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.
Differences between P1 and P2 Pasm

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=OZHuwY3o1A

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.

```
CON
_xintf = 0
_xintf = 1
obj
psm:"parallax serial terminal"
var
long datavar
long ansvar
```

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
Differences between P1 and P2 Pasm

terminal. Now to the PASM method:

```plaintext
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 assign 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, %0
      (   Now assign this address to answervar. )
   mov answer_var, temp_var
   
   (   Next read the value of datavar (spin object) into the Pasm data_var. )
   rdlong data_var, par
   
   (   Finally 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 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.
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:

\texttt{mov temp\_var, par}

This is the \texttt{mov} instruction description:

\textbf{MOV}

\textbf{Instruction:} Set a register to a value.

\textbf{MOV} \textit{Destination}, \#\textit{Value}

\textbf{Result:} \textit{Value} is stored in \textit{Destination}.

\textit{Destination} (d-field) is the register in which to store \textit{Value}.

\textit{Value} (s-field) is a register or a 9-bit literal whose value is stored into \textit{Destination}.

Explanation

MOV copies, or stores, the number in \textit{Value} into \textit{Destination}.

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

So, our first instruction directive will take the address of the spin code \texttt{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.

\begin{verbatim}
asm
  org 0
  "This is the starting point for PASM"

  \(( The \ first \ item \ is \ to \ move \ the \ address \ of \ the \ parameter \ register \ "PAR" \ into \ \)
  \( a \ temporary \ variable \ and \ assign \ it \ to \ the \ variable \ in \ which \ we \ will \ read \ the \ in \)
  \( this \ case \ the \ value \ of \ datavar \ in \ the \ spin \ method. \)\)
  mov temp\_var, par

  \(( Now \ we \ are \ going \ to \ assign \ the \ \texttt{pasm} \ variable, \ \texttt{data\_var}, \ the \ address \ of \ datavar \ in \)
  \( the \ spin \ method. \)\)
  mov data\_var, temp\_var
\end{verbatim}

\textit{Figure 4}

Now we have the address of the \texttt{data\_var} which corresponds to \texttt{datavar} in the spin method.
Differences between P1 and P2 Pasm

```c
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
```

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.

```c
60   { { Next read the value of datavar (spin object) into the pass data_var. } }
61     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 *left to right*.

```c
65   { { Finally write it to the answer_var which is spin’s answervar for printing. } }
66     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.
Differences between P1 and P2 Pasm

RES

**Directive:** Reserve next long(s) for symbol.

\(<\text{Symbol}\) RES \(\langle\text{Count}\rangle\)

- **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:

```c
CON
    _clkmode = xtall + pll10x
    _xinfreq = 6_250_000  "MT BOARD AT 100MHz DIFFERENT CRYSTAL
    _xinfreq = 6_000_000  "QUICKSTART 80 MHZ NORMAL CRYSTAL
    obj
    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

pub main
    datavar := 21  "assign a value to datavar
    datavar2 := 29
    pst.start(115800)  "start the serial terminal object
    waitcnt(clkfreq+cnt)  "hold five sec to open the serial terminal
    cogreq(Pasm, 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
```
Differences between P1 and P2 Pasm

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.

```
var
long datavar
long answervar
long datavar2
long answervar2
```

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:
Differences between P1 and P2 Pasm

```
68  mov temp_var, par
69  ((jump over two longs to get the address of datavar2 in the spin method))
70  add temp_var, #8
71  ((assign the address))
72  mov data_var2, temp_var
73  ((read the value))
74  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.

```
76  mov data_var2, temp_var
77  ((move over one long to get answervar in spin))
78  add temp_var, #4
79  ((assign the address))
80  mov answer_var2, temp_var
81  ((now write the value to answervar2 in spin))
82  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.
We are going to repeat the above code and make some changes:

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

```
CON
_clkmode = xtal1 - p1116x
_xinfreq = 8,250,000  "MY BOARD AT 100MHZ DIFFERENT CRYSTAL
_xinfreq = 5,000,000  "QUICKSTART BO BO X9Z NORMAL CRYSTAL

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

obj

put "parallax serial terminal"

pub main
  x = 30
  y = 45
  wait for(0.15000)
  wait for(clkfreq=5 -cnt) hold five sec to open the
  serial terminal and enable it
  cognax(pasm,%x) start cog at the first variable address
  wait for(clkfreq=2 -cnt) give pass time to do the work
  put str(string("product!"))
  put dec(product+)
  put newline
```

See figure 19 lines 26 and 27.
Differences between P1 and P2 Pasm

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.

Figure 22
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:
Differences between P1 and P2 Pasm

\[
\begin{align*}
\text{results:} & \quad 15 \\
25-10 &= 15 \\
\end{align*}
\]

*Figure 26*

Change subtraction variable to 12.

\[
\begin{align*}
\text{results:} & \quad 13 \\
25-12 &= 13 \\
\end{align*}
\]

*Figure 27*

Multiplication this is from the propeller manual page 380:

```
\begin{verbatim}
((Multiplication based on the propeller manual page 380))

\begin{verbatim}
\textcolor{red}{\text{CON}}
\textcolor{blue}{\text{clkmode = xtal1 + pll16x}}
\textcolor{green}{\text{.xrefreq = 6.2500000 \text{ // MY BOARD AT 100MHZ DIFFERENT CRYSTAL}}}
\textcolor{orange}{\text{.xrefreq = 5.0000000 \text{ // QUICKSTART 00 MHZ NORMAL CRYSTAL}}}
\textcolor{purple}{\text{var}}

\textcolor{brown}{\text{\textquotesingle VARIABLE IN THE PAR ADDRESS TO BE PASSED}}
\textcolor{pink}{\text{long x}}
\textcolor{teal}{\text{long y}}
\textcolor{cyan}{\text{long product}}
\textcolor{gray}{\text{obj}}

\textcolor{yellow}{\text{if} \textcolor{green}{\text{parialx serial terminal}}}
\textcolor{purple}{\text{pub main}}
\textcolor{orange}{\text{x := 3}}
\textcolor{blue}{\text{w := 27}}
\textcolor{red}{\text{waiton}(\text{clkfreq}5 \text{ +cnt}) \text{\textquotesingle hold five sec to open the}}
\textcolor{purple}{\text{serial terminal and enable it}}
\textcolor{orange}{\text{cofgmcu(Maps0,\text{w}) \text{\textquotesingle start cog at the first variable address}}}
\textcolor{blue}{\text{waitcnt(\text{clkfreq}2 \text{ +cnt}) \text{\textquotesingle give pasc time to do the work}}}
\textcolor{green}{\text{put str(string\text{\textquotesingle product})}}
\textcolor{cyan}{\text{put dec(product)}}
\textcolor{magenta}{\text{put newline}}
\end{verbatim}
\end{verbatim}
```

*Figure 28*
Differences between P1 and P2 Pasm

```
(dat

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

  ssm

    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 u_var, temp_var 'repeat assignment and read in value
    rdlong u_var, temp_var 'repeat assignment and read in value
    add temp_var, #4 'jump again to assign the product variable address
    mov product_var, temp_var

Figure 29

))

  multiply shl x_var,#16 'get multiplicand into x[31..16]
  mov t,#18 'ready for 16 multiplier bits
  shr u_var,#1 'get initial multiplier bit into c
  loop
  if_c add u_var, x_var wc 'if c set, add multiplicand to product
  xor u_var,#1 'put next multiplier in c, shift prod.
  djnz t,'loop until done
  wrlong u_var, product_var 'write the product from y[31..0] to the
  product variable for the top object

  'multiply ret ret 'return with product in y[31..0] 'this would be a subroutine
  when used in a program

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=27

The first operation is to shift left, the multiplicand into x[31..16], line 48.)
Di
erences between P1 and P2 Pasm

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

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.
Differences between P1 and P2 Pasm

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.

**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:

\[
\text{(Label) (Condition) Instruction Operands (Effects)}
\]

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:

```assembly
  test  _pins, #020  wc
  and  _pins, #030
  shl  tl, _pins
  shr  _pins, #3
  movd  vcfg, _pins
  if_nc mov  dira, tl
  if_nc mov  dirb, #0
  if_c  mov  dira, #0
  if_c  mov  dirb, tl
```
Differences between P1 and P2 Pasm

The first instruction, test _pins, #820  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_ne 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:
Differences between P1 and P2 Pasm

```assembly
org
mov tempvar, par  ; get the par address into the temporary variable
add tempvar, #4   ; move over to the next long to get the divisor variable
rdlong u, 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 u[15..0] (u[16] must be 0)
; on exit, quotient is in x[15..0] and remainder is in x[31..16]
shl u, #15        ; get divisor into u[30..15]
mov t, #16        ; ready for 16 quotient bits
loop              ; loop until done
    cmpnh x, u    ; u < x? Subtract it, quotient bit in c
    rcl x, #1     ; rotate c into quotient, shift dividend
    djnz t, loop  ; return if used as a subroutine
    remainder in x[31..16]
```

Figure 33

Figure 34
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, \( \text{Value1} = \text{Value1} - \text{Value2} \), 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**.
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.

**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
   .clockmode = x011 + all16x
   .xfreq = 8.250_000   "MY BOARD AT 100MHz"
   .xfreq = 5.000_000   "QUICKSTART 80 MHz"

var
   long count
   obj

pst:"parallax serial terminal"

pub main
   pst.start(115000)
   waitcnt(clkfreq*5+cnc) "hold two sec to open the
   serial terminal and enable it
   cognu(@asm,count)

Figure 37
```
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:

cst.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).
Differences between P1 and P2 Pasm

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.

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

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.

```pasm
1 ((basic array populate the array do some single math ))
2 CON
3 _clkn=xtal + p111fix
4 _xinfreq = 6.250.000 'MY BOARD AT 100MHZ DIFFERENT CRYSTAL
5 _xinfreq = 5.000.000 'QUICKSTART 80 MHZ NORMAL CRYSTAL
6 var
7
8 long data
9 long array[10] 'global variable array 10 cells long array[0].array[9]
10 obj
11 pst:"parallax serial terminal"
12 pub main
13 data := 16
14 pst.start(115500)
15 waitcnt(clknfreq+cnt) 'hold five sec to open the
16 serial terminal and enable it
17 cognew(Rush,#data ) 'start cog at the first variable address
18
19
20 pst.str("array[]")
21 pst.dec(array[1]) 'print the second cell first
22 pst.newline
23 pst.str("array[]")
24 pst.dec(array[0]) 'print the first cell second
25 pst.newline
```

Figure 41

Figure 42
Differences between P1 and P2 Pasm

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.

```
Figure 44

org 0
mov tempvar, par 'get the par address into a temporary variable
mov datvar, tempvar 'assign the address to the datvar in pasm
rdlong datvar, tempvar 'read in the value of the data variable from spin
add tempvar, #4 'move over and get the beginning of the array
mov arrayvar, tempvar 'assign the beginning of the array
wrlong datvar, arrayvar 'write the value from spin to array[0]
add arrayvar, #4 'move over to the next array cell
add datvar, #10 'add 10 to the value in in the data variable from spin
 'in this case 16 + 10 = 26
wrlong datvar, arrayvar 'write the product to the second array cell array[1]
```

```
Figure 45

tempvar long 0
datvar long 0
arrayvar long 10 'global variable array 10 cells long array[0]..array[9]
```

Figure 46

array:26
array:16
Differences between P1 and P2 Pasm

**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://github.com/totalspectrum/spin2gui/releases)


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

```spin2
CON
oscmode = $01000000
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(65, 62, 0, BAUD)
pause(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
```
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.
Differences between P1 and P2 Pasm

```
CON
   oscmode = $010000f8
   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:
Differences between P1 and P2 Pasm

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.
Differences between P1 and P2 Pasm

Subtraction”

1  CON
2  oscmode = $010007f8
3  freq = 160_000_000
4  baud = 230_400
5
6  OBJ
7  ser: "spin/SmartSerial"
8
9  VAR
10  long cog
11  long data
12  long data2
13  long answer
14
15  PUB main
16    clkset(lookmode, freq)
17    ser.start(63, 62, 0, BAUD)
18    pausems(500) ' wait for baud change on host
19
20    ser.str(" FASM demo subtraction: ")
21    ser.nl
22    'SUBTRACT DATA FROM DATA2 if data val is larger it will
23    'generate a negative number
24    data := 35
25    data2 := 100
26    cog := cognew($startasm, $data)
27
28    ser.str(" cog: ")
29    ser.dec(cog)
30    ser.nl
Differences between P1 and P2 Pasm

```
32    ' wait until the COG has finished
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    ser.nl
41    ' ser.str("answer2 = ")
42    ' ser.dec(answer2)
43    ' ser.nl
44
45
46 DAT
47 startasm
48    ' assembly program to subtract a number
49    ' we start with ptra pointing at the number
50    ' the answer must immediately follow the input data
51    ' in the VAR section
52    rdlong value, ptra         'get first value
53
54    add ptra, #4              ' move over to pickup next value
55    rdlong value2, ptra       'get second value
56
57    sub value2, value
58    mov value3,value2
59    add ptra,#4
60    wrlong value3,ptra
61    sub ptra, #8              ' point back to original data
62    wrlong #0, ptra           ' tell original COG we are done
63
64    ' now shut down
65    cogid value
66    cogstop value
```

Multiplication:
Differences between P1 and P2 Pasm

```
1   CON
2       oscmode = $010007f8
3       freq = 160_000_000
4       baud = 230_400
5
6   OBJ
7       ser: "spin/SmartSerial"
8
9   VAR
10      long cog
11      long data
12      long data2
13      long answer
14
15   PUB main
16       clkset(oscmode, freq)
17       ser.start(63, 62, 0, BAUD)
18       pausems(500) ' wait for baud change on host
19
20       ser.str("PASM demo multiplication: ")
21       ser.nl
22 '    MULTIPLY THESE TWO VARIABLES
23      data := 11
24      data2 := 7
25      cog := cognew(@startasm, @data)
26
27      ser.str(" cog: ")
28      ser.dec(cog)
29      ser.nl
30      ' wait until the COG has finished
31      ' it indicates this by clearing "data"
32      repeat while data <> 0
33
34      ' now print results
35      ser.str(" answer = ")
36      ser.dec(answer)
```
Division will be shown on next iteration.
Differences between P1 and P2 Pasm