

## Outline

### PASM for beginners Propeller 102

Preface

Table of contents

#### **Part one What we have to work with**

Book resources

Obex shared resources

Forum

Propeller Chip

This book

Program listings on the Net

Propeller manual and how to use it and what it does not tell

Binary math beginnings

Simple register manipulations

#### **Part two Simple Output**

7404 buffers

The PST

Development board

Duplex Full Serial dfs interfacing to PST

LEDs

2 lines by 16 char display

Extended to 4 lines by 20 chars

Speaker

Servo motor

Frequency

PWM generation

#### **Part three Simple Input**

read one push button

read Switches

Read Keyboard

read Potentiometer

read Voltage

Count pulses.

read frequency

1302 interface

single wire clock/other interface.

LM34 interface

**Part Four The projects**

Solenoids and relays

Self leveling Table

440 cps tone

Metronome

Tachometer

Servo motor R/C

Temperature reading devices

Clock chip interfacing

DC motor

DC Motor with encoder

Stepper motors

Data collection from a solar collector.

Part five Appendices

Equipment needs

PASM words used in text with short meanings.

Epilogue

Index



# An introduction to Propeller Assembly language

By Harprit Singh Sandhu.

## Preface

Since it did not look like any one on the Propeller discussion group was going to write an introduction to the Propeller Assembly language and I needed to learn the language I decided to put down my notes in the Monograph and make it available to the Propeller community.

My sources include the Propeller manual, the internet and a few books on digital/binary mathematics.

Basically I am going to follow the format that I followed in my book on the introduction to the spin language. First we will cover what we have to work with than we will cover basic operations, and then we will cover a project or two to use the information that we have become familiar with.

I am going to be using the professional development Board as my basic hardware configuration. I will try to minimize the use of its ancillary devices so that the cost of learning the language will be kept to a minimum.

As in any assembly language, PASM, the Propeller Assembly Language, has to do with More than anything else you need to be completely familiar with what is in the propeller manual. It will make your life easier, if it is straightforward for you to find the instruction that you want to use. That and learning how to use the data sheets, and you are on your way.

Any assembly language is all about manipulating registers, moving bits back and forth, and the setting the I/O pins. The system does everything in very small steps. You have to define everything you want the processor to do. There are a number of commands in PASM, that can be accessed directly almost like SPIN commands and these copmmands will perform certain simple, often used functions for you. These are listed in the description of the language in the second half off the propeller manual.

After you have executed the first few experiments, you will start to get a feel for a hollow these manipulations are undertaken and further experiments are designed mostly to increase your familiarity with the language and how it is used to build up simple control applications.

As time goes on, you will develop little subroutines that you can plant in your programs. When you need them or you can use as templates for other subroutines that you want to create and you will find that the work starts to go rather quickly, and the results are very fast. You will be glad that you learned how to use assembly language. It's.

PASM has a very rich vocabulary and allows you to make the decisions that go into creating a computer program with ease. However, this large vocabulary means that you do have to have access to the manual at all times. The cause, I doubt that any of us can memorize a large instruction set, unless that's all we do everyday in and out.

There is no accumulated or as such, or a Master register on which all calculations are performed in PA SM. This any register can be accessed and manipulated as you see fit. The fact that all but cogs are identical makes life as easy as one might expect.

In general, you will want to do those things that are done over and over in your program in PAS M and the major portion of the logic and output to other devices in the spin. That along with the ever increasing software, available in the object exchange, maintained by parallax makes it relatively painless to program the propeller chip. It's

## Chapter 1.

### What the manual does not tell us.

I found that the thing that made it hardest for me to get going on the project was that the manual did not tell me enough about what the microprocessor had in the way it's of internal registers. Having this information is absolutely fundamental to understanding what is going on in the language. It's this being the case, the first thing we will do is describe what we have to work with and what the various registers do.

In most microprocessors. The system consists of an array of various types over registers or memory locations if you like that form the core of the machine. Some of these registers occupy information that cannot be changed and some of them are memory locations that form the core memory off the device.

When you start the micro processor or when you press the reset button is the logic engine is designed to jump to a certain memory location. this memory location is specified in one or more registers at one end, all the memory address of the device.

In our particular case, we will be working with eight cogs and we know that each of these cog is similar to every other cog. The assembly language programs that rewrite goes into one of the calls and for all purposes as the beginners we will assume that the called contains only this assembly language programs and contains it in its entirety. So theoretically we could run seventh assembly language programs, the 8<sup>th</sup> cog being used as the controlling cog

The program itself is specified as a number of data points. The program reads in these data points, and when the program is executed these data points represent the program. As a part of this reading process, the microprocessor, completely fills the memory of the cog that we have started for this program. This is described in a little more detail in the propeller manual. See page

Figure 1 illustrates what one cog looks like two an assembly language program.

Figure 1

-----

As we go through this book, we will assume that we have a serious interest in learning PASM. This means that you will undertake all the experiments in the book and that it will not be able to jump around in the book and expect to understand what is going on.

The book will (in general) follow the format, and the programs that were delivered in my book for beginners on learning SPIN. There are a number of advantages in doing it in this way, among them the ability to compare the speed of doing what we do in the two languages.

We will assume that you have the following resources available to you as learning tools.

1. My book on spin, which you will need as a general reference.
2. The propeller manual version 1.0 and the published errata for it. Referred to as PM in this text). See Propeller Tool help tab. It would be best if you also had a hard copy of this resource in your hands.
3. The data sheet for the propeller chip. This too can be downloaded from the help menu tab in the propeller tool. Print it out also.
4. The hardware needed to run all the experiments.
5. A professional development board for the propeller. (referred to as PDB in this text)
6. A printout of the circuitry of the PDB. We will need to refer to this from time to time as we lay out our circuits.

Both printed and digital/electronic copies are useful in that the printed copy is easier to read and the digital/electronic copies are easier and much more convenient to search.

The first program that is always considered in any tutorial on any language is the blinking of an LED. (“Hello World” is more commonly used with larger logic engines but using the LED is easier with micro controllers like the propeller.) The propeller manual provides an example of this on page 340. Here is a listing of that program as provided by Parallax:

```
{{AssemblyToggle.spin}}
con
  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000

pub main
  cognew(@toggle, 0)

dat
toggle      org      0
            mov      dira, pin
            mov      time, cnt
            add      time, #9
:loop       waitcnt  time, delay
            xor      outa, pin
```



```
        jmp        #:loop

pin      long      |<1
delay    long      6_000_000
time     res       1
```

The explanations provided in the manual are not as detailed as they need to be for beginning students of the language. To remedy this, I will provide a line by line description, or if you will, documentation, for this program as the first example of proceeding with our learning process.

I will be setting up all the experiments in this book on the **propeller professional development board**. If you have the education Kit provided by parallax it will provide most of the electronic components that you need for this book. The other components that you will need, have been list in the appendix so that you can have them all one hand as you start your experimentation. It is of course not strictly necessary to use the professional development board but the board does make life a lot easier in that it has an awful lot of the devices that we will be experimenting with built right onto the board, and they are well organized and easy to use. You will find lots of used for this board as you go along. When it is on sale it is an unbeatable value.

Let's go over the first program in the propeller manual a line at the time so that we can understand exactly what the programmer had in mind as he or she wrote the program.

### **{{AssemblyToggle.spin}}**

the first line is a comment, and as such will not executed as a part of the program. This line tell us the name of the program and the fact that it is in PASM. Both SPIN and PASM programs use the .spin identification.

### **CON**

the second line identifies the following two lines as constants. These constants are the same as we see in SPIN, and they define the speed of the processor as 5 X 16 megahertz. We could have defined this as any other speed as long as we used valid values. The PM lists all the valid propeller speeds. Keep in mind that the slower we run the processor, the less energy it uses.

### **PUB Main**

The next line defines the method as a public method and provides the main handle for the program. Each program has to have at least one method within it.

### **cognew(@toggle, 0)**

Note: Cognew is a SPIN command and it does not have a PASM equivalent. All Objects run in the SPIN environment and this is the only way to start a new cog. [@toggle](#) tells the system where to locate the program in the cog memory

The line launches a new cog that will hold the program, that we are addressing, for blinking

the LED. It tells the propeller that the new program is to be installed starting at location identified by "toggle" in the memory of this cog. @toggle is short hand for the memory location in this Cog. It happens to be at location 0 in the RAM but the 0 in this line does not refer to that. That is defined later in the ORG directive.

Next we need to define what the total toggle program consists of. It is defined in the DAT data provided for this program.

The next line.

### **DAT.**

Tells us that the following lines contain the data that will be interpreted as the program by cognew the new cog that we started earlier under MAIN.

### **{toggle P16}**

This line is not a data statement. It is a comment to remind us that we will be toggle line 16 on the propeller with the following data defined program instructions. As our toggling target, we have a choice of any line from one to 31, but we may want to avoid the last four lines in that they have other uses that we may not want to interfere with at this time. We are using line 16.

### **ORG 0**

This line tells us that we are going to start storing our data points into the memory of this cog starting with the first RAM location in the cog. We have the option of starting at any memory location within the cog but at this point, there is really no good reason not to start at location zero. As a general rule almost all PASM programs will start at location 0.

### **Toggle Mov Dira, Pin**

In this line, Toggle is a marker. It marks the beginning of a object. The command MOV tells the processor to move the value of PIN into the DIRA register. On start up and reset a propeller chip has all its pins in the input mode. In this mode, they are high impedance pins that will accept data both from TTL level signals as well as from CMOS level signals.

Remember that the propeller is a CMOS device running at 3.3 Volts. The switching point between high-level and low-level signals in a 3.3 volt device is 1.65 volts or half of 3.3 volts. Further down in the program our constant will be defined as a long or 32 bit value. In this 32 bit register bit 16 is made a one and then the 32-bit number is moved into the register DIRA. DIRA defines the direction of the I/O bits as either inputs or outputs. If the corresponding bit is 0 the line will be used as an input and if it is 1 it will act as and be treated as an output.

There are 16 registers that are pre-defined in PASM. A list of these 16 registers is in the data sheet on page 34. Some of these registers can be written to, and some cannot. We will not go into which is which at this time, but you need to be aware that these registers exist and one of them (DIRA) represent the direction that the pins will be programmed to in our programs.

Before we started, (i.e. on start up and reset) DIRA was

```
00000000_00000000_00000000_00000000
```

After this above line is executed. The register DIRA will contain

```
00000000_00000001_00000000_00000000
```

This tells us that all the pins on our propeller are now inputs except line 16, which is the 17th pin because as a convention we always start counting at zero.

In this discussion, and in all subsequent discussions, we will use binary format as used above (or as convenient) so that it is easy to see which pin is set to what without having to do any mental manipulations. It is also possible to use other number bases but we will stick to binary and decimal values almost exclusively in the text.

#### **mov time cnt**

This instruction moves the current content of the (running) system clock or counter into the Time register. CNT is also one of the 16 registers that was referred to earlier as being predefined in each cog ram. This register contains the current count in the system clock and is a read only register. You cannot set the system clock, you can only read it. All cogs read the same system clock.

#### **add time, #9**

This instruction adds, the number nine to the time variable and stores the result back into the time variable. This is necessary, because there is a short delay between putting the counter into the time variable and starting the looping process. Adding the nine compensates for the delay between the two instructions. In this particular case this number can be 9 or higher. Making it higher, would detract from the accuracy of the first delay and making it lower, would make it necessary for the counter to go all the way around the 32-bit count the before it would respond in the way that we intend it to respond. (This number has to be increased if the delay between setting the wait time and the start of the loop increases.

#### **:loop waitcnt time, delay**

This is the wait instruction that determines the delays between turning pin 16 on and off in the program. This delay is based on the clock frequency that we specified under CON. The delay is 6/80 seconds as specified. The LED stays on for 6/80 seconds and then stays off for 6/80 seconds for a full cycle of 3/80 seconds. The way this works in PASM the delay is dependent on the specified speed of the system clock. (That is the way it is programed in this particular program).

#### **xor outa, pin**

This instruction inverts the signal on pin 16 each time through the loop. (XOR is the exclusive OR operation)

### **jmp #:loop**

This instruction tells the program to jump to the location (line) marked " :loop". This starts the process of converting the target line, and the delay over, and re-begins routine that blinks the LED at location PIN

We still have the business of defining our constants and telling the processor, where we want the information about them to be stored. This is done on the next three lines.

### **Pin Long |<16**

This identifies PIN as location 16 in the register.

### **Delay long 6\_000\_000**

Tells the processor that delay will be a "four byte" long with a value of 6\_000\_000 placed in it.

### **Time res 1**

Tells the system that the time variable will be located within the workspace assigned in RES 1. This is just one of the registers in the resources area. Any number of variables can be stored in the RES 1 area. All un-inialized variables are stored in the RES 1 area so in a way this is the area in which we declare the variables that we are going to use in the program.

All assignments in the RES area must be be defined at the end of the program.

## Experimentation.

Next let us make some modifications to the program and play with it to see what happens when we make changes. We will be doing this with almost all the programs we write to enhance our learning expedience.

It is best not to connect the output pin on a propeller directly to an LED. It would be best to use a resistor of between 120 and 500 ohms to limit the load on the line. The larger the value you use the dimmer the LED will be. On the PDB the LEDs already have resistors in series with them but we will add a resistor in the line just to get in the habit of limiting the LED current. Adding this resistor will not adversely affect the lighting on the LED. I used 220 ohm resistors

The circuitry we want to have in place is as shown in Figure XXX

### Figure XXX

The purpose of these changes is to play with the contents of DIRA. Generally nothing works as well as poking around in small programs, for a good learning experience.

Revise the program so that it reads as follows.

```
con
  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000

pub main
  cognew(@toggle, 0)

dat
  org      0
toggle    mov     dira,   pin
          mov     TIME,   CNT
          add     time,   #9
:loop     waitcnt time,   delay
          xor     outa,   pin
          jmp     #:loop

pin       long    %00000000_00000000_00000000_0000011
delay     long    16_000_000
time      res     1
```

We will run the program repeatedly as we change the various parameters to see what happens. Of particular interest is what we put in DIRA, the number 9, and the number 16\_000\_000. There are other things that you might want to try also (logic changes etc.) but we will not cover them right here.

Next let make some major changes to the program so that we can blink LEDs on lines 0 and 1 alternately. For this experiment set up the circuitry as shown in Figure XXX

### Figure XXX

If you are going to use an oscilloscope to look at what is going on at the pins, you can look at either line 0 or 1. You can just leave the probe there for the next few experiments.

Change the program so that it looks like the following:

```
con
```

```

_clkmode = xtall + pll16x
_xinfreq = 5_000_000

pub main
cognew(@toggle, 0)

dat
    org      0                                'start of the program storage locations
toggle mov   dira,   pin                      'pin now sets lines 0 and 1 as outputs
    mov     time,   cnt                      'sets the delay time to 10m cycles
    add     time,   #9                        'adds 9 to the time count

:loop waitcnt time,   delay                  'the wait instruction
    mov     outa,   pin0on                  'sets output 0 on and 1 off

    waitcnt time,   delay                  'the wait instruction
    mov     outa,   pin1on                  'sets output 1 on and 0 off

    jmp     #:loop                          'go back and loop.

pin      long %00000000_00000000_00000000_00000011    'used to set 0 and 1 as outputs
kine0on  long %00000000_00000000_00000000_00000001    'used to turn on line 0
line1on  long %00000000_00000000_00000000_00000010    'used to turn on line 1
delay    long 10_000_000                            'delay cycles defined
time     res 1                                       'storage location for time

```

### Program XXX Blinking lines alternately.

The above program XXX will blink the LEDs on lines 0 and 1 on and off alternately. This code is easy to read but it is archaic and it is not the best way to do it. An only slightly better way to write this program is shown next in Program XXX. Again the changes are to the looped part of the program

```

con
_clkmode = xtall + pll16x
_xinfreq = 5_000_000

pub main
cognew(@toggle, 0)

dat
    org      0                                'start of the program storage locations
toggle mov   dira,   pin                      'pin now sets lines 0 and 1 as outputs
    mov     time,   cnt                      'sets the delay time to 10m cycles
    add     time,   #9                        'adds 9 to the time count

:loop waitcnt time,   delay                  'the wait instruction
    or      outa,   %01                      'sets output 0 on and 1 off

    waitcnt time,   delay                  'the wait instruction
    or      outa,   %10                      'sets output 1 on and 0 off
    jmp     #:loop                          'go back and loop.

delay    long 10_000_000                            'delay cycles defined
time     res 1                                       'storage location for time

```

**Program XXX a slightly better way to write program XXX  
Blinking lines 0 and 1 alternately.**

When you run the above program you will see that two LEDs can be turned on and off alternately with the above register assignments. The program shows you how to set the lines as Inputs and Outputs with simple statements and how to affect the I/O operation of the pins. However this is both the tedious and archaic way of doing it. A number of better ways to do it follow. In these examples we are using shorthand notation and binary math techniques to manipulate the registers. Handling registers in this way is important in almost all the things that we will be doing. The techniques we use next use the following principles:

There are three basic things we can do to one bit in one operation.

Change it from a low to high-level (0-->1)

Change it from a high to low-level (1-->0)

Invert its state, in other words toggle the bit. Make it high if low and low if high.

Assume that the current 4 bits under consideration are %1010

We can change any **bit in this group to high** with the **or** operation on that bit.

	1010	current content
<b>or</b>	0001	addresses the last bit
	1011	result: the last bit is turned on (changed from 0 to 1)

If we want to **turn a bit off** we can do it with the **andn** operation

	1010	current content
<b>andn</b>	0010	addresses the third bit
	1000	result: the third bit is turned off (changed from 1 to 0)

If we want to **change (or toggle) the state** of a bit we use the **xor** command

	1010	current content
<b>xor</b>	0011	addresses the third and fourth bits.
	1001	result: the third and fourth bit are toggled (changed from 1 to 0 and 0 to 1)

Now let us look at the looping part of the code in our program and see how we can use the above commands to make the LEDs flash alternatively.

The simplest way is to **first set the two bits** that control the LEDs to 01 or 10 with the **outa** command and then do a toggling procedure with the **xor** command.

```

dat
    org        0                'start of the program storage locations
toggle mov     dira,    #%11    'pin now sets lines 0 and 1 as outputs
    mov     time,    cnt        'sets the delay time to 10m cycles
    add     time,    #20        'adds 12 to the time count
    mov     outa,    #%01        'sets the two bits
    
```

```
:loop waitcnt time, delay
      xor outa,  #%11      'toggles bits 0 and 1
      jmp  #:loop
```

Now that we understand the simplest of bit manipulations and the creation of the simplest of loops. Let us expand on these idea to learn how to undertake some other often used techniques.

### Shifting the bits in a register left and right.

The two instructions used to shift bits left and right are **SHL** and **SHR**. Bits can be shifted from 1 to 32 places within the 32 bit registers.

If we use a bit shifting technique to move the ON bit back and forth to alternate the coming on of the two bits we have been considering above, the data part of program would look like the listing shown in Program XXX. Here we have to wait after each shift to duplicate the effect in program XXX above.

```
dat
      org      0          'start of the program storage locations
toggle mov  dira,  #%11   'pin now sets lines 0 and 1 as outputs
      mov  time, cnt      'sets the delay time to 10m cycles
      add  time, #20      'adds 12 to the time count
      mov  outa,  #%01    'sets the two bits
:loop waitcnt time, delay 'delay
      shl  outa,  #%1     'shift left one bit
      waitcnt time, delay 'delay
      shr  outa,  #%1     'shift back right one bit
      jmp  #:loop
```

### Program XXX

#### Controlling off the LEDs by shifting the active bit left and right.

Tony's code

CON

```
_clkmode = xtall + pll16x      'Standard clock mode * crystal frequency = 80
MHz
_xinfreq = 5_000_000
```

```
PUB main
  cognew(@entry,0)
```

DAT

```
entry  mov dira, #%11      'pin p0 and p1 as output
      mov time, #5        'first time we only want to wait 1 clock(+ 4 for
the overhead)
      add time, cnt       'add current counter value to time
```



```

loop    waitcnt time, delay    'wait until counter matches time, when done add
delay to time
        mov outa, #%01        'set pin p0 high, p1 low

        waitcnt time, delay    'wait, time had 15million added to it from above
waitcnt opcode
        mov outa, #%10        'set pin p0 low, p1 high
        jmp #loop            'jmp to loop, don't forget the # sign

delay long 15_000_000        '15 million clock cycles
time res 1

```

## Creating Subroutines/Methods

PASM has the waitcnt command for creating pauses but we are going to ignore that in the immediate discussion.

First let us see how we call a subroutine in PASM. We will make the wait a part of a subroutine and then call the subroutine whenever we need to wait. This needs to be done after each shift command. The complete code for this is

```

con
_clkmode = xtall + pll16x
_xinfreq = 5_000_000

pub main
cognew(@toggle, 0)

dat
org 0 'start of the program storage locations
toggle mov dira,  #%11 'pin now sets lines 0 and 1 as outputs
        mov time,  cnt  'sets the delay time to 10m cycles
        add time,  #200 'adds 12 to the time count
        mov outa,  #%01 'sets the two bits

:loop
        call #clkdelay 'call the delay subroutine
        shl outa,  #%1  'shift left one bit
        call #clkdelay 'call the delay subroutine
        shr outa,  #%1  'shift back right one bit
        jmp #:loop

clkdelay waitcnt time, delay 'the delay subroutine
clkdelay_ret ret 'return for delay subroutine

delay long 10_000_000 'delay cycles defined
time res 1 'location for time

```

### Program XXX

Places the waitcnt command in a subroutine.

Often we need to be able to do something a fixed number of times and then do something else. In order to do this we need to learn how to set up counters. Let us use a counter that uses waits of one quarter (0.25 seconds) repeatedly to make up the delays we need in our programs from time to time. The wait period will need to run through 80\_000\_000\_/4 cycles of an empty loop before exiting. We will then place this method in our blink routine to make sure it works.

Again: our clock is running at 80 MHz, so one second takes 80\_000\_000 cycles and 0.25 seconds take 20\_000\_000 cycles. We also know that the average instruction takes 4 clock cycles. So our 0.25 second subroutine has to have 5\_000\_000 iterations through its loop. (Our counts are not exact because we are not counting every instruction but it will be close enough for what we are trying to learn at this time. We will learn to count exact cycles later on in the book)

```

con
_clkmode = xtall + pll16x
_xinfreq = 5_000_000

pub main
cognew(@toggle3, 0)

dat
toggle3  org      0                'start of the program storage locations
         mov      dira,   %#11     'pin now sets lines 0 and 1 as outputs
         mov      outa,   %#01     'sets the two bits

:loop
         call     #clkdelay        'call the delay subroutine
         shl     outa,   #1        'shift left one bit
         call     #clkdelay        'call the delay subroutine
         shr     outa,   #1        'shift back right one bit
         jmp     #:loop

clkdelay mov      time,  deltime    'the delay subroutine, load deltime into toime
take4    sub      time,  #1         'internal to subroutine flag
         if_nz   jmp      #take4    'sub 1 from time and set flag if 0
         if_nz   jmp      #take4    'if flag not 0 go back to take4
clkdelay_ret  ret                'return for delay subroutine

deltime  long    5_000_000        'time of delay
time     res     1                'location for time

```

### Program XXX

Subroutine creation and use for 0.25 second delay.

+++++

))))))

### Passing a variable from PASM to SPIN

Understanding the passing of a variable between SPIN and PASM with the PAR command.  
(Read Page 283 of PM slowly.)

You can pass a variable from a PARM method to a SPIN method with the PAR command. The best way to do this is with a long variable or 32 bits.

Let us set a variable named “value” in binary notation to

```
%10101010_11111110_11110111_10111011
```

The pattern is made different for each of the four bytes so that we can see this reflected in the output of the display as we display each of the 4 bytes as eight LEDs. The original code as posted on the discussion forum for this book by Kuroneko (#57)

```
PUB null | shared
  cognew(@generate, @shared)      ' start incrementer
  dira[16..23]~~                 ' all LEDs are outputs
  repeat                          ' endless loop
    outa[16..23] := shared.byte[2] ' display byte 2 only

DAT      org      0
generate  add      temp, #1        ' simply increment our value
          wrlong   temp, par      ' share with SPIN cog
          jmp      #generate      ' endless

temp     res      1                ' temporary (yes, blindingly obvious)
```

Kuroneko's code is designed to run on the demonstration board for the propeller as posted. I have modified it (program XXX) to run on the PDB to match the work we are doing in all the experiments. As modified below it displays the selected byte on pins 0..7 of the PDB. These pins are connected to lines 0..7 of the propeller.

The long is transferred as 4 bytes. The 4 bytes are referred to as Byte[0] to Byte[3] in the output of the SPIN routine because we are using only 8 LEDs to display the data. Byte [0] is the least significant byte and is on the right end (the least significant end) in the notation shown.

Value is a random with the binary digits in each bit in a different order so that you can easily see the changes as you run the program with different value in the byte subscript.

```
PUB null | Pot_Value
  cognew(@generate, @pot_Value)  ' start incrementer
  dira[0 ..7]~~                  ' all LEDs are outputs
  repeat                          ' endless loop
    outa[0..7] := pot_Value.byte[1] ' displays byte 1 only

DAT      org      0
generate  mov      temp, value
```

```
wrlong temp, par
jmp generate

value long %1010_1010_1110_1111_1111_0111_1011_1011
temp res 1 ' temporary (yes, blindingly obvious)
```

**Program XXX**  
**Variable shared between SPIN and PASM**  
**Created in PASM and moved to SPIN in this case**

Run the above program to see how the information is read with the SPIN code. Vary the Byte value from 0 to 1 to see all the bits.

We can use the above code to display the value of the potentiometer we are going to read once we have learned to read a potentiometer..

**Reading a potentiometer.**

We will read the potentiometer with the MCP 3208 A to D chip. This next section is about how to read the information from this chip.

In a lot of our experiments we will be using the input from a potentiometer or two to vary the inputs to the program we are experimenting with. The best way to read a couple of potentiometers is by using a chip that is dedicated to doing just that. The Microchip Technologies MCP- 3208 will be discussed in the following experiment. This 16 pin chip is configured as follows

Line 1	Channel 0	Wiper of the 1st potentiometer
Line 2	Channel 0	Wiper of the 2nd potentiometer
Line 3	Channel 0	Wiper of the 3rd potentiometer
Line 4	Channel 0	Wiper of the 4th potentiometer
Line 5	Channel 0	Wiper of the 5th potentiometer
Line 6	Channel 0	Wiper of the 6th potentiometer
Line 7	Channel 0	Wiper of the 7th potentiometer
Line 8	Channel 0	Wiper of the 8th potentiometer
Line 9	Ground	Ground
Line 10	Chip select	Pin 24 of the propeller, made an output
Line 11	Data in	Pin 25 of the propeller, made an output
Line 12	Data out	Pin 26 of the propeller, made an <b>input</b>
Line 13	Clock	Pin 27 of the propeller, made an output
Line 14	Ground	Reference Ground
Line 15	V Ref	Reference Volts (5 Volts)
Line 16	5 Volts	5 Volts power

The circuit for using this chip is shown in Figure XXX

Figure XXX  
Circuitry for reading a MCP 3208

The sequence that you have to follow to read this chip is illustrated in an easy to read diagram on the data sheet (See page 16) . Download this data sheet, print this page, and have in in front of you as you follow the program provided in Program XXX below.

Program XXX  
Reading channel 0 of the MCP 3208

Reading the potentiometer consists of the following steps:

1. Selecting the chip with the Chip Select line. High to low.
2. Clocking in the parameters needed to identify what you want on the Data In line.5 cycles
3. Toggling the **Clock** bit to read the A to D value to 12 bits (0 to 4095). 12 cycles

At the end of the process you will have the reading in a designated register from where it can be used as need be. The routine needs to repeat in its own cog so that the A to D value is available to the rest of the cogs at all times. In our case the value read will be stored in the variable Pot\_Value.

## Circuitry

Reading the data sheet specifies that in order to read the data from the 3208 we need the following wiring connections:

Start with setting up the 4 control lines going into the chip as

	3208	Propeller pin
Make <b>Chip select</b> and output from the propeller	pin 10	24
Make <b>Data in</b> and output from the propeller	pin 11	25
Make <b>Data Out</b> and input into the propeller	pin 12	26 input
Make <b>Clock</b> and output from the propeller	pin 13	27

Since we are using lines 24 to 27 to communicate with the 3208 we can need to set the up as I/O lines. On start up, all the lines are inputs so we have to set up the output lines only. We can do this with the **mov** command

```
mov dira |<24
mov dira |<25
      line 26 is already an input line into the 3208 and does not have to be set.
mov dira |<27
```

The chip select line is left high when the chip is not being used.

To select the 3208 make the **chip select** line low. It can be left low until all the reading is done and we are ready to make the chip dormant again. It uses less power when dormant.

Clear the register that you are going to read the data to.

Start process with making chip select low and then follow the data sheet diagram.

Data is transferred when the Clock goes from high to low so set the condition you want on the **Data in** line and then make the Clock low.

The sequence will be as follows:

```
outa[chipClk]~      'START BIT Clock needs to be low to load data
outa[chipDin]~~    'must start with Din high to set up 3202
outa[chipClk]~~    'Clock high to read data in

outa[chipClk]~      'SINGLE DIFFLow to load
outa[chipDin]~~    '1111 High single diff mode
outa[chipClk]~~    'High to read

outa[chipClk]~      'D2 Low to load
outa[chipDin]~     '0000 low channel 0
outa[chipClk]~~    'High to read
```

```

outa[chipClk]~      'D1 Low to load
outa[chipDin]~     '0000 low channel 0
outa[chipClk]~~    'High to read

outa[chipClk]~     'D0 Low to load
outa[chipDin]~     '0000 msbf high = MSB first
outa[chipClk]~~    'High to read

```

```

outa[chipClk]~
outa[chipClk]~~

```

```

outa[chipClk]~     'Low to load Read the null bit, not stored
outa[chipDin]~     'NULL making line low for rest of cycle
outa[chipClk]~~    'High to read

```

```

DataRed:=0         'Clear out old data
repeat Bitsread    'Reads the data into DataRed in 12 steps
  DataRed <=<= 1   'Move data by shifting left 1 bit. Ready for next bit
  outa[chipClk]~  'Low to load
  DataRed:=DataRed+ina[chipDout] 'Xfer the data from pin chipDout
  outa[chipClk]~~ 'High to read
outa[chipSel]~~   'Put chip to sleep, for low power
Pot0:=DataRed     'Finished data read for display
result:=datared

```

```

PUB null | P_Val
  cognew(@generate, @P_Val)      ' start new cog at "generate" and read variable at P_Val
  dira[0 ..11]~~                ' all 12 lines are outputs. 12 lines needed for 1.5 bytes
  repeat                          ' endless loop to display data
    outa[0..11] := P_Val         ' displays 1.5 bytes of data

```

```

DAT      org      0      'sets the starting point in Cog
generate mov      dira,  set_dira  'sets direction of the prop pins
         call     #chip_sel_lo    'selects chip by pulling line low
         call     #Clk_lo         'START. Clock needs to be low to load data
         call     #Din_hi        'must start with Din high to set up 3208
         call     #Tog_clk       'clk hi to read data

         call     #Din_hi        'SINGLE DIFF Low to load
         call     #Tog_Clk       'toggle clock line hi then low to read in the data

         call     #Din_lo        'D2 Low to load input line selection sequence 000 for line 0
         call     #Tog_Clk       'toggle clock line hi then low to read in the data

         call     #Din_Lo        'D1 Low to load input line selection sequence 000 for line 0
         call     #Tog_Clk       'toggle clock line hi then low to read in the data

         call     #Din_Lo        'D0 Low to load input line selection sequence 000 for line 0
         call     #Tog_Clk       'toggle clock line hi then low to read in the data

```

```

        call    #Din_lo      'blank bit needs a clock cycle, next
        call    #Tog_Clk    'toggle clock line hi then low to read in the data

                                'next toggle is for the null bit, nothing read
        call    #Tog_Clk    'toggle clock line hi then low to read in the data

        mov     dat_red, #0  'Clear register we will read data into
        mov     count,  #12 'Counter for number of bits we will read
read_bit  mov     temp1,  ina   'read in what is in all the input lines
        andn   temp1,  inputmask wz 'mask off everything except Dout line. Set Z flag
        if_nz  add     Dat_red, #1 'if value is still positive add 1 to data register
        ror    Dat_red, #1   'roll register right 1 bit to get ready for next bit
        call   #Tog_Clk      'toggle clock to get next bit ready in Dout
        sub    count,  #1 wz  'decrement the "bits read" counter. Set Z flag
        if_nz  jmp     #read_bit 'go up and do it again if counter not 0
        rol    dat_red, #12  'roll back 12 bits to get data into 12 LSBits of register
        mov    temp,    dat_red 'get data that as read
        wrlong temp,    par    'put it in PAR to share it as P.Val
        call   #Chip_Sel_Hi 'Put chip to sleep by deselecting it, for low power usage
        jmp    #generate     'go back to do it all again

'Subroutines
Clk_Hi    mov     temp,    outa    'Get the OUTA register
        or     outa,    clk_bit  'OR it with the Clock Bit to male high
        mov    outa,    temp     'put it back in OUTA register
Clk_Hi_ret    ret

Clk_Lo    mov     temp,    outa    'Get the OUTA register
        andn   temp,    clk_bit  'ANDN it with the Clock Bi to make lowt
        mov    outa,    temp     'put it back in OUTA register
Clk_Lo_ret    ret 'return from this subroutine

Tog_Clk   call    #Clk_hi      'make clock bit high
        call    #clk_lo      'make clock bit low
Tog_Clk_ret    ret 'return from this subroutine

Din_Hi    mov     temp,    outa    'Get the OUTA register
        or     temp,    din_Bit  'Makes the Din high
        mov    outa,    temp     'put it back in OUTA register
Din_Hi_ret    ret 'return from this subroutine

Din_Lo    mov     temp,    outa    'Get the OUTA register
        andn   temp,    din_Bit  'makes Din low
        mov    outa,    temp     'put it back in OUTA register
Din_Lo_ret    ret 'return from this subroutine

Chip_Sel_Hi    mov     temp,    outa    'Get the OUTA register
        or     temp,    chs_Bit  'Makes Chip select high
        mov    outa,    temp     'put it back in OUTA register
Chip_Sel_Hi_ret    ret 'return from this subroutine

Chip_Sel_Lo    mov     temp,    outa    'Get the OUTA register
        andn   temp,    chs_Bit  'makes chip select low
        mov    outa,    temp     'put it back in OUTA register
Chip_Sel_Lo_ret    ret 'return from this subroutine

Read_Dout  mov     temp,    ina    'Get the INA register
Read_Dout_ret    ret 'return from this subroutine

```



```

Read_Next_Bit mov     temp1, ina      'Get the INA register
               or     temp1, inputmask 'mask all but Din bit
Read_Next_Bit_ret ret              'return from this subroutine

'Constants. This section is similar to the CON block in SPIN
Set_dira      long    %00001011_00000000_00000000_00000000 'Set dira register
Chs_Bit       long    %00000001_00000000_00000000_00000000 'Chip select bit      24
Din_Bit       long    %00000010_00000000_00000000_00000000 'Data in bit          25
Dout_Bit      long    %00000100_00000000_00000000_00000000 'Data out bit         26
Clk_Bit       long    %00001000_00000000_00000000_00000000 'Clock bit            27
inputmask     long    %11111011_11111111_11111111_11111111 'Mask for reading the Dout bit only

'Variables. This section is similar to the VAR block in SPIN
temp          res     1      'temporary storage variable
temp1         res     1      'temporary storage variable
count         res     1      'temporary storage variable
Dat_Red       res     1      'temporary storage variable

```

```

Routine to read a bit
put 12 in counter
Read label
recall ina, |<27 ls 1 add 1 to value
if 0 add z to value call add 0
If 1 add 1 to value call add 1
shift value left one
Sub 1 from counter
if not zero Go to read
put chip to sleep
put value in Par
go to beginning of routine to read again

```

Improvements to the pot reader program.

We can make the program considerably shorter by eliminating some instructions that we added to make it easier to read the program. In the routine that set the bits, the 4 lines can be reduced to 2 as follows

```

Chip_Sel_Lo  mov     temp,  outa      'Get the OUTA register
              andn   temp,  chs_Bit  'makes chip select low
              mov     outa,  temp     'put it back in OUTA register
Chip_Sel_Lo_ret ret              'return from this subroutine

```

this reduces to

```

Chip_Sel_Lo  andn   outa,  chs_Bit  'makes chip select low
Chip_Sel_Lo_ret ret              'return from this subroutine

```

We can do this because we can access the OUTA register directly without having to go through a temp register. OUTA can be addresses just like any other register.

In the early part of the program we are reading channel1 which is specified with 000 in D0, D1 and D2. We specify this by making D0, D1, D2 (on to the null bit) low and toggling them all in wit the Clock bit. Because the Dout line is low in all these transfers we can eliminate a number of lines by toggling just the Clock bit shown below

#### Original code

```

call    #Din_lo    'D2 Low to load input line selection sequence 000 for line 0
call    #Tog_Clk   'toggle clock line hi then low to read in the data

call    #Din_lo    'D1 Low to load input line selection sequence 000 for line 0
call    #Tog_Clk   'toggle clock line hi then low to read in the data

call    #Din_lo    'D0 Low to load input line selection sequence 000 for line 0
call    #Tog_Clk   'toggle clock line hi then low to read in the data

call    #Din_lo    'blank bit needs a clock cycle, next
call    #Tog_Clk   'toggle clock line hi then low to read in the data

call    #Tog_Clk   'next toggle is for the null bit, nothing read
                'toggle clock line hi then low to read in the data

```

#### Reduced code

```

call    #Din_lo    'D2 Low to load input line selection sequence 000 for line 0
call    #Tog_Clk   'toggle clock line hi then low to read in the data
call    #Tog_Clk   'toggle clock line hi then low to read in the data
call    #Tog_Clk   'toggle clock line hi then low to read in the data
call    #Tog_Clk   'toggle clock line hi then low to read in the data
call    #Tog_Clk   'toggle clock line hi then low to read in the data

```

Keep in mind that when we read channels 2 to 8 we will have to put their address in D0, D1 and D2 and will not be able to eliminate all the lines like we did above.

### Creating pulses.

As is our goal we want to do this using as little SPIN as possible. We will still read and display the P\_Val variable in SPIN so we can see what is going on but the generation of the pulses will be in a third Cog in PASM. We will need an oscilloscope to look at the signal that we are creating.

We need to learn about the generation of pulses at different frequencies. Pulses in all sorts of sequences are needed to send information back and forth between various semiconductor devices. To start with let us keep in simple. A fixed pulse with a short pulse length will be repeated at different frequencies controlled by a potentiometer.

Lets write a short program to output our series of pulses. The pseudo code for doing this is as follows

```

Start the program in SPIN in Cog0
repeat

```

display the potentiometer on the LEDs

Cog two

Repeat

read the potentiometer @generate

Cog 3

repeat

toggle the pin

delay based on the pot reading

We will need a delay routine that uses the potentiometer reading. We can use the following lines placed in the appropriate places in our code to provide this delay.

```
waitcnt time, delay 'the wait instruction
```

and in the constants section

```
delay long P_Val 'delay cycles defined
```

The signal will be designed to appear on Pin 12, the first pin after the 0..11 pin LED display.

=====

I know this is not quite right but...

Why does this not open up a third cog and toggle line 12

It compiles but I get a dead line 12

In other ;words how do you start two PASM cogs

[code]

```
PUB null | P_Val, Delay_Val
  cognew(@generate, @P_Val) ' start second cog at "generate"
  cognew(@Toggle, @Delay_Val) 'start third cog at toggle
  repeat
    'do something unrelated to other cogs
DAT
Org 0
generate mov outa, set_dira
          jmp #generate

org 30
toggle mov dira, set_dira
        mov outa, pin_hi
        mov outa, pin_lo
        jmp #toggle

Set_dira long %00000001_00000000_00010000_00000000 'Set dira register
Pin_Hi long %00000000_00000000_00010000_00000000
Pin_Lo long %00000000_00000000_00010000_00000000
        jmp #toggle
```

[/code]

```

=====

CON
_clkmode = xtal1 + pll16x
_xinfreq = 5_000_000

PUB null | P_Val
  cognew(@generate, @P_Val)      ' start new cog at "generate" and read variable at P_Val
  cognew(@Toggle, 0)            ' start new cog at Toggle
  =====
  =====COG 1 works fine=====
  =====
  dira[0 ..11]~~                ' all lines are outputs except 26
  repeat                          ' endless loop to display data
    outa[0..11] := P_Val        ' displays 1.5 bytes of data

DAT
  =====
  =====COG 2 works fine=====
  =====
  org 0                          ' sets the starting point in Cog
generate
  mov      dira, set_dira        ' sets direction of the prop pins
  call    #chip_sel_lo          ' selects chip by pulling line low
  call    #Clk_lo               ' START. Clock needs to be low to load data
  call    #Din_hi               ' must start with Din high to set up 3208
  call    #Tog_clk              ' clk hi to read data

  call    #Din_Hi               ' SINGLE DIFF Low to load
  call    #Tog_Clk              ' toggle clock line hi then low to read in the data

  call    #Din_Lo               ' D2 Low to load input line sequence 000 for line 0
  call    #Tog_Clk              ' toggle clock line hi then low to read in the data

  call    #Din_Lo               ' D1 Low to load input line sequence 000 for line 0
  call    #Tog_Clk              ' toggle clock line hi then low to read in the data

  call    #Din_Lo               ' D0 Low to load input line sequence 000 for line 0
  call    #Tog_Clk              ' toggle clock line hi then low to read in the data

  call    #Din_Lo               ' blank bit needs a clock cycle, next
  call    #Tog_Clk              ' toggle clock line hi then low to read in the data

  call    #Tog_Clk              ' next toggle is for the null bit, nothing read
                                ' toggle clock line hi then low to read in the data

  mov     dat_red, #0           ' Clear register we will read data into
  mov     count, #12            ' Counter for number of bits we will read
read_bit
  mov     temp, ina             ' read in what is in all the input lines
  andn   temp, inputmask wz    ' mask off everything except Dout line. Set Z flag
  if_nz  add  Dat_red, #1       ' if value is still positive add 1 to data register
  ror    Dat_red, #1           ' roll reg rt 1 bit to get ready for next bit
  call   #Tog_Clk              ' toggle clock to get next bit ready in Dout
  sub    count, #1 wz          ' decrement the "bits read" counter. Set Z flag
  if_nz  jmp  #read_bit        ' go up and do it again if counter not 0
  rol    dat_red, #12          ' roll back 12 bits = 12 LSBits of register
  wrlong dat_red, par          ' put it in PAR to share it as P.Val
=====

```

```

        call    #Chip_Sel_Hi      'Put chip to sleep , for low power usage
        mov     temp2, dat_red    'get data that as read
        jmp     #generate        'go back to do it all again

'Subroutines
Clk_Hi      mov     temp,    outa    'Get the OUTA register
           or      temp,    clk_bit 'OR it with the Clock Bit to male high
           mov     outa,    temp   'put it back in OUTA register
Clk_Hi_ret  ret

Clk_Lo      mov     temp,    outa    'Get the OUTA register
           andn   temp,    clk_bit 'ANDN it with the Clock Bi to make lowt
           mov     outa,    temp   'put it back in OUTA register
Clk_Lo_ret  ret

Tog_Clk     call    #Clk_hi      'make clock bit high
           call    #clk_lo      'make clock bit low
Tog_Clk_ret ret

Din_Hi      mov     temp,    outa    'Get the OUTA register
           or      temp,    din_Bit 'Makes the Din high
           mov     outa,    temp   'put it back in OUTA register
Din_Hi_ret  ret

Din_Lo      mov     temp,    outa    'Get the OUTA register
           andn   temp,    din_Bit 'makes Din low
           mov     outa,    temp   'put it back in OUTA register
Din_Lo_ret  ret

Chip_Sel_Hi mov     temp,    outa    'Get the OUTA register
           or      temp,    chs_Bit 'Makes Chip select high
           mov     outa,    temp   'put it back in OUTA register
Chip_Sel_Hi_ret ret

Chip_Sel_Lo mov     temp,    outa    'Get the OUTA register
           andn   temp,    chs_Bit 'makes chip select low
           mov     outa,    temp   'put it back in OUTA register
Chip_Sel_Lo_ret ret

Read_Dout   mov     temp,    ina     'Get the INA register
Read_Dout_ret ret

Read_Next_Bit mov    temp,    ina     'Get the INA register
           or      temp,    inputmask 'mask all but Din bit
Read_Next_Bit_ret ret

'Constants. This section is similar to the CON block in SPIN
Set_dira    long    %00001011_00000000_00000000_00000000 'Set dira register
Chs_Bit     long    %00000001_00000000_00000000_00000000 'Chip select bit      24
Din_Bit     long    %00000010_00000000_00000000_00000000 'Data in bit          25
Dout_Bit    long    %00000100_00000000_00000000_00000000 'Data out bit         26
Clk_Bit     long    %00001000_00000000_00000000_00000000 'Clock bit            27
inputmask   long    %11111011_11111111_11111111_11111111 'Mask for read the Dout bit only

'Variables. This section is similar to the VAR block in SPIN
temp        res     1      'temporary storage variable
temp2       res     1      'temporary storage variable
count       res     1      'temporary storage variable
Dat_Red     res     1      'temporary storage variable

```

```

=====
=====COG 3 can't move temp2 into variable effectively=====
=====
toggle      org      0          'begin a 0
toggle      mov      dira, pin_12 'set pin 12 as output

toggle1     mov      temp1, outa 'read in outa
toggle1     or       temp1, pin_12 'make pin 12 lo
toggle1     mov      outa, temp1 'put it back in outa
toggle1     call     #clkdelay 'call the delay subroutine

toggle1     mov      temp1, outa 'read in outa
toggle1     andn    temp1, pin_12 'make pin 12 high
toggle1     mov      outa, temp1 'put it back in outa
toggle1     call     #clkdelay 'call the delay subroutine

toggle1     jmp      #toggle1 'repeat

clkdelay    mov      time, temp2 'the delay subroutine,load TEMP2 into time ACTS LINE TEMP2=0
clkdelay    add      time, #$ff 'delay added to skip past underflow of CNT
dloop      sub      time, #1 wz 'sub 1 from time and set flag if 0
            if_nz jmp      #dloop 'if flag not 0 go back to take4
clkdelay_ret ret          'return for delay subroutine

'Constants. This section is similar to the CON block in SPIN
Pin_12     long      %00000000_00000000_00010000_00000000
del_time   long      400
'Variables. This section is similar to the VAR block in SPIN
delttime2  res      1      'temporary storage variable
temp1      res      1      'temporary storage variable
time       res      1      'temporary storage variable

```

## PAR Instruction

The purpose of the PAR instruction is to allow SPIN Cogs to communicate with PASM cogs by addressing the same variables. (Yes more than one variable can be shared.)

### Here is how the Par instruction works

Par is one of the 16 SPRs (special purpose registers) in each active Cog in the propeller.

When you start a cog with Cognew or Coginit a designated value is passed to the PAR register. This designated value is the address of the variable Shared in the cognew statement as shown below

Cognew (@Psmcode, @Shared)

PAR is created by the initial SPIN code when the PASM Cog is started and is used by the PASM code to address the shared variable.

Let us look as a typical situation where we start and run a SPIN Cog. ;No PASM cog or cog to start with.

You can actually run this code on your computer to follow along

```

VAR
  long Shared

PUB Main
  dira[0]~~
  shared:=10
  repeat
    !outa[0]
    waitcnt(clkfreq/shared,cnt)

```

Here shared has a value of 10 and is used to control the rate at which pin 0 blinks. The rate is 10 times a second. So far there is nothing fancy about this. There is no PAR interaction, there is not second Cog.

Now let us start **Cog\_1** a PASM Cog that blinks 4 LEDs at a different rate but is completely independent from **Cog\_0** the starting Cog. We will connect their operation together later

Here is the program with the code added

```

VAR
  long Shared

PUB Main
  dira[0]~~
  shared:=15
  cognew(@newCog, @Shared)
  repeat
    !outa[0]
    waitcnt(clkfreq/shared+cnt)

DAT
Org 0
newCog
  mov  dira, pin_0_7      'sets up first 8 line I/O
  mov  400, PAR
:loop
  or   outa, pins        'turn on pin
  call #pause
  andn outa, pins        'turn off pin
  call #pause
  jmp  #:loop            'loop again

pause
  mov   time, DELAY      'the delay subroutine
  add   time, #$f        'delay added to skip past underflow of CNT
delay_loop
  sub   time, #1 wz      'sub 1 from time and set flag if 0
  if_nz jmp #delay_loop  'if flag not 0 go back
pause_ret
  ret

```

```

pin_0_7    long %00000000_00000000_00000000_11110001    'sets OUTAA
pins       long %00000000_00000000_00000000_11110000    'on-off mask using OR and ANDN
delay      long 800000>>1
mem        res 1
time       res 1

```

PAR is the address or a variable/register/parameter. It is the address of the parameter that is passes to the PASM routine when it is started. On out case, PAR is the address to the Shared variable. If we want to read the shared variable we read it with

```
Mov Read_value, PAR
```

If we stick with a 4 (0 to 12) bit variable, we can display the variable on the LEDs that we have connected to the pins of the propeller. We will write to the variable in the SPIN routine in the bits 0-3 and read it in PARM routine and display them at 4-7

=====

Using a debugger.

The general consensus is that the debugger of choice is the PASD debugger by the German group Insonics. The software was written by Andy Schenk and Eric Moyer. It is down loadable from the web site at

[www.insonix.ch/propeller/prop\\_pasd.html](http://www.insonix.ch/propeller/prop_pasd.html)

There is no charge. The manual I available both in English and in German.

Download the debugger and start reading the manual while we put together our first PAM program.

In the propeller system, all programs reside within a SPIN shell. Even a 100% PASM needs to be called from a SPIN instruction that starts the Cog the PASM instructions the executed in and tells the system where to load the program within the target Cog. If any constants will be used or if other OBJECTs will be called by the program, they too are called out in the SPIN part of the program. The program itself is defined as a set of DAT (data) statements that are the PASM instructions that will make up the program.

A typical program might look like this

```

{{
*****
* The first thing we need is a good      *
* description of what the program does, *
* who wrote it and when.                *
* Terms of use if any. Copyrights etc   *
*****
}}
CON

```



```
..Set the clock speed parameters here
  List the constants here
```

```
OBJ
  lists the objects to be use here
```

```
PUB ProgName (sample)
  cognew(@program_Loc, variable) 'launch assembly program in a COG
  display of variables is done here in SPIN
```

```
DAT
org
```

The body of the PASM program goes here  
 It is almost always a loop that need to  
 do something very fast and provide a  
 result in a variable defined earlier as  
 a part of the cognew statement.

Next let us fill in the above program so that we have a working program that we can follow with the debugger. We want to make the program as simple as possible or now so that there are no logical manipulations that are hard to follow

The program will be designed to turn count from 1 to 10 over and over again and display the number every 1/4 second. Here is a listing of the program. You can copy this program and run it to watch its operation. It is described after the listing.

```
{
*****
* Program to turn and LED on and of.      *
* Test program for introducing the use    *
* of a debugger.                          *
* Harprit Sandhu    02 Aug '11            *
* MIT license terms apply.                *
*****
}}
CON
  _clkmode = xtall + pll2x
  _xinfreq = 5_000_000

VAR
  long count, old_count

OBJ
  fds : "FullDuplexSerial"

PUB count_1to100
  fds.start(31,30,0,115200) 'start console at 115200 for debug output
  cognew(@counter,@count) 'start PASM routine in its own cog
  waitcnt(clkfreq/20+cnt) 'to let everything start up and stabilize
  old_count:=0 'set the initial value of the old counter to 0
  repeat
    if(old_count)==count 'print loop
      'check to see if we have a new value
      'if not we do not print value to console
    else 'if so we have to print to the console
      fds.dec(count) 'print value
      fds.tx(" ") 'print a separating space
```

```

        if count==maximum_value  'check to see if we have reached 100
        fds.tx($d)                'new line
        fds.tx($d)                'new line
        old_count:=count          'remember the value as the old value

DAT
counter    org    0                'start at location 0
add_one    mov    current_count, #0 'put a 0 into the counter
          add    current_count, #1 'add 1 to the counter
          call   #delay            'delay to allow print routine to catch up
          'there is a minimum value that is needed for
          'the print routine to get done before proceeding
          wrlong current_count, par 'write the value into the PASM/SPIN shared long
          sub    current_count, maximum_value wz 'subtract the maximum value
          'to be printed and set Z flag
          if_z   jmp    #counter    'if the answer is 0 we are done and start over
          add    current_count, maximum_value 'add the max value back in
          jmp    #add_one         'go back and keep subtracting 1s

delay      mov    delay_counter, delay_value 'load the delay counter
redo       sub    delay_counter, #1        wz 'subtract 1 and set zero test value
          if_z   jmp    #delay_ret      'if it is 0 we are done so return from sub
          jmp    #redo                'if not keep subtracting
delay_ret  ret

delay_value long    6000           'delay value has to be long enough for print
maximum_value long    100         'max value can be anything above 1

current_count res    1             'define variable
delay_counter res    1             'define variable

```

The program is divided into two components, the SPIN method and the PASM method. The counting takes place in the PASM method and the displayed is done in the SPIN method. Since the PASM counting routine is much faster than the ability of the SPIN code to output the values to the console, a delay (of 6000 loops) has to be added in the PASM routine to slow things down. When things are slowed down, the possibility exists that the print routine will print a value more than once. In order to avoid this the print routine makes sure that the old value and the new value are not the same before printing the value.

It is worth the time to vary the delay\_value and observe what happens when it gets too small.

Next we will look at the program with the PASD debugger.

=====posted=====

### Debugging with PASD

The program is designed so that it is easy to add errors into the code. The two main error sources are the timing in the PASM code and the inability of the SPIN code to keep up with the much faster PASM code. The two have been reconciled as listed the last time but let us modify some parameters to make the program error prone to see what happens and then see if we can find the errors with the debugger. The delay routine is also easily modified so that errors can be introduced into it as well.

We may also find that the debugger is not very good at finding timing problems! Things

change when we are not running in real time.

Let us start over.

Just like my spin book. My PASM book will be for bonafide beginners.

The book will be divided into three sections similar to the four sections of my spin book.

The general outline of the book. The basics, we need to understand PASM, and what we had to work with

Basic input.

Basic output.

Building simple projects.

Relelvant appendices

The end product will be a basic understanding of PASM and the ability to write simple programs that will be adequate to control small projects.

Any cog that runs PASM code has to be started from within a cog that are is running SPIN. Here are the few lines of codes needed to start a cog running assembly language. We will be using two commands in the program

The first command is the mov command. Mov moves data between registers. It moves date from the second register mentioned to the first.

```
[code]
    mov into, from
```

```
[/code]
```

will move whatever is in the "from" register the "into" register. All registers are game.

The jmp instruction directs program flow. Its lets you jump to any marked location within the program.

```
[code]
VAR
    long shared_var
```

```
PUB Object_name
cognew(@Prog_start, @shared_Var)
repeat
```

```
DAT org 0
Prog_start
do_again      mov  dira, init_dira
               jmp  #do_again
```

```
init_dira long %00000000_00000000_00000000_00001111
[/code]
```

The program sets the DIRA register so that the first four pins of the propeller are set as outputs again and again.

Here is what we needed to do to accomplish that.

First we defined a variable in a VAR block

We have to have at least one public method in the object so we started with naming the method. As soon as we did that we

can start the PASM cog with the cognew command and the two @ variables. These two variables tell the program where in the cog to start writing the program and provide an address that will point to a variable that can be shared between SPIN and PASM programs.

The program itself is described in a DAT data block. The block starts by telling the cog to start the program at location 0 with the Org 0 statement.

The next two lines are the program. The Prog\_start marker is the target of the cognew command mentioned above (the shared variable has not yet been addressed). The first line sets the DIRA register to the value init\_dira which is defined as a long at the end of the program.

The program then jumps back to the do\_again line.

What we have created is a basic program in its absolute minimum. We still have to define all the other blocks that might be needed to support the program that we are going to write and we will add these as we proceed with the learning process.

Harprit

=====+++++=====+++++=====

### Turning pins on and off and creating delays.

Next we need to learn how to turn propeller pins on and off and how to create a timed delay. Be warned that there are more elegant ways to do this but we are beginners and so we will stick with easy to understand code as is necessary at this time

We can turn pins on and off by **OR**-ing and **ANDN**-ing them with value we are interested in with the **DIRA** register once the **DIRA** register has been defined. The values we are going to use for the binary manipulations have to be pre-defined as constants at the tail end of the program. Any number of pins in any pattern can be turned ON off at one time.

In our previous program we defined pin 0 and pin 1 as outputs. We can turn them on with the **OR** instruction;. Let us consider pin 1 only for now. We identify this pin with the constant

```
pin_1 long %00000000_00000000_000000_00000010
```

we can also identify is in decimal and hex notation as follows

```
pin_1 long 2
pin_1 long $00_00_00_02 (The underscores are ignored)
```

but for now binary notation will be easier for us and I will use it throughout the book for consistency. A binary notation makes it possible to see the function of each pin at a glance without having to do any mental manipulations.

```
Mov dira, init_dira 'the original line of code
or outa, pin_1 'the instruction to turn pin 1 ON
```

And add the following line at the end of the program to identify pin 1

```
pin_1    long    %00000000_00000000_000000_00000010
```

and we can turn the pin off with the ANDN instruction as follows

```
Mov  dira,  init_dira    'the original line of code
anda  outa,  pin_1      'the instruction to turn pin 1 OFF
```

```
do_again  Mov  dira,  init_dira    'the original line of code
          or   outa,  pin_1      'the new instruction to turn pin 1 ON
          andn outa,  pin_1      'the new instruction to turn pin 1 OFF
          jmp  #do_again
```

```
pin_1    long    %00000000_00000000_000000_00000010
```

The above instructions will turn line 1 on and off about as fast as you can with a propeller. You will notice that the delay between ON and OFF is shorter than the delay between OFF and ON because the JMP instruction takes time as we go through the loop.

In order to be able to see an LED connected to line 1 go on and off we need a much longer delay. There are a number of ways of creating a delay but as beginners we can create a conventional delay loop of any length just as we would in a language like BASIC . The code is as follows

```
delay      mov  delay_counter,  delay_count    'load the delay counter
redo       sub  delay_counter,  #1 wz          'subtract 1 and set zero testvalue
          if_z jmp  #delay_ret                'if it is 0 we are done so return from sub
          jmp  #redo                          'if not keep subtracting
delay_ret  ret
```

This is set up as a subroutine here but you can also use the same technique for in line coding. The subroutine loads the delay value and then keeps subtracting 1 each time through the loop till the value reaches 0. It then returns control to the point from where it was called.

Whenever you want to call this delay you put in the line

```
call #delay
```

The length of the delay is determined by the constant delay\_count which is defined along with other constants at the end of the program. We define a ¼ second delay with a count of 2\_000\_000 times through the loop. We do this with

```
count_delay    long.....2_000_000
```

At 80 MHz, 2 million counts should take 1/40 of a second but we have other instructions that

have to be executed as a loop and that adds time to the delay. Making these additions to our program yields the following code.

[code]

```

CON
  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000

VAR
  long shared_var

PUB Object_name
cognew(@Prog_start, @shared_Var)
  repeat

DAT org 0
Prog_start      mov      dira,      init_dira
                mov      outa,      init_dira
do_again        or       outa,      pin_1           'the new
instruction to turn pin 1 ON
                call     #delay
                andn     outa,      pin_1           'the new
instruction to turn pin 1 OFF
                call     #delay
                jmp      #do_again

delay           mov      delay_counter,  delay_count  'load the delay
counter
redo           sub      delay_counter,  #1 wz         'subtract 1 and
set zero testvalue
if_z           jmp      #delay_ret      'if it is 0 we are
done so return from sub
              jmp      #redo           'if not keep

subtracting
delay_ret      ret

pin_1          long     %00000000_00000000_00000000_00000010
init_dira      long     %00000000_00000000_00000000_00000010
delay_count    long     2_000_000

delay_counter  res      1
[/code]

```

Run this program and make changes to see what happens.

WE now know how to write a rudimentary program in PASM. We now know how to turn any propeller line ON and OFF and we know how to add delays into our program when we need to.

Harprit

+++++

### Using the Full Duplex Serial Object

Before we start, we need a good way to be able to look into the programs that we are running. One of the best ways of doing this is to send the information to the PST (parallax serial terminal). This terminal appears in a separate window on your monitor when you run the

Parallax Serial Terminal.exe

program. We can communicate with this terminal with the FDS (full duplex serial) object that is provided by parallax as a part of the object exchange. This software also comes as one of the programs in the parallax tool editing suite. It is also possible to use the parallax serial terminal program,

Parallax Serial Terminal.spin

but the PST uses some non-standard coding, and I avoid it for that reason. The FDS software on the other hand, uses standard serial communications commands and if you are already familiar with them using it is painless. We will use FDS in all our programs that require us to look at what is going on in the program. In fact we will design our programs so that this is one of our primary tools.

In order to use FDS. You have to have the FDS file in the same folder as the other work that you are doing. If you have not already done so, make a copy of the program and add it to your work folder.

In your program, list the PDS in the OBJ block with the following lines of code just as you would have done in a SPIN program. As a matter of fact we are going to be running this software in the SPIN cog in our programs.

OBJ

```
fds : "FullDuplexSerial"
```

The FDS software is activated within your program by issuing the following command

```
fds.start(31,30,0,115200) 'start console at 115200 for debug output
```

Once this has been done, all standard commands that a serial terminal accepts are applicable to the FDS. Here is a list of some sample commands that you will need in almost every program. The entire ASCII coding standard is supported.

```
fds.bin(P_val,12) 'print value as binary to 12 places
fds.tx($d) 'new line
fds.dec(P_val) 'print a decimal value
fds.tx(" ") 'print a space
fds.tx($3) 'clear screen and go to 0,0 position
fds.tx($1) 'go to 0,0 position on screen, do not erase
'the tx prefix supports the entire ASCII set of commands
'and alphanumerics.
```

It is a good idea to provide one or a few spaces after printing a variable to erase any old information that may be left after the last printout of the same data was made. (This happens when the new output is shorter than the old.)

Most printing routines are used in a loop that monitors whatever we are interested in. A 1/60 second delay at the end of the loop is adequate for providing a steady flicker free display.

Harprit

```
+++++
{{
}}
CON
_c1kmode = xta11 + p1116x
_xinfreq = 5_000_000
```

```

VAR
    long p_val2
    long stack2[25]          'space for Cog_PST
OBJ
    fds : "FullDuplexSerial"

PUB null | P_VAL
    fds.start(31,30,0,115200) 'start console at 115200 for debug output
    cognew(@generate, @P_Val) 'start new cog at "generate" and read
    variable at P_Val
    cognew(osco, @stack2)
    dira[0 ..12]~~          ' all 12 lines are outputs. 12 lines needed
    for 1.5 bytes
    repeat
        P_VAL2:=P_VAL          ' endless loop to display data
        outa[0..11] := P_Val    ' displays 1.5 bytes of data
        fds.bin(P_val,12)      'print value
        fds.tx($d)
        fds.dec(P_val)        'print value
        fds.tx(" ")
        fds.tx($d)
        fds.tx(1)             'print a separating spac
        waitcnt(clkfreq/60+cnt)

PRI osco
    dira [12]~~
    repeat
        !outa[12]
        waitcnt(clkfreq/(P_VAL2+5)+cnt)

DAT
generate    org    0          'sets the starting point in Cog
            mov    dira, set_dira 'sets direction of the prop pins
            call   #chip_sel_lo  'selects chip by pulling line low
            call   #Clk_lo      'START. Clock needs to be low to load data
            call   #Din_hi      'must start with Din high to set up 3208
            call   #Tog_clk     'clk hi to read data
            call   #Din_Hi      'SINGLE DIFF Low to load
            call   #Tog_Clk     'toggle clock line hi then low to read data
            call   #Din_Lo      'D2 Low to load input sequence 000 for line 0
            call   #Tog_Clk     'toggle clock line hi then low to read in
the data
            call   #Din_Lo      'D1 Low to load input line selection
sequence 000 for line 0
            call   #Tog_Clk     'toggle clock line hi then low to read in
the data
            call   #Din_Lo      'D0 Low to load input line selection
sequence 000 for line 0
            call   #Tog_Clk     'toggle clock line hi then low to
read in the data
            call   #Din_Lo      'blank bit needs a clock cycle,
next
            call   #Tog_Clk     'toggle clock line hi then low to
read in the data
            call   #Tog_Clk     'next toggle is for the null bit,
nothing read
            call   #Tog_Clk     'toggle clock line hi then low to
read in the data
into        mov    dat_red, #0  'Clear register we will read data
read        mov    count, #12  'Counter for number of bits we will

```



```

read_bit      mov      temp,   ina      'read in what is in all the input
lines
line. Set Z flag andn      temp,   inputmask wz 'mask off everything except Dout
ready for next bit shl      Dat_red, #1      'roll register right 1 bit to get
to data register if_nz add      Dat_red, #1      'if value is still positive add 1
ready in Dout call      #Tog_Clk      'toggle clock to get next bit
counter. Set Z flag sub      count,   #1 wz      'decrement the "bits read"
not 0 if_nz jmp      #read_bit      'go up and do it again if counter
P.Val        wrlong   dat_red,   par      'put it in PAR to share it as
it, for low power usage call    #Chip_Sel_Hi      'Put chip to sleep by deselecting
jmp          #generate      'go back to do it all again

'Subroutines
Clk_Hi       or       outa,   clk_bit      'OR it with the Clock Bit to make
high
Clk_Hi_ret   ret
Clk_Lo       andn    outa ,   clk_bit      'ANDN it with the Clock Bi to
make low
Clk_Lo_ret   ret      'return from this subroutine
Tog_Clk      call    #Clk_hi      'make clock bit high
call        #clk_lo      'make clock bit low
Tog_Clk_ret  ret      'return from this subroutine
Din_Hi       or       outa ,   din_Bit      'Makes the Din high
Din_Hi_ret   ret      'return from this subroutine
Din_Lo       andn    outa ,   din_Bit      'makes Din low
Din_Lo_ret   ret      'return from this subroutine
Chip_Sel_Hi  or       outa ,   chs_Bit      'Makes Chip select high
Chip_Sel_Hi_ret ret      'return from this subroutine
Chip_Sel_Lo  andn    outa,   chs_Bit      'makes chip select low
Chip_Sel_Lo_ret ret      'return from this subroutine
Read_Dout    mov      temp,   ina      'Get the INA register
Read_Dout_ret ret      'return from this subroutine
Read_Next_Bit mov     temp,   ina      'Get the INA register
or          temp,   inputmask      'mask all but Din bit
Read_Next_Bit_ret ret      'return from this subroutine

'Constants. This section is similar to the CON block in SPIN
Set_dira     long     %00001011_00000000_00011111_11111111 'set dira register
Chs_Bit      long     %00000001_00000000_00000000_00000000 'Chip select bit
24
Din_Bit      long     %00000010_00000000_00000000_00000000 'Data in bit
25
Dout_Bit     long     %00000100_00000000_00000000_00000000 'Data out bit
26
Clk_Bit      long     %00001000_00000000_00000000_00000000 'Clock bit
27

```

```

inputmask    long    %11111011_11111111_11111111_11111111    'Mask for reading
the Dout bit only
Pin_12      long    %00000000_00000000_00010000_00000000    'set dira register

'Variables. This section is similar to the VAR block in SPIN
temp        res     1      'temporary storage variable, misc
count       res     1      'temporary storage variable, bit counter
Dat_Red     res     1      'temporary storage variable, data being read
    
```

+++++=+++++

For now, let us set up the following standard for the propeller pins usage.

Pins 0 to 11 would be connected to 12 LEDs on the PDB  
 pins 30 and 31 are reserved for serial communications.  
 Pins, 28 and 29 will be left untouched because we don't want to interfere with system operations.  
 Pins, 24 to 27 will be connected to the MCP 3208 for reading in the potentiometers that will be used in experiments to come. We will read 4 5K pots eventually  
 Pin 23 will be the output for an oscilloscope.  
 Pin 22 will be the output for a small speaker.

This might change from time to time, but for now, you can set this up this way and it should serve for the next few experiments. I will try to stick with this arrangement throughout, but we may have to make changes.

The oscilloscope and speaker are on separate cogs so that the outputs can be tailored to meet the requirements of each device.

All the experiments that we do will be able to be done on a simple breadboard. If that is your preference. However, I will be using the PDB, the professional development Board, provided by parallax, because this board has a lot of ancillary devices on it that make it very easy to use. For example, it has 16 LEDs that already have the resistors, they need in the series connected to them. So that making these LEDs active is a simple matter of running jumpers from the propeller pins to the LED pins. There are also resistors on the push buttons and on the switches, which makes life easier. When we need to pull pins up for any number of purposes.

**Setting up the MCP 3208.**

The 3208 is a chip that allows you to read up to eight channels of analog input in a hurry. It is capable of providing about 100,000 conversions per second. If the software to read this device is written in spin. it slows things down considerably. So, we have an interest in writing software in PASM to speed things up. It's not that you can't read one potentiometer quickly in spin its that if you want to read all eight channels, things can get just a bit sluggish.

In this next exercise we will read channel 0, on the 3208 and display the 12 bits that we read on 12 of the LEDs that be have connected up on the professional development Board. It would be a good idea at this time, if you were to download that data sheet for the 3208 and have it available for easy reference.

The connections to the 16 pin 3208 are as follows.

```

Line 1      Channel 0  wiper of the 1st potentiometer
Line 2      Channel 0  wiper of the 2nd potentiometer
Line 3      Channel 0  wiper of the 3rd potentiometer
Line 4      Channel 0  wiper of the 4th potentiometer
Line 5      Channel 0  wiper of the 5th potentiometer
    
```



## Chapter 6

In order to get effective use from our propellers chip we need to be able to interface them to any number of devices, that react with the real world, and provide us with information that we can use with microprocessor. Though there are any number of devices that you might start with, we going to start with the 3208 because it allows us to read potentiometers into the propellers chip. we will use the information that they provide to drive any number of devices from speakers to motors in our later experiments.

Take a close look at Figure 6-1 herein. Then refer it from time to time so see how the data transfer takes place. Also see page 16 of the data sheet Figure 5-1

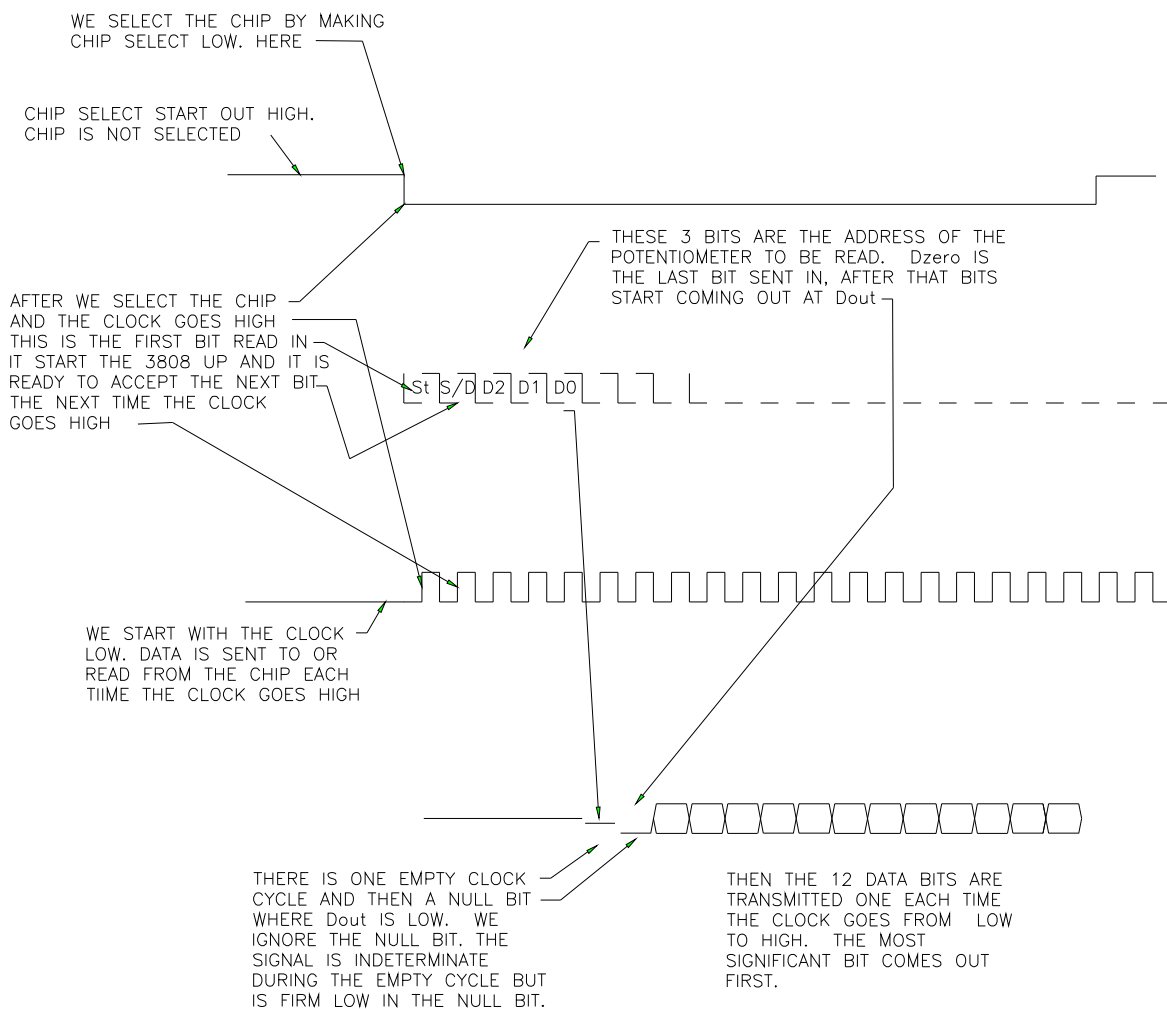


Figure 6-1

Segments of the program to read the 3208 are included in the following text. The entire program is listed at the end of the chapter and can be run on your propeller. The program is also on line in the discussion forum if you want to download it from there it will be easier. (Page 12 post #236)

The 3208 is capable of reading up to eight potentiometers one at a time at about 100,000 reading a second. We will place our potentiometers across 5 V and connect the wipers of the potentiometers to the eight input lines on the 3208. To start with, we will use only one potentiometer and it should be connected to Pin one.

Though we are connecting the potentiometer across 5 V it is not necessary that the potentiometers read across 5 V. There is a ground line, and a reference voltage line dedicated to the use with the input devices. The limitation is that since there is only one reference voltage line for all the potentiometers they all have to play be placed across this same voltage.

We are interested in potentiometers because a potentiometer is a device that is easily manipulated to provide a variable input. When we build other devices and connect them to a propellers chip for whatever purpose we may have in mind we can make the connections through the 3208 to provide the interface. The importance of a variable signal is to be appreciated, because we want to be able to make sure that we are actually reading or manipulating a changing signal. The signal may be an input or may be an output but in either case, we will have to see the results change in some way to make sure that the device is actually working. If nothing changes, not much can be deduced.

The 3208 is particularly well suited to our purpose or reading our first device, because the device is fairly easy to connect to and to use. Here is the procedure for reading the device. Follow along with the diagrams provided so that you can see exactly what we are going to do and how the system will respond.

There are four lines that control the operation of the 30 28. They are.

The clock line.

The data input line.

The data output line. We read the signal that comes out of this line

The chip select line.

The chip is dormant when the chip select line is high. We select the chip by making the chip select line low. When we make the chip select line low the 3028 responds by seeing it as a start signal for the whole next clock cycle.

```

DAT      org      0          'sets the starting point in Cog
generate mov      dira,   set_dira 'sets direction of the prop pins
         call     #chip_sel_lo 'selects chip by pulling line low
         call     #Clk_lo    'START. Clock needs to be low to load data
         call     #Din_hi    'must start with Din high to set up 3208
         call     #Tog_clk   'clk hi-lo to read data
         call     #Din_Hi    'SINGLE DIFF Low to load
         call     #Tog_Clk   'toggle clock line hi then low to read in the data

```

The next bit, we send it is a bit that selects the mode in which we want the 3208 to respond. For our purposes, we are interested in a single response and this is selected by making the data input line, low and toggling the clock chip high and then low it.

We next send out three the more bits. These bits identify one of the eight lines that we are going to read. A three bit signal can select one of the eight lines on the 3208. We will select line 0 in the initial experiment so the address we transmit to make the selection will be 000. Each line is impressed on the Din line and each time the clock is toggled high and then low one bits is read the the 3208.

```

         call     #Din_Lo    'D2 Low to load input line selection sequence 000 for line 0
         call     #Tog_Clk   'toggle clock line hi then low to read in the data
         call     #Din_Lo    'D1 Low to load input line selection sequence 000 for line 0
         call     #Tog_Clk   'toggle clock line hi then low to read in the data
         call     #Din_Lo    'D0 Low to load input line selection sequence 000 for line 0
         call     #Tog_Clk   'toggle clock line hi then low to read in the data
         call     #Din_Lo    'blank bit needs a clock cycle, next
         call     #Tog_Clk   'toggle clock line hi then low to read in the data
         call     #Din_Lo    'next toggle is for the null bit, nothing read
         call     #Tog_Clk   'toggle clock line hi then low to read in the data

```

Once the chip has accepted the three bit signal that the Din line goes into a don't care state. And we have no interest in it for the rest of the reading cycle.

Once the 3208 chip now knows, which line to read. It starts sending us the information about that line as 14 bits released by 14 clock cycles that we sent to the 3208. The first cycle provides indeterminate information and is to be ignored. The next bit as a low bit, to make sure we initiate our reading cycle

properly. This bit is to be considered a null bit but it does tell us that the cycle has started. The next 12 bits are the data that we are interested in, and they are transmitted one bit at a time. Each bit arriving when the clock goes from low to high.

```

        mov     dat_red, #0      'Clear register we will read data into
        mov     count, #12     'Counter for number of bits we will read
read_bit  mov     temp, ina      'read in what is in all the input lines
        andn   temp, inputmask wz 'mask off everything except Dout line. Set Z flag
        shl   Dat_red, #1      'shift reg left 1 bit to get ready for next bit
if_nz    add     Dat_red, #1    'if value is still positive add 1 to data register
        call  #Tog_Clk        'toggle clock to get next bit ready in Dout
        sub   count, #1 wz     'decrement the "bits read" counter. Set Z flag
if_nz    jmp    #read_bit     'go up and do it again if counter not yet 0

```

We read the bits by first clearing the register we are going to read into and then setting a counter to 12 to represent the 12 bits that we are going to read in. The bits are read by reading in the entire I/O register and then masking every bit except that the Dout bit from the 3208. If the masked answer is a one we add one to the register we are reading into and shift the whole register to the left one bit. If the red bit is a zero, we'd just shift all the bits left one bit. This makes the LSB in the register to zero. We do this 12 times and at the end of the 12 cycles. We have the reading from the potentiometer in our register.

```

wrlong   dat_red, par        'write it in PAR to share it as P.Val
call     #Chip_Sel_Hi      'Put chip to sleep , for low power usage
jmp      #generate         'go back to do it all again

```

We then write this information into the PAR register and it becomes available to the SPIN cog in our program, and we can use it for what ever we want. I have written in the code needed to send what is needed to the parallax serial terminal both as 12 bits binary and as a decimal quantity. As you manipulate the control knob of the potentiometer, the readings should go from 0 to 1111\_1111111 on line 1 and from zero to 4095 on line 2.

#### B null | P\_VAL

```

fds.start(31,30,0,115200) 'start console at 115200 for debug output
  cognew(@generate, @P_Val) 'start new cog at "generate" and read variable into P_Val
  cognew(oscope, @stack1)  'open cog to generate osc signals
  cognew(spkr, @stack2)    'open cog to generate speaker signals
  dira[0 ..11]~~         'sets 12 lines as outputs. 12 lines needed for 1.5 bytes

```

```

repeat                                     'loop
  global_value:=P_VAL                     'endless loop to display data
  outa[0..11] := P_Val                    'displays 1.5 bytes of data on the LEDs
  fds.bin(P_val,12)                       'print value to the PST in binary to match LEDs
  fds.tx($d)                              'new line
  fds.dec(P_val)                          'print value as decimal
  fds.tx(" ")                             'spaces
  fds.tx($1)                              'home to 0,0
  waitcnt(clkfreq/60+cnt)                 'flicker free wait

```

Here is the listing of the entire program.

```

{{
Program to read a pot
August 05 2011
Sandhu
Works.
Using a speaker or the o'scope is optional.
LEDs and serial terminal both show what is being read
}}
CON
  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000
  spkr_line=22
  osc_line=23

VAR
  long global_value
  long stack1[25]          'space for oscscope
  long stack2[25]        'space for speaker

OBJ
  fds : "FullDuplexSerial"

PUB null | P_VAL
fds.start(31,30,0,115200)      'start console at 115200 for debug output
cognew(@generate, @P_Val)     'start new cog at "generate" and read variable into P_Val
cognew(oscope, @stack1)      'open cog to generate osc signals
cognew(spkr, @stack2)        'open cog to generate speaker signals
dira[0 ..11]~~              'sets 12 lines as outputs. 12 lines needed for 1.5 bytes
repeat                        'loop
  global_value:=P_VAL        'endless loop to display data
  outa[0..11] := P_Val        'displays 1.5 bytes of data on the LEDs
  fds.bin(P_val,12)           'print value to the PST in binary to match LEDs
  fds.tx($d)                  'new line
  fds.dec(P_val)              'print value as decimal
  fds.tx(" ")                 'spaces

```



```

        fds.tx($1)                'home to 0,0
        waitcnt(clkfreq/60+cnt)    'flicker free wait

PRI oscpe                          'oscilloscope output cog
dira [osc_line]~~                  'set pin direcion as output
repeat                              'loop
    !outa[osc_line]                 'invert line
    waitcnt(clkfreq/(global_value+20)+cnt) 'wait suitable for osc view

PRI spkr                            'speaker oputput cog
dira [spkr_line]~~                 'set pin direcion as output
repeat                              'loop
    !outa[spkr_line]                'invert line
    waitcnt(clkfreq/(global_value+20)+cnt) 'wait suitable for speaker

DAT      org      0                'sets the starting point in Cog
generate mov      dira,  set_dira    'sets direction of the prop pins
        call     #chip_sel_lo       'selects chip by pulling line low
        call     #Clk_lo            'START. Clock needs to be low to load data
        call     #Din_hi            'must start with Din high to set up 3208
        call     #Tog_clk           'clk hi-lo to read data
        call     #Din_Hi           'SINGLE DIFF Low to load
        call     #Tog_Clk          'clock line hi then low to read in the data
        call     #Din_Lo           'D2 Low to load input sel 000 for line 0
        call     #Tog_Clk          'clock line hi then low to read in the data
        call     #Din_Lo           'D1 Low to load input seq 000 for line 0
        call     #Tog_Clk          'clock line hi then low to read in the data
        call     #Din_Lo           'D0 Low to load line seq 000 for line 0
        call     #Tog_Clk          'clock line hi then low to read in the data
        call     #Din_Lo           'blank bit needs a clock cycle, next
        call     #Tog_Clk          'lock line hi then low to read in the data
        call     #Tog_Clk          'next toggle is for the null bit
        call     #Tog_Clk          'clock line hi then low to read in the data
        mov      dat_red, #0        'Clear register we will read data into
        mov      count, #12        'Counter for number of bits we will read
read_bit mov      temp, ina         'read in what is in all the input lines
        andn     temp, inputmask wz 'mask off except Dout line. Set Z flag
        shl     Dat_red, #1        'shift reg left 1 bit, ready for next bit
        if_nz   add     Dat_red, #1 'if still positive add 1 to data register
        call     #Tog_Clk          'toggle clock to get next bit ready in Dout
        sub     count, #1 wz       'decr the "bits read" counter. Set Z flag
        if_nz   jmp     #read_bit   'go up and do it again if counter not yet 0
        wrlong  dat_red, par       'write it in PAR to share it as P.Val
        call     #Chip_Sel_Hi      'Put chip to sleep by de selecting
        jmp     #generate          'go back to do it all again

```

'Subroutines

```

Clk_Hi      or      outa,  clk_bit      'OR it with the Clock Bit to make high
Clk_Hi_ret  ret                               'return from this subroutine

Clk_Lo      andn   outa ,  clk_bit      'ANDN it with the Clock Bi to make low
Clk_Lo_ret  ret                               'return from this subroutine

Tog_Clk     call   #clk_hi              'make clock bit high
            call   #clk_lo              'make clock bit low
Tog_Clk_ret ret                               'return from this subroutine

Din_Hi      or      outa ,  din_bit      'Makes the Din high
Din_Hi_ret  ret                               'return from this subroutine

Din_Lo      andn   outa ,  din_bit      'makes Din low
Din_Lo_ret  ret                               'return from this subroutine

Chip_Sel_Hi or      outa ,  chs_bit      'Makes Chip select high
Chip_Sel_Hi_ret ret                               'return from this subroutine

Chip_Sel_Lo andn   outa,  chs_bit      'makes chip select low
Chip_Sel_Lo_ret ret                               'return from this subroutine

Read_Next_Bit mov    temp,  ina          'Get the INA register
            or     temp,  inputmask     'mask all but Din bit
Read_Next_Bit_ret ret                               'return from this subroutine

```

'Constants. This section is similar to the CON block in SPIN

```

Set_dira    long    %00001011_11000000_00001111_11111111  'Set dira register
Chs_Bit     long    %00000001_00000000_00000000_00000000  'Chip select bit 24
Din_Bit     long    %00000010_00000000_00000000_00000000  'Data in bit      25
Dout_Bit    long    %00000100_00000000_00000000_00000000  'Data out bit     26
Clk_Bit     long    %00001000_00000000_00000000_00000000  'Clock bit        27
inputmask   long    %11111011_11111111_11111111_11111111  'Mask for Dout bit only
Pin_23      long    %00000000_10000000_00000000_00000000  'osc line
Pin_22      long    %00000000_01000000_00000000_00000000  'Speaker line

```

'Variables. This section is similar to the VAR block in SPIN

```

temp        res     1      'temporary storage variable, misc
count       res     1      'temporary storage variable, read bit counter
Dat_Red     res     1      'temporary storage variable, data being read

```