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.

```assembly
1. CON
2. \_xinfreq = \_6.250_000  ; MY BOARD AT 100KHZ
3. \_xinfreq = \_5.000_000  ; QUICKSTART 50 KHZ
4. var
5. long count
6. obj
7. psl:parallax serial terminal
8. pub main
9. psl.start(115200)
10. waitcnt(\_xinfreq*5 +cnt) ’hold two sec to open the serial terminal and enable it
11. cognv(#asm,#count)
12. repeat
13. psl.dec(count~) ’post clear p 157
14. psl.newLine
15. waitcnt(\_xinfreq +cnt)

Figure 1

```

```assembly
16. dat
17. asm org
18. mov addr, par
19. loop add value, #1 ’counting variable
20. wait rdlong prev, addr uz ’what is in par??
21. if nz jmp #wait ’if the value in par “addr” has not been cleared
22. if the value in par “addr” has not been cleared
23. meaning the value that was put in “value” from
24. addr which has the address of par “parameter”
25. urlong value, addr
26. mov write the value to the addr which has been assigned
27. the same address as par and the address of count in
28. memory where the spin program can read it then jump back
to the top of the loop and continue after the variable
29. called count has been cleared to zero
30. jmp #loop
31. addr long 0
32. value long 0
33. 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

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

CALL syntax:

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

- Symbol (z-field) is a 9-bit literal whose value is the address to jump to. This field must contain a JMP 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, symbol, 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.

Properly Assembly does not use a call stack, so the return address must be stored in a different manner. At compile time the assembler loads the destination routine as well as an RET instruction (Symbol and Symbol_ret, respectively) and records those addresses into the CALL instruction’s z-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

3. Assembly Language Reference – CALL

CALL instruction is really just a JMP instruction without a hard-coded destination address, and this run-time section 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

![CALL Procedure Diagram]

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

① The cog stores the return address (PC+1) that of next instruction) into the source (z-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 (z-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 name 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 cannot 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 d-field set (since CALL was 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 (z-field) of the Symbol_ret register unless the NRET effect is specified. Of course, specifying NRET is not recommended for the CALL instruction since it will overwrite the return address of the previous call.

RET

Instructions: Return to previously recorded address.
Chapter 2 Subroutines

Figure 3

```montague
หลวง

Figure 4

```司法

Figure 5

```司法

```
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:

```
Figure 6
```

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.

```
Figure 7
```
Since we now have two subroutines you should see this as seen in 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
Chapter 2 Subroutines

Montague PASM tutorial

((Multiplication based on the propeller manual page 388 as a subroutine))

CON

;Declade = x1+1 @110x
_xxfreq = 6,250,000  ; MY BOARD AT 100Mhz DIFFERENT CRYSTAL
_xxfreq = 5,090,000  ; QUICKSTART 80 MHz NORMAL CRYSTAL

VAR

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

OBJ

;par =serial terminal

pub main
w := 2
put string(255000)
waitcnt(xxfreq*5 ,cont) 'hold five sec to open the
serial terminal and enable it
getcoin(xxfreq*2 ,cont) 'start cop at the first variable address
waitcont(xxfreq*2 ,cont) 'give pasm time to do the work
put str(string("product: "))
pas dec(product)
pas 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 'read the x variable
;rlong x_var, temp_var 'read in the value from top of stack
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
;rlong y_var, temp_var
add temp_var, #4 'jump again to assign the product variable address
mov product_var, temp_var 'test first part prior to subroutine call

Figure 11

multiply

;call multiply
;call writer
;copid copname
;copstop copname

;multiply
;mov t,#16 'set multiplicand into y[31..16]
;shl x_var,#1 'get initial multiplier bit into x
;loop
;mov u_var,#1 'set next multiplier in u, shift prod.
drc t 'loop loop until done
;call writer '<<<ADD
;multiply_ret

;mov t, product_var 'this would be a subroutine
;when used in a program
;call writer '<<<ADD
;writer => ADD
;writer_ret ret '<<<ADD
temp_var res 1
x_var res 1
y_var res 1
product_var res 1
r t res 1
copname res 1

Figure 12

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

Figure 13
Montague PASM tutorial
Chapter 2 Subroutines

```pasm
  .c1node = xall + pl116x
  .xinfreq = 5_000_000
  "QUICKSTART 80 NHZ" NORMAL CRYSTAL

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

obj
  pst : "parallax serial terminal"
  sub main
    dividend := 25
    divisor := 3
    pst.start((153840 + cnt) 'hold five sec to open the
      serial terminal and enable it
    xinval(dividend) 'start cog at the first variable address
    waiton(xinfreq + cnt) 'give top object time to catch up to pace

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

Figure 14

let
  (( NOTE! I have removed three mov commands as I have shown that they are unnecessary
  each "add tempvar, &n" points to the next variable. I got that from the NUTS AND VOLTS
  and appeared that that may not be necessary. )

  let org
    mov tempvar, par 'get the par address into the temporary variable
    rdlong x, tempvar 'read the value into the dividend
    add tempvar, &n '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, &n 'move over to the next long to get the quotient address
    call divide '<<<<<<<<<
    copy copname '<<<<<<<<<
    divide x31..8 by y15..0 (y16 must be 0)
    on exit, quotient is in x15..8 and remainder is in x31..16
    divide shl y, #15 'get divisor into y[30..15]
    mov t, #18 'ready for 18 quotient bits
    loop cmpdup x, y, w 'y <= x? Subtract it, quotient bit in c
      rcl x, #1 'rotate c into quotient, shift dividend
djnz t, #1 'loop until done

  quotient in x15..8, ;return used as a subroutine
  remainder in x31..16

mov quotientvar, x
and quotientvar, endvar2 'isolate lower 16 bits
wrlong quotientvar, tempvar 'write into Spinvar "quotient"
mov remaindervar, x
shr remaindervar, #16 'isolate higher 16 bits
add tempvar, &n 'incr pointer to remainder address
divide next res
  wrlong remaindervar, tempvar 'write into Spinvar "remainder"

endvar2
  long #1111
  tempvar res 1
  x res 1
  res 1
  quotientvar res 1
  remaindervar res 1
  copname res 1

Figure 16

quotient:11
remainder:2

Figure 17
```
Montague PASM tutorial

Chapter 2 Subroutines

```pasm
CON
  _clkode = xtal + p1116x
  _xinfreq = 6250.000
  _xinfreq = 5.000.000
  _xinfreq = 5.000.000

var
  \text{varchar in the par address to be passed}

long x
long y
long product

obj

\text{set 'parallax serial terminal'}

pub main
  x := 38
  y := 45
  \text{wait(}clicfreq*5\text{) hold five sec to open the serial terminal and enable it}
  \text{cogpet(}Room.x\text{) start cog at the first variable address}
  \text{wait(}clicfreq*2\text{) give pass time to do the work}
  \text{pstr(string(}product\text{'\text{'})}
  \text{pstr} \text{dec(product)}
  pstr: \text{realine}

dat

asm
  org

  mov tempvar, par \text{'get the address of x from par'}
  mov xvar, tempvar \text{'assign the address to xvar in par'}
  rdlong xvar, tempvar \text{'read the value that is in x'}
  add tempvar, y \text{'move over one long to get y's address'}
  mov yvar, tempvar \text{'assign that address to yvar'}
  rdlong yvar, tempvar \text{'read the value that is in y'}
  add tempvar, y \text{'move over one long to get the address of product'}
  mov productvar, tempvar \text{'assign the address to productvar'}

  call \text{adder} \text{<<<<HOO}
  call \text{adder} \text{<<<<HOO}
  call \text{adder} \text{<<<<HOO}
  call \text{adder} \text{<<<<HOO}
  call \text{adder} \text{<<<<HOO}

  \text{adder ret ret} \text{<<<<HOO}
  \text{adder ret ret} \text{<<<<HOO}
  \text{adder ret ret} \text{<<<<HOO}
  \text{adder ret ret} \text{<<<<HOO}
  \text{adder ret ret} \text{<<<<HOO}

writer
  \text{wlong xvar, productvar \text{'write x into the product variable and print'}}
  \text{wlong xvar, productvar \text{'write x into the product variable and print'}}

product: 75
```

(Please note that the above code is a PASM tutorial example. The `\text{adder}` function is used to add two long variables, and the `\text{wlong}` function is used to write long variables.)
Chapter 2 Subroutines

```pasm
pub main
    datavar := 18
    subvar := 12
    pat.start(115000) ; start the serial terminal object
    waiton(clicfreq-5,cont) ; hold five sec to open the
    cognau(pano,datavar) ; open a new cog for pasm, where it starts 'ask' and
    waiton(clicfreq-1) ; hold for a second
    ; print routine
    pat.str(string("results"))
    pat.dec(answervar)
    pat.newLine

Figure 18

ask     org     0  ; This is the starting point for PASM

mov tempvar, par
mov data_var, tempo
rrlong data_var, temp_var
add temp_var, #4
mov subvar, tempo
rrlong sub_var,temp_var
add temp_var,#4
mov answer var,temp_var
call subtract
call print
cooldown cognau

Figure 19

subtract     sub data_var, sub_var
subtract_ret   ret

write        wrlong data_var, temp_var
write_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
cognau res 1

Figure 20

Figure 21

results: 18
```