At this point we should be able to get in and out of PASM and do some math and create and target specific array cells.

We are now going to revisit those objects and create subroutines with each one.

Let's start with the counting program that counts up from zero.

```
CON
_clkmode = xtal1 + pll16x
'_xinfreq = 6_250_000   'MY BOARD AT 100MHZ
_xinfreq = 5_000_000  'QUICKSTART 80 MHZ

var

   long count

obj

pst:"parallax serial terminal"

pub main

pst.start(115000)
    waitcnt(clkfreq*5 +cnt)'hold two sed to open the
    'serial terminal and enable it
cognew(@asm,@count)

     repeat
      pst.dec(count~)'post clear p 157
      pst.newline
      waitcnt(clkfreq +cnt)
```

Figure 1

```
dat

asm      org

          mov addr, par
 loop     add value,#1   'counting variable
 wait     rdlong prev, addr  wz 'what is in par??
          if_nz   jmp #wait   'if the value in
          'par is zero continue to next command
          'if the value in par "addr" has not been cleared
          'meaning the value that was put in "value" from
          ' addr which has the address of par "parameter"
          '
          wrlong value, addr
          'now write the value to the addr which has been assigned
          'the same address as par and where the address of count in
          'memory where the spin program can read it then jump back
          'to the top of the loop and continue after the variable
          ' called count has been cleard to zero
          jmp #loop


addr long 0
value long 0
prev  long 0
```

Figure 2

Now we are going to add three lines of code, the code definitions are as follows as seen on lines 44,45 and 61 on the next listing:

CALL

Instruction: Jump to address with intention to return to next instruction.

CALL #*Symbol*

Result: PC + 1 is written to the s-field of the register indicated by the d-field.

- *Symbol* (s-field) is a 9-bit literal whose value is the address to jump to. This field must contain a **DAT** symbol specified as a literal (#symbol) and the corresponding code should eventually execute a **RET** instruction labeled with the same symbol plus a suffix of "_ret" (*Symbol*_ret RET).

Explanation

CALL records the address of the next instruction (PC + 1) then jumps to *Symbol*. The routine at *Symbol* should eventually execute a **RET** instruction to return to the recorded address (PC+1; the instruction following the **CALL**). For the **CALL** to compile and run properly, the *Symbol* routine's **RET** instruction must be labeled in the form *Symbol* with "_ret" appended to it. The reason for this is explained below.

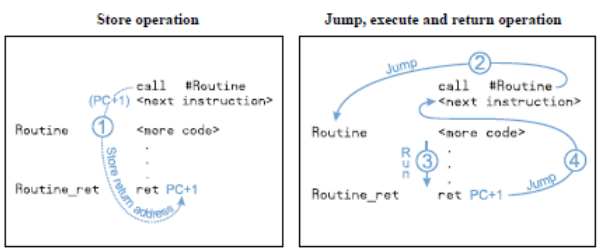
Propeller Assembly does not use a call stack, so the return address must be stored in a different manner. At compile time the assembler locates the destination routine as well as its **RET** instruction (labeled *Symbol* and *Symbol*_ret, respectively) and encodes those addresses into the **CALL** instruction's s-field and d-field. This provides the **CALL** instruction with the knowledge of both where it's going to jump to and exactly where it will return from.

At run time the first thing the **CALL** instruction does is store the return address (PC+1) into the location where it will return from; the "*Symbol*_ret RET" instruction location. The **RET**

instruction is really just a **JMP** instruction without a hard-coded destination address, and this run-time action provides it with the "return" address to jump back to. After storing the return address, **CALL** jumps to the destination address; *Symbol*.

The diagram below uses a short program example to demonstrate the **CALL** instruction's run-time behavior; the store operation (left) and the jump-execute-return operation (right).

Figure 3-1: Run-time CALL Procedure



In this example, the following occurs when the **CALL** instruction is reached at run time:

① The cog stores the return address (PC+1; that of `<next instruction>`) into the source (s-field) of the register at `Routine_ret` (see left image).

② The cog jumps to `Routine` (see right image).

③ `Routine`'s instructions are executed, eventually leading to the `Routine_ret` line.

④ Since the `Routine_ret` location contains a **RET** instruction with an updated source (s-field), which is the return address written by step 1, it returns, or jumps, back to the `<next instruction>` line.

CALL – Assembly Language Reference

This nature of the **CALL** instruction dictates the following:

- The referenced routine must have only one **RET** instruction associated with it. If a routine needs more than one exit point, make one of those exit points the **RET** instruction and make all other exit points branch (i.e., **JMP**) to that **RET** instruction.
- The referenced routine can not be recursive. Making a nested call to the routine will overwrite the return address of the previous call.

CALL is really a subset of the **JMPRET** instruction; in fact, it is the same opcode as **JMPRET** but with the i-field set (since **CALL** uses an immediate value only) and the d-field set by the assembler to the address of the label named *Symbol*_ret.

The return address (PC + 1) is written to the source (s-field) of the *Symbol*_ret register unless the **NR** effect is specified. Of course, specifying **NR** is not recommended for the **CALL** instruction since that turns it into a **JMP** or **RET** instruction.

RET

Instruction: Return to previously recorded address.

RET

JMP

Instruction: Jump to address.

JMP ⟨#⟩ *Address*

- *Address* (s-field) is the register or a 9-bit literal whose value is the address to jump to.

```
CON
_clkmode = xtal1 + pll16x
'_xinfreq = 6_250_000   'MY BOARD AT 100MHZ
_xinfreq = 5_000_000  'QUICKSTART 80 MHZ

var

   long count

obj

pst:"parallax serial terminal"

pub main

pst.start(115000)
    waitcnt(clkfreq*5 +cnt)'hold two sed to open the
     'serial terminal and enable it
cogneu(@asm,@count)

     repeat
      pst.dec(count~)'post clear p 157
      pst.newline
      waitcnt(clkfreq +cnt)
```

Figure 3

```
dat
        {{First subroutine. we are going to add three lines. First line
         to add is call #wait, this tells the program to go and find a set of code named
         "wait". at the bottom of the wait subroutine the following is added
          " wait_ret   ret ", this signals the end of the subroutine and to jump
          back to the next line of code after the "call".
          jmp #loop is added as the next line of code to execute which sends the
          code back to execute an "add" directive.}}
asm      org

          mov addr, par
 loop     add value,#1   'counting variable
```

Figure 4

```
          call #wait   '<<<ADD
          jmp #loop    '<<<ADD
 wait    rdlong prev, addr  wz 'what is in par??
         if_nz   jmp #wait  'if the value in
         'par is zero continue to next command
         'if the value in par "addr" has not been cleared
         'meaning the value that was put in "value" from
         ' addr which has the address of par "parameter"
         '
         wrlong value, addr
         'now write the value to the addr which has been assigned
         'the same address as par and where the address of count in
         'memory where the spin program can read it then jump back
         'to the top of the loop and continue after the variable
         ' called count has been cleard to zero
        '  jmp #loop

wait_ret   ret  '<<<ADD


addr long 0
value long 0
prev  long 0
```

Figure 5

Adding line 44 will call the subroutine named "wait". The routine will execute the code that is listed there. Upon completion of the code routine the "ret" command will send the code back to the next line of code after the "call" in this case it is a "jmp" meaning a jump to the address listed in the jmp command, which in this case is "loop" which is where the "add" command will add 1 to the value. You should see this:

```
0
1
2
3
```

Now, let's get a little deeper and make a couple of other changes. The above code will be modified and will have two subroutines. Figure 7, line 41, add the "repeat_" label with associated code through line 43. Modify lines 44 and 45 as indicated.

These modifications should result in an endless loop that is incrementing a variable. We are going to call the loop routine that does the addition, then call the wait routine that causes a lockstep between PASM and SPIN, then jump back to repeat the loop of calls.

```




CON
_clkmode = xtal1 + pll16x
'_xinfreq = 6_250_000    'MY BOARD AT 100MHZ
_xinfreq = 5_000_000   'QUICKSTART 80 MHZ

var

   long count

obj

pst:"parallax serial terminal"

pub main

pst.start(115000)
    waitcnt(clkfreq*5 +cnt)'hold two sed to open the
    'serial terminal and enable it
cognew(@asm,@count)

     repeat
      pst.dec(count~)'post clear p 157
      pst.newline
      waitcnt(clkfreq +cnt)

```

Figure 6

```
31          {{First subroutine. we are going to add three lines. First line
32           to add is call #wait, this tells the program to go and find a set of code named
33           "wait". at the bottom of the wait subroutine the following is added
34            " wait_ret   ret ", this signals the end of the subroutine and to jump
35            back to the next line of code after the "call".
36            jmp #loop is added as the next line of code to execute which sends the
37            code back to execute an "add" directive.}}
38 asm       org
39
40            mov addr, par
41 repeat_           call #loop  '<<<ADD
42                   call #wait  '<<<ADD
43                   jmp #repeat_
44  loop      add value,#1  'counting variable
45  loop_ret  ret  'ADD <<<<<<<<<
46
47            ' call #wait  '<<<ADD
48            ' jmp #loop   '<<<ADD
49  wait      rdlong prev, addr  wz 'what is in par??
50            if_nz  jmp #wait  'if the value in
51            'par is zero continue to next command
52            'if the value in par "addr" has not been cleared
53            'meaning the value that was put in "value" from
```

Figure 7

```
56            wrlong value, addr
57            'now write the value to the addr which has been assigned
58            'the same address as par and where the address of count in
59            'memory where the spin program can read it then jump back
60            'to the top of the loop and continue after the variable
61            ' called count has been cleard to zero
62          '  jmp #loop
63
64 wait_ret   ret  '<<<ADD
65
66
67 addr long 0
68 value long 0
69 prev  long 0
```

Figure 8

Since we now have two subroutines you should see this as seen in figure 9:

```
0
1
2
3
4
5
6
7
8
```

Figure 9

O.K. we are going to move on to the multiplication, addition, and subtraction code and create subroutines. In order to do this we also have to understand a difference from the continuous addition and doing a single multiplication etc. Thanks to the forums and David Carrier at Parallax, who both pointed out again that we have to stop the cog otherwise the results will be screwed up. I was wondering why I got zeros.

So, let's look at the code. Please refer to chapter one figures 28,29 and 30 and compare with chapters 10 through 13 in this chapter.

You will see the addition on line 62 execution of "multiply_ret ret", line 55 multiply label and lines 51 through 55.

We will now have a subroutine that executes and stops the cog after execution. The result will now be ready for the spin method to print the results.

The code is presented on the next page for your review.

The next code examples will be multiplication, division and

```

{{Multiplication based on the propeller manual page 380 as a subroutine}}

CON
_clkmode = xtal1 + pll16x
'_xinfreq = 6_250_000    'MY BOARD AT 100MHZ  DIFFERENT CRYSTAL
_xinfreq = 5_000_000  'QUICKSTART 80 MHZ  NORMAL CRYSTAL

var

   'VARIABLE IN THE PAR ADDRESS TO BE PASSED
   long x
   long y
   long product
obj

pst:"parallax serial terminal"

pub main
      x := 2
      y := 9
pst.start(115000)
    waitcnt(clkfreq*5 +cnt)'hold five sec to open the
     'serial terminal and enable it
cognew(@asm,@x)'start cog at the first variable address
      waitcnt(clkfreq*2 +cnt)  'give pasm time to do the work
      pst.str(string("product:"))
      pst.dec(product)
      pst.newline

```

Figure 10

```


dat
'' Multiply x[15..0] by y[15..0] (y[31..16] must be 0)
' on exit, product in y[31..0]
'
asm                     org

                        mov temp_var, par  'move par to a temporary variable
                        mov x_var, temp_var 'find the x variable
                        rdlong x_var, temp_var 'read in the value from top object
                        add temp_var, #4     'jump to next long which is the address of the
                                             ' next variable
                        mov y_var, temp_var  'repeat assignment and read in value
                        rdlong y_var, temp_var
                        add temp_var, #4     'jump again to assign the product variable address
                        mov product_var, temp_var
                        wrlong y_var, product_var  'test first part prior to subroutine call
```

Figure 11

```

                         call #multiply
                         call #writer
                         cogid cogname
                         cogstop cogname
 multiply               shl x_var,#16 'get multiplicand into x[31..16]
                        mov t,#16 'ready for 16 multiplier bits
                        shr y_var,#1 wc 'get initial multiplier bit into c
:loop                   if_c add y_var,x_var wc 'if c set, add multiplicand to product
                        rcr y_var,#1 wc 'put next multiplier in c, shift prod.
                        djnz t,#:loop 'loop until done
                        call #writer    '<<<<<<ADD
multiply_ret            ret 'return with product in y[31..0] 'this would be a subroutine
                         ' when used in a program
                        ' call #writer    '<<<<<<ADD

writer {<<<ADD}         wrlong y_var, product_var   'write the product from y[31..0] to the
                                                    'product variable for the top object
writer_ret ret  '<<<ADD

temp_var res 1
x_var res 1
y_var res 1
product_var res 1
t res 1
cogname res 1
```

Figure 12

```
product:27
product:27
product:36
product:27
```

Figure 13

```
CON
  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000          'QUICKSTART 80 MHZ  NORMAL CRYSTAL

var
   long dividend          'VARIABLE IN THE PAR ADDRESS TO BE PASSED
   long divisor
   long quotient
   long remainder

obj
  pst : "parallax serial terminal"

pub main
   dividend := 35
   divisor := 3
   pst.start(115200)
   waitcnt(clkfreq*5 + cnt)  'hold five sec to open the
                             'serial terminal and enable it
   cognew(@asm,@dividend)    'start cog at the first variable address
   waitcnt(clkfreq + cnt)    'give top object time to catch up to pasm


   pst.str(string("quotient:"))
   pst.dec(quotient)
   pst.newline
   pst.str(string("remainder:"))
   pst.dec(remainder)
   pst.newline
```

Figure 14

```

dat
 {{ NOTE: I have removed three mov commands as I have been shown that they are unnessary
 each "add tempvar,#4" points to the next variable. I got that from the NUTS AND VOLTS
 and appears that that may not be necessary.}}

asm       org

          mov tempvar, par      'get the par address into the temporary variable
          rdlong x, tempvar     'read the value into the dividend

          add tempvar, #4       'move over to the next long to get the divisor variable
          rdlong y, tempvar     'read the value of the divisor into the variable
          add tempvar, #4       'move over to the next long to get the quotient address
          call #divide  '<<<<ADD
          wrlong remaindervar,tempvar
           cogid cogname  '<<<<<ADD
          cogstop cogname '<<<<<ADD
' Divide x[31..0] by y[15..0] (y[16] must be 0)
' on exit, quotient is in x[15..0] and remainder is in x[31..16]

divide    shl y,#15             'get divisor into y[30..15]
          mov t,#16             'ready for 16 quotient bits
:loop     cmpsub x,y    wc      'y =< x? Subtract it, quotient bit in c
          rcl x,#1    'rotate c into quotient, shift dividend
          djnz t,#:loop         'loop until done
```

Figure 15

```
'  quotient in x[15..0],  ;return if used as a subroutine
'  remainder in x[31..16]

          mov    quotientvar,x
          and    quotientvar,andvar2    'isolate lower 16 bits
          wrlong quotientvar,tempvar    'write into Spinvar 'quotient'

          mov    remaindervar,x
          shr    remaindervar, #16      'isolate higher 16 bits
          add    tempvar,#4             'incr pointer to remainder address
 divide_ret ret

        '  wrlong remaindervar,tempvar   'write into Spinvar 'remainder'

andvar2       long $ffff
tempvar       res 1
x             res 1
y             res 1
quotientvar   res 1
remaindervar  res 1
t             res 1
cogname res 1
```

Figure 16

```
quotient:11
remainder:2
```

Figure 17

```
{{basic addition in pasm using the add directive SUBROUTINE. Page 259 propeller manual}}

CON
_clkmode = xtal1 + pll16x
'_xinfreq = 6_250_000   'MY BOARD AT 100MHZ  DIFFERENT CRYSTAL
_xinfreq = 5_000_000  'QUICKSTART 80 MHZ  NORMAL CRYSTAL

var
   'VARIABLE IN THE PAR ADDRESS TO BE PASSED
   long x
   long y
   long product
obj

pst:"parallax serial terminal"

pub main
      x := 30
      y := 45
pst.start(115000)
    waitcnt(clkfreq*5 +cnt)'hold five sec to open the
    'serial terminal and enable it
cognew(@asm,@x)'start cog at the first variable address
      waitcnt(clkfreq*2 +cnt)  'give pasm time to do the work

      pst.str(string("product:"))
      pst.dec(product~)
      pst.newline
```

```

dat

asm          org

             mov tempvar, par 'get the address of x from par
             mov xvar, tempvar 'assign the address to the xvar in pasm 'DISABLE
             rdlong xvar, tempvar 'read the value that is in x
             add tempvar, #4  'move over one long to get y's address
            ' mov yvar, tempvar 'assign that address to yvar  'DISABLE
             rdlong yvar, tempvar 'read the value that is in y
             add tempvar, #4 'move over one long to get the address of product
             mov productvar, tempvar 'assign the address to productvar 'DISABLE

             call #adder     '<<<<<<ADD
             call #writer    '<<<<<<ADD
             cogid cogname   '<<<<<<ADD
             cogstop cogname '<<<<<<ADD
adder        add xvar,yvar  'add x and y together answer will be in x    ;<<ADD LABLE
adder_ret    ret '<<<<ADD


writer         wrlong xvar, productvar 'write x into the product variable and print
writer_ret ret   '<<<<ADD

tempvar long 0
xvar long 0
yvar long 0
productvar long 0
flag long 0
cogname res 1   '<<<<ADD
```

```
product:75
```

```



{{ Tutorial on how to pass a number variable  and perform subtraction
with the sub directive
from spin to pasm and back, this works for numbers
from 0 to 256, bigger numbers in a later tutorial}}


CON
         _clkmode = xtal1 + pll16x
        '_xinfreq = 6_250_000   'MY BOARD AT 100MHZ  DIFFERENT CRYSTAL
         _xinfreq = 5_000_000  'QUICKSTART 80 MHZ  NORMAL CRYSTAL
obj

        pst:"parallax serial terminal"

var     'global variables
        long datavar
        long subvar
        long answervar

```

```
pub main
        datavar:= 30            'assign a value to datavar
        subvar := 12

        pst.start(115000)      'start the serial terminal object

        waitcnt(clkfreq*5 +cnt)'hold five sec to open the

        cognew(@asm,@datavar)   ' open a new cog for pasm. where it starts "asm" and
                                ' the address of the first variable
        waitcnt(clkfreq+cnt)    ' hold for a second

                                ' print routine

        pst.str(string("results:"))

        pst.dec(answervar)
        pst.newline

```

Figure 18

```

   asm          org       0    'This is the starting point for PASM


                mov temp_var, par

                mov data_var, temp_var
                rdlong data_var, temp_var

                add temp_var, #4

              '  mov sub_var, temp_var
                rdlong sub_var,temp_var

                add temp_var,#4

               ' mov answer_var,temp_var
                 call #sub_tract
                 call #writer
                 cogid cogname
               cogstop cogname
```

Figure 19

```

sub_tract               sub data_var,sub_var
sub_tract_ret ret


writer               wrlong data_var, temp_var
writer_ret   ret
{{ Reserved variables reserved for PASM's use. }}
   sub_var res 1
   data_var res 1
   answer_var res 1
   temp_var res 1
   cogname res 1
```

Figure 20

```
results:18
```

Figure 21