PARALLAX SEMICONDUCTOR
www.parallaxsemiconductor.com
sales@parallaxsemiconductor.com
support@parallaxsemiconductor.com
phone: 916-632-4664 • fax:916-624-8003

Application Note AN014

Coroutines in Propeller Assembly Language

Abstract: The multicore P8X32A does not require traditional interrupts to manage multiple processes simultaneously; yet multi-tasking in a single core is still a handy option. Coroutines in Propeller Assembly language can support a pair of independent tasks in a single core (cog) through the use of the JMPRET instruction.

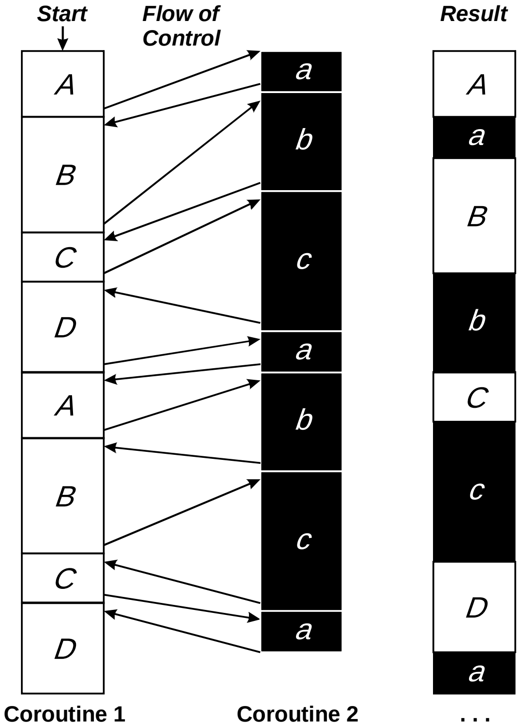
Introduction

The P8X32A Propeller has eight independent processors, called “cogs,” each of which can support multiple cooperative tasks. The simplest way to implement multitasking in Propeller Assembly language (PASM) for two tasks is to write the tasks as coroutines.

Coroutine Principle

Two programs are said to be “coroutines” when they take turns executing in such a way that, when each gets its turn, it resumes from the point where it left off. In the Propeller, coroutines take turns cooperatively, rather than by means of an interrupt, so each must yield its turn to the other in a planned ping-pong fashion. Figure 1 illustrates the principle as a flow chart.

Figure 1: Coroutines Flowchart



Here, execution bounces back and forth between **Coroutine 1** and **Coroutine 2** in such a way that each gets a small slice of time before yielding to its complement. The result is an interleaved execution that can give the appearance that each is operating independently in real time.

Coroutines are often employed in programs that do both input and output independently, such as those that perform full-duplex asynchronous serial I/O, or that monitor an encoder and control a motor.

Propeller Coroutine Implementation

To understand how coroutines work in the Propeller, it is necessary to have a feel for the Propeller's execution sequence. Propeller instructions are processed in four steps:

1. **I**nstruction fetch
2. **W**rite result of previous instruction
3. **S**ource operand fetch
4. **D**estination operand fetch

This means that if the next instruction is the destination of the current instruction's result, the old contents of that instruction slot will be read first before the new value is written. Such behavior is what makes Propeller coroutine switching both simple and fast.

Consider the **`jmpret`** **d,s** instruction. This instruction transfers control to address **s** and stuffs the address of the instruction following the **`jmpret`** into the source field of the instruction at address **d**. The latter address most commonly holds a **`jmp`** instruction, so that **`jmpret`** can be used to effect subroutine calls. In fact, the **`call`** instruction is just a **`jmpret`** in disguise. For example, this:

```
'-------[ Main Program ]----------------------------------------

              ...
              jmpret    subr_ret,#subr   'Call subr.
              ...

'-------[ Subroutine ]------------------------------------------

subr                                      'Subroutine entry point.
              ...
subr_ret      jmp       #0-0              'Return to caller....
```

…is exactly equivalent to this:

```
'-------[ Main Program ]----------------------------------------

              ...
              call      #subr             'Call subr.
              ...

'-------[ Subroutine ]------------------------------------------

subr                                      'Subroutine entry point.
              ...
subr_ret      ret                         'Return to caller.
```

Now, consider what would happen in the situation below, where the target of the `call` and the return statement reside at the same location.

```
              call      #swap
:nxt          ...

swap
swap_ret      ret
```

In this example, the following steps take place at the `call`:

1. The processor reads the instruction at **swap** (i.e. **ret** = **jmp** #ret_addr).
2. The processor stuffs the address of **:nxt** into the source field of **swap**, replacing **ret_addr**.
3. The **jmp** #ret_addr instruction executes.

The next time a **call** issues to **swap**, execution will transfer to **:nxt**, and the caller's return address will get stuffed into **swap**'s source field. And so it goes, back-and-forth. This then is the essence of Propeller coroutine switching.

Program Example

In this example program, two coroutines control two separate LEDs, blinking them at different rates. (This program is designed to run on the Propeller Demo Board[1] but can be modified for other platforms simply by changing values in the `CON` section as appropriate.)

```
CON                                             'Spin setup code

   _clkmode       = xtal1 + pll16x
   _xinfreq       = 5_000_000

   PING_LED       = 16
   PONG_LED       = 17

PUB  Start

  ping_period := clkfreq * 11 / 16              'Set periods for both LEDs.
  pong_period := clkfreq * 13 / 16

  cognew(@pingpong, 0)

DAT                                             'PASM code

pingpong      or        dira,ping_mask          'Enable PING_LED output.
              or        dira,pong_mask          'Enable PING_LED output.
              mov       ping_time,cnt           'Initialize timers.
              mov       pong_time,ping_time

'-------[ ping coroutine ]-----------------------------------------------------

ping          call      #swap                   'Give the pong coroutine a chance.
              'Pong coroutine returns here.
              mov       acc,cnt                 'Is it time to change state?
              sub       acc,ping_time
              cmp       acc,ping_period wc
        if_c  jmp       #ping                   '  No:  Keep checking.

              add       ping_time,ping_period   '  Yes: Add to get the next time.
              or        outa,ping_mask          '       Turn LED on.
```

```
ping_on       call      #swap                   'Give the pong coroutine a chance.
              'Pong coroutine returns here.
              mov       acc,cnt                 'Is it time again to change state?
              sub       acc,ping_time
              cmp       acc,ping_period wc
        if_c  jmp       #ping_on

              add       ping_time,ping_period   '  Yes: Add to get the next time.
              andn      outa,ping_mask          '       Turn LED off.
              jmp       #ping                   'Loop back...

'-------[ pong coroutine ]-------------------------------------------------------

pong          mov       acc,cnt                 'Is it time to change state?
              sub       acc,pong_time
              cmp       acc,pong_period wc
        if_nc add       pong_time,pong_period   '  Yes: Add to get next time.
        if_nc xor       outa,pong_mask          '       Invert output pin.
              call      #swap                   'Give the ping coroutine a chance.
              jmp       #pong                   'Loop back...

'-------[ coroutine swapper ]----------------------------------------------------

swap
swap_ret      jmp       #pong                   'Initialize swap to point to pong.

'-------[ variables ]------------------------------------------------------------

ping_mask     long      1 << PING_LED
pong_mask     long      1 << PONG_LED
ping_period   long      0-0
pong_period   long      0-0

ping_time     res       1
pong_time     res       1
acc           res       1
```

In this program, **ping** and **pong** do the same things with their respective LEDs. However, **ping** explicitly turns its LED on and off, while **pong** simply toggles its LED. The **ping** coroutine, with two distinct calls to **swap**, illustrates that execution picks up from where it left off when the other coroutine returns.

Here are a couple other points worth noting:

1. Both **ping** and **pong** use the shared variable **acc**. This is okay, so long as the value of a shared variable does not have to span a call to **swap**, because the complementary coroutine might clobber it.
2. Never rely upon the states of the **c** and **z** flags across calls to **swap**. Without extremely careful programming, the complementary coroutine will almost certainly clobber them. If it should be necessary to retain the zero and/or carry flag state(s) across a call to **swap**, use the flag save and restore instructions illustrated below. (The **$**'s in **restore_z** and **restore_c** refer to the current instruction address.)

```
'Save Z and C in their restore instructions:

save_z        muxnz     restore_z,#1            'Save NZ bit in restore_z, bit 0.
              call      #swap                   'Swap to alternate coroutine.
restore_z     test      $,#0-0 wz               'Test to restore Z flag.
```

```
save_c        muxc     restore_c,#1          'Save C bit in restore_c, bit 0.
              call     #swap                 'Swap to alternate coroutine.
restore_c     test     $,#0-0 wc             'Test to restore C flag.

'Save Z and C in an external long:

save_z        muxnz    flags,#%10            'Save NZ bit in flags, bit 1.
save_c        muxc     flags,#%01            'Save C bit in flags, bit 0.
              call     #swap                 'Swap to alternate coroutine.
restore_z     test     flags,#%10 wz         'Restore Z from flags, bit 1.
restore_c     test     flags,#%01 wc         'Restore C from flags, bit 0.

              'or

restore_zc    shr      flags,#1 wz,wc,nr     'Restore Z and C from flags, bits 1 and 0.
                                             'Bits 31..2 of flags must be zero.

flags         long     0
```

Resources

Download the coroutine example code from www.parallaxsemiconductor.com/an014.

References

1. Propeller Demo Board; Parallax #32100, www.parallax.com

Revision History

Version 1.0: original document.