|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Lesson 3 OK03  The OK03 lesson builds on OK02 by teaching how to use functions in assembly to make more reusable and rereadable code. It is assumed you have the code for the [Lesson 2: OK02](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok02.html) operating system as a basis.   |  | | --- | | **Contents**   * [1 Reusable Code](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok03.html#reusable) * [2 Our First Function](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok03.html#firstfunction) * [3 A Big Function](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok03.html#bigfunction) * [4 Another Function](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok03.html#anotherfunction) * [5 A New Beginning](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok03.html#newbeginning) * [6 Onwards](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok03.html#onwards) |   1 Reusable Code  So far we've made code for our operating system by typing the things we want to happen in order. This is fine for such tiny programs, but if we wrote the whole system like this, the code would be completely unreadable. Instead we use functions.  Functions explained  A function is a piece of code that can be reused to compute a certain kind of answer, or perform a certain action. You may also hear them called procedures, routines or subroutines. Although these are all different, people rarely use the correct term.  You should already be happy with the concept of a function from mathematics. For example the cosine function applied to a number gives another number between -1 and 1 which is the cosine of the angle. Notationally we write cos(x) to be the cosine function applied to the value x.  In code, functions can take multiple inputs (including none), give multiple outputs (including none), and may cause side effects. For example a function might create a file on the file system, named after the first input, with length based on the second.  Function as black boxes  Functions are said to be 'black boxes'. We put inputs in, and outputs come out, but we don't need to know how they work.  In higher level code such as C or C++, functions are part of the language itself. In assembly code, functions are just ideas we have.  Ideally we want to be able to set our registers to some input values, branch to an address, and expect that at some point the code will branch back to our code having set the registers to output values. This is what a function is in assembly code. The difficulty comes in what system we use for setting the registers. If we just used any system we felt like, each programmer may use a different system, and would find other programmers' work hard to understand. Further, compilers would not be able to work with assembly code as easily, as they would not know how to use the functions. To prevent confusion, a standard called the Application Binary Interface (ABI) was devised for each assembly language which is an agreement on how functions should be run. If everyone makes functions in the same way, then everyone will be able to use each others' functions. I will teach that standard here, and from now on I will code all of my functions to meet the standard.  The standard says that r0,r1,r2 and r3 will be used as inputs to a function in order. If a function needs no inputs, then it doesn't matter what value it takes. If it needs only one it always goes in r0, if it needs two, the first goes in r0, and the second goes on r1, and so on. The output will always be in r0. If a function has no output, it doesn't matter what value r0 takes.  Further, it also requires that after a function is run, r4 to r12 must have the same values as they had when the function started. This means that when you call a function, you can be sure the r4 to r12 will not change value, but you cannot be so sure about r0 to r3.  When a function completes it has to branch back to the code that started it. This means it must know the address of the code that started it. To facilitate this, there is a special register called lr (link register) which always holds the address of the instruction after the one that called this function.   | Table 1.1 ARM ABI register usage | | | | | --- | --- | --- | --- | | **Register** | **Brief** | **Preserved** | **Rules** | | r0 | Argument and result | No | r0 and r1 are used for passing the first two arguments to functions, and returning the results of functions. If a function does not use them for a return value, they can take any value after a function. | | r1 | Argument and result | No | | r2 | Argument | No | r2 and r3 are used for passing the second two arguments to functions. There values after a function is called can be anything. | | r3 | Argument | No | | r4 | General purpose | Yes | r4 to r12 are used for working values, and their value after a function is called must be the same as before. | | r5 | General purpose | Yes | | r6 | General purpose | Yes | | r7 | General purpose | Yes | | r8 | General purpose | Yes | | r9 | General purpose | Yes | | r10 | General purpose | Yes | | r11 | General purpose | Yes | | r12 | General purpose | Yes | | lr | Return address | No | lr is the address to branch back to when a function is finished, but this does have to contain the same address after the function has finished. | | sp | Stack pointer | Yes | sp is the stack pointer, described below. Its value must be the same after the function has finished. |   Often functions need to use more registers than just r0 to r3. But, since r4 to r12 must stay the same after the method has run, they must be saved somewhere. We save them on something called the stack.  Stack explained  Stack diagram  A stack is a metaphor we use in computing for a method of storing values. Just like in a stack of plates, you can only remove items from the top of a stack, and only add items to the top of the stack.  The stack is a brilliant idea for storing registers on when functions are running. For example if I have a function which needs to use registers r4 and r5, it could place the current values of those registers on a stack. At the end of the method it could take them back off again. What is most clever is that if my function had to run another function in order to complete and that function needed to save some registers, it could put those on the top of the stack while it ran, and then take them off again at the end. That wouldn't affect the values of r4 and r5 that my method had to save, as they would be added to the top of the stack, and then taken off again.  The terminology we used to refer to the values put on the stack by a particular method is that methods 'stack frame'. Not every method needs a stack frame, some don't need to store values.  Because the stack is so useful, it has been implemented in the ARMv6 instruction set directly. A special register called sp (stack pointer) holds the address of the stack. When items are added to the stack, the sp register updates so that it always holds the address of the first item on the stack. **push {r4,r5}** would put the values in r4 and r5 onto the top of the stack and **pop {r4,r5}** would take them back off again (in the correct order).  2 Our First Function  Now that we have some idea about how functions work, let's try to make one. For a basic first example, we are going to make a function that takes no input, and gives an output of the GPIO address. In the last lesson, we just wrote in this value, but it would be better as a function, since it is something we might need to do often in a real operating system, and we might not always remember the address.  Copy the following code into a new file called 'gpio.s'. Just make the new file in the 'source' directory with 'main.s'. We're going to put all functions related to the GPIO controller in one file to make them easier to find.  .globl GetGpioAddress GetGpioAddress: ldr r0,=0x20200000 mov pc,lr  **.globl lbl** makes the label **lbl** accessible from other files.  **mov reg1,reg2** copies the value in **reg2** into **reg1**.  This is a very simple complete function. The **.globl GetGpioAddress** command is a message to the assembler to make the label **GetGpioAddress** accessible to all files. This means that in our main.s file we can branch to the label **GetGpioAddress** even though it is not defined in that file.  You should recognise the **ldr r0,=0x20200000** command, which stores the GPIO controller address in r0. Since this is a function, we have to give the output in r0, so we are not as free to use any register as we once were.  **mov pc,lr** copies the value in **lr** to **pc**. As mentioned earlier **lr** always contains the address of the code that we have to go back to when a method finishes. **pc** is a special register which always contains the address of the next instruction to be run. A normal branch command just changes the value of this register. By copying the value in **lr** to **pc** we just change the next line to be run to be the one we were told to go back to.  A reasonable question would now be, how would we actually run this code? A special type of branch **bl** does what we need. It branches to a label like a normal branch, but before it does it updates **lr** to contain the address of the line after the branch. That means that when the function finishes, the line it will go back to will be the one after the **bl** command. This makes a function running just look like any other command, it simply runs, does whatever it needs to do, and then carries on to the next line. This is a really useful way of thinking about functions. We treat them as 'black boxes' in that when we use them, we don't need to think about how they work, we just need to know what inputs they need, and what outputs they give.  For now, don't worry about using the function, we will use it in the next section.  3 A Big Function  Now we're going to implement a bigger function. Our first job was to enable output on GPIO pin 16. It would be nice if this was a function. We could simply specify a pin and a function as the input, and the function would set the function of that pin to that value. That way, we could use the code to control any GPIO pin, not just the LED.  Copy the following commands below the GetGpioAddress function in gpio.s.  .globl SetGpioFunction SetGpioFunction: cmp r0,#53 cmpls r1,#7 movhi pc,lr  Suffix **ls** causes the command to be executed only if the last comparison determined that the first number was less than or the same as the second. Unsigned.  Suffix **hi** causes the command to be executed only if the last comparison determined that the first number was higher than the second. Unsigned.  One of the first things we should always think about when writing functions is our inputs. What do we do if they are wrong? In this function, we have one input which is a GPIO pin number, and so must be a number between 0 and 53, since there are 54 pins. Each pin has 8 functions, numbered 0 to 7 and so the function code must be too. We could just assume that the inputs will be correct, but this is very dangerous when working with hardware, as incorrect values could cause very bad side effects. Therefore, in this case, we wish to make sure the inputs are in the right ranges.  To do this we need to check that r0 <= 53 and r1 <= 7. First of all, we can use the comparison we've seen before to compare the value of r0 with 53. The next instruction, **cmpls** is a normal comparison instruction that will only be run if r0 was lower than or the same as 53. If that was the case, it compares r1 with 7, otherwise the result of the comparison is the same as before. Finally we go back to the code that ran the function if the result of the last comparison was that the register was higher than the number.  The effect of this is exactly what we want. If r0 was bigger than 53, then the **cmpls** command doesn't run, but the **movhi** does. If r0 is <= 53, then the **cmpls** command does run, and so **r1** is compared with 7, and then if it is higher than 7, **movhi** is run, and the function ends, otherwise **movhi** does not run, and we know for sure that r0 <= 53 and r1 <= 7.  There is a subtle difference between the **ls** (lower or same) and **le** (less or equal) as well as between **hi** (higher) and **gt** (greater) suffixes, but I will cover this later.  Copy these commands below the above.  push {lr} mov r2,r0 bl GetGpioAddress  **push {reg1,reg2,...}** copies the registers in the list **reg1,reg2,...** onto the top of the stack. Only general purpose registers and lr can be pushed.  **bl lbl** sets **lr** to the address of the next instruction and then branches to the label **lbl**.  These next three commands are focused on calling our first method. The **push {lr}** command copies the value in **lr** onto the top of the stack, so that we can retrieve it later. We must do this because when we call GetGpioAddress, we will need to use **lr** to store the address to come back to in our function.  If we did not know anything about the GetGpioAddress function, we would have to assume it changes r0,r1,r2 and r3, and would have to move our values to r4 and r5 to keep them the same after it finishes. Fortunately, we do know about GetGpioAddress, and we know it only changes r0 to the address, it doesn't affect r1,r2 or r3. Thus, we only have to move the GPIO pin number out of r0 so it doesn't get overwritten, but we know we can safely move it to r2, as GetGpioAddress doesn't change r2.  Finally we use the **bl** instruction to run GetGpioAddress. Normally we use the term 'call' for running a function, and I will from now. As discussed earlier **bl** calls a function by updating the lr to the next instruction's address, and then branching to the function.  When a function ends we say it has 'returned'. When the call to GetGpioAddress returns, we now know that r0 contains the GPIO address, r1 contains the function code and r2 contains the GPIO pin number. I mentioned earlier that the GPIO functions are stored in blocks of 10, so first we need to determine which block of ten our pin number is in. This sounds like a job we would use a division for, but divisions are very slow indeed, so it is better for such small numbers to do repeated subtraction.  Copy the following code below the above.  functionLoop$:  cmp r2,#9 subhi r2,#10 addhi r0,#4 bhi functionLoop$  **add reg,#val** adds the number **val** to the contents of the register **reg**.  This simple loop code compares the pin number to 9. If it is higher than 9, it subtracts 10 from the pin number, and adds 4 to the GPIO Controller address then runs the check again.  The effect of this is that r2 will now contain a number from 0 to 9 which represents the remainder of dividing the pin number by 10. r0 will now contain the address in the GPIO controller of this pin's function settings. This would be the same as GPIO Controller Address + 4 × (GPIO Pin Number ÷ 10).  Finally, copy the following code below the above.  add r2, r2,lsl #1 lsl r1,r2 str r1,[r0] pop {pc}  Argument shift **reg,lsl #val** shifts the binary representation of the number in **reg** left by **val** before using it in the operation before.  **lsl reg,amt** shifts the binary representation of the number in **reg** left by the number in **amt**.  **str reg,[dst]** is the same as **str reg,[dst,#0]**.  **pop {reg1,reg2,...}** copies the values from the top of the stack into the register list **reg1,reg2,...**. Only general purpose registers and pc can be popped.  This code finishes off the method. The first line is actually a multiplication by 3 in disguise. Multiplication is a big and slow instruction in assembly code, as the circuit can take a long time to come up with the answer. It is much faster sometimes to use some instructions which can get the answer quicker. In this case, I know that r2 × 3 is the same as r2 × 2 + r2. It is very easy to multiply a register by 2 as this is conveniently the same as shifting the binary representation of the number left by one place.  One of the very useful features of the ARMv6 assembly code language is the ability to shift an argument before using it. In this case, I add **r2** to the result of shifting the binary representation of **r2** to the left by one place. In assembly code, you often use tricks such as this to compute answers more easily, but if you're uncomfortable with this, you could also write something like **mov r3,r2**; **add r2,r3**; **add r2,r3**.  Now we shift the function value left by a number of places equal to r2. Most instructions such as **add** and **sub** have a variant which uses a register rather than a number for the amount. We perform this shift because we want to set the bits that correspond to our pin number, and there are three bits per pin.  We then store the the computed function value at the address in the GPIO controller. We already worked out the address in the loop, so we don't need to store it at an offset like we did in OK01 and OK02.  Finally, we can return from this method call. Since we pushed **lr** onto the stack, if we pop **pc**, it will copy the value that was in **lr** at the time we pushed it into **pc**. This would be the same as having used **mov pc,lr** and so the function call will return when this line is run.  The very keen may notice that this function doesn't actually work correctly. Although it sets the function of the GPIO pin to the requested value, it causes all the pins in the same block of 10's functions to go back to 0! This would likely be quite annoying in a system which made heavy use of the GPIO pins. I leave it as a challenge to the interested to fix this function so that it does not overwrite other pins values by ensuring that all bits other than the 3 that must be set remain the same. A solution to this can be found on the downloads page for this lesson. Functions that you may find useful are **and** which computes the Boolean and function of two registers, **mvns** which computes the Boolean not and **orr** which computes the Boolean or.  4 Another Function  So, we now have a function which takes care of the GPIO pin function setting. We now need to make a function to turn a GPIO pin on or off. Rather than having one function for off and one function for on, it would be handy to have a single function which does either.  We will make a function called SetGpio which takes a GPIO pin number as its first input in r0, and a value as its second in r1. If the value is 0 we will turn the pin off, and if it is not zero we will turn it on.  Copy and paste the following code at the end of 'gpio.s'.  .globl SetGpio SetGpio: pinNum .req r0 pinVal .req r1  **alias .req reg** sets **alias** to mean the register **reg**.  Once again we need the **.globl** command and the label to make the function accessible from other files. This time we're going to use register aliases. Register aliases allow us to use a name other than just r0 or r1 for registers. This may not be so important now, but it will prove invaluable when writing big methods later, and you should try to use aliases from now on. **pinNum .req r0** means that **pinNum** now means **r0** when used in instructions.  Copy and paste the following code after the above.  cmp pinNum,#53 movhi pc,lr push {lr} mov r2,pinNum .unreq pinNum pinNum .req r2 bl GetGpioAddress gpioAddr .req r0  **.unreq alias** removes the alias **alias**.  Like in SetGpioFunction the first thing we must do is check that we were actually given a valid pin number. We do this in exactly the same way by comparing **pinNum** (**r0**) with 53, and returning immediately if it is higher. Once again we wish to call GetGpioAddress, so we have to preserve **lr** by pushing it onto the stack, and to move **pinNum** to **r2**. We then use the **.unreq** statement to remove our alias from **r0**. Since the pin number is now stored in **r2** we want our alias to reflect this, so we remove the alias from **r0** and remake it on **r2**. You should always **.unreq** every alias as soon as it is done with, so that you cannot make the mistake of using it further down the code when it no longer exists.  We then call GetGpioAddress, and we create an alias for **r0** to reflect this.  Copy and paste the following code after the above.  pinBank .req r3 lsr pinBank,pinNum,#5 lsl pinBank,#2 add gpioAddr,pinBank .unreq pinBank  **lsr dst,src,#val** shifts the binary representation of the number in **src** right by **val**, but stores the result in **dst**.  The GPIO controller has two sets of 4 bytes each for turning pins on and off. The first set in each case controls the first 32 pins, and the second set controls the remaining 22. In order to determine which set it is in, we need to divide the pin number by 32. Fortunately this is very easy, at is the same as shifting the binary representation of the pin number right by 5 places. Hence, in this case I've named **r3** as **pinBank** and then computed **pinNum** ÷ 32. Since it is a set of 4 bytes, we then need to multiply the result of this by 4. This is the same as shifting the binary representation left by 2 places, which is the command that follows. You may wonder if we could just shift it right by 3 places, as we went right then left. This won't work however, as some of the answer may have been rounded away when we did ÷ 32 which may not be if we just ÷ 8.  The result of this is that **gpioAddr** now contains either 2020000016 if the pin number is 0-31, and 2020000416 if the pin number is 32-53. This means if we add 2810 we get the address for turning the pin on, and if we add 4010 we get the address for turning the pin off. Since we are done with **pinBank**, I use **.unreq** immediately afterwards.  Copy and paste the following code after the above.  and pinNum,#31 setBit .req r3 mov setBit,#1 lsl setBit,pinNum .unreq pinNum  **and reg,#val** computes the Boolean and function of the number in **reg** with **val**.  This next part of the function is for generating a number with the correct bit set. For the GPIO controller to turn a pin on or off, we give it a number with a bit set in the place of the remainder of that pin's number divided by 32. For example, to set pin 16, we need a number with the 16th bit a 1. To set pin 45 we would need a number with the 13th bit 1 as 45 ÷ 32 = 1 remainder 13.  The **and** command computes the remainder we need. How it does this is that the result of an and operation is a number with 1s in all binary digits which had 1s in both of the inputs, and 0s elsewhere. This is a fundamental binary operation, and is very quick. We have given it inputs of **pinNum** and 3110 = 111112. This means that the answer can only have 1 bits in the last 5 places, and so is definitely between 0 and 31. Specifically it only has 1s where there were 1s in **pinNum**'s last 5 places. This is the same as the remainder of a division by 32. It is no coincidence that 31 = 32 - 1.  binary division example  The rest of this code simply uses this value to shift the number 1 left. This has the effect of creating the binary number we need.  Copy and paste the following code after the above.  teq pinVal,#0 .unreq pinVal streq setBit,[gpioAddr,#40] strne setBit,[gpioAddr,#28] .unreq setBit .unreq gpioAddr pop {pc}  **teq reg,#val** checks if the number in **reg** is equal to **val**.  This code ends the method. As stated before, we turn the pin off if pinVal is zero, and on otherwise. **teq** (test equal) is another comparison operation that can only be used to test for equality. It is similar to cmp but it does not work out which number is bigger. If all you wish to do is test if to numbers are the same, you can use **teq**.  If **pinVal** is zero, we store the **setBit** at 40 away from the GPIO address, which we already know turns the pin off. Otherwise we store it at 28, which turns the pin on. Finally, we return by popping the **pc**, which sets it to the value that we stored when we pushed the link register.  5 A New Beginning  Finally, after all that work we have our GPIO functions. We now need to alter 'main.s' to use them. Since 'main.s' is now getting a lot bigger and more complicated, it is better design to split it into two sections. The '.init' we've been using so far is best kept as small as possible. We can change the code to reflect this easily.  Insert the following just after **\_start:** in main.s:  b main  .section .text main: mov sp,#0x8000  The key change we have made here is to introduce the **.text** section. I have designed the makefile and linker scripts such that code in the .text section (which is the default section) is placed after the .init section which is placed at address 800016. This is the default load address and gives us some space to store the stack. As the stack exists in memory, it has to have an address. The stack grows down memory, so that each new value is at a lower address, thus making the 'top' of the stack, the lowest address.  The 'ATAGs' section in the diagram is a place where information about the Raspberry Pi is stored such as how much memory it has, and what its default screen resolution is.  Layout diagram of operating system  Replace all the code that set the function of the GPIO pin with the following:  pinNum .req r0 pinFunc .req r1 mov pinNum,#16 mov pinFunc,#1 bl SetGpioFunction .unreq pinNum .unreq pinFunc  This code calls SetGpioFunction with the pin number 16 and the pin function code 1. This has the effect of enabling output to the OK LED.  Replace any code which turns the OK LED on with the following:  pinNum .req r0 pinVal .req r1 mov pinNum,#16 mov pinVal,#0 bl SetGpio .unreq pinNum .unreq pinVal  This code uses SetGpio to turn off GPIO pin 16, thus turning on the OK LED. If we instead used **mov pinVal,#1**, it would turn the LED off. Replace your old code to turn the LED off with that.  6 Onwards  Hopefully now, you should be able to test what you have made on the Raspberry Pi. We've done a large amount of code this time, so there is a lot that can go wrong. If it does, head to the troubleshooting page.  When you get it working, congratulations. Although our operating system does nothing more than it did in [Lesson 2: OK02](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok02.html), we've learned a lot about functions and formatting, and we can now code new features much more quickly. It would be very simple now to make an Operating System that alters any GPIO register, which could be used to control hardware!  In [Lesson 4: OK04](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok04.html), we will address our wait function, which is currently imprecise, so that we can gain better control over our LED, and ultimately over all of the GPIO pins. | 第三课 OK03  第3课是基于第2课的。这一课主要讲解如何在汇编文件中利用函数来使得代码更加容易重用和更加容易阅读。我们假设你已经看过第2课，并做过相应的实践。  目录   1. 可重用的代码 2. 第一个函数 3. 一个巨大的函数 4. 其他函数 5. 一个新的开始 6. 同志仍需努力 7. 可重用代码   就我们的操作系统来说，我们已经输入了一些代码，并期望它按照我们预想的那样执行。这么一个小程序这么写还是很有趣的。但是如果我们要实现的是整个操作系统的话，这么写代码将变得完全不可阅读，甚至将一个灾难的开始。取而代之的是利用函数。  函数解析  （图）  函数被说成是黑盒子。我们给它一些参数，并且它给我们一些输出，这就足够了，我们并不需要直到它是如何工作的 。  在更高抽象级别的语言中（例如C或者C++），函数是它们语言本身的一个部分。而在汇编语言中，函数就是一种想法。  我们直觉上应该有个想法：我们可以把一些数值放进某些寄存器中，并以此作为输入，而后由一段代码进行执行一系列动作，最后再跳转回来，并把输出放到某些寄存器中。这就是汇编语言中函数的本质。函数真正的难点是在于设置寄存器时我们使用的系统是什么。如果我们任意选择我们喜欢的系统，那么每个程序员可能就会使用不同的系统，并且很难和其他的程序员来进行协调。更甚者，编译器都无法轻易地协调这些代码。因为并不知道你自定义的函数结构是如何工作的。为了消除这些混淆，一种被称为应用程序二进制结构（ABI）被发明出来，并在使用汇编语言编写者中达成协议。ABI规定了汇编语言中函数应该如何工作。如果程序员都按照同样的协定进行编程，那么函数将变得很通用。我会在这个地方示范这个标准的写法，并从此以后，我所有的函数都将采用这个标准写法。  标准是这么说的：寄存器r0、r1、r2和r3依照次序作为函数的输入参数。如果函数不需要输出，那么寄存器r0、r1、r2和r3里的数值是什么就无所谓了。如果函数只需要一个输入，那么就使用r0；如果需要二个输入，那么就使用r0和r1，以此类推。但是输出永远都是再r0里。还是那句话，如果函数没有输出，那么寄存器r0里是什么，也就无所谓了。  还有些规定：当函数开始前，寄存器r4和寄存器r12里的是数值什么，函数结束后，r4和r12里的数值仍要是什么，请你写的代码确保这一点。寄存器r0和r3的数值，你就无法确保了。  当函数结束后，它必须跳转回函数代码开始的位置。函数从哪里来的，就要回到哪里去。为了实现这个原则，硬件上有个寄存器lr（link register）总是再代码调用函数时，保存着调用指令的地址。  表1.1 ARM ABI寄存器用法  而大多数情况下，函数需要用到比r0到r3更多的寄存器。因为r4到r12必须保持数值不变，而函数又需要使用它们，所以，要想办法把r4到r12的数值保存到别的地方，以便给我们的函数腾出空间来。在函数结束之前，我们想办法恢复r4到r12中的数值就可以了。我们经常把r4到r12中的数值保存在一个叫做栈的地方。  栈的扩展阅读  因为栈相当有用，所以ARMv6指令集中直接实现了它，并且用一个特殊寄存器sp（stack pointer）来存储栈的地址。当把数据项存入栈中时，寄存器sp就会更新，以便总是指向栈中第一个项（也就是保存该项的地址数值）。指令push {r4, r5}将会把寄存器r4和r5中数值保存到栈顶。指令pop {r4, r5}将按照正确的顺序把数值从栈顶放回到寄存器r4和r5中。   1. 第一个函数   既然我们已经有了函数如何工作的大致想法，那么就来试着做一个看看吧。作为打基础的第一个例子，我们从没有输入的函数开始，并且让它把GPIO控制器的地址作为输出。在本课程的最后部分，我们将要利用这个控制器的地址。因为我们的操作系统要经常用到这个地址并且这个地址并不是那么容易记住，所以，好办法就是把获得该控制器的地址的代码块来写成一个函数形式。  把下面的代码复制到一个新的文件里，并取名为“gpio.s”。这个文件要和“main.s”放在一个目录里。我们将把和GPIO控制器相关的所有函数都放在一个文件里，这样用的时候容易找到。  .globl GetGpioAddress  GetGpioAddress:  ldr r0, =0x20200000  mov pc, lr  指令.globl lbl的作用是：让其他文件里的代码可以访问到这个文件里的标签lbl。（因为GetGpioAddress函数要被别的代码用到，所以，你必须要让别的文件了的代码能访问到这个函数吧。）  指令mov reg1, reg2的作用是把寄存器reg2里的数值拷贝到reg1寄存器中。  这个函数相当简单。指令.globl GetGpioAddress给汇编器发送一条消息，告诉其他文件GetGpioAddress函数在“gpio.s”文件里。这意味着在文件“main.s”里，我们甚至可以跳转到标签GetGpioAddress处，尽管我们并没有在“main.s”文件里定义这个标签。  你应该对指令ldr r0, =0x20200000并不陌生。它的作用是把GPIO控制器的地址数值存入寄存器r0中。因为我们在函数里写这条指令，所以，我们不能随意用别的寄存器，只能按照标准选用寄存器r0。  指令mov pc, lr意思是把lr寄存器里的数值拷贝到pc寄存器中。之前提到过，寄存器lr里存储着一个函数要返回的地址数值。而寄存器pc是另外一个特殊寄存器。该寄存器总是存储着处理器要执行代码的下一行代码的地址。一个跳转指令就会改变这个特殊寄存器的数值，并因此改变处理器执行的路径。如果把寄存器lr的数值拷贝给pc寄存器，这就会把要执行的下一行代码的地址修改为函数要返回的地址。  问题来了：我们的代码该如何调用这些函数呢？一条新的指令bl被用来做这些事情。该指令会跳转到一个标签处去执行。但在跳转之前，该指令会更新lr寄存器的数值，用的数据就是bl指令下一行代码的地址。也就是说，当函数执行完毕后，程序的执行路径会回来，并从bl指令的下一行开始。这就给我们一种这样的感觉：我们自己写的函数（或者是别人写好的函数）就像一条指令一样，做它们想做的，而后返回来继续执行。你这么考虑函数就对了。这种思维模式叫做自顶而下。我们一般把要用到的工具（包括生活中所有的工具）看作是个黑盒子，我们并不需要知道它们是如何实现这样的功能的，我们只关心它们输出什么和它们如何输出。就好比我们都有朋友，我们并不太了解朋友是如何构成的，只要知道他们可以倾诉和什么时间可以交流就足够了。  目前，你不要太担心如何运用这些函数，下面我将给你展示这项技能。   1. 一个巨大的函数   现在，我们要去实现一个更大的函数。我们的首要工作就是让GPIO的第16号引脚可以输出。如果这个做成函数，那结果将很美妙。我们可以把函数要处理的引脚号码作为函数的输入，那样函数将把接收到的数值来对相应的引脚进行设置。这样我们可以利用该函数进行设置任何GPIO引脚了，而不是局限到LED灯。  把下面的代码复制到文件“gpio.s”中GetGpioAddress函数的下面。  .globl SetGpioFunction  SetGpioFunction:  cmp r0, #53  cmpls r1, #7  movhi pc, lr  指令后缀ls指示出：要运行当前的命令，要满足的条件是：当且仅当第一个参数小于等于第二个参数，参数无符号。  指令后缀hi指示出：要运行当前的命令，要满足的条件是：当且仅当第一个参数大于第二个参数，参数无符号。  当我们写函数时，我们总是应该想着我们函数的输入。要是我们把输入搞错了，会怎么样呢？我们的这个例子中，只有一个输入，它表示GPIO引脚的引脚号码。因为有54个引脚，所以它的取值范围应该时0到53之间，包括临界值。每个引脚有8个函数与其相关，这些函数用标注了0到7这8个数字来区别，自然函数的代码也应该据此来区别不同的引脚。我们可以假定输入时正确的，而并不对错误的输入进行处理。但是，这么做很危险，尤其是在处理硬件时。因为，当错误的输入产生时，将导致很严重的边界错误。我们的这个例子中，我们要确保输入在合理的范围内。  为了做到这一点，我们需要检查寄存器r0里的数值小于等于53并且寄存器r1的数值小于等于7。我们可以利用之前学到的比较指令来比较寄存器r0的数值和数值53。接下来的指令cmpls时个普通的比较指令，它仅在寄存器r0的数值小于等于53时才执行。如若是这样，指令cmpls将比较寄存器r1和数值7；否则，比较的结果和之前的相同。如果最近的一次比较的结果是大于寄存器里的数，那么我们的代码执行流程就会返回到调用函数的地方。  这就是我们想要的。如果寄存器r0的数值大于53，指令cmpls将不会执行，但是movhi指令却可以得到执行。如果寄存器r0的数值小于等于53，那么指令cmpls将运行，并且寄存器r1的数值将和7进行比较，如果大于7，movhi指令将会得到执行，并且函数就此结束。否则，指令movhi将不会得到执行，并且我们确信寄存器r0的数值一定小于等于53，并且寄存器r1的数值一定小于等于7。  这里有个不容易察觉的区别存在于ls（小于或者相同）和le（少于和等于），同样的区别也存在于hi（高于）和gt（大于）后缀之间。别担心，我随后会提及。  把下面的代码拷贝到上面代码的下面。  Push {lr}  Mov r2, r0  Bl GetGpioAddress  指令push {reg1, reg2, …}意思是拷贝寄存器reg1、reg2，…中的数值到栈的顶部。只有普通用途寄存器和lr寄存器的数值可以被压入栈中。  指令bl lbl意思是设置寄存器lr数值为下一行代码的地址并且跳转到标签lbl处。  下面的三行代码的主要功能是调用我们的第一个函数。指令push {lr}的意思是拷贝寄存器lr中的数值到栈的顶部，以便后面用到时可以恢复。我们必须这么做，因为当我们调用函数GetGpioAddress时，我们需要运用寄存器lr来存储返回我们函数的地址。  如果我们对GetGpioAddress函数一无所知，我们就可以做这样的假设：函数将改变寄存器r0、r1、r2和r3的数值，并且把它们的数值移动到r4和r5中，以便在函数返回时，这些数值保持和进入函数时一致。幸运的是，我们的确了解函数GetGpioAddress，并且我们也了解到该函数仅仅改变了寄存器r0的数值，将其改变为一个地址，它并不影响寄存器r1、r2和r3的数值。所以，我们仅仅需要把GPIO引脚数移出寄存器r0来防止它被复写。既然函数并不影响寄存器r2，那么我们就把该数值放到r2里吧。  最终我们还是要运用指令bl去调用函数GetGpioAddress。通常，我们用术语“call”来运行一个函数，并且以后我也会这个术语。这个术语的实质就是之前讨论的那样：利用指令bl来调用函数，并用bl指令的下一行代码的地址数据来更新特殊寄存器lr，然后跳转到函数中去。  当一个函数结束时，我们就称这个函数要“返回”了。当调用函数GetGpioAddress返回时，我们清楚地知道，在寄存器r0中保存着GPIO控制器的地址，r1保存着函数代码，r2保存着GPIO引脚数值。我之前提到过：GPIO函数保存在10个块中，因此我们需要去决定10个块中的那个块是我们需要的引脚号。这听起来像是要用到除法，但是除法执行起来相当慢。一个更加明智的做法是用循环的减法来实现它。  把下面的代码拷贝到之前代码的下面。  functionLoop$:  cmp r2, #9  subhi r2, #10  addhi r0, #4  bhi functionLoop$  指令add reg, #val意思是把数值val加到寄存器reg中。  这一小片循环代码把引脚号和数值9进行比较。如果高于9，则从引脚号里减去10，并且把GPIO控制器地址加上4，然后再次执行检测。  这段代码将会让寄存器r2中保存一个从0到9的数值，这个数值暂存了被10整除的余数。寄存器r0会保存这个引脚函数设置的GPIO控制器的地址。这个数值和公式：GPIO控制器地址+4\*（GPIO引脚号/10）所表达的数值相同。  最后，把下面的代码复制到上面的代码的下面。  add r2, r2, lsl #1  lsl r1, r2  str r1, [r0]  pop {pc}  指令参数移动reg, lsl #val将会移动一个在寄存器reg中保存的数值的二进制表示以val单位，而且这个动作是在执行真正指令之前进行的。  指令lsl reg, amt将移动一个在寄存器reg中保存的数值的二进制表示以amt个单位。  指令str reg, [dst]和指令str reg, [dst, #0]相同。  指令pop {reg1, reg2, …}将把栈顶的数值依次拷贝到寄存器reg1, reg2, …中去。只有普通用途寄存器和pc寄存器可以使用pop指令。  这个函数将以这段代码终结。第一行看起来是个加法指令，但是却是个用加法指令伪装的乘法指令，实际上是乘以3操作。因为在硬件电路层，实现一个乘法并求得结果将耗费很长一段时间，因此，汇编语言中乘法指令是一个胖且慢的指令。但是有时候，用一些其他的指令，可以更加快速得到我们想要的求解答案。在本例中，寄存器r2中数值x3操作和寄存器r2中的数值x2然后再加上一个r2的运算结果是一样的。而且，乘2操作很便捷，因为仅仅需要把数值的二进制表示左移一个单位即可。  ARMv6汇编语言一个很非常有用的特征是：在真正执行一个指令前，可以先对参数进行移动。本例中，我把寄存器r2中的数值加上了r2中的数值的二进制表示左移一个单位的结果。在汇编语言中，我们经常会用到这样的小技巧来快速求解。但是如果我们对此并不适应，你可以直觉地利用下面的代码：  mov r3, r2  add r2, r3  add r2, r3  其作用和本例是一样一样的。  现在，我们已经移动了函数的数值，且移动的数值为寄存器r2中保存的数值。大多数指令（比如add和sub）都使用寄存器来当作一个变量，而不是一个数值。我们之所以移动这个数值，是因为我们要设置和引脚号相对应的位。记得每个引脚对应三个位。  然后，我们把经计算而得的GPIO控制器地址数值进行保存。我们已经在一个循环中计算得到这个地址，所以没有必要像OK01和OK02课程里介绍的那样存储在一个偏移地址里。  最终，我们可以把经过函数调用计算得到的数值返回了。因为我们已经把寄存器lr的数值压入栈中，所以当我们pop pc后，lr中的数值就会拷贝到pc中去。这就相当于使用指令mov pc, lr，并且当这行代码执行时，函数调用也将返回。 |