its size. This was done in order to show how to concatenate the symbolic parameter &SIZE as a prefix.

```
35 MACRO
36 &L1 STKINIT &SIZE
37 &L1 ST R3,STKSAV3
38 SR R3,R3
39 STH R3,STKCOUNT
40 L R3,STKSAV3
41 B L&SYSNDX
42 STKCOUNT DC H'0'
43 STKSIZE DC H'&SIZE'
44 STKSAV2 DC F'0'
45 STKSAV3 DC F'0'
46 STKSAV4 DC F'0'
47 THESTACK DC &SIZE.F'0'
48 L&SYSNDX SLA R3,0
49 MEND
```

Note the “.” in the definition of THESTACK as DC &SIZE.F'0’. This concatenates the value of the symbolic parameter with “F’0’”, as in “128F’0’”

The Second Revision of STKPUSH
Here is the final version of the macro for pushing onto the stack.

```
&L2 STKPUSH &ARG,&TYP
&L2 ST R3,STKSAV3
LH R3,STKCOUNT GET COUNT OF ITEMS ON THE STACK
CH R3,STKSIZE IS THE STACK FULL?
BNL Z&SYSNDX YES, DO NOT ADD ANOTHER.
ST R4,STKSAV4 NO, WE CAN PUSH ANOTHER ITEM.
ST R2,STKSAV2 START BY SAVING THE OTHER 2 REGISTERS
SLA R3,2 MULTIPLY THE INDEX BY 4.
LA R2,THESTACK
L&TYP R4,&ARG FORM THE ADDRESS
ST R4,0(3,2) STORE THE ITEM
LH R3,STKCOUNT GET THE OLD COUNT OF ITEMS
AH R3,=H'1' INCREMENT THE COUNT BY 1
STH R3,STKCOUNT STORE THE CURRENT COUNT
L R4,STKSAV4 RESTORE THE REGISTERS.
L R2,STKSAV2
Z&SYSNDX L R3,STKSAV3
MEND
```
Conditional Assembly
Consider the problem of generalizing STKPOP. We shall want to pop the following from the stack: register values, halfwords, and fullwords. The type for the argument refers to the destination; an address can be popped into either a register or fullword. In order to see the problem for STKPOP, consider the “key instruction”.

Halfword:  \texttt{STH \ R4, \&ARG}  
Fullword:  \texttt{ST \ R4, \&ARG}  
Register:  \texttt{LR \&ARG, R4  No STR for store register}.

We could write a STR macro, but I want to use another solution. We have already seen how concatenation can be used to construct different instructions in a macro expansion. We now investigate conditional assembly, in which the expansion of a macro can lead to a number of distinct code sequences.

Conditional assembly permits the testing of attributes such as data format, data value, or field length, and to use the results of such testing to generate source code that is specific to the case in question. This chapter will focus on five specific conditional assembly instructions.

\begin{itemize}
  \item \texttt{AGO}  an unconditional branch
  \item \texttt{AIF}  a conditional branch. This means “Ask If”.
  \item \texttt{ANOP}  A NOP that can be the branch target for either \texttt{AGO} or \texttt{AIF}.
  \item \texttt{MNOTE}  print a programmer defined message at assembly time
  \item \texttt{MEXIT}  exit the macro definition.
\end{itemize}

Attributes for Use by Conditional Assembly
The assembler can generate code specified by certain attributes of the arguments to the macro definition at the time it is expanded. There are six types of attributes that can be associated with a parameter. Here are three if the more useful attributes.

\begin{itemize}
  \item \texttt{L’}  Length  The length of the symbolic parameter
  \item \texttt{I’}  Integer  The integer attribute of a fixed–point, floating–point, or packed decimal number.
  \item \texttt{T’}  Type  The type of the parameter, as specified by the DC or DS declaration with which it is defined.
\end{itemize}

Some types for the \texttt{T’} attribute are as follows.
\begin{itemize}
  \item \texttt{A}  Address  \texttt{H} Halfword
  \item \texttt{B}  Binary  \texttt{I} Instruction
  \item \texttt{C}  Character  \texttt{P} Packed Decimal
  \item \texttt{F}  Fullword  \texttt{X} Hexadecimal
\end{itemize}

The Sequence Symbol
Conditional assembly is built on the ability to generate conditional branching in the code generation process. In this, it is not that branch assembler language statements are used, but that entire segments of code will not even be assembled.

The assembler uses sequence symbols, denoted by the “.” (period) prefix. More on this later.
The Ask If (AIF) Instruction
The AIF instruction has two parts.
1. A logical expression in parentheses, and
2. A sequence symbol immediately following, which serves as the branch target.

The AIF logical expression may use the following relational operators, which are quite similar to those seen in early versions of the FORTRAN language.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>Equal To</td>
</tr>
<tr>
<td>LT</td>
<td>Less Than</td>
</tr>
<tr>
<td>GT</td>
<td>Greater Than</td>
</tr>
<tr>
<td>NE</td>
<td>Not Equal To</td>
</tr>
<tr>
<td>LE</td>
<td>Less Than or Equal To</td>
</tr>
<tr>
<td>GE</td>
<td>Greater Than or Equal To</td>
</tr>
</tbody>
</table>

If the type of &AMT is packed, go to .B23PACK
\[
\text{AIF}(T' \&AMT \text{ EQ } 'P').B23PACK
\]

If the type of &LINK is not an instruction, go to .R30ERROR
\[
\text{AIF}(T' \&LINK \text{ NE } 'I').R30ERROR
\]

Here, each of .B23PACK and .R30ERROR are sequence symbols.

Testing the Value of a Symbolic Parameter
What we want for the STKPOP instruction is a conditional assembly based on the value of the second parameter. The prototype for the macro will be something like
\[
&L1 \text{ STKPOP } &ARG, &TYP
\]

What we want to issue is an AIF statement such as
\[
\text{AIF} (\&TYP \text{ EQ } 'R').\text{ISREG}
\]

There is a well-known peculiarity in any assembler language, not just in the IBM Assembler, that disallows this straightforward construct.

We must put the symbolic parameter in single quotes. The statement is thus:
\[
\text{AIF} ('\&TYP' \text{ EQ } 'R').\text{ISREG}
\]

If &TYP is the character R, the logical expression becomes ('R' EQ 'R'), which immediately evaluates to True, and the branch is taken. [Page 384, R_17]

Targets for Use by Conditional Assembly
Each of the AGO and AIF instructions is a branch instruction that takes effect at assembly time. Neither persists into the assembly source code. It should be expected that the targets for either of these conditional assembly branch instructions should be of a distinct type. The targets for these are called sequence symbols.

The format of a sequence symbol is as follows. A sequence symbol begins with a period (.) followed by one to seven letters or digits, the first of which must be a letter. Unlike the symbols created by use of the &SYSNDX system symbol, sequence symbols do not persist into assembly time, and thus cannot generate a name conflict for the assembler.
A Sample of Conditional Assembly
Here is the DIVID macro, with conditional assembly instructions to insure that it is expanded only for parameters that are packed decimal.

```
MACRO
&LABEL DIVID &QUOT,&DIVIDEND,&DIVISOR
AIF (T'&QUOT NE 'P').NOTPACK
AIF (T'&DIVIDEND NE T'&QUOT).NOTPACK
AIF (T'&DIVISOR NE T'&QUOT).NOTPACK
AGO .DOIT
.NOTPAK MNOTE 'ONE PARAMETER IS NOT PACKED DECIMAL'
MEXIT
.DOIT ANOP
&LABEL ZAP &QUOT,&DIVIDEND
CP &DIVISOR,=P'0' IS IT ZERO
BNE A&SYSNDX NO, DIVISION IS OK
ZAP &QUOT,=P'0' YES, SET QUOTIENT TO 0
B B&SYSNDX
A&SYSNDX DP &QUOT,&DIVISOR
B&SYSNDX NOPR R3 DO NOTHING
MEND
```

Some Examples of the Conditional Assembly Divide Macro
In the following, assume that each of $X$, $Y$, and $Z$ is defined by a DC statement as packed decimal, but that $A$, $B$, and $C$ are defined as halfwords. Here are some possible expansions.

```
F10DOIT DIVID X,Y,Z
+F10DOIT ZAP X,Y
  + CP Z,=P'0' IS IT ZERO
  + BNE A0032 NO, DIVISION IS OK
  + ZAP X,=P'0' YES, SET QUOTIENT TO 0
  + B B0032
+A0032 DP X,Z
+B0032 NOPR R3 DO NOTHING
F25NODO DIVID A,B,C
+ONE PARAMETER IS NOT PACKED DECIMAL
```

The Original Definition of Macro STKPOP
We now begin our redefinition of the STKPOP macro.

We begin with the original definition, which popped a value into a register.

```
*STKPOP
MACRO
&L3 STKPOP &R
&L3 LH 3,STKCOUNT GET THE STACK COUNT
SH 3,=H'1' SUBTRACT 1 WORD OFFSET OF TOP
STH 3,STKCOUNT STORE AS NEW SIZE
SLA 3,2 BYTE OFFSET OF STACK TOP
LA 2,THESTACK ADDRESS OF STACK BASE
L &R,0(3,2) LOAD ITEM INTO THE REGISTER.
MEND
```
Again, this macro has one symbolic parameter: &R. Again, a register number. We want to expand this definition in a number of ways. We begin by introducing the type &TYP. At this point, it will become necessary to have another work register.

**Mechanics of the Revised STKPOP**

The new design will use register R4 to transfer the value at the top of the stack.

The new prototype will be as follows.

```
&L3   STKPOP &ARG, &TYP
```

Each type of instruction will include the following as the first statement in the “key code” – that which actually places the value into the destination.

```
L     R4, 0(3,2) LOAD ITEM INTO REGISTER R4.
```

The second statement of the “key code” depends on the type of the destination.

- &TYP == H  STH R4, &ARG
- &TYP == F  ST R4, &ARG
- &TYP == A  ST R4, &ARG (SAME AS FULLWORD)
- &TYP == R  LR &ARG, R4 COPY R4 INTO REGISTER

Here is the key code section, with the conditional assembly.

The first statement is common to all types.

```
L     R4, 0(3,2) LOAD ITEM INTO REGISTER R4.
AIF ('&TYPE' EQ 'R').ISREG
ST&TYP R4, &ARG
AGO .CONT
.ISREG LR &ARG, R4
.CONT The next statement.
```

**STKPOP: Revision 2**

Here I am going to add some code to save and restore the work registers.

```
MACRO
&L3   STKPOP &ARG, &TYP
&L3   ST R2, STKSAV2
ST R3, STKSAV3
ST R4, STKSAV4
LH R3, STKCOUNT GET THE STACK COUNT
SH R3, =H'1' SUBTRACT 1 WORD OFFSET OF TOP
STH R3, STKCOUNT STORE AS NEW SIZE
SLA R3, 2 BYTE OFFSET OF STACK TOP
LA R2, THESTACK ADDRESS OF STACK BASE
L     R4, 0(3,2) LOAD ITEM INTO REGISTER R4.
AIF ('&TYPE' EQ 'R').ISREG
ST&TYP R4, &ARG
AGO .CONT
.ISREG LR &ARG, R4
.CONT L     R4, STKSAV4
L     R3, STKSAV3
L     R2, STKSAV2
MEND
```
STKPOP: The Complete Version

MACRO

&L3 STKPOP &ARG, &TYP

&L3 ST R3, STKSAV3

LH R3, STKCOUNT GET THE STACK COUNT

CH R3, =H'0' IS THE COUNT POSITIVE

BNH Z&SYSNDX NO, WE CANNOT POP.

SH R3, =H'1' SUBTRACT 1 WORD OFFSET OF TOP

STH R3, STKCOUNT STORE AS NEW SIZE

SLA R3, 2 BYTE OFFSET OF STACK TOP

ST R2, STKSAV2 SAVE REGISTER R2

ST R4, STKSAV4 SAVE REGISTER R4

LA R2, THESTACK ADDRESS OF STACK BASE

L R4, 0(3, 2) LOAD ITEM INTO REGISTER R4.

AIF (’&TYPE’ EQ ’R’).ISREG

ST&TYP R4, &ARG

AGO .CONT

.ISREG LR &ARG, R4

.CNT L R4, STKSAV4

L R2, STKSAV2

Z&SYSNDX L R3, STKSAV3

MEND
Original Code for the Macro Expansions

MACRO DEFINITIONS

MACRO

STKPUSH &ARG, &TYP

LH R3, STKCOUNT

SLA R3, 2

LA R2, THESTACK

ST &TYP R4, &ARG

ST R4, 0 (3, 2)

LH R3, STKCOUNT

AH R3, =H'1'

STH 3, STKCOUNT

MEND

SOME MACRO INVOCATIONS

STKPUSH R7, R

LH R3, STKCOUNT

SLA R3, 2

LA R2, THESTACK

LR R4, R7

ST R4, 0 (3, 2)

LH R3, STKCOUNT

AH R3, =H'1'

STH 3, STKCOUNT

STKPUSH HHW, H

LH R3, STKCOUNT

SLA R3, 2

LA R2, THESTACK

LR R4, HHW

ST R4, 0 (3, 2)

LH R3, STKCOUNT

AH R3, =H'1'

STH 3, STKCOUNT

STKPUSH FFW

LH R3, STKCOUNT

SLA R3, 2

LA R2, THESTACK

L R4, FFW

ST R4, 0 (3, 2)

LH R3, STKCOUNT

AH R3, =H'1'

STH 3, STKCOUNT

STKPUSH FFW, A

LH R3, STKCOUNT

SLA R3, 2

LA R2, THESTACK

LA R4, FFW

ST R4, 0 (3, 2)

LH R3, STKCOUNT

AH R3, =H'1'

STH 3, STKCOUNT

STKPUSH FF, A

LH R3, STKCOUNT

SLA R3, 2

LA R2, THESTACK

LA R4, FFW

ST R4, 0 (3, 2)

LH R3, STKCOUNT

AH R3, =H'1'

STH 3, STKCOUNT

136 ********************************************
Revised Code for the Macros

The next few pages show the listing of the final forms of the macros, as actually coded and tested. These are followed by listings of the expanded macros.

```
002900 *
002910  MACRO
002911 &L1 STKINIT
002912 &L1 ST R3,STKSAV3
002913 SR R3,R3
002914 STH R3,STKCOUNT       CLEAR THE COUNT
002915 L R3,STKSAV3
002920  MEND
002930 *

003000 MACRO
003100 &L2 STKPUSH &ARG,&TYP
003110 &L2 ST R3,STKSAV3 SAVE REGISTER R3
003200 LH R3,STKCOUNT GET THE CURRENT SIZE
003210 CH R3,STKSIZE IS THE STACK FULL?
003220 BNL Z&SYSNDX YES, DO NOT PUSH
003230 ST R4,STKSAV4 OK, SAVE R2 AND R4
003240 ST R2,STKSAV2
003250 SLA R3,2 MULTIPLY BY FOUR
003260 LA R2,THESTACK ADDRESS OF STACK START
003270 L&TYP R4,&ARG LOAD R4 WITH VALUE
003280 ST R4,0(3,2) STORE INTO THE STACK
003290 LH R3,STKCOUNT
003300 AH R3,=H'1'
003310 STH 3,STKCOUNT
003320 L R4,STKSAV4
003330 L R2,STKSAV2
003336 Z&SYSNDX L R3,STKSAV3
003337  MEND
003338 *
003339 *
```
S/370 Assembler Language

Writing Macros

MACRO
003340 &L3 STKPOP &ARG,&TYP
003342 &L3 ST R3,STKSAV3
003343 LH R3,STKCOUNT GET THE STACK COUNT
003344 CH R3,=H'0' IS THE COUNT POSITIVE?
003345 BNH Z &SYSNDX NO, WE CANNOT POP
003346 SH R3,=H'1' SUBTRACT 1 WORD OFFSET
003347 STH R3,STKCOUNT STORE THE NEW SIZE
003348 SLA R3,2 BYTE OFFSET OF STACK TOP
003349 ST R3,STKSAV2 SAVE REGISTER R2
003350 ST R4,STKSAV4 SAVE REGISTER R4
003351 LA R2,THESTACK ADDRESS OF STACK BASE
003352 L R4,0(3,2) LOAD ITEM INTO R4
003353 AIF ('&TYP' EQ 'R').ISREG
003354 ST&TYP R4,&ARG
003355 AGO .CONT
003356 .ISREG LR &ARG,R4
003357 .CONT L R4,STKSAV4
003358 L R2,STKSAV2
003359 Z &SYSNDX L R3,STKSAV3
003360 MEND
003361 *

Revised Code for the Macro STKINIT
Here is an expansion of the newer definition of STKINIT, which allows the stack size to be specified.

STKINIT 128

138 STKINIT 128

STKINIT 128

00004A 5030 C05E 00064 139+ ST R3,STKSAV3
00004E 1B33 140+ SR R3,R3
000050 4030 C056 0005C 141+ L R3,STKSAV2
000054 5830 C05E 00064 142+ L R3,STKSAV3
000058 5830 C266 0026C 143+ B L0009
00005C 0000 144+ STKCOUNT DC H'0'
00005E 0080 145+ STKSZ DC H'128'
000060 00000000 146+ STKSAV2 DC F'0'
000064 00000000 147+ STKSAV3 DC F'0'
000068 00000000 148+ STKSAV4 DC F'0'
Revised Code for the Macro Expansions

128 * SOME MACRO INVOCATIONS
129 *
130
00004A 5030 C22E 00234 131+ ST R3,STKSAV3
00004E 1B33 132+ SR R3,R3
000050 4030 C226 0022C 133+ STH R3,STKCOUNT
000054 5830 C22E 00234 134+ L R3,STKSAV3
135 *

* Stack Push with a Register as an Argument

136 STKPUSH R7,R
000058 5030 C22E 00234 137+ ST R3,STKSAV3
00005C 4830 C226 0022C 138+ LH R3,STKCOUNT
000060 4930 C228 0022E 139+ CH R3,STKSIZE
000064 47B0 C0BC 00092 140+ BNL Z0010
000068 5040 C232 00238 141+ ST R4,STKSAV4
00006C 5020 C22A 00230 142+ ST R2,STKSAV2
000070 8B30 0002 00002 143+ SLA R3,2
000074 4120 C236 0023C 144+ LA R2,THESTACK
000078 1847 145+ LR R4,R7
00007A 5043 2000 00000 146+ ST R4,0(3,2)
00007E 4830 C226 0022C 147+ LH R3,STKCOUNT
000082 4A30 C5A2 005A8 148+ AH R3,=H'1'
000086 4030 C226 0022C 149+ STH 3,STKCOUNT
00008A 5840 C232 00238 150+ L R4,STKSAV4
00008E 5820 C22A 00230 151+ L R2,STKSAV2
000092 5830 C22E 00234 152+ Z0010 L R3,STKSAV3

* Stack Push with a Halfword as an Argument

153 STKPUSH HHW,H
000096 5030 C22E 00234 154+ ST R3,STKSAV3
00009A 4830 C226 0022C 155+ LH R3,STKCOUNT
00009E 4930 C228 0022E 156+ CH R3,STKSIZE
0000A2 48B0 C0CC 000D2 157+ BNL Z0011
0000A6 5040 C232 00238 158+ ST R4,STKSAV4
0000AA 5020 C22A 00230 159+ ST R2,STKSAV2
0000AE 8B30 0002 00002 160+ SLA R3,2
0000B2 4120 C236 0023C 161+ LA R2,THESTACK
0000B6 4840 C33A 00340 162+ LH R4,HHW
0000BA 5043 2000 00000 163+ ST R4,0(3,2)
0000BE 4830 C226 0022C 164+ LH R3,STKCOUNT
0000C2 4A30 C5A2 005A8 165+ AH R3,=H'1'
0000C6 4030 C226 0022C 166+ STH 3,STKCOUNT
0000CA 5840 C232 00238 167+ L R4,STKSAV4
0000CE 5820 C22A 00230 168+ L R2,STKSAV2
0000D2 5830 C22E 00234 169+ Z0011 L R3,STKSAV3
Stack Push with a Fullword as an Argument

```
0000D6  5030  C22E 00234  171+  ST  R3, STKSAV3
0000DA  4830  C226 0022C  172+  LH  R3, STKCNT
0000DE  4930  C228 0022E  173+  CH  R3, STKSIZE
0000E2  47B0  C10C 00112  174+  BNL  Z0012
0000E6  5040  C232 00238  175+  ST  R4, STKSAV4
0000EA  5020  C22A 00230  176+  ST  R2, STKSAV2
0000EE  8B30  0002 00002  177+  SLA  R3, 2
0000F2  4120  C36 0023C  178+  LA  R2, THESTACK
0000F6  5840  C336 0033C  179+  L  R4, FFW
0000FA  5043  2000 00000  180+  ST  R4, 0(3,2)
0000FE  4830  C226 0022C  181+  LH  R3, STKCNT
000102  4A30  C5A2 005A8  182+  AH  R3, =H'1'
000106  4030  C226 0022C  183+  STH  3, STKCNT
00010A  5840  C332 00238  184+  L  R4, STKSAV4
00010E  5820  C22A 00230  185+  L  R2, STKSAV2
000112  5830  C22E 00234  186+Z0012  L  R3, STKSAV3
```

Stack Pop with a Register as an Argument

```
000116  5030  C22E 00234  188+  ST  R3, STKSAV3
00011A  4830  C226 0022C  189+  LH  R3, STKCNT
00011E  4930  C228 0022E  190+  CH  R3, STKSIZE
000122  47B0  C14C 00152  191+  BNL  Z0013
000126  5040  C232 00238  192+  ST  R4, STKSAV4
00012A  5020  C22A 00230  193+  ST  R2, STKSAV2
00012E  8B30  0002 00002  194+  SLA  R3, 2
000132  4120  C36 0023C  195+  LA  R2, THESTACK
000136  4140  C336 0033C  196+  LA  R4, FFW
00013A  5043  2000 00000  197+  ST  R4, 0(3,2)
00013E  4830  C226 0022C  198+  LH  R3, STKCNT
000142  4A30  C5A2 005A8  199+  AH  R3, =H'1'
000146  4030  C226 0022C  200+  STH  3, STKCNT
00014A  5840  C332 00238  201+  L  R4, STKSAV4
00014E  5820  C22A 00230  202+  L  R2, STKSAV2
000152  5830  C22E 00234  203+Z0013  L  R3, STKSAV3
```

Stack Pop with a Register as an Argument

```
000156  5030  C22E 00234  206+  ST  R3, STKSAV3
00015A  4830  C226 0022C  207+  LH  R3, STKCNT
00015E  4930  C5A4 005A0  208+  CH  R3, =H'0'
000162  47B0  C186 0018c  209+  BNL  Z0014
000166  4B30  C5A2 005A8  210+  SH  R3, =H'1'
00016A  4030  C226 0022C  211+  STH  R3, STKCNT
00016E  8B30  0002 00002  212+  SLA  R3, 2
000172  5020  C22A 00230  213+  ST  R2, STKSAV2
000176  5040  C232 00238  214+  ST  R4, STKSAV4
00017A  4120  C36 0023C  215+  LA  R2, THESTACK
00017E  5843  2000 00000  216+  L  R4, 0(3,2)
000182  1884  217+  LR  R8, R4
000184  5840  C232 00238  218+  L  R4, STKSAV4
000188  5820  C22A 00230  219+  L  R2, STKSAV2
00018C  5830  C22E 00234  220+Z0014  L  R3, STKSAV3
```
* Stack Pop with a Fullword as an Argument

221

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>000190</td>
<td>000194</td>
<td>000198</td>
</tr>
<tr>
<td>C22E</td>
<td>C226</td>
<td>C25A4</td>
</tr>
<tr>
<td>STKPOP FFW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R3, STKSAV3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R3, STKSAV3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R3, STKSAV3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R3, STKSAV3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R3, STKSAV3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R3, STKSAV3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R3, STKSAV3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001CC</td>
<td>0001D0</td>
<td>0001D4</td>
</tr>
<tr>
<td>5030</td>
<td>4830</td>
<td>4930</td>
</tr>
<tr>
<td>STKPOP HHW, H</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R3, STKSAV3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R3, STKSAV3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R3, STKSAV3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R3, STKSAV3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R3, STKSAV3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R3, STKSAV3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R3, STKSAV3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R3, STKSAV3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R3, STKSAV3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R3, STKSAV3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R3, STKSAV3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R3, STKSAV3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R3, STKSAV3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Stack Pop with a Halfword as an Argument

237

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>000190</td>
<td>000194</td>
<td>000198</td>
</tr>
<tr>
<td>C22E</td>
<td>C226</td>
<td>C25A4</td>
</tr>
<tr>
<td>STKPOP HHW, H</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R3, STKSAV3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R3, STKSAV3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R3, STKSAV3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R3, STKSAV3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R3, STKSAV3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R3, STKSAV3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R3, STKSAV3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R3, STKSAV3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R3, STKSAV3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R3, STKSAV3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R3, STKSAV3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R3, STKSAV3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST R3, STKSAV3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Copyright © by Edward L. Bosworth, Ph.D.

Page 364 Chapter 18 Revised August 3, 2009