|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Lesson 4 OK04  The OK04 lesson builds on OK03 by teaching how to use the timer to flash the 'OK' or 'ACT' LED at precise intervals. It is assumed you have the code for the [Lesson 3: OK03](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok03.html) operating system as a basis.   |  | | --- | | **Contents**   * [1 A New Device](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok04.html#newdevice) * [2 Implementation](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok04.html#implementation) * [3 Another Blinking Light](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok04.html#abl) |   1 A New Device  The timer is the only way the Pi can keep time. Most computers have a battery powered clock to keep time when off.  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.  Just like the GPIO Controller, the timer has an address. In this case, the timer is based at 2000300016. 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 |   Flowchart of the system timer's operation  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  A bit is a name for a single binary digit. As you may recall, a single binary digit is either a 1 or a 0.  A byte is the name we give for a collection of 8 bits. Since each bit can be one of two values, there are 28 = 256 different possible values for a byte. We normally interpret a byte as a binary number between 0 and 255 inclusive.  Diagram of GPIO function select controller register 0.  A bit field is another way of interpreting binary. Rather than interpreting it as a number, binary can be interpreted as many different things. A bit field treats binary as a series of switches which are either on (1) or off (0). If we have a meaning for each of these little switches, we can use them to control things. We have actually already met bitfields with the GPIO controller, with the setting a pin on or off. The bit that was a 1 was the GPIO pin to actually turn on or off. Sometimes we need more options than just on or off, so we group several of the switches together, such as with the GPIO controller function settings (pictured), in which every group of 3 bits controls one GPIO pin function.  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.   Issues like these are called concurrency problems, and can be almost impossible to fix.  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).  2 Implementation  Large Operating Systems normally use the Wait function as an opportunity to perform background tasks.  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.  The following code blocks are examples.  ldrd r0,r1,[r2,#4]  **ldrd regLow,regHigh,[src,#val]** loads 8 bytes from the address given by the number in **src** plus **val** into **regLow** and **regHigh** .  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 999,999,999,99910 = 11101000110101001010010100001111111111112, **r1** would contain 111010002 and **r0** would contain 110101001010010100001111111111112.  The most sensible way to implement 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  Borrowing the idea from the GPIO controller, the first function we should write should be to get the address of this system timer. An example of this is shown below:  .globl GetSystemTimerBase GetSystemTimerBase:  ldr r0,=0x20003000 mov pc,lr  Another function that will prove useful would be one that returns the current counter value in registers **r0** and **r1**:  .globl GetTimeStamp GetTimeStamp: push {lr} bl GetSystemTimerBase ldrd r0,r1,[r0,#4] pop {pc}  This function simply uses the GetSystemTimerBase function and loads in the counter value using **ldrd** like we have just learned.  Now we actually want to code our wait method. First of all, we need to know the counter value when the method started, which we can now get using GetTimeStamp.  delay .req r2 mov delay,r0 push {lr} bl GetTimeStamp start .req r3 mov start,r0  This code copies our method's input, the amount of time to delay, into **r2**, and then calls GetTimeStamp, which we know will return the current counter value in **r0** and **r1**. It then copies the lower 4 bytes of the counter's value to **r3**.  Next we need to compute the difference between the current counter value and the reading we just took, and then keep doing so until the gap between them is at least the size of **delay**.  loop$:  bl GetTimeStamp elapsed .req r1 sub elapsed,r0,start cmp elapsed,delay .unreq elapsed bls loop$  This code will wait until the requested amount of time has passed. It takes a reading from the counter, subtracts the initial value from this reading and then compares that to the requested delay. If the amount of time that has elapsed is less than the requested delay, it branches back to **loop$**.  .unreq delay .unreq start pop {pc}  This code finishes off the function by returning.  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](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok05.html) we shall use all we have learned to flash out a pattern on the LED. |  |