Get into Parallel Processing with the Parallax Propeller™

Introducing the Propeller

Every now and again something different comes along. Microcontroller chip development has proceeded down the same paths for many years now: either the same basic 'core' processor being surrounded by more and peripherals or the processor itself being made more and more powerful. Examples of the former include 8051-core devices, the latter ARM-based microcontrollers. The common feature is a single processor supported by specialist, dedicated logic providing features like Pulse Width Modulation output and pulse

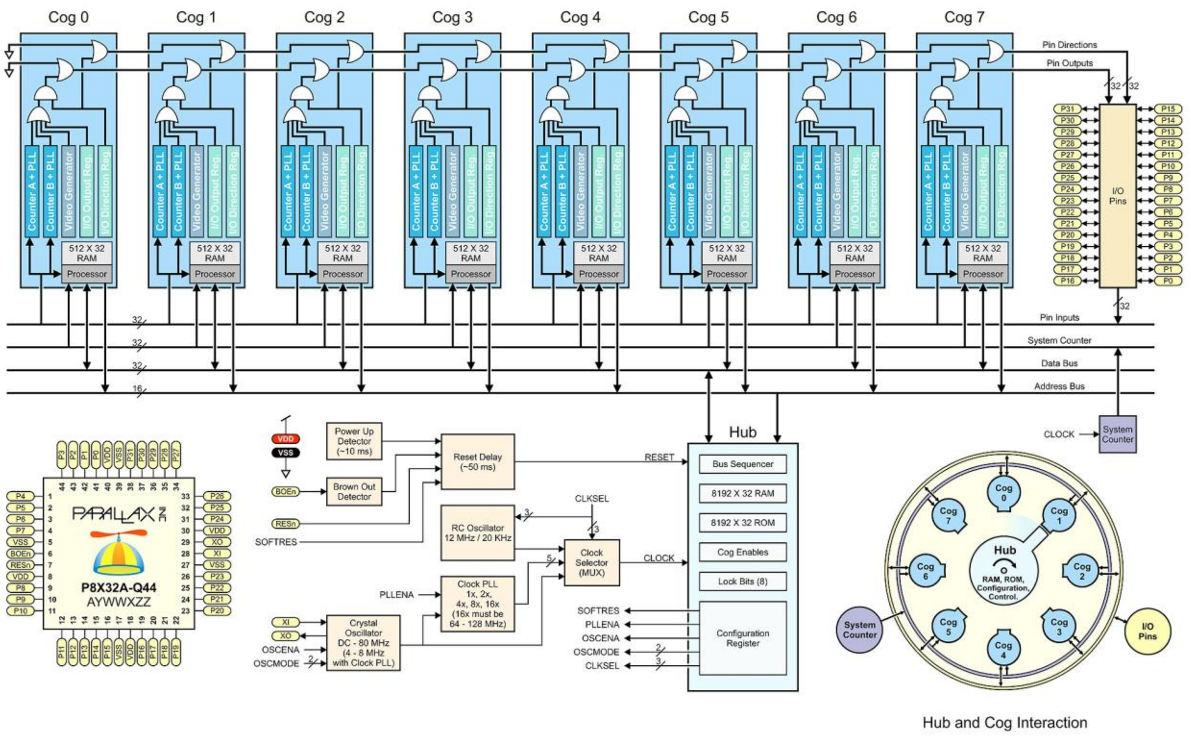


Fig.1 Propeller internal architecture

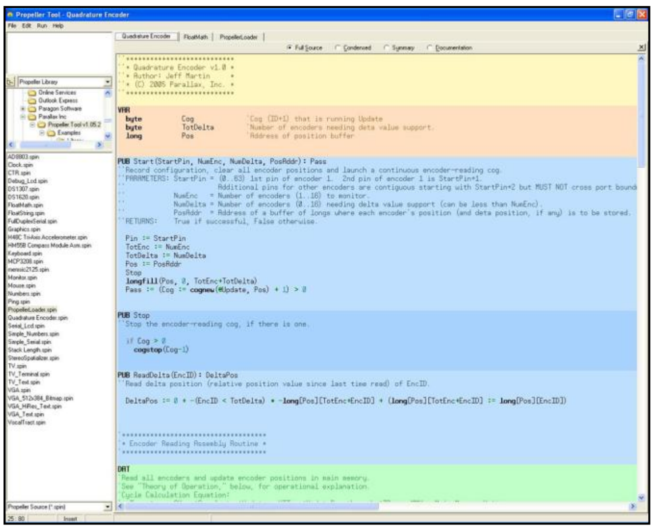
counting input. Even Cypress' PSoC® system still involves a single embedded control core. The Propeller from Parallax represents a major change in design philosophy. This device contains eight 32-bit processors or 'COGs' with minimal support logic and only the most basic I/O hardware (Fig.1). A first reaction to this layout might be: 'Great, I can implement that Neural network project with each COG running essentially the same program'.

While pure parallel processing may indeed be a good use for the Propeller, I don't believe this was the main driver behind the design. The idea is to give the engineer maximum control over the *peripheral* system in a particular application. You may still have a single

COG running the top-level program, farming out lower-level tasks such as serial I/O to another COG as and when required. This is the really fascinating feature of this device: the ability to reconfigure itself under program control to suit the requirements at a particular time and then to shut processes down when no longer needed, perhaps re-assigning processor resources to a completely different task. The processor clock is also under program control so power consumption can be reduced if high speed is not needed when implementing slow I/O such as RS-232.

The Starter Kit Hardware

The kit contains a very small demonstration board packed with various I/O sockets, some rather surprising: VGA output to a monitor, TV output, PS/2 keyboard and mouse sockets. The video outputs are provided because the chip contains as part of its central resource ROM the look-up table of a character generator. The only ‘conventional’ I/O port is USB derived from the on-board FTDI chip. The UART function that drives this device, is of course implemented entirely in software and runs on one of the COGs.



```
''***************************
''* Quadrature Encoder v1.0 *
''* Author: Jeff Martin     *
''* (C) 2005 Parallax, Inc. *
''***************************

VAR
  byte        Cog          'Cog (ID+1) that is running Update
  byte        TotDelta     'Number of encoders needing deta value support.
  long        Pos          'Address of position buffer

PUB Start(StartPin, NumEnc, NumDelta, PosAddr): Pass
''Record configuration, clear all encoder positions and launch a continuous encoder-reading cog.
''PARAMETERS: StartPin = (0..63) 1st pin of encoder 1.  2nd pin of encoder 1 is StartPin+1.
''                       Additional pins for other encoders are contiguous starting with StartPin+2 but MUST NOT cross port bound
''            NumEnc   = Number of encoders (1..16) to monitor.
''            NumDelta = Number of encoders (0..16) needing delta value support (can be less than NumEnc).
''            PosAddr  = Address of a buffer of longs where each encoder's position (and deta position, if any) is to be stored.
''RETURNS:    True if successful, False otherwise.

  Pin := StartPin
  TotEnc := NumEnc
  TotDelta := NumDelta
  Pos := PosAddr
  Stop
  longfill(Pos, 0, TotEnc+TotDelta)
  Pass := (Cog := cognew(@Update, Pos) + 1) > 0

PUB Stop
''Stop the encoder-reading cog, if there is one.

  if Cog > 0
    cogstop(Cog-1)

PUB ReadDelta(EncID): DeltaPos
''Read delta position (relative position value since last time read) of EncID.

  DeltaPos := 0 + -(EncID < TotDelta) * -long[Pos][TotEnc+EncID] + (long[Pos][TotEnc+EncID] := long[Pos][EncID])


'**********************************
'* Encoder Reading Assembly Routine *
'**********************************

DAT
'Read all encoders and update encoder positions in main memory.
'See "Theory of Operation," below, for operational explanation.
'Cycle Calculation Equation:
```

Fig.2 Source code in Propeller Tool

All communication with the IDE software on the PC - Propeller Tool – is via the USB port. There is a serial EEPROM on-board which communicates with the Propeller via an I^2C bus which, you guessed it, is implemented in software run by a COG. It provides non-volatile memory for user programs. These I/O routines are loaded from system ROM at Reset to allow programs to be downloaded from the PC or from the EEPROM, *but are then shut down* before the user’s program begins execution. If your program requires these I/O resources, then it will have to load them and assign COG(s) as appropriate. This may seem awkward at first, but why have unwanted resources cluttering up memory space if you don’t need them?

Propeller Tool

The IDE that comes with the starter kit is called Propeller Tool and provides program editing, compilation of the high-level language Spin, and downloading to the demo board. You have the option of programming in Spin, assembly language or a combination of both. Obviously the assembler produces more efficient, faster operation and there is the usual trade-off between faster development and faster operation.

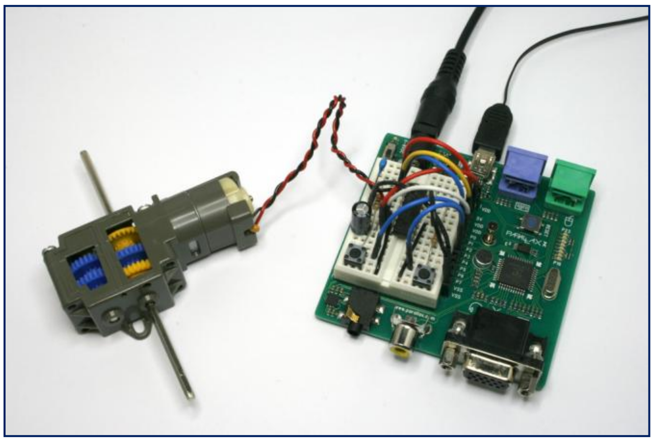


Fig.3 The wired-up demo board

The editor screen is very colourful (Fig.2) and the automatic assignment of different colours to code blocks is supposed to aid understanding. A number of radio buttons allow you to view sub-sets of the full source code, for example, just the comments. There are two options for downloading and running: compile and run in COG RAM, and compile and send to the EEPROM from where it is automatically loaded into RAM by the device bootloader.

The first method is best for development, only transferring to non-volatile memory when the code works.

Using the Demo Board

To illustrate some of the main features of Propeller programming a task was devised involving the speed control of a small DC motor using PWM. Two pushbuttons provide Speed Up and Speed Down inputs. The drive capability of the I/O ports is insufficient for the motor used so an H-Bridge circuit was constructed from half of an L293D quad driver chip. This was mounted on the breadboard together with two ‘Tact’ switches, pull-up resistors and decoupling capacitors (Fig.3). Note the use of the D-variant of this chip which has built-in protection diodes for driving inductive loads. Only Ports 0 to 7 of the Propeller are available to the user out of a possible 32, the others being committed to EEPROM busses, etc. on this demo board. The Propeller is a +3.3V device although both +3.3V and +5V regulated supplies are available. Hence the logic of the L293D works off the +3.3V supply, while its separate motor supply pin is connected to +5V. A small but very useful feature is the Ground or 0V post which takes the croc clip from a ‘scope probe.

```
''       ****************************************************************
''       * Simple DC motor speed controller for the Parallax Propeller *
''       *                    W.G.Marshall 2010                        *
''       ****************************************************************

''Port 0 = PWM output
''Port 1 = Speed Up button input
''Port 2 = Speed Down button input

CON
_clkmode = xtal1 + PLL4X
_clkfreq = 20_000_000

VAR
  word Ratio                           'Ratio links the routines in COGs 0 & 1
  long Stack[9]                        'Make stack space for COG 1

PUB Main
''Initialisation of ports and program start
    dira[0..2] := %100                 'Set Ports 0 = output, 1 & 2 = input
    Ratio := 10000                     'Initial PWM 50% with frequency = 1 kHz
    cognew(Buttons, @Stack)            'Start COG 1 running Buttons routine
    Toggle                             'while COG 0 runs PWM generator routine

PUB Toggle | Time
''COG 0 produces PWM signal with Mark/Space ratio set by variable Ratio
    Time := cnt                        'Set base time from system counter
    repeat                             'Repeat next 4 lines forever
     waitcnt(Time += 20000 - Ratio)    'Wait for interval set by Ratio
     outa[0]~~                         'Set Port 0 = 1
     waitcnt(Time += Ratio)            'Wait for interval set by Ratio
     outa[0]~                          'Set Port 0 = 0

PUB Buttons | Width
''COG 1 monitors two pushbuttons to derive value for Ratio
    repeat                             'Repeat next 8 lines forever
     Width := Ratio
     waitpne(%110, %110, 0)            'Wait for button press
     if ina[1] == 0                    'If Speed UP button pressed
       Width := Width + 1 <# 17500     'then increment Width to max 17500
     else                              'Speed DOWN button pressed
       Width := Width - 1 #> 2500      'so decrement Width to min 2500
     Ratio := Width
     waitcnt(10000 + cnt)              'Wait before checking buttons again
```

Listing 1. PWM generator using just Wait statements

Programming in Spin

A first attempt for the program to drive the motor is given in Listing 1. It is not presented as optimal solution but does illustrate some of the key features of Propeller programming. The aim is to use two COGs; one driving the PWM output with a mark/space ratio set by the global variable Ratio, the second monitoring two push button inputs and setting the value of Ratio. The PWM frequency is to be 1kHz.

The CON statements set up two system constants and fix the clock speed. We decided on a 20 MHz clock so the internal PLL multiplier is set to 4 given the 5 MHz crystal supplied with the board. Next, the VAR statements set up global variables: Ratio as mentioned, and Stack which assigns stack space for the second COG.

The first public method, PUB Main performs the usual initialization tasks including setting an initial value of Ratio equivalent to 50% PWM. Now comes the first really interesting instruction: COGNEW. This is what launches the second COG. Up to now COG 0 has been doing everything, running the boot-loader and then the first part of our program. COGNEW tells it to load the public method Buttons into the next free COG, in this case COG 1, and set it running. Once it has done that it launches the Toggle method and runs that from now on. A feature peculiar to the Propeller is the sharing of the 32 GPIO port lines by all processors. Each COG has its own Port Direction register, each output of which is 'OR-ed' with the next COG's register (see Fig.1). A COG requiring an output port needs to set the appropriate bit in its Direction register to logic 1. Once set it also enables the corresponding output from the COG I/O register to drive the I/O pin. Care must be taken to ensure that two COGs don't try and use the same port line for output, as program operation will not be as expected! Port input is completely independent and any COG may read the state of any port pin at any time. A COG can check its own output or indeed monitor what other COGs are doing on port pins they have set as outputs.

The Propeller has no interrupt system so there are a number of Wait instructions which cause program execution to pause until some event takes place. WAITCNT suspends operation for the specified number of system clock cycles by checking the value of a target figure against the value of the System Counter CNT. This is how the PWM waveform on Port 0 is generated by Toggle.

```
''       **************************************************************
''       * Simple DC motor speed controller using counters for timing *
''       *             PWM mark/space ratio from 0 to 100%            *
''       *                        W.G.Marshall 2010                   *
''       **************************************************************

''Port 0 = PWM output
''Port 1 = Speed Up button input
''Port 2 = Speed Down button input

CON
_clkmode = xtal1 + PLL4X
_clkfreq = 20_000_000

VAR
  word Ratio                           'Ratio = PWM pulse width
  word Period                          'Period = PWM period
  long Stack[9]                        'Make stack space for COG 1

PUB Main
''Initialisation of ports, counters and program start
    Ratio := 10000                     'Initial PWM 50%
    Period := 20000                    'Set PWM period
    ctra[30..26] := %00100             'Configure Counter A to NCO/PWM mode
    ctra[5..0] := %00000               'Direct Counter APIN to Port 0
    frqa := 1                          'Set counter increment to 1
    dira[0..2] := %100                 'Set Ports 0 = output, 1 & 2 = input
    cognew(Buttons, @Stack)            'Start COG 1 running Buttons routine
    Toggle                             'COG 0 runs PWM generator routine

PUB Toggle | Time
''COG 0 produces PWM signal with pulse width set by variable Ratio
    Time := cnt                        'Set base time from System Counter
    repeat                             'Repeat next 3 lines forever
     phsa := -Ratio                    'Load negated Pulse width into PHS
     Time += Period                    'Time = Time + Period
     waitcnt(Time)                     'Wait for interval set by Time

PUB Buttons | Width
''COG 1 monitors two pushbuttons to derive value for Ratio
    repeat                             'Repeat next 8 lines forever
     Width := Ratio
     waitpne(%110, %110, 0)            'Wait for button press
     if ina[1] == 0                    'If Speed UP button pressed
       Width := Width + 1 <# Period    'then increment Width to max Period
     else                              'Speed DOWN button pressed
       Width := Width - 1 #> 0         'so decrement Width to min 0
     Ratio := Width
     waitcnt(6000 + cnt)               'Wait before checking buttons again
```

Listing 2. Improved PWM generator using a hardware counter

The WAITPNE instruction in the Buttons method waits for Port 1 or Port 2 (or both) to go to a logic 0. In other words it waits for a button to be pressed. The beauty of these Wait instructions is that the COG operation is suspended with its power consumption reduced by over 85%. You can see that the COG running Buttons spends most of its time ‘asleep’, only waking when necessary. Because of instruction overheads the value of the parameter in the WAITCNT instructions of Toggle must not be less than 2500. This imposes a maximum and minimum for the PWM mark/space ratio. Hence the max (<#) and min (#>) statements in Buttons.

Using the COG’s counters

The program in Listing 1 is one solution, but it has a major drawback: because of the minimum size requirement for the parameter of WAITCNT statements the PWM range is limited from about 10% to 90% mark/space ratio. Suppose we need the full 0 to 100% range? Each COG does have a simple ‘Count/Capture Unit’ made from some registers and a few bits of logic. There are two identical counters, A and B each consisting of three registers CTR, FRQ and PHS. CTR sets the operational mode, PHS is the accumulator holding the current value and FRQ is added to PHS when required. Refer to Listing 2 to see how Counter A of COG 0 helps generate the PWM signal.

The counter is set up in PUB Main. First, the CTRA register is set to select PWM mode and Bit 31 of PHSA connected to output Port 0. FRQA is set to 1 so that PHSA is incremented by one for each cycle of System Clock.

In PUB Toggle PHSA is loaded with the negative (2’s complement) value of Ratio. This of course sets Bit 31 or the ‘sign-bit’ of PHSA to logic 1. As this bit is connected to Port 0, the PWM output also goes high. PHSA is now automatically incremented at the System Clock rate by having FRQA added to it. After Ratio clock cycles, PHSA reaches zero and Bit 31 changes to logic 0. That is the end of the PWM pulse. While all this is happening the COG is sat in the WAITCNT statement for the duration of Period. Of course PHSA continues to increment, but the end of Period will be reached long before PHSA reaches a value setting Bit 31 high again. When WAITCNT times out the cycle repeats, with PHSA being reloaded with –Ratio. We have thus added some more parallel operation by having the Counter determine the pulse width, while independently the COG program is setting the period.

The max and min limits in PUB Buttons can now be set to Period and zero respectively as the WAITCNT restriction is removed.

Adding more features

Supposing you want to add a serial output feature that dumps data when a button is pressed? Just add a new public method that acts as a software UART and contains the WAITPNE instruction to monitor a third pushbutton. Then add a COGNEW calling this method after the existing COGNEW in the Main method. When this executes, the next free COG, in this COG 2 will be loaded with the new method and set running. COG 2 will then power down waiting for the dump data button to be pressed. When it is pressed, the serial out routine will run with no impact on the other two methods running in COGs 0 and 1.

Speeding it up

You would normally expect programs run by an on-board interpreter, in this case SPIN, to be slower than those in native assembler. The unique architecture of the Propeller does to some extent widen the speed gap. This is because user SPIN code is held in shared central RAM while each COG runs the interpreter in its own local memory. The hub provides access to central resources in a strict time sequence and a particular COG may be held up waiting its turn. Machine code from the assembler is stored and run in the COG local memory resulting in a considerable increase in throughput.

Summary

- Eight 32-bit processors (COGs) running independently at a maximum 20 MIPS each, giving a maximum for the whole chip of 160 MIPS.
- A hub controller allows COGs access to shared RAM/ROM in a fixed time sequence which means that all timing is predictable.
- 32 GPIO port lines are shared by all COGs.
- Each COG has simple Count/Capture hardware that can be linked to the GPIO lines.
- No interrupt system, extensive use of Wait instructions for software timing.
- Makes programming parallel processes extremely simple.
- Programming in both high-level language (SPIN) and assembler possible.

Essential reading

Propeller Application Note AN001 – Propeller counters v1.0

Downloadable from Parallax.com

Programming and Customizing the Multicore Propeller Microcontroller

Shane Avery *et al*   ISBN 978-0-07-166450-9   McGraw Hill