Introduction

This document is, and always will be, a work in progress. It’s designed to be a compendium of tricks stumbled into and traps fallen into while learning to program Parallax’s Propeller chip. At this early stage in my own learning curve, it may seem ruefully pretentious to produce such a document. But I wanted to get the information down while it was still fresh in my mind and to have a vehicle for recording my adventures and misadventures with the Propeller as they happened. I know if I wait, I’ll forget both my discoveries and my mistakes and probably have to repeat them. Hopefully, this will be a time-saver for other programmers, as well, as they probe the Propeller’s many wonders and mysteries.

As yet, this document is not organized by category but is, rather, just a jumbled potpourri of miscellania. As more stuff gets added, it may become apparent how better to organize it.

Contributions

Anyone encountering this document who would like to contribute or make corrections to it is welcome to do so. Just leave a posting in the “Tricks & Traps” thread in Parallax’s Propeller forum. I will include it as my time allows and as my sole editorial discretion deems it suitable. Credit will be given to contributors. Just make sure to include how you’d like to be credited.

Copyright

This document, including any future additions, corrections, and contributions, regardless of their source, is Copyright 2006 by Philip C. Pilgrim. Permission is hereby granted to any and all to make and distribute (for free – not for compensation) as many copies of it as you like. The right to modify this document in any way for distribution or any purpose other than personal use is expressively reserved by the original author.

Disclaimer

This document is offered in the hopes that it might prove helpful to others. Although every effort has been made to insure accuracy, it is being written by a Propeller neophyte, after all, and may contain errors. Heck, it probably contains errors! Therefore, neither the original author, nor Bueno Systems, Inc., nor any other contributor makes any warranties, express or implied, as to its accuracy or suitability for any purpose whatsoever. This means that you, the reader, accept all risks for using the information contained herein.

Trademarks

“Propeller”, as applied to a microcontroller chip, and the propeller beanie logo are trademarks of Parallax, Inc. The beanie logo on the cover page is used with permission.
Assembly Tricks & Traps

**Trick:** You can use immediate addressing when your source operand is 9 bits or less. Just prepend the constant with a pound sign:

```
MOV Dest, #53
```

This statement loads the value **53** into **Dest**.

**Trap:** Make sure, when using immediate addressing, that your operand doesn’t exceed nine bits in length. If it does, only the least-significant nine bits will be used. The following code loads **Dest** with the value $1A5.

```
MOV Dest, #$A5A5A5A5
```

*Thanks to André LaMothe*

**Trick:** You can use the **MOVI**, **MOV D**, and **MOVS** commands to load the Instruction (bits 31 .. 23), Destination (bits 17 .. 9), or Source (bits 8 .. 0) fields of any writable memory destination. This technique is a handy way to achieve indirect addressing:

```
MOVS :Indirect,Address
... some other instruction(s) ...
:Indirect MOV Dest,0-0
```

In this example, the contents of the memory at **Address** are stuffed into the source field of the **MOV** instruction at **:Indirect**. When the **MOV** instruction executes, the contents of the memory at **that** address will be loaded into **Dest**. (The **0-0** is just a way to indicate that something will be stuffed into that location from elsewhere.)

**Trap:** Never modify the instruction following the one doing the modifying. If the prior example had been written:

```
MOVS :Indirect,Address
:Indirect MOV Dest,0-0
```

it would not have worked as expected. The reason is that the Propeller uses a two-stage pipeline. As one instruction is executing, the next one is being fetched. By the time the **MOVS** is finished executing, the **MOV** instruction with its prior source field contents will already have been loaded. Always insert at least one additional instruction between modifier and modified. Use a **NOP** if you have to.
**Trick:** You can use inline subroutine arguments to improve a program’s readability and encapsulation:

```
CALL #Subroutine
LONG Arg1
LONG Arg2

.. 

Subroutine

MOVS :GetArg1,Subroutine_ret
ADD Subroutine_ret,#1
:GetArg1
MOVS :Arg1,0-0
MOV :Arg1,0-0
MOVS :GetArg2,Subroutine_ret
ADD Subroutine_ret,#1
:GetArg2
MOV :Arg2,0-0
MOV :Arg2,0-0

.. 

JMP Subroutine_ret

:Arg1 LONG 0-0
:Arg2 LONG 0-0

Subroutine_ret RET
```

In this example, **Subroutine** is called with two arguments, having values **Arg1** and **Arg2**. When **Subroutine** is **CALL**ed, the address of **Arg1** is written into memory at **Subroutine_ret**. The first instructions in **Subroutine** use indirect addressing to retrieve **Arg1** and **Arg2** and place their values in **Subroutine**’s local variables, :**Arg1** and :**Arg2**, respectively. One advantage to this method is that you don’t need to use global variables to pass parameters to a subroutine.

Notice that the **JMP Subroutine_ret** is an indirect jump. It jumps to the address contained in **Subroutine_ret**. We could have used immediate addressing, but that would have resulted in two jumps to exit the routine, rather than one.

This is an example of “call by value”. You can also implement “call by reference”, wherein **Arg1**, for example, might contain the address of the value you wish to pass. In this case, you would have to do the indirect addressing twice to get the value within the subroutine.
Trap: It’s so easy to forget that the source field of a JMP follows the same rules as any other instruction. If you’re jumping directly to a location in memory, use immediate addressing:

```
JMP  #Here
  ..
Here
```
Failure to include the pound sign (#) can be a frequent source of headaches.

Trick: Indirect JMP addressing can come in handy for implementing things like state machines, though. Here’s an example of a three-state loop, whose state variable is the address contained in the source field of memory location StateMachine:

```
Init     MOVS  StateMachine,#State1
         JMP   StateMachine

State1   do state1 stuff
         MOVS  StateMachine,#State2
         JMP   #Common

State2   do state2 stuff
         MOVS  StateMachine,#State3
         JMP   #Common

State3   do state3 stuff
         MOVS  StateMachine,#State1
         JMP   #Common
  ..

Common   common post-state code
  ..
         JMP   StateMachine
  ..

StateMachine  RES  1
```

Of course, more complicated state machines can be designed by choosing the next state based on some condition. But this simple example should be enough to get started with.

*Thanks to M. Park for suggesting improvements to this example.*
**Trap:** The Propeller has a TEST instruction that lets you set flags based on the outcome of ANDing the instruction’s source and destination arguments. *But be sure to tell it which flags to set using WZ and/or WC.* This doesn’t happen automatically, just because it’s a test instruction. The same applies to the compare instructions (CMP, CMPS, CMPSX, and CMPX).

**Trap:** You can use the compare and test instructions to test port input bits. *But beware statements like the following:*

```
TEST INA,#%0010     WZ
```

It just won’t work. Why? Because INA is read-only. And you can’t use read-only locations in the destination field of an instruction. You will have to do the following instead:

```
TEST Mask,INA       WZ

Mask                LONG %0010
```

The read-only locations include PAR, CNT, INA, and INB. PHSA and PHSB should also not be used in the destination field of these instructions.

**Trap:** The DAT section of a .spin file can contain assembly code for more than one cog. Each cog, when loaded, gets 512 longs from hub memory, beginning at the address given in COGNEW or COGINIT. The beginning of each cog’s assembly code should begin with an ORG statement. This resets the assembler’s address counter so that subsequent addresses will be correct when the code beginning at the ORG is loaded into its cog for execution. Now, you may be tempted to share code and data among different ORGed sections, as in the following example:

```
ORG
Cog1     CALL    #Common
..       .       .
ORG
Cog2     CALL    #Common
..       .       .
Common   common subroutine
..       .       .
Common_ret
```

*But resist the temptation.* Even though Common may be included with Cog1 when it loads, the address computed for it by the assembler will be relative to the beginning address of Cog2, because of the subsequent ORG. And it’s *that* (incorrect) address that gets assembled into Cog1’s CALL statement.
Trap: In Propeller assembly code, the usual direction of data transfer is from source to destination. The `WRBYTE`, `WRWORD`, and `WRLONG` are exceptions to the rule. These statements all write data from cog memory at `from_cog` to hub memory at `to_hub`:

```
WRBYTE from_cog, to_hub
WRWORD from_cog, to_hub
WRLONG from_cog, to_hub
```

Thanks to Paul Baker

Trap: Be sure to place `RES` statements at the end of any `ORG` segment that uses them. The following example does it wrong:

```
MOV  Time,CTR
ADD  Time,Offset
...
Time  RES  1
Offset  LONG  230
Other_value  LONG  123

ORG
Another_cog  ...
```

In this example, `Offset` will get clobbered when `CTR` is copied to `Time`, and `123` will get added to it instead of `230`. Why? Because when the assembler encounters a `RES`, it reserves space in cog memory, but not in hub memory where the program is stored. Consequently, `230` will occupy `Time`’s address in hub memory, and `123` will occupy `Offset`’s. Do this instead:

```
MOV  Time,CTR
ADD  Time,Offset
...
Offset  LONG  230
Other_value  LONG  123
Time  RES  1

ORG
Another_cog  ...
```

In this example, whatever’s assembled at `Another_cog` will get loaded into `Time` when the first cog loads. But we don’t care, since `Time` will get written over anyway.

Thanks to Beau Schwab
**Trap:** The Propeller assembler won’t let you **CALL** a subroutine with a local label. For example, the following is not allowed:

```
MOV :Arg, #1
CALL #:Subr
..

:Subr TEST :Arg, #%101
..

:Subr_ret RET

:Arg LONG 0-0
```

But there *is* a work-around. Just use **JMPRET** instead:

```
MOV :Arg, #1
JMPRET :Subr_ret, #:Subr
..

:Subr TEST :Arg, #%101
..

:Subr_ret RET

:Arg LONG 0-0
```

It’s not as elegant-looking as a **CALL**, but it *does* work.

**Trap:** The **PAR** register’s bits 1..0 *always* read as zero. This is okay as long as you’re passing a long address to a newly-created cog via **COGNEW** or **COGINIT**. But if you use **PAR** to pass any other kind of parameter, you will need to shift it left by two to avoid getting its two LSBs zeroed, then shift it back in the assembly routine:

```
COGNEW(@new_cog, arg << 2)
..

DAT

new_cog MOV my_arg, PAR
SHR my_arg, #2
```

Also, because the upper 16 bits of the 32-bit parameter field are ignored, this effectively limits argument size to 14 bits.

*Thanks to Rayman and BradC for pointing out that bits 31 .. 16 are not passed.*
**Trap:** The “@” prefix can be used in both Spin code and assembly to denote the address of some memory location. But the same expression, `@Address`, can mean two different things, depending on where it’s used. Here’s an example:

```
VAR
    LONG var_begin

PUB start
    var_begin := @begin
    COGNEW(var_begin, 0)

DAT
    begin JMP #begin
    my_begin LONG @begin
```

In the Spin code, `@begin` refers to the actual location of `begin` in hub memory, and `var_begin` will be assigned that value. In the assembly code, one might expect `my_begin` to hold the same value, but it does not. What it contains is the offset of `begin` from the beginning of the current object, and that will not be the same as its location in hub memory. The rule is this (quoting the manual):

“It is important to note that this is a special operator that behaves differently in variable expressions than it does in constant expressions. At run-time, ... it returns the absolute address of the symbol following it. This run-time, absolute address consists of the object’s program base address plus the symbol’s offset address.

“In constant expressions, it only returns the symbol’s offset within the object. It can not return the absolute address, effective at run-time, because that address changes depending on the object’s actual address at run-time.”
**Trick:** Suppose we want to start a new cog from an assembly program. To do so, we need the hub address of that program, so we can pass it to \texttt{COGINIT}. But in the prior trap, we discovered that \texttt{@program} won’t give us that address. What to do?

Here’s where the \texttt{@@} notation comes to the rescue. This notation, prepended to an address (\texttt{@@addr}), means, “Return the value obtained from adding the current object’s base address in the hub, to \texttt{addr}’s offset from that base address: in other words, \texttt{addr}’s hub address. By this logic, then, \texttt{@@0} will refer to the hub address of the beginning of the object. We can use that to advantage:

```
CON
    _clkmode   = xtal1 + pll16x
    _xinfreq   = 5_000_000
PUB start
    COGNEW(@begin, @@0)
DAT
begin         ADD       :new_cog,PAR
               SHL       :new_cog,#2
               OR        :new_cog,#%1000
               COGINIT   :new_cog
               COGID     :self
               COGSTOP   :self
:new_cog      LONG      @new_begin
:self         RES       1
new_begin     MOV       dira,#1
:loop         XOR       outa,#1
               JMP       #:loop
```

The code at \texttt{begin} receives the object’s base address in hub memory (\texttt{@@0}) in \texttt{PAR}. It adds this to \texttt{@new_begin}, the offset of \texttt{new_begin} from the beginning of the object, to get its hub address. It then constructs the rest of the argument to \texttt{COGINIT}, which is used to start \texttt{new_begin} running in a new cog. Finally, it stops its own cog.

*Thanks to Chip Gracey*
**Trap:** Beware the “≤” and “≥” operators. In many computer languages, these mean “less than or equal to” and “greater than or equal to”, respectively. Not so in Spin. Here, these are assignment operators and work like this:

\[
\begin{align*}
\text{a} &\text{ <= b} \quad \text{is equivalent to} \quad \text{a} := \text{a < b} \\
\text{a} &\text{ >= b} \quad \text{is equivalent to} \quad \text{a} := \text{a > b}
\end{align*}
\]

For “less than or equal to”, use “=≤”; for “greater than or equal to”, “=≥”.

*Thanks to Beau Schwab*

**Trap:** Failure to account for the clock speed can lead to program lockup, especially with spin code. For example, the following snippet out of `serialDemo.spin` (in the Tips & Traps forum thread) will lock up if the baud rate is set too high for the selected clock frequency. Here is the offending code:

```spin
\[
\begin{align*}
\text{t} &= \text{cnt} \\
\text{repeat 10} &\text{ '10 bits.} \\
\text{waitcnt(t += bittime)} &\text{ 'Time for next bit edge: +=} \\
\text{outa[tx]} &\text{ := (b >>= 1) & 1 'Get the next bit.}
\end{align*}
\]
```

The trap there is that the value of the system counter, \texttt{cnt} is captured in the variable \texttt{t} before entering this routine. The idea then is that serial bits will be sent out at regular times, \texttt{t + bittime}, \texttt{t + 2 * bittime}, \texttt{t + 3 * bittime}, etc. However, the spin code in the line following the \texttt{waitcnt} instruction, plus the \texttt{repeat} loop overhead, can take longer than \texttt{bittime}. If that happens, \texttt{cnt} will already have passed \texttt{t + bittime} when it hits the \texttt{waitcnt} instruction, and it will have to wait there for the whole cycle of 32 bits to roll back to the value, for each bit. This limits the baud rate to 19200 for an 80 MHz clock (5 MHz crystal, plus 16x PLL), or 1200 baud with a 5 MHz clock. Of course assembly code is faster, but this is still a trap for fast actions that use repeated \texttt{waitcnt}.

*Thanks to Tracy Allen*
**Trap:** Code does not run as expected when using Hyperterminal. As with the BASIC Stamp, the DTR line resets the Propeller; and when reset, the code stored in EEPROM will be loaded, replacing whatever code was previously in the Propeller RAM. Hyperterminal brings DTR high when it connects and returns it low when it disconnects. With the FDTI programming adapter at least, the Propeller resets when DTR goes from high to low.

A better option is to use the DEBUG window in the Stamp IDE, since DTR is not automatically set. Here, you can manipulate DTR manually through the DTR checkbox to see the effect of a reset on the Propeller.

*Thanks to Tracy Allen*

**Trick:** When a Spin cog is started with COGNEW, you have to give it the address of an array in hub RAM to use as a stack. But how big does this array have to be? The answer to that depends on the Spin code in the cog and how deeply it nests procedure calls. To get an empirical idea how big to make the stack area, you can use the following Spin object:

```spin
CON
  filler = $5aa5a55a

VAR
  word  my_stack[32]

PUB start(addr, size)
  longfill(addr, filler, size)
  return cognew(monitor(addr, size), @my_stack) => 0

PUB monitor(addr, size) | used, i
  dira := $FF0000
  outa := 0
  repeat
    used := 0
    repeat i from addr to addr + size * 4 step 4
      used -= long[i] <> filler
    outa := used << 16
  The procedure start should be called from your top-level program before you call COGNEW and launch your Spin cog. It takes two arguments: the address of the stack array used by your cog, and its size in LONGs. It then fills this area with a pattern before it launches its own cog that does the monitoring. This cog will continuously examine the stack array, counting the number of LONG locations in which the pattern has been changed. It then displays this number on the development board’s LEDs in binary. You can then use this number, padded appropriately for safety, as the dimension of your stack array.
**Trick:** You want to define an array in the **DAT** area, initialized to a certain value. How do you go about it? Use the following array constructor:

```plaintext
CON
  MySize = 64

DAT
  MyVar   LONG  $AA55[MySize]
```

This defines a block of **MySize** **LONG**s in cog memory, initialized to **$AA55** (assuming it's part of an assembler program that gets its own cog). 

But, perhaps more interestingly, it also defines a single instance of an array in hub RAM, initialized to **$AA55**, that can be shared by multiple instances of the defining object -- or anyone else, for that matter, if they know its address.

---

**Trick:** Instead of using a separate pointer variable in assembly language array indexing, you can embed the pointer right into the code, *viz*:

```plaintext
movd :loop, #buffer
mov  :i, #BufferSize

:loop
  mov  0-0, ina
  add :loop, :d_inc
  djnz :i, #:loop

:d_inc   long 1 << 9   'LSB of destination field.
:i       res  1
:buffer  res  BufferSize
```

The loop is first initialized by writing the buffer address into the destination field of the **mov** instruction at :loop. (The “0-0” just signifies that something will be written into that field.) The loop then copies the value of **ina** to the next position of the buffer. The **add** instruction increments the destination field of the **mov** instruction, effectively advancing the pointer into the buffer.

*Thanks to Paul Baker*
**Trap:** In working with the Propeller, the biggest trap of all is the set of preconceived notions brought to bear from work with other microcontrollers. That this is particularly true of the Propeller’s shift and rotate instructions (shl, shr, sar, rol, ror, rcl, and rcr) is an understatement. The trap here is the handling of the carry flag. In most micros, the carry flag is handled as an extension of the register being shifted or rotated. Right shifts/rotates treat the carry simply as a less-significant bit than the LSB of the register; left shifts/rotates, as a more significant bit than the MSB of the register. And rotates through the carry treat it as just another bit linking the register’s LSB and MSB in a continuous loop. For the Propeller, however, this is not a useful model to keep in mind. In fact it will be more hindrance than help. Granted, shifts/rotates of a single bit position will behave just like they do on other micros. But that’s where the similarity ends. This is because, no matter how many positions are shifted, the carry (if written) always gets the initial value of bit 0 for right shifts/rotates or bit 31 for left shifts/rotates. Here’s an illustration of each instruction, starting with the following configuration in Data and Carry, and showing the result of each given command:

<table>
<thead>
<tr>
<th>Data:</th>
<th>Carry:</th>
</tr>
</thead>
<tbody>
<tr>
<td>abcdefghijklm-------nopqrstuvwxyz</td>
<td></td>
</tr>
<tr>
<td>shl Data,#4 wc</td>
<td>efgijklm-------nopqrstuvwxyz0000</td>
</tr>
<tr>
<td>shr Data,#4 wc</td>
<td>0000abcdefhijklm-------nopqrstuvwxyz</td>
</tr>
<tr>
<td>sar Data,#4 wc</td>
<td>aaaaaabcdefhijklm-------nopqrstuvwxyz</td>
</tr>
<tr>
<td>rol Data,#4 wc</td>
<td>efgijklm-------nopqrstuvwxyzabcdef</td>
</tr>
<tr>
<td>ror Data,#4 wc</td>
<td>wxyzabcdefhijklm-------nopqrstuvwxyz</td>
</tr>
<tr>
<td>rcl Data,#4 wc</td>
<td>efgijklm-------nopqrstuvwxyzCCCC</td>
</tr>
<tr>
<td>rcr Data,#4 wc</td>
<td>CCCCCabcdefhijklm-------nopqrstuvwxyz</td>
</tr>
</tbody>
</table>

Note especially the rcl (rotate carry left) and rcr (rotate carry right) instructions. These are not actually rotates at all, but shifts, in which the initial value of the carry bit (instead of 0) fills the vacated positions.

**Trick:** You can use the rcr instruction to advantage when averaging signed numbers, thus:

```
mov average, value0
add average, value1 wc
rcr average, #1
```

rcr, in this case, acts like sar but with 33 bits instead of 32, the “sign” bit being the carry.
**Trick:** You can extend the sign from any bit position in both Spin and Assembly by doing two shifts. For example, suppose you have a 24-bit number whose MSB (bit 23) is the sign, and you want to create a 32-bit signed value from it. In Spin, do this:

```
Value := Value << 8 ~> 8
```

In assembly, the equivalent would be:

```
shl Value,#8
sar Value,#8
```

In each case, `Value` is first shifted left to get its sign bit into bit 31 of the long. Then an arithmetic right shift propagates the sign into the 8 MSBs.

*Thanks to Chip Gracey via Jon Williams*

**Note:** In Spin, two sign extension operators are pre-defined. To extend the sign from bit 7 (byte), do this:

```
~Value
```

And to extend from bit 15 (word), do this:

```
~~Value
```

*Thanks to M. Park*