A Beginner's Guide to Manipulating I/O Pins in PASM

By Michael J. Hanagan

Preface: I encountered the speed limitations of SPIN while interfacing the Propeller with a Microchip MCP3202 chip. I could only achieve measurement speeds on the order of 1-2,000 Hz but the ADC itself was capable of measurement speeds up to 100,000 Hz. The bottleneck was reading the serial transmitted data from the ADC. The slow interpreted SPIN language could not manipulate the SPI clock and data pins fast enough. The solution of course is programming the I/O function with the ADC in the much faster language known as Propeller Assembly, or PASM.

When I began this venture well over a year ago there was not much out there on learning the very basics of PASM. I read a few nice pdfs by “potatohead” and “desilva”, and many PASM code examples in the OBEX library which help me to get started. I still struggled for a long time trying to learn enough PASM to achieve high speed SPI communications with the ADC. During this time I collected a set of working notes which outlines some of the introductory basics of PASM. While I have some programming experience I am no software engineer. My involvement with the Propeller chip is primarily on a hobby/enthusiast level. As such I often need to refer back to these and other notes as reminders when I am writing a snippet of PASM code. I thought I would assemble a few of these notes into a short text in hopes it might help others trying to learn the basics of PASM. This first set of notes focuses on how to manipulate the Propeller’s I/O pins using PASM.

If you find any errors in this text you can reach me on the Propeller Forum under my profile name “MJHanagan”. I greatly appreciate those on the forum who caught errors in my original draft and made suggestions for clarifications, specifically “Mark_T” and “tonyp12”.

If time permits in the future I will write a follow on set of notes covering the various ways of passing data from cog RAM to main RAM. And if I ever get the chance I hope to also write up my experience in interfacing the Propeller with the MDC320x family of ADCs using both SPIN and PASM.

MJH 26-Jan-2014
Rutland, MA

An overview of I/O pin manipulation using PASM

One of the most useful features of the Propeller chip is its high number of I/O pins coupled with its eight coprocessors called cogs. After boot up each of the Propeller's 32 pins can be programmed to act as an input or an output (during boot up pins 28-31 are used load the program and data from the EEPROM). Each of the eight coprocessors has shared access to all 32 I/O pins via three 32-bit special registers: **DIRA**, **OUTA** and **INA**. Bits 0-31 in each register corresponds to pin numbers P0-P31 on the chip. The **DIRA** register is a read-write variable whose bit settings define pins as inputs (0) or outputs (1). The **OUTA** register is also a read-write variable whose bit setting determine of an output pin is low (0) or high (1). The **INA** register is a read-only variable whose bit setting indicates the high/low status of all 32 I/O pins, inputs as well as outputs.

Each cog maintains its own independent **DIRA** and **OUTA** registers. When a cog starts all bits in its **DIRA** are set to 0 (inputs) and all bits in its **OUTA** register are also set to 0 (low). You must set each bit in the **DIRA** to a 1 if its corresponding pin is to act as an output. Once a pin is set as an output in **DIRA** you can set it high (+3.3V) by setting the corresponding bit in **OUTA** to 1 and set it low (0 V, i.e. Vcc) again setting the bit to 0. If the same output pin is to also be controlled high and low by a different cog then each cog that uses that pin must set that bit high in its own **DIRA** register. If a cog does not declare a pin as an output in its **DIRA** it will not be able to manipulate its output state high or low. Each of the eight cog's **DIRA**s are ORed together with the result being the final **DIRA** for all 32 pins. The **DIRA** of any inactive or stopped cog is automatically cleared to 0 so only active cogs are factored into the final **DIRA** for the chip.

If any active cog sets a pin to be an output it supersedes any input setting in all other active cogs. However, the contents of the other cog's **DIRA**s are not altered, only the final ORed **DIRA** for the chip is affected. Any pin set as an output cannot be used as an input by any cog. Within a cog you can switch a pin's direction as needed for synchronous communication via a single input/output data pin (provided no other active cog has it set as an output). The **DIRA** and **OUTA** for each cog are ANDed together, then these eight results are then ORed together to ultimately determine which output pins are high and low. The like the **DIRA** registers the **OUTA** registers are local so reading these registers will not reveal what other setting have been made by other cogs. Only the read-only **INA** reflects the final state of all the I/O pins regardless of their input/output settings.

Here is another way of summarizing the overall effect of the eight independent read-write **DIRA** and **OUTA** registers:

- Setting a bit high in a cog's **OUTA** register will only result in the corresponding pin outputting +3.3V if the same bit is also set to a 1 in the cog's **DIRA** register.
- If any cog (including Spin) has a bit set high in both **DIRA** and **OUTA** the corresponding pin will output +3.3V regardless of any other cog's attempt to set it low or make it an input pin.
- A cog's **OUTA** register does not reveal the true electrical state of the pins, only the **INA** register will reflect the high/low state of each pin.

Additional details regarding I/O pins, **DIRA**, **OUTA** and **INA** can be found in the Propeller Manual on pages 26-27 (general I/O pin discussion), pp 104-106 (SPIN **DIRA**), pp 118-119 (SPIN **INA**), pp 175-177 (SPIN **OUTA**), p 289 (PASM **DIRA**), p 297 (PASM **INA**) and p 330 (PASM **OUTA**). If you can read electrical engineering flow diagrams and symbols there is a block diagram of the Propeller chip on pages 20-21.

Methods for manipulating specific bits/pins in DIRA and OUTA

Since pin direction (input/output) and output status (high/low) are determined by the bit settings in **DIRA** and **OUTA** we will need to learn how to set specific bits in these two registers low (0) and high (1). The bits in these registers numbered 0 through 31 correspond to I/O pins labelled P0 through P31. The general method for manipulating bits in **DIRA** and **OUTA** utilizes the PASM commands **OR, ANDN** and occasionally **XOR**. The **OR** command is used to force bits high, **ANDN** forces bits low, and the **XOR** command inverts the status of bits (if they are high they get set low and if already low it sets them high). You will often see these bitwise functions shown as truth tables:

Initial Bit		Value1 Bit Result			
Value1	Value2	AND	ANDN	OR	XOR
A	B	A & B	A & !B	A OR B	A XOR B
0	0	0	0	0	0
0	1	0	0	1	1
1	0	0	1	1	1
1	1	1	0	1	0

The first step to controlling individual bits/pins is to setup what is often referred to as a bit mask. This is a 32-bit variable which typically has one bit set high corresponding to the pin it is meant to control. For example, assume we want to manipulate P10, the pin mask would have a binary value of %0100_0000_0000 (decimal value 1024, or hexadecimal value $400). We can use the **LONG** command in the **DAT** section to assign a value to a variable using any one of these numerical methods:

```
PMask   LONG   %0100_0000_0000
PMask   LONG   1024
PMask   LONG   $400
```

In general, individual bit mask variables tend to be more useful, but setting more than one bit high in a variable does allow for the control of multiple pins using a single command. For example if we wanted to simultaneously control P10 and P12 you would use a bit mask variable with a binary value of %0001_0100_0000_0000 (decimal 5120, or $1400). With a bit mask variable defined with a single bit or multiple bits we can now use it to manipulate that same bit or bits in **DIRA** and **OUTA** using a variety of PASM commands.

To set bits contained in the bit mask variable high (1) in **DIRA** or **OUTA** without altering any others use the **OR** command (p 327):

```
OR Value1, Value2
```

All 32 bits in **Value1** are bitwise ORed with **Value2** and the result stored back in **Value1**. Assuming we want to set P10 as an output we would use **DIRA** as **Value1** and PMask as **Value2**:

```
OR DIRA, PMask
```

```
     Value1:  DIRA = %xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
     Value2: PMask = %00000000_00000000_00000100_00000000
ORed Value1:  DIRA = %xxxxxxxx_xxxxxxxx_xxxxx1xx_xxxxxxxx
```

The "x" bit designation means "doesn't matter". If a bit is low (0) in **DIRA** it will remain low after the ORing it with any 0 in the pin mask and if it is high (1) it will remain high after the ORing it with a 0. The only bits directly affected in **DIRA** are whose corresponding bits in PMask are set high since any value ORed with 1 is 1, which in this example is only bit 10. If bit 10 in **DIRA** was low or high it will be forced high by the **OR** command.

On the other hand, to set a specific bit low (0) in **DIRA** without altering the direction of any other bits use the **ANDN** command (p 267):

```
ANDN Value1, Value2
```

All 32 bits in **Value1** are bitwise ANDed with the inverted value (bitwise NOT) of **Value2** and the result stored back in **Value1**. Assuming we want to set P10 as an input pin we would use **DIRA** as **Value1** and PMask as **Value2**:

```
ANDN DIRA, PMask
```

The first step of **ANDN** command takes the inverse (bitwise NOT) of the PMask:

```
      Value2: PMask = %00000000_00000000_00000100_00010000
     !Value2: PMask = %11111111_11111111_11111011_11111111
```

The second part of the **ANDN** command takes !PinMask and ANDs it with **DIRA**:

```
      Value1:  DIRA = %xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
     !Value2: PMask = %11111111_11111111_11111011_11111111
ANDed Value1:  DIRA = %xxxxxxxx_xxxxxxxx_xxxxx0xx_xxxxxxxx
```

Again, the "x" bit designation means "doesn't matter". If a bit is low (0) it will remain low after the ANDing it with any bit set to a 1 in the inverted pin mask. Likewise, a high (1) bit will remain high after the ANDing it with any bit set to 1 in the inverted pin mask. The only bits affected in **DIRA** are whose corresponding bits in PMask are set low (0) after the bitwise **NOT** since any value ANDed with 0 is 0, which in this example is only bit 10. If bit 10 in **DIRA** was low or high it will be forced low by the **ANDN** command.

The same **OR** and **ANDN** commands are used on **OUTA** to set output pins high (+3.3V) or low (0V). To set an output pin high use the **OR** command:

```
OR OUTA, PMask
```

```
      Value1:  OUTA = %xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
      Value2: PMask = %00000000_00000000_00000100_00000000
 ORed Value1:  OUTA = %xxxxxxxx_xxxxxxxx_xxxxx1xx_xxxxxxxx
```

To set an output pin low use the **ANDN** command:

```
ANDN OUTA, PMask
```

The first step of **ANDN** command takes the inverse (bitwise NOT) of the PMask:

```
      Value2: PMask = %00000000_00000000_00000100_00010000
     !Value2: PMask = %11111111_11111111_11111011_11111111
```

The second part of the **ANDN** command takes !PMask and ANDs it with **OUTA**:

```
      Value1:  OUTA = %xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
     !Value2: PMask = %11111111_11111111_11111011_11111111
ANDed Value1:  OUTA = %xxxxxxxx_xxxxxxxx_xxxxx0xx_xxxxxxxx
```

To invert the state of an output pin use the **XOR** command:

```
XOR OUTA, PMask
```

If the output pin is already high (1) the **XOR** command makes it low (0):

```
      Value1:  OUTA = %xxxxxxxx_xxxxxxxx_xxxxx1xx_xxxxxxxx
      Value2: PMask = %00000000_00000000_00000100_00010000
XORed Value1:  OUTA = %xxxxxxxx_xxxxxxxx_xxxxx0xx_xxxxxxxx
```

If the output pin is already low (0) the **XOR** command makes it high (1):

```
      Value1:  OUTA = %xxxxxxxx_xxxxxxxx_xxxxx0xx_xxxxxxxx
      Value2: PMask = %00000000_00000000_00000100_00010000
XORed Value1:  OUTA = %xxxxxxxx_xxxxxxxx_xxxxx1xx_xxxxxxxx
```

There is also a handy set of four **MUX** commands that can be used to conditionally set bits in **DIRA** and **OUTA** based on a pin mask variable and the state of the C and/or Z flags:

```
MUXC  Value1, Value2   Sets bits in Value1 equal to the C flag state (p 315)
MUXNC Value1, Value2   Sets bits in Value1 equal to the !C flag state (p 316)
MUXZ  Value1, Value2   Sets bits in Value1 equal to the Z flag state (p 317)
MUXNZ Value1, Value2   Sets bits in Value1 equal to the !Z flag state (p 318)
```

Like in the **OR** and **ANDN** commands shown above the only bits affected in **Value1** are those whose corresponding bits in **Value2** (the pin mask) are set high (1). All bits other bits in **Value1** remain unaffected. Prior to executing the **MUX** command we must first set the C and/or Z flags as desired. Many PASM commands have the option of setting the C and Z flags based on the results of the instruction.

By way of example assume we are setting the output state of P10 high or low based on the value of the variable CountNum. If CountNum is <15 set P10 high, and if CountNum equals or exceeds 15 set P10 low. For this example using the **CMP** command provides a convenient way of setting the C and Z flags based on the comparison of two unsigned values (p 272):

```
CMP Value1, Value2   wc, wz
```

If **Value1** is less than **Value2** and the **WC** is specified the C flag is set (1), otherwise the C flag is not set (0). If **Value1** equals **Value2** and the **WZ** is specified the Z flag is set (1), otherwise it is not set (0). For this example we want to compare the value in CountNum to 15:

```
CMP CountNum, #15   wc
```

If CountNum is less than 15 the C flag is set (1), if CountNum has a value of 15 or higher the C flag is not set (0). In this case **WZ** was not specified so the Z flag result is not written. With the C flag now set use the **MUXC** on **OUTA** to set the output of P10 based the C flag setting:

```
MUXC OUTA, PMask
```

Case 1: CountNum is <15, C flag is set (1) making all masked bits high (1):

```
        Value1:  OUTA = %xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
        Value2: PMask = %00000000_00000000_00000100_00000000
MUXCed Value1:  OUTA = %xxxxxxxx_xxxxxxxx_xxxxx1xx_xxxxxxxx
```

Case2: CountNum is =>15, C flag is not set (0) making all masked bits low (0):

```
        Value1:  OUTA = %xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
        Value2: PMask = %00000000_00000000_00000100_00000000
MUXCed Value1:  OUTA = %xxxxxxxx_xxxxxxxx_xxxxx0xx_xxxxxxxx
```

If we want the opposite effect on the output pins use the **MUXNC** command. In this case the !C flag is used in place of C.

```
MUXNC OUTA, PMask
```

Case 1: CountNum is <15, C flag is set (1), thus !C is set (0) making all masked bits low (0):

```
        Value1:  OUTA = %xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
        Value2: PMask = %00000000_00000000_00000100_00000000
MUXCed Value1:  OUTA = %xxxxxxxx_xxxxxxxx_xxxxx0xx_xxxxxxxx
```

Case 2: CountNum is =>15, C flag is not set (0), thus !C is set (1) making all masked bits high (1):

```
        Value1:  OUTA = %xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
        Value2: PMask = %00000000_00000000_00000100_00000000
MUXCed Value1:  OUTA = %xxxxxxxx_xxxxxxxx_xxxxx1xx_xxxxxxxx
```

The **MUXZ** and **MUXNZ** commands work in the same way but using the status of the Z flag.

Other comparative commands include **cmps**, **cmpx**, **cmpsx**, **cmpsub**, **test** and **testn**. In addition, most other commands have the ability to set the C and Z flags according to the resulting value.

Methods for reading the status of an input pin in INA

Reading the status of a single input pin is typically done using the **TEST** command (p362):

```
TEST    Value1, Value2    wc, wz
```

This command performs a bitwise AND of **Value1** and **Value2** and sets the Z and C flags based on the ANDed value. If the **WZ** effect is specified, the Z flag is set (1) if the ANDed result is 0. If the **WC** effect is specified, the C flag is set (1) if the result contains an odd number of high (1) bits. Since the C flag only reflects the odd or even number of high bits in the result it will only work properly when the pin mask has a single bit set. If two or more bits are set in the pin mask and they are both high in INA the AND result will have even parity therefore the C flag result is 0.

If we want to see if a specific input pin is high or low use the **TEST** command on **INA** with a pin mask variable. For example, assume we want to see if input pin P5 is set high or low. PMask should contain the value %0010_0000:

```
TEST  PMask, INA    wc
```

Case 1: input pin P5 is low (0):

```
Value1: PMask = %00000000_00000000_00000000_00100000
Value2:   INA = %xxxxxxxx_xxxxxxxx_xxxxxxxx_xx0xxxxx
ANDed result: = %00000000_00000000_00000000_00000000
 Result value = 0
       Z flag = 1 (result not written)
       C Flag = 0 even parity - no bit set(result written)
```

Case 2: input pin P5 is high (1):

```
Value1: PMask = %00000000_00000000_00000000_00100000
Value2:   INA = %xxxxxxxx_xxxxxxxx_xxxxxxxx_xx1xxxxx
ANDed result: = %00000000_00000000_00000000_00100000
 Result value = 32
       Z flag = 0 (result not written)
       C Flag = 1 odd parity - one bit set(result written)
```

With the status of the input pin now captured in the C flag we can conditionally execute a subsequent command or save the 0 or 1 C flag value in a variable. For example, if we are reading the input pin as a form of data transmission we can use the **RCL** (p 333) or **RCR** (p 334) command to insert the value of the C flag into a variable.

`RCL Value, <#>N` Shift bits in Value left "N" positions with vacated bits filled in with C flag value.

`RCR Value, <#>N` Shift bits in Value right "N" positions with vacated bits filled in with C flag value.

Assuming the data bit transmission is coming in MSB to LSB we will want to insert the just read C flag result in the LSB position to maintain proper bit order, therefore use the **RCL** command:

```
RCL ReadBits, #1

            Value: ReadBits = %abcxxxxx_xxxxxxxx_xxxxxxxx_xxxxxdef
 Rotate left 1 bit: ReadBits = %bcxxxxxx_xxxxxxxx_xxxxxxxx_xxxxdef?
```

Note that the value in bit 31 (a) is lost forever during the rotate left command (it gets sent to the proverbial "bit bucket"). All other bit values are retained but shifted to the left by 1. Now the C flag value is inserted into the vacated bit 0 (LSB) position:

Case 1: input pin was low so C flag is not set (0):

```
  C flag inserted: ReadBits = %bcxxxxxx_xxxxxxxx_xxxxxxxx_xxxxdef0
```

Case 2: input pin was high so C flag is set (1):

```
  C flag inserted: ReadBits = %bcxxxxxx_xxxxxxxx_xxxxxxxx_xxxxdef1
```

On the other hand if the data bit transmission is coming in LSB to MSB we will want to insert the C flag result in the MSB position to maintain proper bit order, therefore use the **RCR** command:

```
RCR ReadBits, #1
```

```
           Initial: ReadBits = %abcxxxxx_xxxxxxxx_xxxxxxxx_xxxxxdef
Rotate right 1 bit: ReadBits = %?abcxxxxxx_xxxxxxxx_xxxxxxxx_xxxxde
```

This time the value in bit 0 (f) is lost forever during the rotate right command. All other bit values are retained but shifted to the right by 1. Now the C flag value is inserted into the vacated bit 31 (MSB) position:

Case 1: input pin was low so C flag is not set (0):

```
   C flag inserted: ReadBits = %0abcxxxxxx_xxxxxxxx_xxxxxxxx_xxxcde
```

Case 2: input pin was high so C flag is set (1):

```
   C flag inserted: ReadBits = %1abcxxxxxx_xxxxxxxx_xxxxxxxx_xxxcde
```

In the case where the bits are coming in LSB to MSB we will probably be capturing fewer than 32 bits so after the last bit is inserted into bit 31 in ReadBits we will need to shift the contents to the right moving the first read bit (LSB) down to the 0 bit position. This can be done using the **SHR** command (p 348). Assume we captured a 12-bit transmission (a-l), now the contents of ReadBits need to be shifted 20 bits to the right (32-12=20 bits):

```
SHR ReadBits, #20
```

```
            Initial: ReadBits = %abcdefgh_ijklxxxx_xxxxxxxx_xxxxxxxx
Shift right 20 bits: ReadBits = %00000000_00000000_0000abcd_efghijkl
```

Note that in the above example 20 0's are shifted in from the left.

The **INA** is a read-only global register just like **CNT**. It not only reflects which input pins are at a voltage above VDD/2, but also indicates which output pins are currently set high as well. As such, each cog has the same view to the voltage status of all 32 I/O pins. Each active cog can "see" the status of all pins in **INA** regardless of their input or an output designation in their local **DIRA** and **OUTA** registers. This feature can be leveraged in a way so one cog can effectively communicate or coordinate activities with another cog. For example cog A could wait for cog B to set a specific output pin high before performing some function by monitoring that pin using **INA** and a pin mask.

Here are a couple of “quirks” regarding **INA**. The following command to write **INA** to a variable in main RAM will not work:

```
WRLONG INA, MainAddr
```

However, this two-step method will work:

```
MOV Tmp, INA
WRLONG Tmp, MainAddr
```

And for the really oddity, this two-step method will also work:

```
MOV INA, INA
WRLONG INA, MainAddr
```

It has something to do with INA being a "shadow register" (and that is beyond my capability to explain).

Summary of I/O pin manipulation commands

Set pins low (0) in **DIRA** (input pin) or **OUTA** (pin low) according to a pin mask variable:

```
ANDN DIRA, PMask
```
Pins in PMask are set to inputs.

```
ANDN OUTA, PMask
```
Output pins in PMask are set low (0V).

Set bits high (1) in **DIRA** (output pin) or **OUTA** (pin on) according to a pin mask variable:

```
OR DIRA, PMask
```
Pins in PMask are set to outputs.

```
OR OUTA, PMask
```
Output pins in PMask are set high (+3.3V).

Inverts bits in **OUTA** according to a pin mask variable:

```
XOR DIRA/OUTA, PMask
```
Pins in PMask are toggled (low becomes high, and high becomes low).

Read the status of a single input pin in **INA** and insert the result (0 or 1) into the LSB of the variable ReadBits:

```
TEST PinMask, INA    wc
RCL  ReadBits, #1
```

Read the status of a single input pin in **INA** and insert the result (0 or 1) into the MSB of the variable ReadBits:

```
TEST PinMask, INA    wc
RCR  ReadBits, #1
```