UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Lesson 4 OK04

The OK04 lesson builds on OK03 by teaching how to use the timer to flash the 'OK' LED at precise intervals. It is assumed you have the code for the Lesson 3: OK03 operating system as a basis.

Contents

1 A New Device
2 Implementation
3 Another Blinking Light

1 A New Device

So far, we've only looked at one piece of hardware on the Raspberry Pi, namely the GPIO Controller. I've simply told you what to do, and it happened. Now we're going to look at the timer, and I'm going to lead you through understanding how it works.

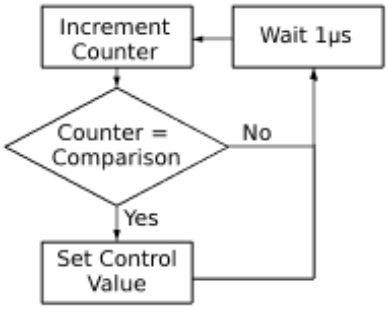
The timer is the only way the Pi can keep time. Most computers have a battery powered clock to keep time when off.

Just like the GPIO Controller, the timer has an address. In this case, the timer is based at 20003000_{16}. Reading the manual, we find the following table:

Table 1.1 GPIO Controller Registers

Address	Size / Bytes	Name	Description	Read or Write
20003000	4	Control / Status	Register used to control and clear timer channel comparator matches.	RW
20003004	8	Counter	A counter that increments at 1MHz.	R
2000300C	4	Compare 0	0th Comparison register.	RW
20003010	4	Compare 1	1st Comparison register.	RW
20003014	4	Compare 2	2nd Comparison register.	RW
20003018	4	Compare 3	3rd Comparison register.	RW

This table tells us a lot, but the descriptions in the manual of the various fields tell us the most. The manual explains that the timer fundamentally just increments the value in Counter by 1 every 1 micro second. Each time it does so, it compares the lowest 32 bits (4 bytes) of the counter's value with the 4 comparison registers, and if it matches any of them, it updates Control / Status to reflect which ones matched.



For more information about bits, bytes, bit fields, and data sizes expand the box below.

Bits explained

Our goal is to implement a function that we can call with an amount of time as an input that will wait for that

amount of time and then return. Think for a moment about how we could do this, given what we have.

I see there being two options:

1. Read a value from the counter, and then keep branching back into the same code until the counter is the amount of time to wait more than it was.
2. Read a value from the counter, add the amount of time to wait, store this in one of the comparison registers and then keep branching back into the same code until the Control / Status register updates.

Both of these strategies would work fine, but in this tutorial we will only implement the first. The reason is because the comparison registers are more likely to go wrong, as during the time it takes to add the wait time and store it in the comparison register, the counter may have increased, and so it would not match. This could lead to very long unintentional delays if a 1 micro second wait is requested (or worse, a 0 microsecond wait).

Issues like these are called concurrency problems, and can be almost impossible to fix.

2 Implementation

I will largely leave the challenge of creating the ideal wait method to you. I suggest you put all code related to the timer in a file called 'systemTimer.s' (for hopefully obvious reasons). The complicated part about this method, is that the counter is an 8 byte value, but each register only holds 4 bytes. Thus, the counter value will span two registers.

Large Operating Systems normally use the Wait function as an opportunity to perform background tasks.

The following code blocks are examples.

```
ldrd r0,r1,[r2,#4]
```

An instruction you may find useful is the **`ldrd`** instruction above. It loads 8 bytes of memory across 2 registers. In this case, the 8 bytes of memory starting at the address in **`r2`** would be copied into **`r0`** and **`r1`**. What is slightly complicated about this arrangement is that **`r1`** actually holds the highest 4 bytes. In other words, if the counter had a value of $999{,}999{,}999{,}999_{10}$ = $11101000110101001010010100001111111111111_2$, **`r1`** would contain 11101000_2 and **`r0`** would contain $11010100101001010000111111111111_2$.

`ldrd regLow,regHigh, [src,#val]` loads 8 bytes from the address given by the number in **`src`** plus **`val`** into **`regLow`** and **`regHigh`**.

The most sensible way to implment this would be to compute the difference between the current counter value and the one from when the method started, and then to compare this with the requested amount of time to wait. Conveniently, unless you wish to support wait times that were 8 bytes, the value in **`r1`** in the example above could be discarded, and only the low 4 bytes of the counter need be used.

When waiting you should always be sure to use higher comparisons not equality comparisons, as if you try to wait for the gap between the time the method started and the time it ends to be exactly the amount requested, you could miss the value, and wait forever.

If you cannot figure out how to code the wait function, expand the box below for a guide.

Wait function implementation

3 Another Blinking Light

Once you have what you believe to be a working wait function, change 'main.s' to use it. Alter everywhere you wait to set the value of r0 to some big number (remember it is in microseconds) and then test it on the Raspberry Pi. If it does not function correctly please see our troubleshooting page.

Once it is working, congratulations you have now mastered another device, and with it, time itself. In the next and final lesson in the OK series, Lesson 5: OK05 we shall use all we have learned to flash out a

pattern on the LED.