Z-100 LifeLine
Since 1989

April 2022

#WEB

This article was first published in issue #72, October 2000

Calling .ASM Routines From ZBASIC

by Steven W. Vagts
Editor, "Z-100 LifeLine"

Calling .ASM Routines From ZBASIC

ZBASIC, like most BASICs and other high-level languages, is able to call assembly language routines. But I have always shied away from doing any of this, because just reading the procedures made my head swim.

I had resumed work on my Two-Terminal Program, however, and quickly found myself up against some limitations of ZBASIC. I decided I needed to shake the cobwebs from my head and look into creating a separate routine for displaying information under ZBASIC.

Dusting off the books and manuals, I read the very terse section in Appendix E of the ZBASIC Manual, and quickly reaffirmed my belief that this was not to be easy.

When we are done here, though, I hope you will find the following useful and easier to understand. Add this article to your ZBASIC Manual. For those using ZBASIC without the manual, I will start from scratch, so do not fear. The manual is not required.

If you are ready, here we go...

Assembly Language Subroutines

All versions of Zenith BASIC have provisions for interfacing with assembly language subroutines via either the USR function or the CALL statement.

The USR function allows assembly language subroutines to be called in the same way BASIC Intrinsic Functions are called. However, it is suggested that the old style user-call USR(n) not be used.

The CALL statement is the recommended way of interfacing 8086 machine language programs with BASIC. It is compatible with more languages than is the USR function call, it produces more readable source code, and it can pass multiple arguments.

The USR function procedures are outside the scope of this article. Please refer to the ZBASIC Manual for additional information on these procedures.

Memory Allocation

Memory space must be set aside for an assembly language subroutine before it can be loaded.

When invoking ZBASIC, if an assembly language subroutine will be used, you must enter the highest memory location minus the amount of memory needed for the subroutine(s) with the /M: switch.

For our programs, this works:

ZBASIC /M:32768

In addition to the BASIC interpreter code area, ZBASIC uses up to 64K of memory beginning at its Data Segment (DS).

If, when an assembly language subroutine is called, more stack space is needed, BASIC's stack can be saved and a new stack set up for use by the assembly language subroutine. BASIC's stack must be restored, however, before returning from the subroutine.

The assembly language subroutine may be loaded into memory two ways:

1. By means of the operating system. Here the object files (.OBJ) for the subroutine and calling program are actually linked into one executable (.EXE) file.

If the user has the Zenith Utility Software Package, the routines may be assembled with the MACRO-86 assembler and linked using the MS-LINK Linker, but not loaded. To load the program file, the user should observe these guidelines:

- The subroutines must not contain any long references.
- Skip over the first 512 bytes of the MS-LINK output file, then read in the rest of the file.

2. By the BASIC POKE statement, where the calling program loads the subroutine, in binary form, into memory for use. The examples that follow will use this method.

As mentioned earlier, the CALL statement is the recommended way of interfacing 8086 machine language programs with BASIC.

CALL Statement Format:

CALL <variable name> [(<argument list>)]

Where:
<**variable name**> contains the segment offset that is the starting point in memory of the subroutine being CALLed.

<**argument list**> contains the variables or constants, separated by commas, that are to be passed to the routine.

The CALL statement conforms to the INTEL PL/M-86 calling conventions outlined in Chapter 9 of the INTEL PL/M-86 Compiler Operator's Manual. BASIC follows the rules described for the MEDIUM case (summarized in the following discussion).

Invoking the CALL statement causes the following to occur:

1. For each parameter in the argument list, the two-byte offset of the parameter's location within the Data Segment (DS) is PUSHed onto the Stack.

2. BASIC's return address Code Segment (CS), and offset (IP) are PUSHed onto the Stack.

3. Control is transferred to the user's routine via an 8086 FAR CALL to the segment address given in the last DEF SEG statement and the offset given in <**variable name**>.

Here, the ZBASIC Manual includes two diagrams to show the state of the Stack at the time of the CALL statement and the Stack's condition during execution of the called routine. But I found this most confusing.

Instead, the following contains excerpts from The Waite Group's Microsoft Macro Assembler Bible, by Nabajyoti Barkakati, and published by Howard W. Sams & Company in 1989.

Retrieving and Interpreting Arguments

When the assembly language procedure is called, you find the arguments followed by the return address on the 8086 Stack as you go from higher to lower addresses (See the diagram of Figure 1).

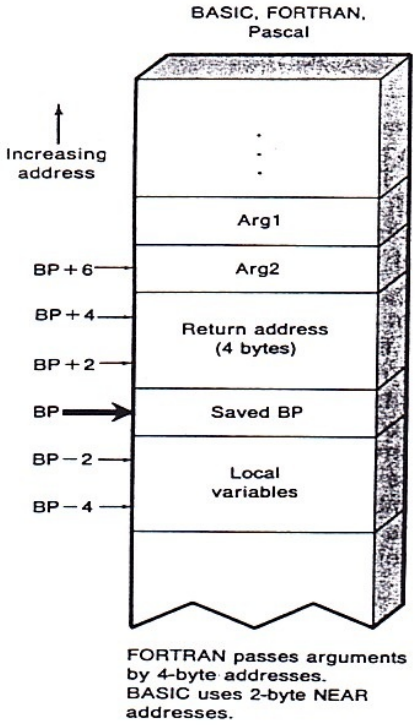

Figure 1.
BASIC Stack Frame

The first thing the assembly language procedure should do is establish the BP register as a pointer to the stack frame containing the return address and the arguments.

This can be done with the following code:

```
PUSH   BP
MOV    BP,SP
```

BP is then used to access the arguments from the Stack, without disturbing the stack pointer.

The contents of the stack frame at this point are determined by the convention used by a language to pass arguments. For BASIC, the stack frame is shown in Figure 1, above. The conventions for BASIC are below:

~~~~~~

Argument-Passing Convention for BASIC

For BASIC, the arguments are passed by reference, using **near pointers**, and are pushed on the Stack in the order of their occurrence in the CALL. Thus, the last argument is right above the return address.

The return address is 4 bytes because all procedures are accessed by **FAR CALLs.**
~~~~~~

The procedure must remove the arguments using a RET n instruction, where n is the number of bytes occupied by the arguments on the Stack.

~~~~~~

After setting up BP, space can be reserved on the stack for local variables. This is done by subtracting from the stack pointer, SP, the number of bytes needed for local storage (variables needed only within the procedure).

As shown in Figure 1, the arguments passed to the procedure are at positive offsets from BP and the local variables are accessed using negative offsets. Thus, [BP-2] gets you the first word-sized local variable.

Before the procedure does any processing, care must be taken to preserve registers to be used in the procedure. Microsoft's high level languages require that the registers SI, DI, SS, DS, and BP be preserved. DS, SS, and BP already have assigned tasks, but SI and DI must be saved. The registers can be saved by PUSHing them on the Stack and restored by POPping them in the reverse order before returning from the procedure.

To access arguments and local variables from the Stack, consult the stack frame given in Figure 1, or draw a sketch of the stack frame and identify the location of each argument by counting the bytes occupied by each argument.

Remember that anything PUSHed on the Stack uses an even number of bytes, but the Stack, like all of 8086 memory, is addressed by byte.

Finally, the ZBASIC and GW-BASIC Manuals have a section on string arguments:

- If the argument is a string, the parameter's offset points to three bytes called the "String Descriptor". Byte zero of the string descriptor contains the length of the string (0 to 255). Bytes one and two, respectively, are the lower and upper eight-bits of the string starting address in string space.

- If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy your program this way! To avoid unpredictable results, add +" " to the string literal in the program. (I could not confirm this during my experimentation.)

EXAMPLE:

```
20 A$ = "BASIC"+""
```

This will force the string literal to be copied into string space. Now the string may be modified without affecting the program.

Strings may be altered by user routines, but the length **MUST NOT** be changed. ZBASIC and GW-BASIC cannot correctly manipulate strings if their lengths are modified by external routines.

Returning Values

When arguments are passed, the procedure may directly alter arguments to reflect the processing done in it. In many other cases, the result is returned from the procedure.

Most Microsoft high-level languages expect values to be returned in the AX and DX registers. One byte results are returned in AL, while word-sized results are returned in AX. Any pointers are returned with the segment address in DX and the offset in AX. Near pointers fit in AX only.

Microsoft BASIC handles large-sized return values differently. The calling program reserves space for the return value in the Stack before calling the procedure. The 2-byte offset (from the Stack segment register SS) of this space is PUSHed on the Stack just before calling the procedure. This means that in BASIC the location [BP+6] in the stack frame holds the near pointer to the location where the calling program expects the return value (see Figure 2).

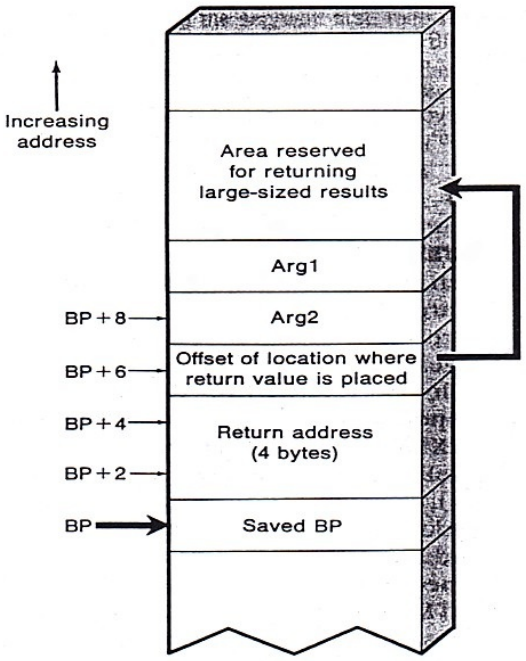

Figure 2.
Stack Frame When Returning
Large-sized Values in Microsoft BASIC

In the assembly language procedure, the return value should be copied to the location whose address is in [BP+6] and then [BP+6] be copied to AX and the SS register copied to DX. This is necessary because the calling BASIC program expects the address of the return value to be in DX:AX.

HOWEVER, according to the ZBASIC and GW-BASIC Manuals, ZBASIC and GW-BASIC have what appear to be a simpler approach. **ALL values are returned to ZBASIC** by including in the argument list, the variable name(s) which will receive the result.
~~~~~~

This argument is the last PUSHed, so it still resides at [BP+6]. I have NOT checked this difference with other BASIC languages.

Cleaning up and Exiting

A procedure is exited by using a RET instruction, but before returning, certain cleanup operations must be performed. Registers that were saved by PUSHing must be restored with POP instructions. Remember that the POPs must be in reverse order of the PUSHes.

Next, the process of setting up the BP as the stack frame pointer must be reversed:

```
MOV   SP,BP    ; Which also abandons
               ; the local variables
POP   BP
```

Finally, the syntax of the return instruction, RET, also depends upon the language. For procedures to be called from Microsoft BASIC, you must clean up the stack (remove the arguments PUSHed by the calling program) by using the return instruction of the form **RET n**, were **n** is the total number of bytes to be removed from the Stack.

Since Stack PUSHes occur in word-sized values, **n** must be a multiple of 2. Determine the value of **n** from your knowledge of the arguments that your procedure requires.

Programming

Let us walk through the steps for writing an assembly module for BASIC.

As mentioned earlier, there are two methods. The first requires compiling the BASIC program because the object (.OBJ) files are combined. The second involves loading our new procedure's binary (.BIN) file while running our BASIC program.

Say we need to compute a certain 16-bit quantity called STEP many times. We want this done as quickly as possible, so we would like an assembly language procedure FASTSTEP.

In the BASIC program we can insert the statement CALL FASTSTEP(STEP), where the variable STEP is the 16-bit quantity to be computed by the new procedure.

Now, we need to write the assembly language procedure FASTSTEP. FASTSTEP.ASM would be the source file containing this procedure.

We assemble this file to get the object module FASTSTEP.OBJ.

If we use the first method, we use the BASIC compiler to also generate an object module for the BASIC program that calls FASTSTEP. Then we use the linker (LINK) to build an executable by linking the two object modules.

If we use the second method, we use LINK to generate an executable procedure (.EXE file), then use EXE2BIN to generate a binary (.BIN) file to be loaded and called by the BASIC program running under the interpreter only. This method is used by the new batch file, ASM2BIN, presented later.

So, how do we write FASTSTEP.ASM?

As we have seen earlier, BASIC calls the procedure by PUSHing the argument's address on the Stack and uses a FAR CALL to invoke FASTSTEP. Because BASIC passes the argument's address (complete segment:offset address) on the Stack, it expects the FASTSTEP procedure to alter the contents of the variable STEP directly. Thus, in this case, you need not return any values in any register.

Since BASIC uses FAR CALLs, the procedure is declared with a PROC FAR directive.

The first task is to set up a fixed reference point to address the argument that is on the stack. You cannot use the stack pointer (SP) for this purpose because SP is altered if you use PUSH or POP instructions in the procedure.

A better way is to use the BP register. First PUSH it on the Stack to save its old value. Then copy the current SP into BP. This results in BP pointing to the word just below the return address, establishing BP as the pointer to the base of a stack frame (the contents of the Stack between the saved BP and the PUSHed arguments).

Once we know the layout of the stack frame, we can access the argument, in this case at location BP+6.

Here's what our outline for FASTSTEP may look like:

```
FASTSTEP  PROC    FAR       ; BASIC uses FAR CALLS
          PUSH    BP        ; Save BP
          MOV     BP,SP     ; Set up BP as pointer
                            ; to stack frame
; --------        Start of procedure's code
          PUSH    DI        ; Save registers, if used
          PUSH    BX
          LES     BX,[BP+6] ; Since seg:offset addr
                            ; of argument is at BP+6,
                            ; this will get address
                            ; into ES:DI

; Compute new value... and save it back
; Suppose new value is in AX
          MOV     ES:[BX],AX

          POP     DI        ; Restore saved registers
          POP     BX        ; in reverse order

; --------        End of procedure's code
          MOV     SP,BP     ; So SP points to saved BP
          POP     BP        ; Now restore BP and
          RET     n         ; return
FASTSTEP  ENDP
```

At this point, let us cover a few coding rules and recap a few thoughts.

- For naming the new assembly language procedure, BASIC uses up to 40 uppercase alphanumeric characters.

- BASIC invokes all external procedures with FAR CALL instructions. Thus, the assembly language procedure has to be declared as FAR.

- BASIC passes all arguments to the procedure by NEAR references. Because BASIC uses FAR CALLs, this means that in the stack frame, the first argument's offset address is at BP+6. If there are two arguments, the first is PUSHed on the Stack, then the second, so the offset of the first becomes BP+8, and the second is BP+6. And so on.

- After BP is established as a pointer to the stack frame, subtract from SP the number of words of local storage space needed for local variables. In the procedure, refer to these variables using negative displacements from BP. Thus, the first word of local storage is [BP-2], the second one is [BP-4], and so on.

Here is how the code might look:

```
aProc     PROC    FAR
          PUSH    BP
          MOV     BP,SP     ; Sets up the stack frame
          SUB     SP,8      ; Sets up 8 bytes for
                            ; local variables
; Access local variables as [BP-2], [BP-4], ...
; For example, to copy 2 into the first 16-bit local
; variable, use:
          MOV     WORD PTR [BP-2],2
```

- Values are returned to ZBASIC and GW-BASIC by including, in the argument list, the variable name(s) which will receive the result. This differs from other BASICs, so be careful.

- Use a RET instruction to return from the procedure. Before returning, however, you have to restore the stack to its state before the procedure call. For BASIC, you must provide, as an operand to RET, the number of bytes occupied by the arguments to the procedure. Thus, if your procedure requires two 16-bit integers (4 bytes of space), the return instruction is:

```
          RET     4    ; Return and remove 4 bytes
                       ; occupied by arguments
```

This returns from the procedure and adds 4 to the stack pointer SP.

Note: The operand in **RET n** is always a multiple of 2 because stack space is used a word at a time.

- In ZBASIC or GW-BASIC, if the argument is a string, the parameter's offset points to three bytes called the "String Descriptor". Byte zero of the string descriptor contains the length of the string (0 to 255). Bytes one and two, respectively, are the lower and upper eight-bits of the string starting address in string space.

Examples

Before we get into some programming examples, here is a helpful utility.

The assembly language routine must be **either** LINKed with the BASIC protgram, combining their .OBJ files into one executable file, or placed in BINARY form so it can be loaded into the BASIC program and run within the interpreter. This conversion to binary (.BIN) form is just an additional step using the EXE2BIN utility.

For troubleshooting purposes, I used the binary file and loaded it into the ZBASIC program, but each time I made a change to the assembled program, I had to do all the assembly steps again. This got old fast.

For those who remember my assembly batch files, ASM.BAT and ASMCOM.BAT, here is another fashioned in the same manner - **ASM2BIN.BAT**.

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
REM **** ASM2BIN.BAT - S.W. VAGTS 12/00 ****
REM  Assembling an ASM calling routine for BASIC
ECHO OFF
REM First check that a file is specified.
IF NOT "%1" == "" GOTO ASSEMBLE
:USAGE
ECHO ! USAGE: ASSEMBLE filename
ECHO !
ECHO ! where filename is the program you are testing.
ECHO !       (do not include extension, .ASM assumed)
ECHO !
GOTO END
REM The above routine runs if no filename was
    specified.
REM The following routine runs the assembly process.
:ASSEMBLE
IF EXIST %1.ASM GOTO FILEFOUND
ECHO ! ASSEMBLE: File %1.ASM not found.  Exiting...
GOTO END
:FILEFOUND
ECHO ! Now starting MASM...
MASM %1.ASM,%1.OBJ;
IF EXIST %1.OBJ GOTO FILEFND2
ECHO ! LINK: File %1.OBJ not found.  Exiting...
GOTO END
:FILEFND2
ECHO ! Now starting LINK...
LINK %1.OBJ,%1.EXE;
IF EXIST %1.EXE GOTO FILEFND3
ECHO ! File %1.EXE not completed.  Exiting...
GOTO END
:FILEFND3
DEL %1.OBJ
ECHO ! File %1.EXE completed successfully.
ECHO ! Now starting EXE2BIN...
EXE2BIN %1.EXE;
IF EXIST %1.BIN GOTO FILEFND4
ECHO ! File %1.BIN not completed.  Exiting...
GOTO END
:FILEFND4
DEL %1.EXE
ECHO ! File %1.BIN completed successfully.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

By simply typing ASM2BIN MULT, it assembles, links, and converts to binary the MULT.ASM routine with no additional commands!

Now, let us get down to working examples. I started first with the ZBASIC program from page E.8 of the ZBASIC Manual, but modified it after finally getting a working version.

MULT2.BAS uses an assembly routine that I called MULT2.ASM.
The ZBASIC program, **MULT2.BAS**, and the assembly language routine, **MULT2.ASM**:

```
10   REM ************************************************************
20   REM ***         MULT2.BAS        --  Steven W. Vagts   12/00        ***
30   REM ***         211 Sean Way, Hendersonville, NC  28792             ***
40   REM *** Note:   Program to test MULT2.ASM for multiplying 2 numbers ***
50   REM *** Invoke: ZBASIC /M:32768 to use the first 32K memory         ***
60   REM ************************************************************
70   DEFINT A-Z
100  DEF SEG = &H2F00
110  REM Set base of CALL to 2F00:0000 (for a 192K Z-100)
120  REM          or &H6F00 for 448k; &HBF00 for 768k RAM
1000 REM Load the BINary Assembly Language Routine
1010 OPEN "R",1,"MULT2.BIN",2              ' Open Binary File
1020 FIELD #1, 2 AS A$                     ' Set 2-byte field
1030 FOR I=&H0 TO (LOF(1)+1) STEP 2        ' For/next to read every byte
1040 GET #1,I/2+1                          ' Get next pair of bytes
1050 Q=CVI(A$)                             ' Convert to 16-bit integer
1060 M=Q MOD 256                           ' Split into 8 high and
1070 L=INT(Q/256)                          '   8 low-bits
1080 POKE I,M AND &HFF                     ' Poke data into memory
1090 POKE I+1,L AND &HFF                   '   locations
1100 NEXT I                                ' Get next pair
1110 CLOSE #1                              ' Close file
2000 INPUT "Input 2 numbers (1-99) or zero to end. ",A$,B$
2010 IF A$="0" OR B$="0" GOTO 9999         ' Test if done
2020 X=FIX(VAL(A$)):Y=FIX(VAL(B$))         ' Make integers, if not
2030 IF NOT (X>0 AND X<100) THEN 2000      ' Test if integer
2040 IF NOT (Y>0 AND Y<100) THEN 2000      ' Test if integer
2050 MULT = &H0                            ' Set address of program
2060 CALL MULT2(X,Y,Z)                     ' Call routine
2070 PRINT X,Y,Z                           ' Print result
2080 GOTO 2000
9999 END                                   ' Done
```

TITLE MULT2.ASM Multiplication Subroutine for ZBASIC by S.W. Vagts 12/00

```
; ZBASIC callable routine to multiply two simple integer numbers.
; Modified sample program from the ZBASIC Manual, Appendix E.

CODE    SEGMENT                    ; Start CODE segment
        ASSUME      CS:CODE

MULT2   PROC  FAR                  ; BASIC uses FAR CALL instruction
        PUSH  BP                   ; Save BP pointer
        MOV   BP,SP                ; Set BP as pointer to Stack frame
; Start procedure's code:
        PUSH  AX                   ; Save any registers used
        PUSH  BX
        PUSH  SI
; Note: BASIC passes arguments to the stack as a 2-byte offset of the
; parameter's location within the data segment (DS). For example, if
; last argument is P3, then P1 is PUSHed first, then P2, then P3, so:
; P1 ends up at BP+10, P2 at BP+8, and P3 at BP+6.
        MOV   SI,[BP+10]           ; SI=pointer to parameter1
        MOV   BX,WORD PTR[SI]      ; AX =integer value
        MOV   SI,[BP+8]            ; SI=pointer to parameter2
        MOV   AX,WORD PTR[SI]      ; AX =integer value
        MOV   SI,[BP+6]            ; SI=pointer to parameter3
        MUL   BX                   ; AX = AX * BX
        MOV   WORD PTR[SI],AX      ; Save AX in parameter3 (BP+6)
        POP   SI                   ; Recall original registers
        POP   BX
        POP   AX
; End of procedure's code
        MOV   SP,BP                ; So that SP points to saved BP
        POP   BP                   ; Recall BP pointer
        RET   6                    ; Return and remove 2 bytes/argument
MULT2   ENDP                       ; from stack by the CALL procedure.

CODE    ENDS
        END   MULT2
```

Save this last program as MULT2.ASM, then convert to a BINary file with:

ASM2BIN MULT2{CR}

Where {CR} is simply pressing {RETURN}.

To try out the program pair, type:

ZBASIC MULT2 /M:32768{CR}

ZBASIC loads and runs MULT2.BAS. When the program gets to line 1010, it loads the MULT2.BIN file into memory.

At line 2060, it CALLs the new subroutine.

I modified the original program, MULT.BAS, to query for two numbers, test that they were valid, and then perform the multiplication:

The next pair of programs prints a string:

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

10  REM ************************************************************************
20  REM ***         PSTRNG.BAS       --  Steven W. Vagts   12/00        ***
30  REM ***         211 Sean Way, Hendersonville, NC  28792             ***
40  REM *** Note:   This program tests PCHR.ASM to print a string.      ***
50  REM *** Invoke: ZBASIC /M:32768 to use the first 32K memory         ***
60  REM ************************************************************************
70  DEFINT A-Z
100 DEF SEG = &H2F00
110 REM  Set base of CALL to 2F00:0000 (for a 192K Z-100)
120 REM           or &H6F00 for 448k; &HBF00 for 768k Z-100
200 REM Define some ESCape codes:
210 E$=CHR$(27): RED$=E$+"m72": GRN$=E$+"m04": BLU$=E$+"m71": BLK$=E$+"m70"
220 Y$=ESC$+"Y": HOME$=Y$+CHR$(32)+CHR$(32) ' Set screen "home"
230 GC$=E$+"F": NC$=E$+"G"                   ' Set Graphics/Normal characters
240 CRLF$=CHR$(10)+CHR$(13)                  ' Set Carriage Return and Line Feed
250 CLR$=E$+"E": E25$=E$+"x1": D25$=E$+"y1" ' Clr Scrn/enable/disable 25th line.
260 DEF FNPOSIT$(L1,L2)=CHR$(27)+"Y"+CHR$(31+L1)+CHR$(31+L2)
1000 REM Load Assembly Language Routines
1010 OPEN "R",1,"PCHR.BIN",2                 ' Open Binary File
1020 FIELD #1, 2 AS A$                       ' Set 2-byte field
1030 FOR I=&H0 TO (LOF(1)+1) STEP 2          ' For/next to read every byte
1040 GET #1,I/2+1                            ' Get next pair of bytes
1050 Q=CVI(A$)                               ' Convert to 16-bit integer
1060 M=Q MOD 256                             ' Split into 8 high and
1070 L=INT(Q/256)                            '   8 low-bits
1080 POKE I,M AND &HFF                       ' Poke data into memory
1090 POKE I+1,L AND &HFF                     '   locations
1100 NEXT I                                  ' Get next pair
1110 CLOSE #1                                ' Close file
2000 REM Construct screen message:
2010 M1$=FNPOSIT$(12,12)+"*** "+RED$+"MERRY CHRISTMAS"+BLK$+" ***"
2020 M2$=CHR$(10)+"*** and "+GRN$+"HAPPY NEW YEAR"+BLK$+" ***"
2030 M3$=CHR$(10)+BLU$+"2001"+BLK$
2040 A$=M1$+M2$+M3$
3000 PCHR=0
3010 CALL PCHR(A$)                           ' Display string.
3020 A$=INKEY$: IF A$="" GOTO 3020
9999 END
```

```
TITLE  PCHR.ASM  Print Char Subroutine for ZBASIC by S.W. Vagts 12/16/00
; ZBASIC callable routine to print a character string to the screen.

CODE    SEGMENT                     ; Start CODE segment
        ASSUME    CS:CODE

PCHR    PROC  FAR                   ; BASIC uses FAR CALL instruction
        PUSH  BP                    ; Save BP pointer
        MOV   BP,SP                 ; Set BP as pointer to Stack frame
; Start procedure's code:
        PUSH  AX               ; Save any registers used
        PUSH  CX
        PUSH  DX
        PUSH  SI
        PUSH  DI
; Note: BASIC passes arguments to the stack as a 2-byte offset of the
; parameter's location within the data segment (DS). For example, if
; last argument is P3, then P1 is PUSHed first, then P2, then P3, so:
```

```
; P1 is at BP+10, P2 is at BP+8, and P3 is at BP+6.
; For our single argument, A$ is saved at the offset BP+6.
; But, according to Appx E of the ZBASIC Manual, if the argument is a string,
; the parameter's offset points to 3 bytes, called the "String Descriptor".
; Byte 0 of the string descriptor contains the length of the string (0-255).
; Bytes 1 & 2 are the lower & upper 8-bits of the string starting address.
        MOV   SI,[BP+6]            ; BP+6 has address to String Descriptor
        MOV   CL,BYTE PTR[SI]      ; CL = number of bytes in string
        MOV   DI,[SI+1]            ; Set address of String Descriptor in DI
PC1:    MOV   DL,BYTE PTR[DI]      ; Get byte from string
        MOV   AH,06h               ; Display char on screen (can also be 02h)
        INT   21h                  ; Call DOS
        INC   DI                   ; Move to next address
        DEC   CL                   ; Decrement count
        JNZ   PC1                  ; Not done, do again
        POP   DI                   ; Recall original registers
        POP   SI
        POP   DX
        POP   CX
        POP   AX
; End of procedure's code
        MOV   SP,BP                ; So that SP points to saved BP
        POP   BP                   ; Recall BP pointer
        RET   2                    ; Return and remove 2 bytes from stack
PCHR    ENDP                       ; by the CALL argument.
CODE    ENDS
        END   PCHR
```

A few last thoughts:

- Due to space constraints, I deleted the blank lines between ZBASIC line blocks. I like to add the blank lines because I think they improve readability, and ZBASIC doesn't care.

- The DEF FN statement of line 260 might catch you by surprise. It took me quite a while to get it to work. Here it is again:

260 DEF FNPOSIT$(L1,L2)=CHR$(27)+"Y" +CHR$(31+L1)+CHR$(31+L2)

We need it to position our text anywhere on the screen, in a manner similar to the LOCATE x,y command. I tried to shorten the CHR$(27)+ "Y" to Y$ earlier, but for some reason, it would not work. The function is then used in line 2010 to place our message.

There is a problem, however. Without this function at the start of our string, for some reason, PCHR **ALWAYS** places the string on line 25! I have not been able to figure out what causes this. If you have some ideas, please give me a call.

I have learned a lot from this little project and hope you will give the routines presented here a try. Enjoy...

Note: This project was completed 22 years ago. I may have inadvertently created an error during this update. Please let me know if you find or have a problem. Thanks.

If you have any questions or comments, please email me at:
z100lifeline@swvagts.com

Cheers,

Steven W. Vagts