|  |  |  |
| --- | --- | --- |
| Lesson 1 OK01  The OK01 lesson contains an explanation about how to get started and teaches how to enable the 'OK' or 'ACT' **LED** on the Raspberry Pi board near the RCA and USB ports. This light was originally labelled OK but has been renamed to ACT on the revision 2 Raspberry Pi boards.   |  | | --- | | **Contents**   * [1 Getting Started](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok01.html#gs) * [2 The Beginning](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok01.html#beginning) * [3 The First Line](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok01.html#firstline) * [4 Enabling Output](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok01.html#enablingoutput) * [5 A Sign Of Life](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok01.html#signlife) * [6 Happily Ever After](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok01.html#happy) * [7 Pi Time](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok01.html#pitime) |   1 Getting Started  I am assuming at this point that you have already visited the [Downloads](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/downloads.html) page, and got the necessary GNU Toolchain. Also on the downloads page is a file called OS Template. Please download this and extract its contents to a new directory.  2 The Beginning  The '.s' file extension is commonly used for all forms of assembly code, it is up to us to remember this is ARMv6.  Now that you have extracted the template, create a new file in the 'source' directory called 'main.s'. This file will contain the code for this operating system. To be explicit, the folder structure should look like:  build/  *(empty)*  source/  main.s  kernel.ld  LICENSE  Makefile  Open 'main.s' in a text editor so that we can begin typing assembly code. The Raspberry Pi uses a variety of assembly code called ARMv6, so that is what we'll need to write in.  Copy in these first commands.  .section .init .globl \_start \_start:  As it happens, none of these actually do anything on the Raspberry Pi, these are all instructions to the assembler. The assembler is the program that will translate between assembly code that we understand, and binary machine code that the Raspberry Pi understands. In Assembly Code, each line is a new command. The first line here tells the Assembler[[1]](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok01.html" \l "note1" \o "Note 1) where to put our code. The template I provided causes the code in the section called **.init** to be put at the start of the output. This is important, as we want to make sure we can control which code runs first. If we don't do this, the code in the alphabetically first file name will run first! The **.section** command simply tells the assembler which section to put the code in, from this point until the next **.section** or the end of the file.  In assembly code, you may skip lines, and put spaces before and after commands to aid readability.  The next two lines are there to stop a warning message and aren't all that important.[[2]](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok01.html" \l "note2" \o "Note 2)  3 The First Line  Now we're actually going to code something. In assembly code, the computer simply goes through the code, doing each instruction in order, unless told otherwise. Each instruction starts on a new line.  Copy the following instruction.  ldr r0,=0x20200000  **ldr reg,=val** puts the number **val** into the register named **reg**.  That is our first command. It tells the processor to store the number 0x20200000 into the register r0. I shall need to answer two questions here, what is a register, and how is 0x20200000 a number?  A single register can store any integer between 0 and 4,294,967,295 inclusive on the Raspberry Pi, which might seem like a large amount of memory, but it is only 32 binary bits.  A register is a tiny piece of memory in the processor, which is where the processor stores the numbers it is working on right now. There are quite a few of these, many of which have a special meaning, which we will come to later. Importantly there are 13 (named r0,r1,r2,...,r9,r10,r11,r12) which are called General Purpose, and you can use them for whatever calculations you need to do. Since it's the first, I've used r0 in this example, but I could very well have used any of the others. As long as you're consistent, it doesn't matter.  0x20200000 is indeed a number. However it is written in Hexadecimal notation. To learn more about hexadecimal expand the box below:  Hexadecimal explained  Hexadecimal is an alternate system for writing numbers. You may only be aware of the decimal system for writing numbers in which we have 10 digits: 0,1,2,3,4,5,6,7,8 and 9. Hexadecimal is a system with 16 digits: 0,1,2,3,4,5,6,7,8,9,a,b,c,d,e and f.  567 is 5 hundreds, 6 tens and 7 units.  You may recall being taught how decimal numbers work in terms of place value. We say that the rightmost digits is the 'units' digits, the next one left is the 'tens' digit, the next is the 'hundreds' digit, and so on. What this actually meant is, the number is 100 × the value in the 'hundreds' digit, plus 10 × the value in the 'tens' digit, plus 1 × the value in the units digit.  567 is 5x10^2+6x10^1+7x10^0  More mathematically, we can now spot the pattern and say that the rightmost digit is the 100=1s digit, the next left is the 101=10s digit, the next is 102=100s digit, and so on. We have all agreed on the system that 0 is the lowest digit, 1 is the next and so on. But what if we used a different number instead of 10 in these powers? Hexadecimal is just the system in which we use 16 instead.  567 = 5x10^2+6x10^1+7x10^0 = 2x16^2+3x16^1+7x16^0  The mathematics to the right shows that the number 567 in decimal is equivalent to the number 237 in hexadecimal. Often when we need to be clear about what system we're using to write numbers in we put 10 for decimal and 16 for hexadecimal. Since it's difficult to write small numbers in assembly code, we use 0x instead to represent a number in hexadecimal notation. So 0x237 means 23716.  So where do a,b,c,d,e and f come in? Well, in order to be able to write every number in hexadecimal, we need extra digits. For example 916 = 9×160 = 910, but 1016 = 1×161 + 1×160 = 1610. So if we just used 0,1,2,3,4,5,6,7,8 and 9 we would not be able to write 1010, 1110, 1210, 1310, 1410, 1510. So we introduce 6 new digits such that a16 = 1010, b16 = 1110, c16 = 1210, d16 = 1310, e16 = 1410, f16 = 1510  So, we now have another system for writing numbers. But why did we bother? Well, it turns out that since computers always work in binary, hexadecimal notation is very useful because every hexadecimal digit is exactly four binary digits long. This has the nice side effect that a lot of computer numbers are round numbers in hexadecimal, even though they're not in decimal. For example, in the assembly code just above I used the number 2020000016. If I had chose to write this in decimal it would have been 53896806410, which is much less memorable.  To convert numbers from decimal to hexadecimal I find the following method easiest:  Conversion example   1. Start with the decimal number, say 567. 2. Divide by 16 and calculate the remainder. For example 567 ÷ 16 = 35 remainder 7. 3. The remainder is the last digit of the answer in hexadecimal, in the example this is 7. 4. Repeat steps 2 and 3 again with the result of the last division until the result is 0. For example 35 ÷ 16 = 2 remainder 3, so 3 is the next digit of the answer. 2 ÷ 16 = 0 remainder 2, so 2 is the next digit of the answer. 5. Once the result of the division is 0, you can stop. The answer is just the remainders in the reverse order to which you got them, so 56710 = 23716.   To convert hexadecimal numbers back to decimal, it is easiest to expand out the number, so 23716 = 2×162 + 3×161 +7 ×160 = 2×256 + 3×16 + 7×1 = 512 + 48 + 7 = 567.  So our first command is to put the number 2020000016 into r0. That doesn't sound like it would be much use, but it is. In computers, there are an awful lot of chunks of memory and devices. In order to access them all, we give each one an address. Much like a postal address or a web address this is just a means of identifying the location of the device or chunks of memory we want. Addresses in computers are just numbers, and so the number 2020000016 happens to be the address of the GPIO controller. This is just a design decision taken by the manufacturers, they could have used any other address (providing it didn't conflict with anything else). I know this address only because I looked it up in a manual[[3]](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok01.html" \l "note3" \o "Note 3), there is no particular system to the addresses (other than that they are all large round numbers in hexadecimal).  4 Enabling Output  A diagram showing key parts of the GPIO controller.  Having read the manual, I know we're going to need to send two messages to the GPIO controller. We need to talk its language, but if we do, it will obligingly do what we want and turn on the OK LED. Fortunately, it is such a simple chip, that it only needs a few numbers in order to understand what to do.  mov r1,#1 lsl r1,#18 str r1,[r0,#4]  **mov reg,#val** puts the number **val** into the register named **reg**.  **lsl reg,#val** shifts the binary representation of the number in **reg** by **val** places to the left.  **str reg,[dest,#val]** stores the number in **reg** at the address given by **dest** + **val**.  These commands enable output to the 16th GPIO pin. First we get a necessary value in r1, then send it to the GPIO controller. Since the first two instructions are just trying to get a value into r1, we could use another **ldr** command as before, but it will be useful to us later to be able to set any given GPIO pin, so it is better to deduce the value from a formula than write it straight in. The OK LED is wired to the 16th GPIO pin, and so we need to send a command to enable the 16th pin.  The value in r1 is needed to enable the LED pin. The first line puts the number 110 into r1. The **mov** command is faster than the **ldr** command, because it does not involve a memory interaction, whereas **ldr** loads the value we want to put into the register from memory. However, **mov** can only be used to load certain values[[4]](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok01.html" \l "note4" \o "Note 4). In ARM assembly code, almost every instruction begins with a three letter code. This is called the mnemonic, and is supposed to hint at what the operation does. **mov** is short for move and **ldr** is short for load register. **mov** moves the second argument **#1** into the first **r1**. In general, **#** must be used to denote numbers, but we have already seen a counterexample to this.  The second instruction is **lsl** or logical shift left. This means shift the binary representation for the first argument left by the second argument. In this case this will shift the binary representation of 110 (which is 12) left by 18 places (making it 10000000000000000002=26214410).  If you are unfamiliar with binary, expand the box below:  Binary explained  Just like hexadecimal binary is another way of writing numbers. In binary we only have 2 digits, 0 and 1. This is useful for computers because we can implement this in a circuit by saying that electricity flowing through the circuit means 1, and not means 0. This is how computers actually work and do maths. Despite only having 2 digits binary can still be used to represent every number, it just takes a lot longer.  567 in decimal = 1000110111 in binary  The image shows the binary representation of the number 56710 which is 10001101112. We use 2 to denote numbers written in binary.  One of the quirks of binary that we make heavy use of in assembly code is the ease by which numbers can be multiplied or divided by powers of 2 (e.g. 1,2,4,8,16). Normally multiplications and divisions are tricky operations, however these special cases are very easy, and so are very important.  13*4 = 52, 1101*100=110100  Shifting a binary number left by **n** places is the same as multiplying the number by 2**n**. So, if we want to multiply by 4, we just shift the number left 2 places. If we want to multiply by 256 we could shift it left by 8 places. If we wanted to multiply by a number like 12, we could instead multiply it by 8, then separately by 4 and add the results (N × 12 = N × (8 + 4) = N × 8 + N × 4).  53/16 = 3, 110100/10000=11  Shifting a binary number right by **n** places is the same as dividing the number by 2**n**. The remainder of the division is the bits that were lost when shifted right. Unfortunately dividing by a binary number that is not an exact power of 2 is very difficult, and will be covered in [Lesson 9: Screen04](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/screen04.html).  Binary Terminology  This diagram shows common terminology used with binary. A bit is a single binary digit. A nibble is 4 binary bits. A byte is 2 nibbles, or 8 bits. A half is half the size of a word, 2 bytes in this case. A word refers to the size of the registers on a processor, and so on the Raspberry Pi this is 4 bytes. The convention is to number the most significant bit of a word 31, and the least significant bit as 0. The top, or high bits refer to the most significant bits, and the low or bottom bits refer to the least significant. A kilobyte (KB) is 1000 bytes, a megabyte is 1000 KB. There is some confusion as to whether this should be 1000 or 1024 (a round number in binary). As such, the new international standard is that a KB is 1000 bytes, and a Kibibyte (KiB) is 1024 bytes. A Kb is 1000 bits, and a Kib is 1024 bits.  The Raspberry Pi is little endian by default, meaning that loading a byte from an address you just wrote a word to will load the lowest byte of the word.  Once again, I only know that we need this value from reading the manual[[3]](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok01.html" \l "note3" \o "Note 3). The manual says that there is a set of 24 bytes in the GPIO controller, which determine the settings of the GPIO pin. The first 4 relate to the first 10 GPIO pins, the second 4 relate to the next 10 and so on. There are 54 GPIO pins, so we need 6 sets of 4 bytes, which is 24 bytes in total. Within each 4 byte section, every 3 bits relates to a particular GPIO pin. Since we want the 16th GPIO pin, we need the second set of 4 bytes because we're dealing with pins 10-19, and we need the 6th set of 3 bits, which is where the number 18 (6×3) comes from in the code above.  Finally the **str** 'store register' command stores the value in the first argument, **r1** into the address computed from the expression afterwards. The expression can be a register, in this case **r0**, which we know to be the GPIO controller address, and another value to add to it, in this case **#4**. This means we add 4 to the GPIO controller address and write the value in **r1** to that location. This happens to be the location of the second set of 4 bytes that I mentioned before, and so we send our first message to the GPIO controller, telling it to ready the 16th GPIO pin for output.  5 A Sign Of Life  Now that the LED is ready to turn on, we need to actually turn it on. This means sending a message to the GPIO controller to turn pin 16 off. Yes, *turn it off*. The chip manufacturers decided it made more sense[[5]](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok01.html" \l "note5" \o "Note 5) to have the LED turn on when the GPIO pin is off. Hardware engineers often seem to take these sorts of decisions, seemingly just to keep OS Developers on their toes. Consider yourself warned.  mov r1,#1 lsl r1,#16 str r1,[r0,#40]  Hopefully you should recognise all of the above commands, if not their values. The first puts a 1 into **r1** as before. The second shifts the binary representation of this 1 left by 16 places. Since we want to turn pin 16 off, we need to have a 1 in the 16th bit of this next message (other values would work for other pins). Finally we write it out to the address which is 4010 added to the GPIO controller address, which happens to be the address to write to turn a pin off (28 would turn the pin on).  6 Happily Ever After  It might be tempting to finish now, but unfortunately the processor doesn't know we're done. In actuality, the processor never will stop. As long as it has power, it continues working. Thus, we need to give it a task to do forever more, or the Raspberry Pi will crash (not much of a problem in this example, the light is already on).  loop$:  b loop$  **name:** labels the next line **name**.  **b label** causes the next line to be executed to be **label**.  The first line here is not a command, but a label. It names the next line **loop$**. This means we can now refer to the line by name. This is called a label. Labels get discarded when the code is turned into binary, but they're useful for our benefit for referring to lines by name, not number (address). By convention we use a **$** for labels which are only important to the code in this block of code, to let others know they're not important to the overall program. The **b** (branch) command causes the next line to be executed to be the one at the label specified, rather than the one after it. Therefore, the next line to be executed will be this **b**, which will cause it to be executed again, and so on forever. Thus the processor is stuck in a nice infinite loop until it is switched off safely.  The new line at the end of the block is intentional. The GNU toolchain expects all assembly code files to end in an empty line, so that it is sure you were really finished, and the file hasn't been cut off. If you don't put one, you get an annoying warning when the assembler runs.  7 Pi Time  So we've written the code, now to get it onto the pi. Open a terminal on your computer and change the current working directory to the parent directory of the source directory. Type **make** and then press enter. If any errors occur, please refer to the troubleshooting section. If not, you will have generated three files. kernel.img is the compiled image of your operating system. kernel.list is a listing of the assembly code you wrote, as it was actually generated. This is useful to check that things were generated correctly in future. The kernel.map file contains a map of where all the labels ended up, which can be useful for chasing around values.  To install your operating system, first of all get a Raspberry PI SD card which has an operating system installed already. If you browse the files in the SD card, you should see one called kernel.img. Rename this file to something else, such as kernel\_linux.img. Then, copy the file kernel.img that **make** generated onto the SD Card. You've just replaced the existing operating system with your own. To switch back, simply delete your kernel.img file, and rename the other one back to kernel.img. I find it is always helpful to keep a backup of you original Raspberry Pi operating system, in case you need it again.  Put the SD card into a Raspberry Pi and turn it on. The OK LED should turn on. If not please see the troubleshooting page. If so, congratulations, you just wrote your first operating system. See [Lesson 2: OK02](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok02.html) for a guide to making the LED flash on and off.   1. [1][^](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok01.html#note1a) OK, I'm lying it tells the linker, which is another program used to link several assembled files together. It doesn't really matter. 2. [2][^](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok01.html#note2a) Clearly they're important to you. Since the GNU toolchain is mainly used for creating programs, it expects there to be an entry point labelled **\_start**. As we're making an operating system, the **\_start** is always whatever comes first, which we set up with the **.section .init** command. However, if we don't say where the entry point is, the toolchain gets upset. Thus, the first line says that we are going to define a symbol called **\_start** for all to see (globally), and the second line says to make the symbol **\_start** the address of the next line. We will come onto addresses shortly. 3. [3][^](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok01.html#note3a)[^](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok01.html#note3b) This tutorial is designed to spare you the pain of reading it, but, if you must, it can be found here [SoC-Peripherals.pdf](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/downloads/SoC-Peripherals.pdf). For added confusion, the manual uses a different addressing system. An address listed as 0x7E200000 would be 0x20200000 in our OS. 4. [4][^](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok01.html#note4a) Only values which have a binary representation which only has 1s in the first 8 bits of the representation. In other words, 8 1s or 0s followed by only 0s. 5. [5][^](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok01.html#note5a) A hardware engineer was kind enough to explain this to me as follows:   The reason is that modern chips are made of a technology called CMOS, which stands for Complementary Metal Oxide Semiconductor. The Complementary part means each signal is connected to two transistors, one made of material called N-type semiconductor which is used to pull it to a low voltage and another made of P-type material to pull it to a high voltage. Only one transistor of the pair turns on at any time, otherwise we'd get a short circuit. P-type isn't as conductive as N-type, which means the P-type transistor has to be about 3 times as big to provide the same current. This is why LEDs are often wired to turn on by pulling them low, because the N-type is stronger at pulling low than the P-type is in pulling high.  There's another reason. Back in the 1970s chips were made out of entirely out of N-type material ('NMOS'), with the P-type replaced by a resistor. That means that when a signal is pulled low the chip is consuming power (and getting hot) even while it isn't doing anything. Your phone getting hot and flattening the battery when it's in your pocket doing nothing wouldn't be good. So signals were designed to be 'active low' so that they're high when inactive and so don't take any power. Even though we don't use NMOS any more, it's still often quicker to pull a signal low with the N-type than to pull it high with the P-type. Often a signal that's 'active low' is marked with a bar over the top of the name, or written as SIGNAL\_n or /SIGNAL. But it can still be confusing, even for hardware engineers!  Spot a mistake? You can help improve this tutorial on [GitHub](https://github.com/chadderz121/bakingpi-www).  [Creative Commons Licence](http://creativecommons.org/licenses/by-sa/3.0/deed.en_GB) Baking Pi: Operating Systems Development by Alex Chadwick is licensed under a [Creative Commons Attribution-ShareAlike 3.0 Unported License](http://creativecommons.org/licenses/by-sa/3.0/deed.en_GB).  Based on contributions at <https://github.com/chadderz121/bakingpi-www>. | 第一课 OK01  如何入门上道？如何点亮“OK”或“ACT”放光二极管（这个发光二极管在树莓派板子的RCA引脚和USB引脚的附近）？这一课给出了说明。说明下：要点亮的这颗二极管原来标注OK字样，但是在树莓派第二版的板子上却改名为ACT字样。（别迷糊，只是名字不同罢了。）（从CPU引出的GPIO引脚是相同的。）  目录   1. 入门启程 2. 开始 3. 第一行代码 4. 输出点什么 5. 生命信号 6. 从此以后开始了幸福的生活 7. 吃派的时间到了 8. 入门启程   在正式进入正题前，我假设你已经做了两点工作：一是访问了本课程的下载页，二是获得了必要的GNU工具链。记得还有一个叫OS Template的文件也在下载页里。把它下载下来，并把这个压缩包解压到一个新文件夹里。   1. 开始   后缀为“.s”的文件一般都是汇编源码文件。不过，就目前的项目，我们只要记住后缀为“.s”的文件是ARMv6的汇编文件就可以了。  既然你已经把下载的OS template文件解压到了一个新文件夹里了，不妨现在就来创建一个文件，并起个“main.s”的名字给它。这个文件就是用来存放我们要做的那个操作系统的源代码的。简单起见，现在你应该拥有这样的目录结构：  build/  (empty)  source/  main.s  kernel.ld  LICENSE  Makefile  用记事本开着这个文件。后续的工作就是在这个文件里编辑汇编代码。树莓派使用一种命叫ARMv6的汇编指令集，所以我们的工作也就是在这个记事本里编写ARMv6指令代码。  复制下面的三行代码到记事本里。  .section .init  .global \_start  \_start:  上面的代码并不能让树莓派做什么。汇编指令集有些指令是用来告诉汇编器如何对待它们以后的指令的。而这些代码并不实际参与树莓派工作时的动作。哦，这里提到了汇编器。汇编器是用来将我们人类可以理解的汇编指令文件转换成树莓派可以理解的二进制代码文件的一种软件。而且，不仅仅是转换成树莓派可以理解的，其他的汇编器可以转换成某种CPU可以理解的二进制代码文件。这里我们只对树莓派研究，所以，把眼界收缩到这里来。其他的暂时放放，别分心。在汇编文件里，每一行代表一条新的指令。上面三行中的第一行（.section .init）意思是告诉汇编器把转换后的二进制文件放在哪里。Template文件中.init行的意思是把该段代码生成的二进制输出放在各种段的最开始的位置。这一点很重要，因为这样可以确保那段代码首先得到运行。如果我们不这么做，同目录下的文件中，谁的首字母在字母表的位置靠前，谁的代码就会被先执行。这显然不是我们想要的。.section指令仅仅是告知汇编器，在个代码段落里将要放置什么类型的代码。代码段落的起始是从该行开始，终止在下一个.section指令行之前的一行，或者是文件的结尾处。  在汇编代码源文件中，你可以使用空白行插入到指令行的前面或后面，而且空白行的行数也不限制。这么做并不改变汇编文件的本意，仅仅是为了人类阅读代码的便捷性。  接下来的两行指令是用来关闭一条警告提示信息。这点你不用多虑，附注里有提及。   1. 第一行代码   现在，我们要真正地开始编码了。对于计算机来说，它仅仅是把每行汇编源代码对应的二进制指令执行一下，按照汇编文件里的指令顺序一行一行地逐条执行。如果你在汇编文件里制定了其他的执行顺序就另当别论。  把下列指令拷贝到你的记事本里。  ldr r0,=0x20200000  形似这样ldr reg, =val的指令，其意思是把数值val放到一个叫reg的寄存器中去。  这就是我们的第一行代码。它指示处理器把数值0x20200000放入r0寄存器。你可能有两个疑惑：一是什么是寄存器，另一个是0x20200000是个什么样子的数。  树莓派上的单个寄存器可以存储介于0和4,294,967,295之间的任何整数。如此巨大的数值都可以存储，是不是寄存器要很“大”才可以做到？嗯，不用。仅仅需要32个二进制bit位就可以做到。是不是很神奇？  一个寄存器是处理器中负责存储信息的存储单元的一个小小部分。寄存器主要用来存储处理器正在或者刚刚使用过的信息。处理器里有那么一些这样的寄存器，这些寄存器有些是有特殊名字的，关于这一点，以后会详细提及。他们中最重要的有13个，名字分别为r0，r1，r2，…，r9，r10，r11，r12。这些寄存器被称为普通用途寄存器。在接下来的任何计算过程中，你都可以使用他们。就因为r0是第一个普通用途寄存器，所以我就用它了。你尽管用其他普通用途寄存器，没有任何问题。只要你的汇编代码前后一致地用你选择的普通用途寄存器就可以了。  现在说说0x20200000是个什么鸟。看起来，它确实像个数。对，它就是个数而已，只不过用到了16进制书写罢了。你把它当作一个很大的十进制数，没什么问题。如果要详细深入了解16进制的知识，那么，就浏览下面的文字。  16进制小知识  我们的第一行代码就是把数值2020000016复制到r0普通用途寄存器。看起来，这个寄存器不是太常用到，但事实正好相反。在计算机里，有大量的存储器和设备。为了存取它们，我们给每个存储器单元或者设备都赋予一个地址。就像邮箱地址或者网页地址一样，这些地址仅仅是用来让我们区分各个存储器单元或者设备的。计算机里的地址就是一个数值。因此数值2020000016正好是GPIO控制器的地址。之所以这个数值是GPIO控制器的地址，是由制造商决定的。并且他们可以使用任何其他地址，只要不和其他地址冲突就可以。你问我是怎么知道这个神奇的数值就GPIO控制器的地址的？哈哈，我是从手册中看到的。对于地址来说，并没有什么是特殊的，全部都是一个数值来代表的地址而已。   1. 输出点什么   下图是GPIO控制器的示意图。  （图1）  读过手册后，我就明确了一点：我们需要给GPIO控制器发送两条消息。既然我们要和GPIO通信，自然，我们就要讲GPIO控制器能够听得明白的语言。如果我们按照他们的语言来交流，他们自然就听我们的话。如果运气好的话，我们只需要一些很简单的数值就可以告诉他们，我们想让他们做的任何事情，其中自然包括点亮一只名叫OK的LED灯。  mov r1, #1  lsl r1, #18  str r1, [r0, #4]  指令“mov reg, #val”的意思是：把val数值存入reg寄存器中。  指令“lsl reg, #val”的意思是：左移或者右移reg寄存器中二进制数值val个位。  指令“”的意思是：reg寄存器中数值，存储到地址为dest + val的存储器或者设备的寄存器中。  这些指令总的作用是让第16号GPIO引脚具有输出功能。我们通常的做法就是先在r1寄存器中存入必要的数值，然后把r1中的数值发送到GPIO控制器中。因为头两条指令的目的是把一个数值存入r1寄存器中，那么，我们有更好更直接的指令ldr来代替之前的两行代码。在之后的学习中，ldr指令对于设置任何GPIO引脚是相当有用的。一个更好的得到数值的办法是从公式中计算获得，而不是把数值直接写进代码中。因为OK LED灯和GPIO的第16号引脚相连，所以，我们要让第16号引脚具有输出功能。  寄存器r1中的数值要能让LED引脚（GPIO的第16号引脚）工作。第一行代码的意思是把十进制数1存入r1寄存器中。指令mov的执行速度要快于ldr指令，这是因为mov指令不需要和内存打交道，而ldr指令却需要利用内存来执行。不要想当然地以为内存是个快速设备，和CPU比较起来，内存的速度就好像是个蜗牛。但是，mov指令只能用于装载确定数值。在ARM汇编文件中，几乎所有指令都是以3个字母开始的。实际上这仅仅是个助记符号，用来提示该指令用来干什么的。mov是move的简写，而ldr是load register的简写。mov指令搬运该指令中第二个参数#1到第一个参数r1寄存器中。一般来说，#必须被用来标识一个数值，但是我们已经看到了一个反例。  第二行代码中指令lsl是logical shift left的简写。它的意思是：把第一个参数的二进制表示，进行第二个参数个的左移。当前的这个例子中，其表示：把十进制的数值1（二进制表示为1）左移18次，其结果为：10000000000000000002，十进制为：26214410。  如果你对二进制不熟悉，看看下面的扩展阅读。  二进制探索  曾经，我只知道从手册中找到我们需要的数值。手册上说GPIO控制器有24字节的寄存器，并且可以设置这24个字节的内容来达到设置GPIO引脚功能的目的。头4个字节和前10个GPIO引脚相关，第二个4字节寄存器和接下来的10个GPIO引脚相关，以此类推。因为有54个GPIO引脚，所以，需要6套4字节寄存器来设置。6\*4=24字节就是为什么GPIO控制器有24个字节的寄存器的原因了。在每4个字节中，每3个位是和一个特定的GPIO引脚相关。所以，我们需要设置第16号引脚，也就是要设置第二套里的第6个引脚。自然，第6个引脚之前的的5个引脚分别占用了3\*5=15个位。那么，从第16位、17位、18位就是给第6个引脚准备的。所以，我们需要18这个数值。第二行代码的目的就是产生一个第18位为1，其他位为0的数值。原因就在这里。  最终，我们用指令str（store register）来把第一个参数中的数值存储到r1寄存器里表示地址的寄存器里去。当然，由于第16号引脚是第二套引脚，也就是说要第二个4字节，所以，要在r1的基础上加上4个字节的增加量。到现在为止，我们已经给GPIO寄存器发送了第一条信息，告诉它把第16号引脚准备好，我们要输出点什么啦！   1. 生命信号   现在，LED已经准备就绪，就等着我们把它点亮了。我们需要实际做的就是告诉GPIO控制器，把第16号引脚关闭。怎么回事？不是要输出吗？怎么是关闭呢？关闭了又如何进行输出呢？没错！就是关闭该引脚。这是制造商所设计的。如果你想让LED灯亮，设置成输出，并关闭它。硬件工程师经常会采取一些诸如此类的惯常做法，看起来好像是让操作系统开发人员更加警觉一些——当心脚下！  mov r1, #1  lsl r1, #16  str r1, [r0, #40]  如果你仔细阅读了之前的文字，以上三行代码的意思，你就会很明了，所不同的仅仅是参数的数值而已。第一行和之前的完全一样。第二行则是把1的二进制表示左移16次。因为我们想要关闭第16号引脚，所以要在第16个位上设置为1。最后，我们把r1寄存器里的数值发送到比GPIO控制器地址大40个单位的地址上去。该地址的寄存器数值就是用来控制第16号引脚输出或者关闭的。（大GPIO控制器地址28个地址的寄存器是用来打开第16号引脚的。）   1. 王子和公主从此以后过上了幸福的生活   是不是可以结束了。如果是这样的话，处理器就懵了。它还不知道你到底想要做什么。实际上，处理器是从不会停下来的。只要电源接通，处理器就会永无休止的运转下去。我们应该在代码的结尾处写一个无限循环代码片段，否则树莓派就会崩溃掉。（在我们这个例子中，这到没什么。反正LED灯会一直亮。）  loop$:  b loop$  形如“name:”指令标识的是其下一行代码的地址。而“b label”指令是指挥CPU，让其下一条指令仍然取label标识的地址处的代码。本例中就是b loop$。这样处理器就会在这条指令的指挥下，反复执行该行。就好像陷入了出不来的循环一样。  这里的第一行并不是一条指令，而是一个标签，用来标识其下一行代码的地址。（你完全没有必要搞清楚该地址具体是什么。这些都是由编译器来维护的。）这样的话，我们要访问第二行代码的手段就多了一个——可以通过标签来访问了。当汇编器把汇编文件转换成二进制代码的时候，标签会被丢弃，但是，当我们需要通过标签访问代码行的时候，标签将很有用。标签后的$只是对这一块代码起作用。指令b（branch的简写）将导致处理器的下一步去执行b后面标签处的指令，而不是b指令下一行的指令。这将导致去执行的还是本行，并且永无止境。这样处理器就被稳定在一个无限循环中，直到它被安全地切换出去。  该段代码的最后部分的新行是故意放置的。GNU工具链期望所有的汇编文件都是以空行结尾，以此来确保文件真的结束了。如果没有这么做，在汇编器转换时，你将得到一个恼人的警告信息。   1. 吃派的时间到了   代码写好了，是时候让它在树莓派上转转了。在你的电脑上打开一个终端，并把工作目录切换到当前目录的上一级。输入make并且按下回车。如果出现了错误，请仔细阅读错误定位提示。如果没有，恭喜你，你将获得三个生成的新文件。他们名字分别为：kernel.img，kernel.list和kernel.map。我们逐个介绍这三个文件的用处。Kernel.img是你的编译好了的操作系统镜像。Kernel.list是你书写的汇编文件的代码的清单。逐个文件的主要用处是在未来的某个时刻检查是否正确生成了文件。Kernel.map文件包含所有标签结束位置的映射，当跟踪变量的值时，很有用。  想要按照你刚刚编写的操作系统，首先要做的是找到一个SD卡，其上已经安装好了一个操作系统。把该SD卡插入你的工作电脑里，你会看到一个名叫kernel.img的文件。先把它改个名字，譬如kernel\_linux.img。而后，复制你编译好的kernel.img文件到该位置处。你刚刚所做的事情是把该SD卡中原有的操作系统取代为你自己的操作系统。如果想把该SD卡还原为原来的操作系统，仅仅是把你的kernel.img文件移除或者改名，把已经改名为kernel\_linux.img的文件改回kernel.img即可。记得做好备份很由必要，因为你总是要回来。  把这张SD卡插入树莓派的SD卡插槽里，并接通电源。不出什么意外，OK LED灯就会亮起来。如果没有亮，你就应该看看问题提示页面了。如果亮了，那么恭喜你，你刚刚写了一个你自己的操作系统。既然你已经走到了这一步，不妨继续我们的下一课：让我们的LED灯闪烁。 |