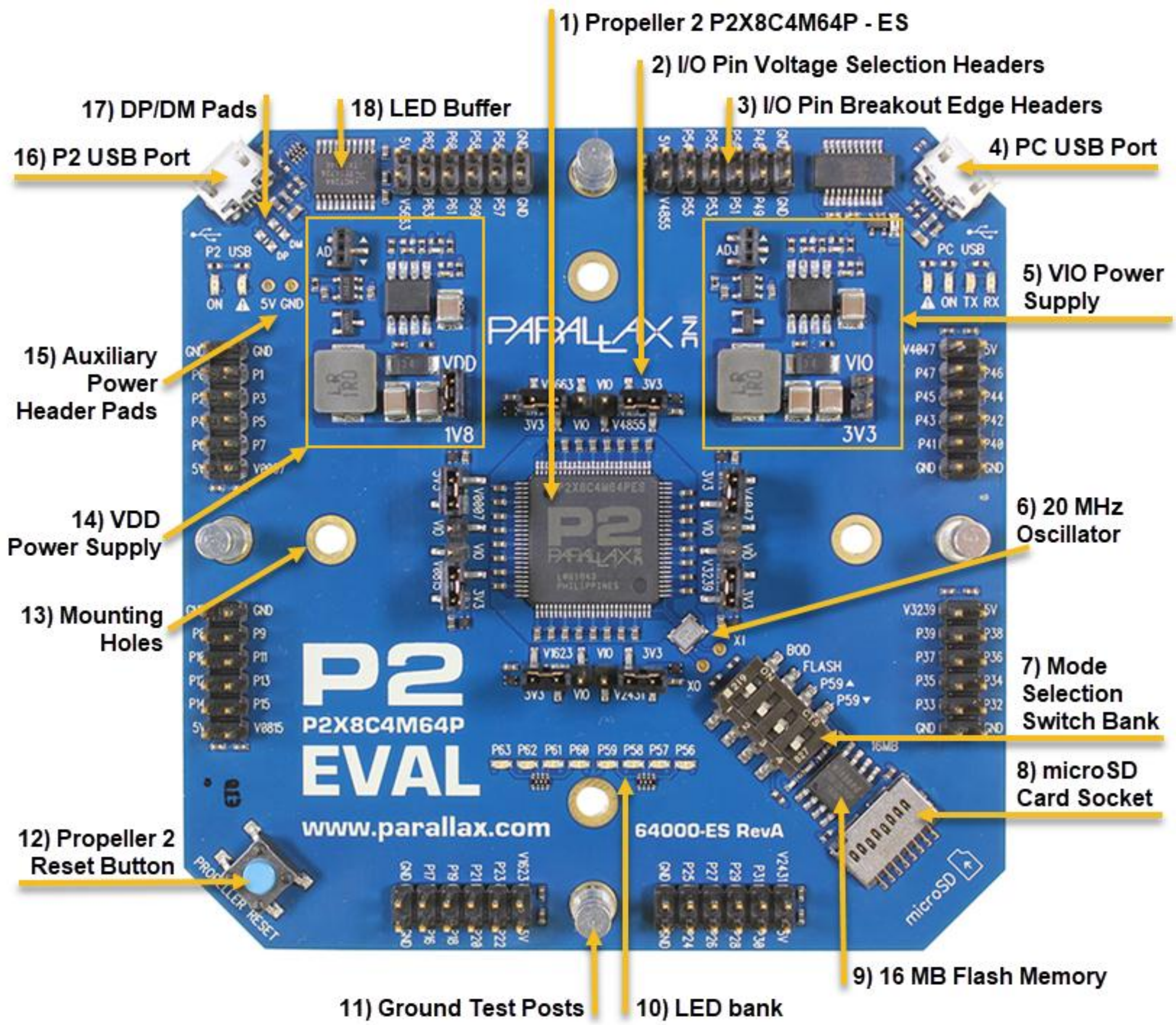
This is a second tutorial regarding propeller pasm programming. I have had since December the P2 Engineering Sample. There are many differences between both the P1 and P2.

The P2 is 64 bit and 16 cores and 64 pins.



My first tutorial as well as with this one was for myself, but I shared it to help anybody who was confused and frustrated with getting into assembly language programming. I am adding to it some of the basic differences using the same approach.

I will use the same approach. Leaving the first tutorial in place, the P2 pasm code will be shown after page 22.

Still no Blinky lights in the beginning!!!! They will come later for both P1 and P2 heading towards running stepper motors.

The first thing one will need is a copy of the propeller manual that is in the propeller tool and can be found here: https://www.parallax.com/product/122-32000.

Here is a link to Jeff Martin's webinar I uploaded to YouTube: https://www.youtube.com/watch?v=OZHuWYW3o1A

The first exercise will encompass passing variables from a spin method to a pasm method and back.

This the first piece of code that I came up with. There may be better ways to do this so bear with me.

I setup two global variables one for the spin method and the other for the pasm method. A five second waitcnt is used so as to have time to open the serial terminal when launching the code.

In order to launch the pasm code into a new cog this command is needed:

cognew(@asm,@datavar). The cognew means open the next cog, the @asm is the beginning of the assembly routine and the @datavar is the address of the first global variable.

```


{{ Tutorial 1 how to pass a number variable 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
        long datavar
        long answervar


```

Figure 1

The next steps are to start the serial terminal wait five seconds to allow one to open the serial terminal and then launch the cog. The code will then take the value in data var and print on the terminal. Now to the PASM method:

```
pub main
        datavar:= 25            'assign a value to datavar


        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("answer:"))
         pst.newline
        pst.dec(answervar)
        pst.newline

```

Figure 2

The datavar is assigned a value, in this case 256 which is the maximum pasm will handle without extra work. I will tackle that at a later time. We want to keep it simple at this time. This is also because many of the other tutorials I have seen get really complicated very quickly and do not take it in baby steps. I want to make sure that everybody can grasp the concept before getting into complicated code and get lost.

```
dat


   asm          org       0    'This is the starting point for PASM

                {{ The first item is to move the address of the parameter register "PAR" into
                a temporary variable and assigne it to the variable in which we will read the in
                this case the value of datavar in the spin method. }}
                mov temp_var, par
                {{ Now we are going to assign the pasm variable, data_var, the address of datavar in
                the spin method. }}
                mov data_var, temp_var

                {{ Now we have to move over to the next long to get the address of answervar in the
                spin object and assign it to answer_var in the pasm code.}}
                add temp_var, #4
                {{ Now assign this address to answer_var. }}
                mov answer_var,temp_var

                {{ Next read the value of datavar (spin object)  into the pasm data_var. }}
                rdlong data_var, par


                {{ Finaly write it to the answer_var which is spin's answervar for printing. }}
                wrlong data_var, answer_var

{{ Reserved variables reserved for PASM's use. }}

   data_var res 1
   answer_var res 1
   temp_var res 1
```

Figure 3

The pasm code starts in a "dat" section of spin. The "asm" "org" "0" indicates the beginning of the pasm code. In the cognew there is also an @datavar expression. This tells the pasm code the address of the first variable and that address will be stored in the "par" value. "par" from what I have found means parameter.

There is a very nice webinar done by Jeff Martin in 2009 that explains a lot of information regarding pasm code. I uploaded it to YouTube: https://www.youtube.com/watch?v=OZHuWYW3o1A.

Starting at:

mov temp_var, par

This is the mov instruction description:

MOV

Instruction: Set a register to a value.
MOV ***Destination***, □#□ ***Value***
Result: *Value* is stored in *Destination*.

Destination (d-field) is the register in which to store *Value*.

Value (s-field) is a register or a 9-bit literal whose value is stored into *Destination*.
Explanation
MOV copies, or stores, the number in *Value* into *Destination*.

If the WZ effect is specified, the Z flag is set (1) if *Value* equals zero. If the WC effect is specified, the C flag is set to *Value's* MSB. The result is written to *Destination* unless the NR effect is specified.

So, our first instruction directive will take the address of the spin code datavar variable in the registers and pass it to a temporary variable that we can manipulate. The code is commented so as to follow the progression and I am using full words instead of abbreviations so as one could more easily follow the progression.

```
asm         org       0    'This is the starting point for PASM

                      {{ The first item is to move the address of the parameter register "PAR" into
                      a temporary variable and assigne it to the variable in which we will read the in
                      this case the value of datavar in the spin method. }}
                      mov temp_var, par
                      {{ Now we are going to assign the pasm variable, data_var, the address of datavar in
                      the spin method. }}
                      mov data_var, temp_var
```

Figure 4

Now we have the address of the data_var which corresponds to datavar in the spin method.

```
{{ Now we have to move over to the next long to get the address of answervar in the
spin object and assign it to answer_var in the pasm code.}}
add temp_var, #4
{{ Now assign this address to answer_var. }}
mov answer_var,temp_var
```

Figure 5

As you can see, we move over and get the address of the spin code answervar variable and assign it's address to the pasm code answer_var variable. This is done by adding 4 to the temporary variable. Adding 4 moves to the next adjacent long where the answer var is located in the hub.

We are next going to use the rdlong and wrlong directives. The rdlong directive will read from a location and copy the value into a destination field as is shown in the propeller manual listing.

RDLONG ***Value***, □#□ ***Address***
Result: Long is stored in *Value*.

Value (d-field) is the register to store the long value into.

Address (s-field) is a register or a 9-bit literal whose value is the main memory address to read from.

The rdlong goes from right to left. We are reading the value that is in the par register which has the location of datavar and it's contents.

```
{{ Next read the value of datavar (spin object)  into the pasm data_var. }}
rdlong data_var, par
```

Figure 6

Lastly, we are going to write the value to the answer_var location that corresponds with answervar in the spin method and then print the results in a new variable. Note: wrlong works from ***<u>left to right.</u>***

```
{{ Finaly write it to the answer_var which is spin's answervar for printing. }}
wrlong data_var, answer_var
```

Figure 7

You should get a value on the serial terminal. I used 256 as this is the largest value for a single long, which is four bytes in size.

```
answer:
256
```

Figure 8

Changing the value of datavar to 25 in the spin method to verify.

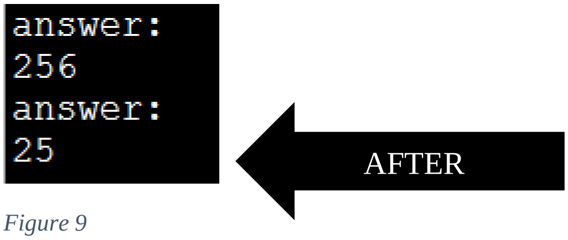


Figure 9

RES

Directive: Reserve next long(s) for symbol.

⟨***Symbol***⟩ **RES** ⟨***Count***⟩

- ***Symbol*** is an optional name for the reserved long in Cog RAM.
- ***Count*** is the optional number of longs to reserve for *Symbol*. If not specified, **RES** reserves one long.

RES: We need to reserve space for the pasm variables this is self-explanatory.

Now we can manipulate two variables and print them in succession. This is the new code:

```

{{ Tutorial 2 how to pass two number variables 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
        long datavar            {{each of these are one long apart. Have to move over one long
                                  so as to access them}}
        long answervar
        long datavar2
        long answervar2

```

Figure 10

```
pub main
        datavar:= 21                'assign a value to datavar
        datavar2 := 29

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

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

        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("answer:"))
        pst.newline
        pst.dec(answervar)
        pst.newline
        pst.str(string("answer:"))
        pst.newline
        pst.dec(answervar2)
        pst.newline

```

Figure 11

```
dat

  asm         org       0     'This is the starting point for PASM

              {{ The first item is to move the address of the parameter register "PAR" into
              a temporary variable and assigne it to the variable in which we will read the in
              this case the value of datavar in the spin method. }}
              mov temp_var, par
              {{ Now we are going to assign the pasm variable, data_var, the address of datavar in
              the spin method. }}
              mov data_var, temp_var
              rdlong data_var, temp_var
              {{ Now we have to move over to the next long to get the address of answervar in the
              spin object and assign it to answer_var in the pasm code.}}
              add temp_var, #4
              {{ Now assign this address to answer_var. }}
              mov answer_var,temp_var
               {{write the value to the answervar in spin}}
              wrlong data_var, answer_var
               {{go back and get the par address to access the next variable}}
              mov temp_var, par
```

Figure 12

```
              {{jump over two longs to get the address of datavar2 in the spin method}}
              add temp_var, #8
               {{assign the address}}
              mov data_var2, temp_var
              {{read the value}}
              rdlong data_var2, temp_var
              {{skip over one long to get answervar in spin}}
              add temp_var, #4
               {{assign the address}}
              mov answer_var2,temp_var
              {{now write the value to answervar2 in spin}}
              wrlong data_var2, answer_var2


{{ Reserved variables reserved for PASM's use. }}

  data_var res 1
  data_var2 res 1
  answer_var res 1
  answer_var2 res 1
  temp_var res 1
```

Figure 13

We have added a couple of items. First a new datavar named datavar2 and a new answervar named answervar2 as well, figure 10 and 14 lines 19 and 20, as their counterparts in the pasm method. In the print area answervar2 has been added also.

```
15 var
16         long datavar          {{each of these are one long apart. Have to move over one long
17                                 so as to access them}}
18         long answervar
19         long datavar2
20         long answervar2
21
```

Figure 14

Note the order of the global variables. This will make it easy to find them in the pasm method.

The pasm routine begins just like before and we get the location of datavar from par into the temporary variable and assign the location to data_var and read the value from par to data_var.

Now we have to move over a couple of longs to get the new variables and values, figure 13 and 15:

```
mov temp_var, par
{{jump over two longs to get the address of datavar2 in the spin method}}
add temp_var, #8
 {{assign the address}}
mov data_var2, temp_var
{{read the value}}
rdlong data_var2, temp_var
```

Figure 15

Now we can write the value to the second answer_var. Remember wrlong is from left to right as opposed to rdlong and other directives which are right to left.

```
{{move over one long to get answervar in spin}}
add temp_var, #4
 {{assign the address}}
mov answer_var2,temp_var
{{now write the value to answervar2 in spin}}
wrlong data_var2, answer_var2
```

Figure 16

This is what you should see on the serial terminal:

```
answer:
21
answer:
29
```

Figure 17

Changing the two datavar's values:

```
answer:
150
answer:
256
```

Figure 18

It works.

Now that we can get in and out of spin and pasm, I will present some examples of simple math.

I am trying to avoid the jump to really complicated programs with the assumption that the reader has a total comprehension of coding in assembly language of any type. I have found many tutorials do that.

These tutorials were good but confusing when they jump ahead and get very complex. Since I am a teacher, I teach flying and aircraft mechanics, I have to assess the background of each student. Academic learning can be difficult and painful, so if the instructor keeps it simple and explains the concept with easy examples that build up slowly, the student has a better chance of understanding and correlating the subject matter. That results in a much better outcome.

First, we will visit addition.

ADD

Instruction: Add two unsigned values.

ADD ***Value1*, 〈#〉 *Value2***

Result: Sum of unsigned *Value1* and unsigned *Value2* is stored in *Value1*.

- ***Value1*** (d-field) is the register containing the value to add to *Value2* and is the destination in which to write the result.
- ***Value2*** (s-field) is a register or a 9-bit literal whose value is added into *Value1*.

We are going to repeat the above code and make some changes:

Note the global variable name changes, figure 19 lines 10,11and 12.

```
1  {{basic addition in pasm using the add directive. Page259 propeller manual}}
2  
3  CON
4  _clkmode = xtal1 + pll16x
5  '_xinfreq = 6_250_000   'MY BOARD AT 100MHZ  DIFFERENT CRYSTAL
6  _xinfreq = 5_000_000  'QUICKSTART 80 MHZ  NORMAL CRYSTAL
7  
8  var
9     'VARIABLE IN THE PAR ADDRESS TO BE PASSED
10    long x
11    long y
12    long product
13 obj
14 
15 pst:"parallax serial terminal"
16 
17 pub main
18       x := 30
19       y := 45
20 pst.start(115000)
21     waitcnt(clkfreq*5 +cnt)'hold five sec to open the
22     'serial terminal and enable it
23 cognew(@asm,@x)'start cog at the first variable address
24       waitcnt(clkfreq*2 +cnt)  'give pasm time to do the work
25 
26       pst.str(string("product:"))
27       pst.dec(product~)
28       pst.newline
```

Figure 19

See figure 19 lines 26 and 27.

```
29
30 dat
31
32 asm        org
33
34             mov tempvar, par 'get the address of x from par
35             mov xvar, tempvar 'assign the address to the xvar in pasm
36             rdlong xvar, tempvar 'read the value that is in x
37             add tempvar, #4  'move over one long to get y's address
38             mov yvar, tempvar 'assign that address to yvar
39             rdlong yvar, tempvar 'read the value that is in y
40             add tempvar, #4 'move over one long to get the address of product
41             mov productvar, tempvar 'assign the address to productvar
42             add xvar,yvar  'add x and y together answer will be in x
43             wrlong xvar, productvar 'write x into the product variable and print
44
45
46 tempvar long 0
47 xvar long 0
48 yvar long 0
49 productvar long 0
50 flag long 0
```

See figure 20 lines 40 to 43. With these changes both variables will be added together.

Figure 20

```
product:75
```

Figure 21

Subtraction:

Changing line 65 in figure 24 does the trick.

```
{{ 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 answervar
        long subvar
```

Figure 22

```
23 pub main
24         datavar:= 25              'assign a value to datavar
25         subvar := 10
26
27         pst.start(115000)        'start the serial terminal object
28
29         waitcnt(clkfreq*5 +cnt) 'hold five sec to open the
30
31         cognew(@asm,@datavar)    ' open a new cog for pasm. where it starts "asm" and
32                                  ' the address of the first variable
33         waitcnt(clkfreq+cnt)     ' hold for a second
34
35                                  ' print routine
36
37         pst.str(string("results:"))
38          pst.newline
39         pst.dec(answervar)
40         pst.newline
```

Figure 23

```
41
42 dat
43
44
45    asm          org       0    'This is the starting point for PASM
46
47                 {{ The first item is to move the address of the parameter register "PAR" into
48                 a temporary variable and assigne it to the variable in which we will read the in
49                 this case the value of datavar in the spin method. }}
50                 mov temp_var, par
51                 {{ Now we are going to assign the pasm variable, data_var, the address of datavar in
52                 the spin method. }}
53                 mov data_var, temp_var
54
55                 {{ Now we have to move over to the next long to get the address of answervar in the
56                 spin object and assign it to answer_var in the pasm code.}}
57                 add temp_var, #4
58                 {{ Now assign this address to answer_var. }}
59                 mov answer_var,temp_var
60                 add temp_var,#4   'move over to the next long and get the subtraction variable address
61                 mov sub_var, temp_var 'assign the address to the variable
62                 rdlong sub_var,temp_var   'read the value in that address
63                 {{ Next read the value of datavar (spin object)  into the pasm data_var. }}
64                 rdlong data_var, par   'go back and get the value from the data variable that is in the par register
65                 sub data_var,sub_var   ''perform the subtraction data-subvar= xxx
66
67                 {{ Finaly write it to the answer_var which is spin's answervar for printing. }}
68                 wrlong data_var, answer_var
69
```

Figure 24

```
70 {{ Reserved variables reserved for PASM's use. }}
71    sub_var res 1
72    data_var res 1
73    answer_var res 1
74    temp_var res 1
```

Figure 25

What we have done is simply, at lines 60 and 61, added a new variable as well at line 71, these will be the subtraction variables. Next perform the subtraction and then write to our answer variable.

You should get this:

```
results:
15
```

25-10=15

Figure 26

Change subtraction variable to 12.

```
results:
13
```

25-12=13

Figure 27

Multiplication this is from the propeller manual page 380:

```
{{Multiplication based on the propeller manual page 380}}

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 := 3
      y := 27
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 28

```
30
31
32 dat
33 '' Multiply x[15..0] by y[15..0] (y[31..16] must be 0)
34 ' on exit, product in y[31..0]
35 '
36 asm                     org
37
38                         mov temp_var, par   'move par to a temporary variable
39                         mov x_var, temp_var 'find the x variable
40                         rdlong x_var, temp_var 'read in the value from top object
41                         add temp_var, #4      'jump to next long which is the address of the
42                                               ' next variable
43                         mov y_var, temp_var  'repeat assignment and read in value
44                         rdlong y_var, temp_var
45                         add temp_var, #4      'jump again to assign the product variable address
46                         mov product_var, temp_var
```

Figure 29

```
47
48                      multiply shl x_var,#16 'get multiplicand into x[31..16]
49                      mov t,#16 'ready for 16 multiplier bits
50                      shr y_var,#1 wc 'get initial multiplier bit into c
51 :loop                if_c add y_var,x_var wc 'if c set, add multiplicand to product
52                      rcr y_var,#1 wc 'put next multiplier in c, shift prod.
53                      djnz t,#:loop 'loop until done
54                      wrlong y_var, product_var   'write the product from y[31..0] to the
55                                                  'product variable for the top object
56
57                      'multiply_ret ret 'return with product in y[31..0] 'this would be a subroutine
58                      ' when used in a program
59 temp_var res 1
60 x_var res 1
61 y_var res 1
62 product_var res 1
63 t res 1
```

Figure 30

product:81 3*27=81

Figure 31 Change 27 to 9.

product:27 3*9=27

Figure 32 Basically, we are doing multiplication by addition:

27+27+27=81

3+3+3+3+3+3+3+3+3=27

The first operation is to shift left, the multiplicand into x[31..16], line 48.

SHL

Instruction: Shift value left by specified number of bits.

SHL *Value*, ⟨#⟩ *Bits*

Result: *Value* is shifted left by *Bits*.

- *Value* (d-field) is the register to shift left.
- *Bits* (s-field) is a register or a 5-bit literal whose value is the number of bits to shift left.

Next because this is 16 bit multiplication, so we are going to load a variable with the number 16, line 49: mov t,#16 'ready for 16 multiplier bits.

We are going to shift the carry into y by 1 each time we add the variables. So, on line 50 the first iteration will be loaded. This is done by shifting y right by one to get the carry flag set with the first number that will eventually be the result of the multiplication.

SHR: There is a shift right and shift left these are self-explanatory in the propeller manual as shown. The code will shift left or right by the number specified.

Line 50: shr y_var,#1 wc 'get initial multiplier bit into c

SHR

Instruction: Shift value right by specified number of bits.

SHR *Value*, ⟨#⟩ *Bits*

Result: *Value* is shifted right by *Bits*.

- *Value* (d-field) is the register to shift right.
- *Bits* (s-field) is a register or a 5-bit literal whose value is the number of bits to shift right.

Now we are going to ask if the carry flag is set when we add x and y. this will loop until the carry flag is not set an we will loop back and perform the operation again. Each addition will be counted until finished. When completed the carry will be the result of the multiplication. The carry will be discussed in the “if” conditional in the next paragraphs.

Now the loop:

If the carry flag is set, we will loop back and perform an add instruction and check the carry flag after each iteration. This conditional jump will be performed by the DJNZ directive what will evaluate the carry. If the carry in this case is set it will jump back to the beginning of the loop where the RCR instruction will rotate the carry flag, RCR, over into y at the end the value in y will be the answer. Basically, it adds up the carry bits. . If the carry is not set it will NOP, NO OPERATION, and drop out of the loop and go to the next instruction which in this case is to write the results to the variable, product_var and will be printed.

Which in the end of the loop, would be the answer if one did multiplication via the addition process.

RCR:

RCR

Instruction: Rotate C right into value by specified number of bits.

RCR Value, ⟨#⟩ Bits

Result: *Value* has *Bits* copies of C rotated right into it.

- ***Value*** (d-field) is the register in which to rotate C rightwards.
- ***Bits*** (s-field) is a register or a 5-bit literal whose value is the number of bits of *Value* to rotate C rightwards into.

CONDITIONAL STATEMENTS:

IF_x (Conditions)

Every Propeller Assembly instruction has an optional "condition" field that is used to dynamically determine whether or not it executes when it is reached at run time. The basic syntax for Propeller Assembly instructions is:

*⟨Label⟩ ⟨Condition⟩ **Instruction Operands** ⟨Effects⟩*

The optional *Condition* field can contain one of 32 conditions (see Table 3-3) and defaults to IF_ALWAYS when no condition is specified. The 4-bit **Value** shown for each condition is the value used for the **-CON-** field in the instruction's opcode.

This feature, along with proper use of instructions' optional *Effects* field, makes Propeller Assembly very powerful. Flags can be affected at will and later instructions can be conditionally executed based on the results. Here's an example:

```
                test    _pins, #$20      wc
                and     _pins, #$38
                shl     t1, _pins
                shr     _pins, #3
                movd    vcfg, _pins
        if_nc   mov     dira, t1
        if_nc   mov     dirb, #0
        if_c    mov     dira, #0
        if_c    mov     dirb, t1
```

The first instruction, `test _pins, #$20 wc`, performs its operation and adjusts the state of the C flag because the **WC** effect was specified. The next four instructions perform operations that could affect the C flag, but they do not affect it because no **WC** effect was specified. This means that the state of the C flag is preserved since it was last modified by the first instruction. The last four instructions are conditionally executed based on the state of the C flag that was set five instructions prior. Among the last four instructions, the first two `mov` instructions have `if_nc` conditions, causing them to execute only "if not C" (if C = 0). The last two `mov` instructions have `if_c` conditions, causing them to execute only "if C" (if C = 1). In this case, the two pairs of `mov` instructions are executed in a mutually exclusive fashion.

When an instruction's condition evaluates to **FALSE**, the instruction dynamically becomes a **NOP**, elapsing 4 clock cycles but affecting no flags or registers. This makes the timing of multi-decision code very deterministic.

DJNZ:

DJNZ

Instruction: Decrement value and jump to address if not zero.

DJNZ ***Value*****, ⟨#⟩** ***Address***

Result: *Value*-1 is written to *Value*.

- ***Value*** (d-field) is the register to decrement and test.
- ***Address*** (s-field) is the register or a 9-bit literal whose value is the address to jump to when the decremented *Value* is not zero.

This directive allows for repetition while decrementing a particular value of choice and when the result is not zero jump to a particular point in the code until the result is zero. At that point the code will drop down to the next instruction in line.

We run the loop until the carry flag is empty. This is repeated addition. Jeff and Dave at Parallax told me that there are many ways to do this. I am working on this myself. Basically, it is repetitive addition and that can be done in a loop until the number of iterations required are completed.

Division:

```
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 := 211
   divisor := 6
   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 33

```

dat


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


' 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

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

Figure 34

```
55
56          mov     quotientvar,x
57          and     quotientvar,andvar2        'isolate lower 16 bits
58          wrlong quotientvar,tempvar         'write into Spinvar 'quotient'
59
60          mov     remaindervar,x
61          shr     remaindervar, #16          'isolate higher 16 bits
62          add     tempvar,#4                 'incr pointer to remainder address
63          wrlong remaindervar,tempvar        'write into Spinvar 'remainder'
64
65 andvar2        long $ffff
66 tempvar        res 1
67 x              res 1
68 y              res 1
69 quotientvar    res 1
70 remaindervar   res 1
71 t              res 1
72
```

Figure 35

```
quotient:35
remainder:1
```

Figure 36

The division will be a continued subtraction algorithm that will subtract the divisor from the dividend until the divisor is either zero or there is a remainder less than the divisor. The answer will now be in the quotient the low bits, with the remainder in the high bits.

On line 47 we are going to shift left the divisor by 15 bits to get it into the high end of y. Then move the number 16 into t because t will be our iterations for the DNJZ directive which will perform the loop function 16 iterations. Now the compare and subtract, cmpsub, will subtract y from x and see if it is zero, the carry flag will answer the condition. At each iteration we will rotate carry left, RCL, by one. At the end of all operations x will have the quotient and y will have the remainder.

CMPSUB

Instruction: Compare two unsigned values and subtract the second if it is lesser or equal.

CMPSUB *Value1*, ⟨#⟩ *Value2*

Result: Optionally, *Value1* = *Value1* – *Value2*, and Z and C flags = comparison results.

- ***Value1*** (d-field) is the register containing the value to compare with that of *Value2* and is the destination in which to write the result if a subtraction is performed.
- ***Value2*** (s-field) is a register or a 9-bit literal whose value is compared with and possibly subtracted from *Value1*.

RCL

Instruction: Rotate C left into value by specified number of bits.

RCL *Value*, ⟨#⟩ *Bits*

Result: *Value* has *Bits* copies of C rotated left into it.

- *Value* (d-field) is the register in which to rotate C leftwards.
- *Bits* (s-field) is a register or a 5-bit literal whose value is the number of bits of *Value* to rotate C leftwards into.

The AND operation takes $FFFF and masks off high bits so as to get the quotient, we later shift the naked remainder by 16 to get the remainder.

AND – Assembly Language Reference

AND

Instruction: Bitwise AND two values.

AND *Value1*, ⟨#⟩ *Value2*

Result: *Value1* AND *Value2* is stored in *Value1*.

- *Value1* (d-field) is the register containing the value to bitwise AND with *Value2* and is the destination in which to write the result.
- *Value2* (s-field) is a register or a 9-bit literal whose value is bitwise ANDed with *Value1*.

Counting up and down:

```
{{counting up example, have to slow pasm. Introducing conditionals
and jmp command}}



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)

```

Figure 37

```
26      repeat
27       pst.dec(count~)'post clear p 157
28       pst.newline
29       waitcnt(clkfreq +cnt)
30
31 dat
32
33 asm      org
34
35           mov addr, par
36  loop     add value,#1   'counting variable
37  wait     rdlong prev, addr  wz 'what is in par??
38           if_nz   jmp #wait  'if the value in
39           'par is zero continue to next command
40           'if the value in par "addr" has not been cleared
41           'meaning the value that was put in "value" from
42           ' addr which has the address of par "parameter"
43           '
44           wrlong value, addr
45           'now write the value to the addr which has been assigned
46           'the same address as par and where the address of count in
47           'memory where the spin program can read it then jump back
48           'to the top of the loop and continue after the variable
49           ' called count has been cleard to zero
50           jmp #loop
51
```

Figure 38

```
52
53 addr  long 0
54 value long 0
55 prev  long 0
```

Figure 39

Since spin is much slower than pasm, we have to interrupt pasm so spin can keep up. With that in mind we are going to look at line 27 and 37 to 51.

Line 27:

pst.dec(count~)'post clear p 157.

Y := X~ + 2

The Post-Clear operator in this example clears the variable to 0 (all bits low) after providing its current value for the next operation. In this example if X started out as 5, X~ would provide the current value for the expression (5 + 2) to be evaluated later, then would store 0 in X. The expression 5 + 2 is then evaluated and the result, 7, is stored into Y. After this statement, X equals 0 and Y equals 7.

Since Sign-Extend 7 and Post-Clear are always assignment operators, the rules of Intermediate Assignments apply to them (see page 147).

```
37  wait      rdlong prev, addr   wz
38            if_nz   jmp #wait
```

Figure 40

So, if the line 27 instruction has not cleared, pasm will jump back to the loop until it is cleared then pasm will perform the operation again.

Change line: 36 add to sub and you will have a continuous loop of subtraction.

```
36 loop      add value,#1   'counting variable
```

Figure 41

Arrays:

We are now able to add, subtract, multiply and divide. Basic math skills that we will now take to a next level but in a slow process. Next let's create an array and do some math while learning to populate the array and print selected arrays cells.

Simple array.

```

{{basic array populate the array do some simple math }}


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

var


  long data
  long array[10] 'global variable array 10 cells long array[0]..array[9]


obj

pst:"parallax serial terminal"


pub main
      data := 16

pst.start(115000)
    waitcnt(clkfreq*5 +cnt)'hold five sec to open the
    'serial terminal and enable it
   cognew(@asm,@data )'start cog at the first variable address

```

Figure 43

```

      pst.str(string("array:"))
      pst.dec(array[1])  'print the second cell first

      pst.newline

      pst.str(string("array:"))
      pst.dec(array[0])  'print the first cell second

      pst.newline



```

Figure 42

```
44
45
46
47 dat
48
49
50  asm         org   0
51
52           mov tempvar,par    'get the par address into a temporary variable
53
54           mov datvar,tempvar 'assign the address to the datvar in pasm
55
56           rdlong datvar,tempvar 'read in the value of the data variable from spin
57
58           add tempvar,#4    'move over and get the beginning of the array
59
60           mov arrayvar,tempvar 'assign the beginning of the array
61
62           wrlong datvar,arrayvar  'write the value from spin to array[0]
63
64           add arrayvar,#4 'move over to the next array cell
65
66           add datvar, #10  'add 10 to the value in in the data variaable from spin
67           'in this case 16 + 10 = 26
68
69           wrlong datvar,arrayvar 'write the product to the second array cell array[1]
70
```

Figure 44

```
71
72
73
74
75 tempvar long 0
76 datvar long 0
77
78 arrayvar long 10  'global variable array 10 cells long array[0]..array[9]
```

Figure 45

We are going to start as before, and this time have two global variables. One is the data to be passed with a value from spin to pasm. The other is an array that is 10 cells long. That means that each cell will be a long in size.

As you can see in the spin method and the pasm method both are declared. Standard entry to get the addresses and values entered.

The line 56, read in the value to the datavar variable.

Line 62 write it to the first array cell, array[0].

Now to access the second array cell, array[1], we have to move over to the next long, line 64 by adding 4 bytes. Now for a little math to make it interesting we are going to add the littoral number 10 to the variable that is stored in the datavar which is 16. So 16+10=26.

The spin method is going to print them in reverse order which shows that we can manipulate the array.

```
array:26
array:16
```

Figure 46

P2 PASM SECTION

The P2 does not have an official GUI at this time. Coding is done in “Pnut” or Spin2Gui and a few others. The Pnut is on P2Pasm only. Serial interface has to be coded. Spin2Gui supports .spin2 files, Fastspin, C and Basic. It has a serial terminal as opposed to Pnut.

https://github.com/totalspectrum/spin2gui/releases

https://www.parallax.com/product/64000-es

The demo code here is in .spin2 and using the Spin2gui

```
CON
  oscmode = $010007f8
  freq = 160_000_000
  baud = 230_400

OBJ
  ser: "spin/SmartSerial"

VAR
   long cog
   long data
   long answer

PUB main
   clkset(oscmode, freq)
   ser.start(63, 62, 0, BAUD)
   pausems(500) ' wait for baud change on host

   ser.str("PASM demo: ")

   data := 12
   cog := cognew(@startasm, @data)

   ser.str("cog: ")
   ser.dec(cog)

   '' wait until the COG has finished
   '' it indicates this by clearing "data"
   repeat while data <> 0

   '' now print results
   ser.str(" answer = ")
   ser.dec(answer)
   ser.nl

```

```
37  DAT
38 startasm
39          '' assembly program to add 1 to a number
40          '' we start with ptra pointing at the number
41          '' the answer must immediately follow the input data
42          '' in the VAR section
43          rdlong value, ptra
44          add     value, #5
45          add     ptra, #4         ' point to answer
46          wrlong  value, ptra
47          sub     ptra, #4         ' point back to original data
48          wrlong  #0, ptra         ' tell original COG we are done
49
50          '' now shut down
51          cogid   value
52          cogstop value
53
54 value    long    0
55
```

Lines 2,3,4 and 15 are to setup the clock speed and baud rate for the serial terminal. Subsequent lines are similar to regular Spin language. Lines 43 through 48 is where the magic happens.

Instead of using cog ram to share variables we are going to do it with HUB RAM. There is no PAR variable with the P2. There is ptra/ptrb.

We can use the ptra as the start address like in P1 but we are going to work directly with it. So no need to have an intermediary variable to mov things around.

At line 43 we read the value from the ptra directly into the variable that is shared with the .spin2 object and P2 PASM.

So on line 43 we read in the value from spin, then we add 5 to that value , move over to the next long where the answer variable resides and write it to it so it will be printed.

Simple.

```
CON
  oscmode = $010007f8
  freq = 160_000_000
  baud = 230_400

OBJ
  ser: "spin/SmartSerial"

VAR
   long cog
   long data
   long data2
   long answer


PUB main
   clkset(oscmode, freq)
   ser.start(63, 62, 0, BAUD)
   pausems(500) ' wait for baud change on host

   ser.str(" PASM demo addition: ")
   ser.nl
'   ADD THESE TWO VALUES
   data := 200
   data2 :=36500
   cog := cognew(@startasm, @data)

   ser.str(" cog: ")
   ser.dec(cog)
   ser.nl

   '' wait until the COG has finished
   '' it indicates this by clearing "data"
   repeat while data <> 0

   '' now print results
   ser.str(" answer = ")
   ser.dec(answer)
```

Now here we are going to take two variables and add them, Both will come from the top object:

```
33    '' it indicates this by clearing "data"
34    repeat while data <> 0
35
36    '' now print results
37    ser.str(" answer = ")
38    ser.dec(answer)
39    ser.nl
40
41
42  DAT
43 startasm
44         '' assembly program to add numbers
45         '' we start with ptra pointing at the number
46         '' the answer must immediately follow the input data
47         '' in the VAR section
48         rdlong value, ptra     'get first value
49
50         add     ptra, #4        ' move over to pickup next value
51                                 'in the next long
52         rdlong  value2, ptra    'get second value
53
54         add     value2, value
55         mov     value3,value2  'move to answer var in next long
56         add ptra,#4
57         wrlong value3,ptra      'write it to the pointer
58
59 '*****HERE IS THE QUESTION*********
60         sub     ptra, #8        ' point back to original data????
61         wrlong  #0, ptra        ' tell original COG we are done
62
63         '' now shut down
64         cogid   value
65         cogstop value
66
67 value   long    0    'data
68 value2  long     0    'data2
69 value3  long     0    'answer
70
```

Observe line 8 get the first value. Move over 4 long sfor the second. Pick up the second. Add them move over another long to get to the answer to be printed write it and shut down.

Subtraction"

```
CON
  oscmode = $010007f8
  freq = 160_000_000
  baud = 230_400

OBJ
  ser: "spin/SmartSerial"

VAR
   long cog
   long data
   long data2
   long answer


PUB main
   clkset(oscmode, freq)
   ser.start(63, 62, 0, BAUD)
   pausems(500) ' wait for baud change on host

   ser.str(" PASM demo subtraction: ")
   ser.nl
  'SUBTRACT DATA FROM DATA2 if data val is larger it will
  'generate a negative number
   data := 35
   data2 := 100
   cog := cognew(@startasm, @data)

   ser.str(" cog: ")
   ser.dec(cog)
   ser.nl
```

```

  '' wait until the COG has finished
  '' it indicates this by clearing "data"
  repeat while data <> 0

  '' now print results
  ser.str(" answer = ")
  ser.dec(answer)
  ser.nl
  ser.nl
 ' ser.str("answer2 = ")
 ' ser.dec(answer2)
 ' ser.nl


 DAT
startasm
        '' assembly program to subtract a number
        '' we start with ptra pointing at the number
        '' the answer must immediately follow the input data
        '' in the VAR section
        rdlong value, ptra      'get first value

        add     ptra, #4         ' move over to pickup next value
        rdlong  value2, ptra     'get second value

        sub     value2, value
        mov     value3,value2
        add ptra,#4
        wrlong value3,ptra
        sub     ptra, #8         ' point back to original data
        wrlong  #0, ptra         ' tell original COG we are done

        '' now shut down
        cogid   value
        cogstop value
```

Multiplication:

```
CON
  oscmode = $010007f8
  freq = 160_000_000
  baud = 230_400

OBJ
  ser: "spin/SmartSerial"

VAR
   long cog
   long data
   long data2
   long answer


PUB main
   clkset(oscmode, freq)
   ser.start(63, 62, 0, BAUD)
   pausems(500) ' wait for baud change on host

   ser.str("PASM demo multiplication: ")
   ser.nl
'     MULTIPLY THESE TWO VARIABLES
   data := 11
   data2 :=7
   cog := cognew(@startasm, @data)

   ser.str(" cog: ")
   ser.dec(cog)
   ser.nl
   '' wait until the COG has finished
   '' it indicates this by clearing "data"
   repeat while data <> 0

   '' now print results
   ser.str(" answer = ")
   ser.dec(answer)
```

```
   ser.nl



 DAT
startasm
        '' assembly program to multiply numbers
        '' we start with ptra pointing at the number
        '' the answer must immediately follow the input data
        '' in the VAR section
        rdlong value, ptra     'get first value

        add     ptra, #4        ' move over to pickup next value
        rdlong  value2, ptra    'get second value

        mul     value2, value
        mov     value3,value2   'mov results to answer
        add ptra,#4
        wrlong value3,ptra
        sub     ptra, #8        ' point back to original data
        wrlong  #0, ptra        ' tell original COG we are done

        '' now shut down
        cogid   value
        cogstop value

value   long    0    'data
value2  long     0    'data2
value3  long     0    'answer

```

Division will be shown on next iteration.