RTOS and the Parallax Propeller

Rich Lundeen
CS504: Real Time Operating Systems

Introduction

In CS 504:Real Time Operating Systems, the class built a functional Real Time Operating system based on the UIK API. In my implementation, I heavily relied on the source code of FreeRTOS. The process focused primarily on demonstrating the atmel processor, multitasking, and basic interprocess communication, which are fundamental parts of an RTOS.

In this project, the same problems are achieved with a very different architecture. Namely, all the first programming projects – generating a sound wave, counting LEDs, timing how long buttons were pressed, and handling hardware events – were programmed similarly and multi-tasked on the propeller chip. This project included interfacing with the ntsc and peripheral drivers, writing a sound driver, learning spin and propeller assembly, and comprehending the propeller architecture.

Architecture

Overview

The Propeller chip has 8 cores and is a symmetrical multiprocessing microcontroller. It uses "round robin" non pre-emptive shared memory access scheme. All cores run simultaneously and independently when not accessing main memory. The propeller chip is a 32 bit RISC architecture without pipelining. The chip runs at 80 MHz and the on chip memory is 64 KB. 32 KB is RAM, the second 32KB is ROM containing the spin interpreter, data tables, and other run-time objects needed. Both RAM and ROM share the same addressing scheme. In practicality, this means that most programs need to fit into 32 KB or make use of the expansion car triage.

There are 32 I/O pins. the cogs are each gated to the I/O control with AND and OR gates, so any conflict results in this logic. The programmer needs to take care so that a process running on one cog does not conflict with another. Each Cog also has its own 512x32 bit register space.

The Hydra kit I am using has both a 3.3 V power supply and a 5V power supply. Both are necessary because the propeller chip runs at 3.3 V, but the keyboard and mouse use 5V (as well as possible others.) The power is fed through a 9V DC wall adapter.

There are also peripherals for connecting NTSC, RJ-11, NES controllers, VGA, and PS2-Interfaces in the hydra kit. For the most part, there is very little hardware related to these. They are simply interfaces to the propeller chip. The chip does all the work.

Programming the board is done using a USB connection. This was chosen because USB is becoming more standard than serial ports on modern computers. However, USB is also more complicated. Because of this, there is a 0717 chip on board the hydra and the FTDI driver for Windows makes the USB interface appear to be a serial port to both the PC and the microcontroller.

Unfortunately, the primary language spin uses to initialize its functions is proprietary, and the only IDE and compiler to develop it only works in Microsoft Windows.

Interrupts

There are no interrupts on the propeller chip. The documentation recommends to "just assign some cogs to individual, high-bandwidth tasks and keep other cogs free." They claim the result is better response time and ease of maintenance [manual1.01]. In traditional architectures, external interrupt lines are fed to a controller which are handled by Interrupt Service Routines. With the Parallax processor the result is slightly different. Any I/O line can be configured as an interrupt device, while a cog can essentially wait for an interrupt to occur using polling.

In this project, the input was simple. One cog simply repeatedly checks for the NES controller to see if any buttons have been pressed. There are many drivers included with the documentation that handle input in a similar way. However, because input devices are usually slow compared with processing speed, it would be fairly simple to combine these and have several inputs read from one cog.

The loop might look something like the following pseudo spin code:

```
repeat
  'check if nes button is pressed
    'handle any input
  'check if mouse input
    'handle any input
  ...
```

Shared Data

Unlike the Atmel processors, much of the data access is automatic. It is assumed no more than one task is running per cog. There is a hub spinning in a round robin way, so each task can access the data according to that. The hub spins at half the clock frequency, so each cog can access data for 2/16 instructions. This eliminates some race conditions, but there are still some to consider.

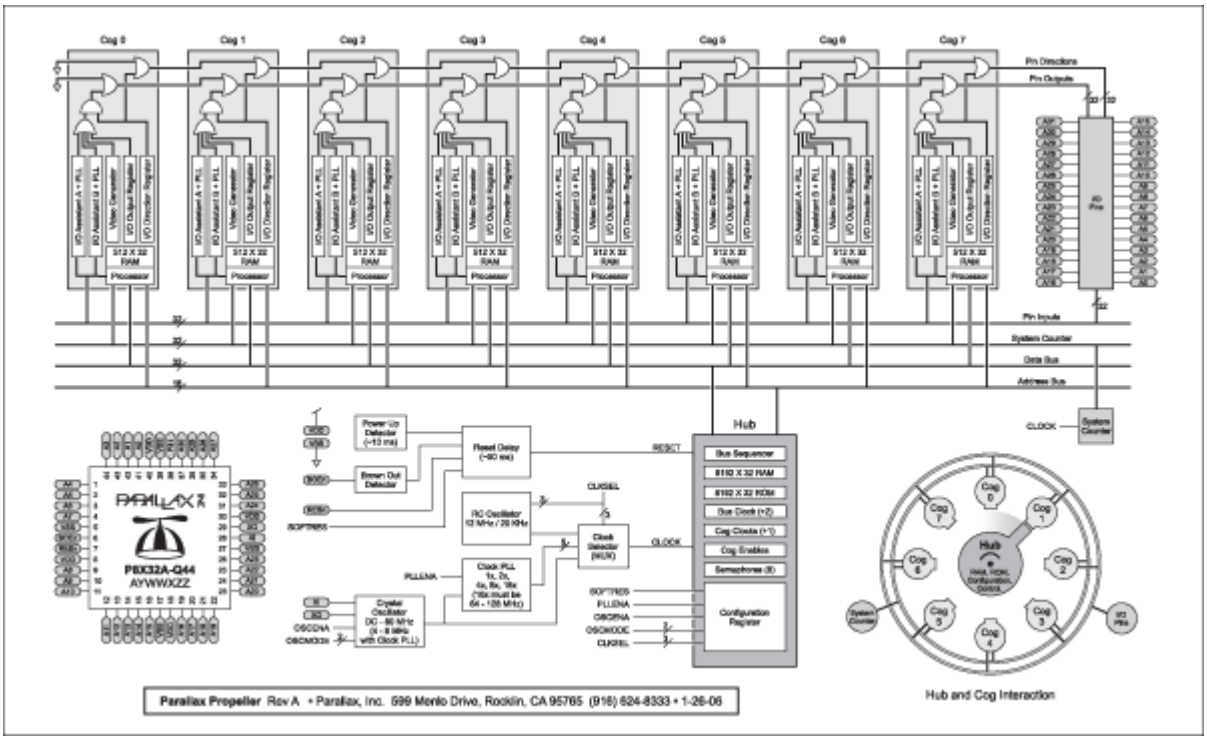

Figure 1: Overview of the Propeller Architecture, including the hub in the bottom right, from the Propeller Manualv1.01

The propeller is a Multiple Instruction Multiple Data (MIMD) architecture, which is common in parallel processing. To keep things simple, many times cogs are used as more of a task - eg one cog often does sound, one is going to do video, one does I/O, and so on. Many times algorithms are not parallelized. A bigger concern is to enable running tasks to communicate with each other. This is done with the 64k of shared memory. Locking mechanisms are built into the hardware (eg LOCKNEW, LOCKRET, LOCKSET, LOCKCLR). Locks are single bits that can only be accessed by one cog at a time, therefore, the hardware guarantees that no two cogs can access a lock at once. So locks can be used by cog programs to share resources. This lock is essentially equivalent to the binary semaphore programmed in class. Also, because hub uses a round robin technique, there is no possibility for any two cogs to ever access memory at the same time.

Programming

Assembly

Spin is an interpreted language, and not only executes up to 20 times slower than assembly, but also uses additional memory. Because of this, programs often use assembly for performance – especially when programming drivers or other low level functionality.

Because of the difficulty involved with assembly programming, most of the programming for this project was done using spin. An exception is constant locations in memory. This included information like the frequencies of notes, string data, etc. Since this involves little logic, a performance gain is present without any additional difficulty.

Spin

Spin is a very small interpreted language, optimized for space. It is indent oriented, and contains very few libraries (it has to fit into 512 longs). Every propeller program starts with spin. When the propeller boots, it first determines where to read the program from (using external data line) and then it boots cog 0 and loads the spin interpreter into it. The first code executed begins with the Public declaration.

Spin has a learning curve, but is fairly intuitive. In this project, a basic skeleton template written by Andre LaMothe was used and added to (greatly simplifying things)

Below is a simple hello world program that blinks the debug LED using 2 cogs.

```
CON
  _clkmode = xtal1 + pll4x       'enable external clock and pll *4
  _xinfreq = 10_000_000          'set freq to 10 MHz
  _stack   = 40

VAR
long blink_stack[20]

'///////////////////////////////////////////////////////////////////////
PUB Start

  COGNEW (Blink(5_000_000), @blink_stack[0])   'start one cog
  COGNEW (Blink(1_500_000), @blink_stack[10])  'start another cog

  repeat while TRUE

'///////////////////////////////////////////////////////////////////////
PUB Blink(rate)  'parallel function
  DIRA[0] := 1
  repeat while TRUE
    OUTA[0] := !OUTA[0]
    waitcnt(CNT+rate)
```

Even with a program as simple as this, multitasking is inherent in the architecture. A new cog is acquired with a call to COGNEW.

As mentioned earlier, most data is addressed globally, so is not passed between cogs. In fact, in this project, two cogs never really needed access to the same piece of data. Regardless, to acqure a lock, something like the following is done

```
repeat while (LOCKSET(0)==1)
  'do work with lock
LOCKCLR(0)
```

Compare this to a more traditional approach like the atmel. Although similar, we do not need to create our own semaphore type object or give up processing (though that certainly is a possibility).

Measuring Latency

By definition, an RTOS works in real time. This implies that it is often very important to have tasks executed within a certain amount of time. This can be measured in many different ways, including latency. This can also be described as achieving a hard deadline.

With the Atmel, latency was measured a number of ways depending on how tasks were handled. Usually, the latency was measured as a measurement of a worst case scenario (eg what would happen if this were interrupted, and this resource was taken, etc.) Similarly, latency on the Propeller chip largely depends on the application.

On the propeller, the latency for any task can be estimated by that task loop's worst case scenario. For example, if a task were to read the NES controller state, read the mouse state, and read the keyboard state, the worst case latency for the NES controller is the time it takes to finish reading the other devices.

The propeller may be ideal in many situations. Because of the multiple cores, it is possible to assign one cog to monitor tasks that require little or no latency. For example, in a reactor one cog could read the temperature at regular intervals, and one cog could be assigned to constantly monitor that temperature to see if it is over a threshold. The monitoring cog would realistically have very little latency. Also, this is much simpler than a more traditional approach, so problems would likely be easier to see.

This is a very different approach from the atmel, where a lot of focus is given to parameters like the timing and length of interrupt service routines.

Graphics

Graphics Driver

Included with the bundled software is an NTSC driver for analog output to Televisions, a VGA driver, and a higher level graphics driver which works on top both of the others. Because VGA refreshes at such a fast rate, it is generally simpler to program on the NTSC drivers using RCA input on a television.

The general purpose graphics driver contains many useful functions.

graphics_demo_10.spin by Chip Gracey has many excellent examples. The driver includes functions for drawing pixels, lines, triangles, etc. It can be a bit confusing to begin with, since it does use polar coordinates for many things.

Visually Stunning Demo

This demonstration is a “hello world” for the graphics driver. The driver was written by Chip Gracey and Andre LaMonthe. It is important because it is the template used as the basis of the Multitasking Demonstration. This can be found in *graphics_template_020.spin*.

Additional programs like *FROGGER_DEMO_030.spin*, *CP_HYDRAMAN_005c.spin*, and others were used to base some of the project's functions on.

Multitasking Demonstration

This demonstration is meant to show equivalent functionality to the RTOS designed in class. It performs multiple tasks, including the following:

- Uses an NES driver to poll for input from the controller
- Uses a graphics driver to output to the television

- Uses a sound driver (coded by me) that plays a scale when a button is pressed
- Blinks an LED at once per second
- Displays a counting number on the screen
- If the A button is pressed, time how long it is pressed and display a similar counting number on another part of the screen
- If the user enters the secret code, print “you will die in 7 days”

The code written for this is available in the files *lundeen_avr_demo_007.spin* for the main program and *lundeen_snd_002.spin* for the sound driver. Below describes some of the interesting aspects of the program.

Sound Driver

The sound driver is written in spin for simplicity, although most realistic drivers should probably be written in assembly.

In the propeller, an intuitive way to write a driver is to make a program start a cog which infinitely loops and checks statuses. In this sound driver's case, global variables are checked. These global variables are changed from the main program, which constitutes an event (which could have arrived from another COG).

For example, the main program checks if the NES B button has been pressed at every loop (which is being monitored by another cog). If it has been pressed, the main program updates a global variable, which is read from the sound driver – which then plays a scale.

Here is an excerpt from the main loop:

```
PUB start : okay
  'start cog if it is available
  stop
  okay := cogon := (cog := cognew(loop, @snd_stack[0])) > 0

PUB loop
  'These are global variables that are updated in the main cog
  _playscaleonce := FALSE
  _playscaleforever := FALSE

  'set pin 7 to output
  DIRA := %00000000_00000000_00000000_10000000
  'set CTRA mode to NCO single mode "00100", Pin B 0, Pin A = 7
  CTRA := %0_00100_000_00000_000000000_000000111
  repeat
    if _playscaleonce  == TRUE
      _playscaleonce_func(10_000_000)
    if _playscaleforever == TRUE
      _playscaleonce_forever(20_000_000)
```

Additional code could obviously be added to the loop in a similar method.

This driver includes very basic code similar to the code that was written for a homework assignment. It generates sound using a square wave by alternating the frequencies and using the timer. It has a function to play a scale, and functions to play individual notes.

There already exists a much more complete sound driver for the propeller which includes functions for

virtually every note, different types of waves (including sawtooth waves, triangle waves), and is probably better written. However, *NS_sound_drv_040.spin* is 320 longs, and *lundeen_snd_01.spin* is only 52 longs – and includes the functionality to play a scale. The point is, when space is at a premium, it becomes a necessity to write drivers. Although it was not necessary in this demo, for a bigger program it very well might be.

Looping

In this demo and in many spin programs, one cog acts a bit like a master cog. It directs the activities of the other cogs, reads and deals with their input, etc. This is a simple way to utilize parallel processing. However, it can also add some complications.

One complication is the screen update. When this was initially programmed, it would update the screen at every iteration. The problem with this is, the more tasks you have to process, the slower the iteration becomes, which gives the screen a choppy effect. To smooth things out, the following algorithm is implemented:

```
repeat
  'timer_cnt makes the refresh consistent
  timer_cnt := CNT + 2_000_000

  'handle everything, hopefully it takes less than 2_000_000

  ' synchronize to frame rate
  waitcnt(timer_cnt)
```

The value for each frame was achieved with trial and error, but when the propeller is running at the full 80 MHz, this worked perfectly, and any animation seemed very smooth.

Another serious complication is the speed of the loop. This made logic much more difficult. Imagine this: you would like the user to enter a sequence of keys “A A B B B”. Any sequence other than this should fail, but if the user enters this sequence then some event should happen. On a standard computer this is a trivial problem. However, because of the nature of the loop, if a user presses “A”, then likely they will still be pressing “A” on the next iteration, because most people press a button longer than 1/40 seconds. Because of this, the demo ignores if a button is pressed multiple times, and keeps track of the state the button sequence is in with a simple finite state machine-like variable.

Disadvantages

Though on the whole, the propeller seems to be a good candidate for achieving RTOS functionality, there may be some disadvantages to consider.

- Spin is a proprietary language and the development tools only seem to run on Windows
- Compared with the atmel, there is very little support for the language, although the hydra kit came with many more examples than were readily available for the atmel.

Conclusions

Using the propeller architecture as a form of an RTOS is a novel and viable alternative. Using basic programming, in this project it was possible to achieve the primary functionality of a simple RTOS using the architecture itself to build a somewhat useful program.

Primary References

Game Programming for the Propeller Powered Hydra, by Andre LaMothe

Propeller Manual v1.1, by Chip Gracey