|  |  |  |
| --- | --- | --- |
| Lesson 5 OK05  The OK05 lesson builds on OK04 using it to flash the SOS Morse Code pattern (...---...). It is assumed you have the code for the [Lesson 4: OK04](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok04.html) operating system as a basis.   |  | | --- | | **Contents**   * [1 Data](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok05.html#data) * [2 Time Flies When You're Having Fun...](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok05.html#tfwyhf) |   1 Data  So far, all we've had to do with our operating system is provide instructions to be followed. Sometimes however, instructions are only half the story. Our operating systems may need data.  Some early Operating Systems did only allow certain types of data in certain files, but this was generally found to be too restrictive. The modern way does make programs a lot more complicated however.  In general data is just values that are important. You are probably trained to think of data as being of a specific type, e.g. a text file contains text, an image file contains an image, etc. This is, in truth, just an idea. All data on a computer is just binary numbers, how we choose to interpret them is what counts. In this example we're going to store a light flashing sequence as data.  At the end of 'main.s' copy the following code:  .section .data .align 2 pattern: .int 0b11111111101010100010001000101010  **.align num** ensures the address of the next line is a multiple of 2**num** .  **.int val** outputs the number **val** .  To differentiate between data and code, we put all the data in the .data. I've included this on the operating system memory layout diagram here. I've just chosen to put the data after the end of the code. The reason for keeping our data and instructions separate is so that if we eventually implement some security on our operating system, we need to know what parts of the code can be executed, and what parts can't.  I've used two new commands here. **.align** and **.int**. **.align** ensures alignment of the following data to a specified power of 2. In this case I've used **.align 2** which means that this data will definitely be placed at an address which is a multiple of 22 = 4. It is really important to do this, because the **ldr** instruction we used to read memory only works at addresses that are multiples of 4.  The **.int** command copies the constant after it into the output directly. That means that 111111111010101000100010001010102 will be placed into the output, and so the label pattern actually labels this piece of data as pattern.  One challenge with data is finding an efficient and useful representation. This method of storing the sequence as on and off units of time is easy to run, but would be difficult to edit, as the concept of a Morse - or . is lost.  As I mentioned, data can mean whatever you want. In this case I've encoded the Morse Code SOS sequence, which is ...---... for those unfamiliar. I've used a 0 to represent a unit of time with the LED off, and a 1 to represent a unit of time with the LED on. That way, we can write some code which just displays a sequence in data like this one, and then all we have to do to make it display a different sequence is change the data. This is a very simple example of what operating systems must do all the time; interpret and display data.  Copy the following lines before the **loop$** label in 'main.s'.  ptrn .req r4 ldr ptrn,=pattern ldr ptrn,[ptrn] seq .req r5 mov seq,#0  This code loads the pattern into **r4**, and loads 0 into **r5**. **r5** will be our sequence position, so we can keep track of how much of the pattern we have displayed.  The following code puts a non-zero into **r1** if and only if there is a 1 in the current part of the pattern.  mov r1,#1 lsl r1,seq and r1,ptrn  This code is useful for your calls to SetGpio, which must have a non-zero value to turn the LED off, and a value of zero to turn the LED on.  Now modify all of your code in 'main.s' so that each loop the code sets the LED based on the current sequence number, waits for 250000 micro seconds (or any other appropriate delay), and then increments the sequence number. When the sequence number reaches 32, it needs to go back to 0. See if you can implement this, and for an extra challenge, try to do it using only 1 instruction (solution in the download).  2 Time Flies When You're Having Fun...  You're now ready to test this on the Raspberry Pi. It should flash out a sequence of 3 short pulses, 3 long pulses and then 3 more short pulses. After a delay, the pattern should repeat. If it doesn't work please see our troubleshooting page.  Once it works, congratulations you have reached the end of the OK series of tutorials.  In this series we've learnt about assembly code, the GPIO controller and the System Timer. We've learnt about functions and the ABI, as well as several basic Operating System concepts, and also about data.  You're now ready to move onto one of the more advanced series.   * The [Screen](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/screen01.html) series is next and teaches you how to use the screen with assembly code. * The [Input](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/input01.html) series teaches you how to use the keyboard and mouse.   By now you already have enough information to make Operating Systems that interact with the GPIO in other ways. If you have any robot kits, you may want to try writing a robot operating system controlled with the GPIO pins! | 第5课OK05  OK05课建立在OK04课程的基础上，并运用本课程去发射SOS莫尔斯电码。假设你已经拥有了第4课OK04操作系统代码的基础。  目录   1. 数据 2. 当你乐呵时的时间文件 3. 数据   至今，我们已经针对我们的操作系统所做的就是按照指示提供相应代码。虽然有时是这个样子，但是代码只是故事的一半儿。我们操作系统还需要数据。  一些早期的操作系统仅仅允许在一些确定文件里的确定数据类型，但是这样太严格了。现代的方法已经取得了很大的进步。  一般而言，数值就是很重要的数据。你可能会认为数值是某种特定的数据类型，例如包含文本的文本文件，包含一张图片的图片文件等等。事实上，这可能仅仅是一个想法。计算机上所有的数据都仅仅是些二进制数值，区别仅仅在于我们是如何解释它们的。在本例子中，我们把闪烁灯的序列存储为数据。  把下面的代码复制到文件“main.s”的末尾处。  .section .data  .align 2  Pattern:  .int  0b11111111101010100010001000101010  指令.align num的意思是确保下一行代码的地址是2的num次幂的整数倍。  指令.int val的意思是输出数值val。  为了凸显数据和代码的区别，我把所有的数据都规制在.data中。在操作系统的内存布局图中已经引入了这个文件。现在仅仅是把这些数据放置在代码的后面。把数据和指令进行隔离的原因是最终的操作系统要有一些安全措施，我们需要知道那些部分可以被执行，那些不行。  这里我使用了一些新的指令。比如指令.align和.int。指令.align可以确保以后的数据地址是2的某个整数幂。这里我使用的指令是.align 2，它的意思是明确地指出这些数据存放的地址应该为22 = 4的整数次幂。这么做很重要，这是因为用于从内存中读取数据的指令ldr仅仅对4的整数次幂有效。  指令.int将会把其后面的常数直接复制到输出。这意味着111111111010101000100010001010102将会被直接放置到输出，并且这些标签样式将实际地标识这些数据为电码。  处理这些数据的一个挑战是找到一个效率高的且有用的表达方式。这种把每个单位时间的开或关的动作存储为一个序列的方法很容易去执行，但是却编辑起来却很困难，因为有关莫尔斯码的-或者.的内涵丢失了。  我之前提及过，数据可以表示你的任何想法。本例中，我已经把SOS的莫尔斯码序列进行了编码，它的源码是并不常见的...---...样式。我用0来表示一个单位时间内LED灯的灭，用1来表示一个单位时间内的LED灯的亮。运用这个办法，我们就可以把灯的动作序列编写成一个序列，那么接下来我改变序列的值，就可以改变灯的动作序列，这样数据的序列就和动作的序列对应上了。这就是一个很简单的例子，用于阐述操作系统必须总是在干什么——解释和显示数据。  把下面的代码拷贝并复制到文件“main.s”里标签loop$之前的位置处。  ptrn .req r4  ldr ptrn,=pattern  ldr ptrn, [ptrn]  seq .req r5  mov seq, #0  这段代码将把电码装载到寄存器r4中，并把数值0装载到寄存器r5中。寄存器r5将保存序列的位置值，这样我们就可以跟踪我们已经显示了多少电码。  下面的代码当且仅当当前电码是1时，将会把一个非零值存入到寄存器r1中。  mov r1, #1  lsl r1, seq  and r1, ptrn  这段代码对于你调用函数SetGpio很有帮助，那个非零值将会关闭LED灯，零值将会打开LED灯。 |