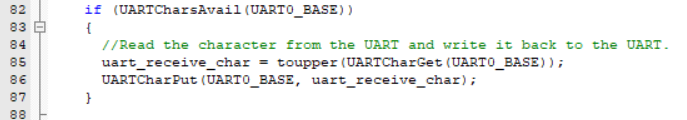
# ARM实验三 UART串行通讯口实验

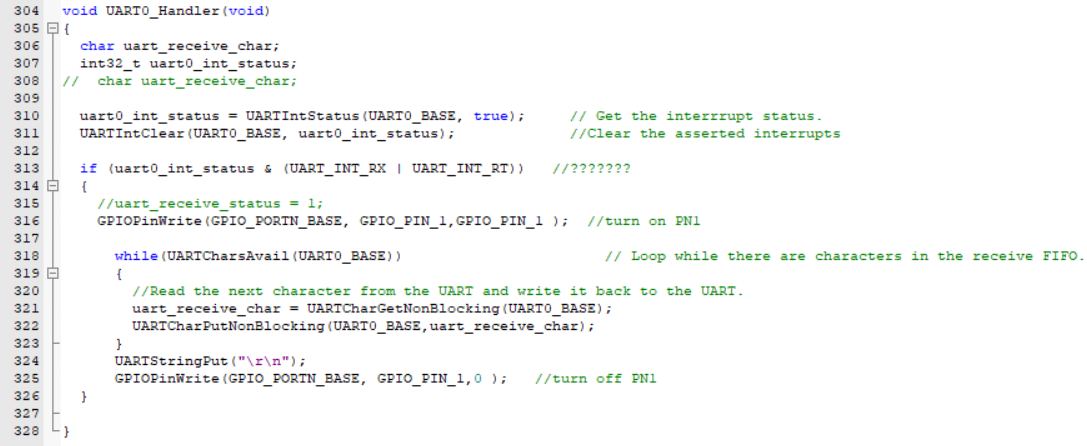
1. **实验主要程序段展示与分析**

实验3.1：



该实验比较简单，只需要在例程的返回字符前做一个变为大写的操作即可。当然在代码开始要加上#include <ctype.h>这个头文件。

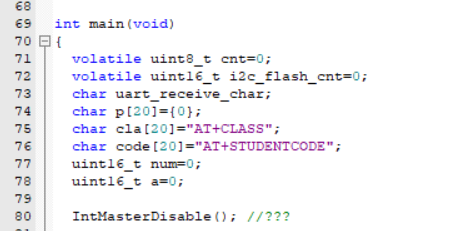
实验3.2：



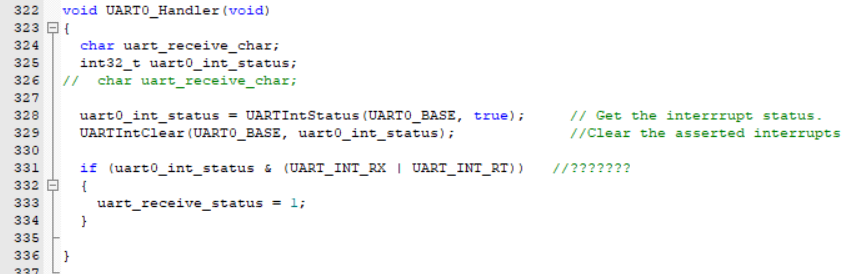
将任务1代码移到中断服务程序内。实验结果显示，当发送字符超过八个，会分为两行返回。这是因为中断条件中将接收FIFO的触发深度设为了4/8，也就是8位。每隔八位会中断一次，cpu收取一次接收FIFO里的字符并将其送上发送FIFO，最后送上回车和换行。最后一行不足八位的也能返回是因为设置了接收超时作为另一个中断条件。

实验结果上看，是否把任务1代码放入中断服务程序没有什么区别。但是从结构上看两种方法各有优势。如果将代码放入中断服务程序，可以保证中断时能执行到这一代码，否则在中断服务程序外可能受到其他代码的干扰；如果不将代码放入中断服务程序，前后台程序之间显得更合理，毕竟中断服务程序作为异常，应当简洁，尽量不要过多干扰系统正常运行，否则所有的工作都被中断服务程序完成，有喧宾夺主之感。

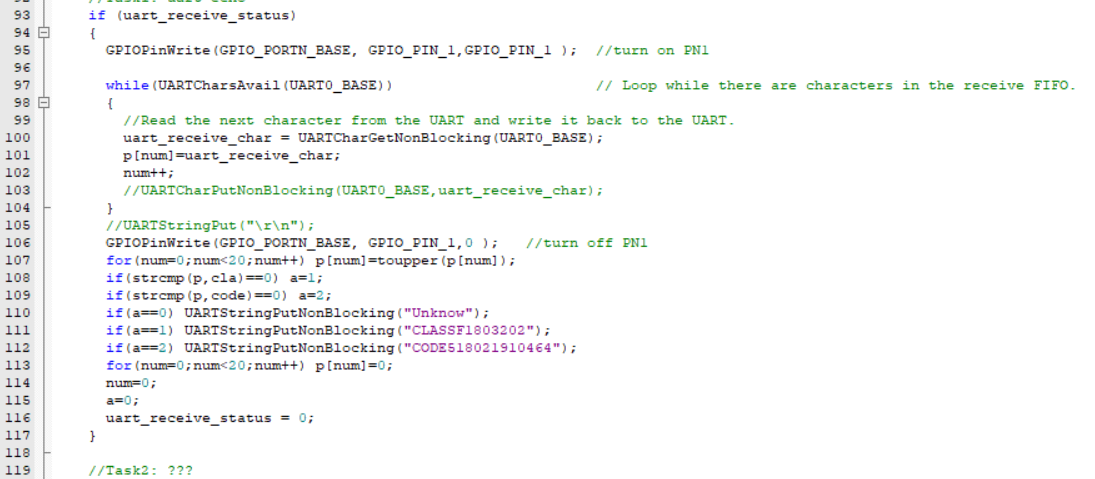
实验3.3：



首先定义变量。p[20]数组用来按位接收pc发来的字符。下面的cla以及code数组用来记录指令，用以比较。整形num是针对p[20] 的计数器，a是状态标志。

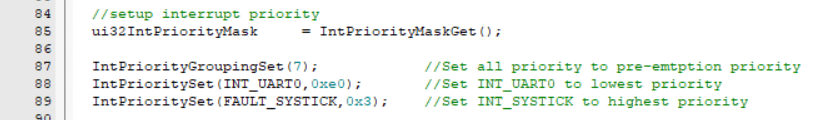


UARTO中断和例程一致。处理函数放在主函数内。



编写task1的内容。第107行开始是关键。107行利用toupper函数将输入字符都变成大写存入p数组内。后两行将p数组与cla、code比较，确定pc发来了哪条指令。后三行针对不同的情况发回数据作为响应。为了准备下一次响应，113行到116行的恢复初始状态的代码是必要的。

实验3.4：



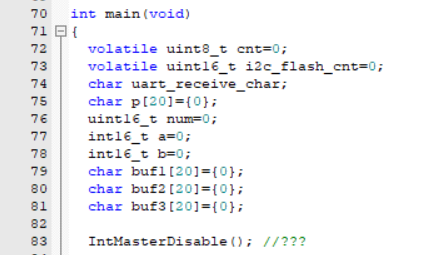
在例程代码中，systick中断优先级低于UART0中断。简单交换两者优先级就可以使得systick中断优先级高于UART0中断。

实验现象：不按任何按键时，D1亮，走马灯正常运行，从pc端发送信息可以正常回应；长按USR\_SW1时，D1和D2亮，走马灯暂停，从pc端发送信息无回应，松开按键后D2灭，实验板发回了信息，但如果pc发送的信息超过十六位，实验板只能回应前十六位；长按USR\_SW2时，D1亮，当pc端发送信息后，实验板能正常回应，走马灯暂停。

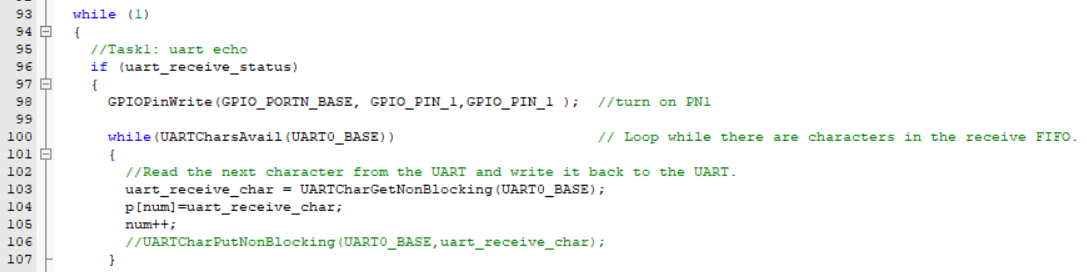
解释：长按USR\_SW1时，程序陷入systick中断服务程序中，由于systick中断优先级高于UART0中断，因此pc发来数据使接收fifo达到设定深度或超时时不会触发UART0中断进行接收，更不会再发送回来，因此实验板无响应信息，task3不会执行。同时由于程序在systick中断服务程序内，因此不会触发task1和下一个systick中断，故跑马灯暂停。松开按键后，程序在systick定时中断的间隔中可以进入UART0中断，接收到接收fifo内的字符并发回。由于实验时，一般是在pc端发送完全部字符后再松开的按键，因此，十六位之后的字符因为接收fifo已满而丢失，所以实验板只能回应前十六位。

长按USR\_SW2时，由于systick中断优先级高，程序每隔一个systick定时就进入一次systick中断，之后回到UART0中断，task2能够正常运行，由于关闭D1的语句在UART0中断服务程序的while循环外，回到UART0中断直接回到while内而不会执行关闭操作，故D1表现出连续发光。其余时间程序都在UART0中断服务程序内，因此task1不可能执行，所以跑马灯暂停。由于程序大部分时间都在UART0接收状态，且收发函数也写在UART0中断服务程序while内，因此实验板能发送回字符。

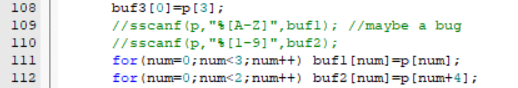
实验3.5：



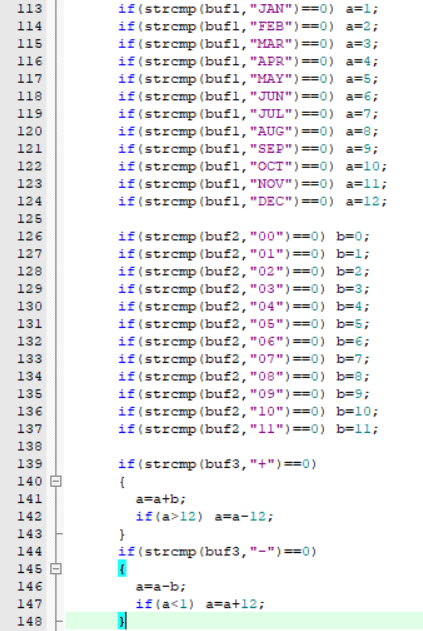
首先需要定义一些变量。p数组用来按顺序存放实验板接收的字符，与实验3.3相同。定义buf1~3数组用来存放处理p数组的结果，将p数组分别拆分为月，加减符号和数字符（如MAY、+和01）。变量a和b用来存放整型化的月份数字和数字符。



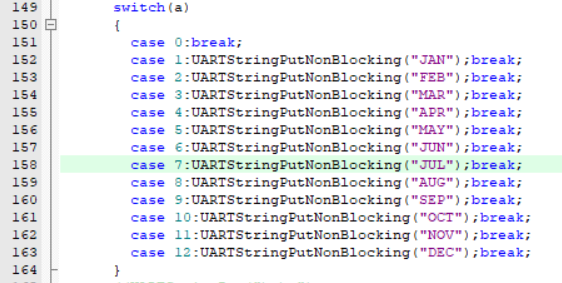
利用中断接收字符，存入p数组，这个操作和3.3类似。



由于发来的命令是格式确定的，可以比较容易地拆分出来。不知什么原因，我的keil环境一运行sscanf函数就容易闪退，而相同的代码在codeblocks里能正常工作。所以我被迫采用了for循环等按位拆分并存放的方法，没有使用sscanf这一格式化处理的方法。

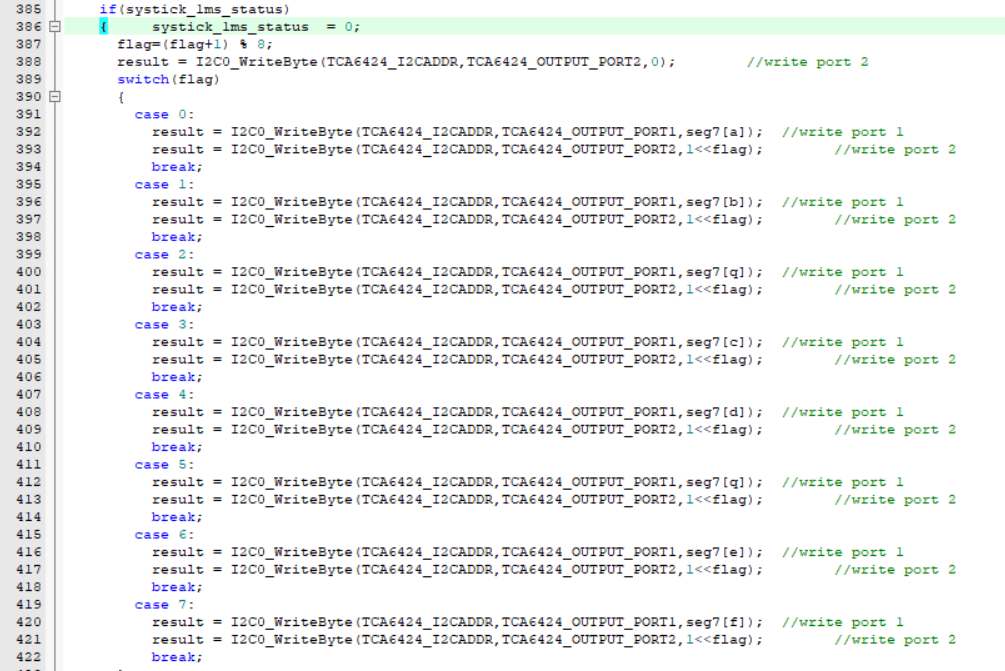


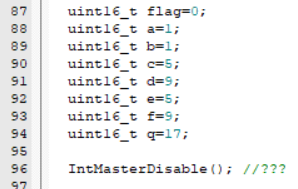
接下来是对比字符串buf1和buf2以确定a和b的值，然后判断buf3中存放的符号，确定运算是加还是减，并做模12的操作。这里就可以看出定义变量时有必要把a定义为有符号量。



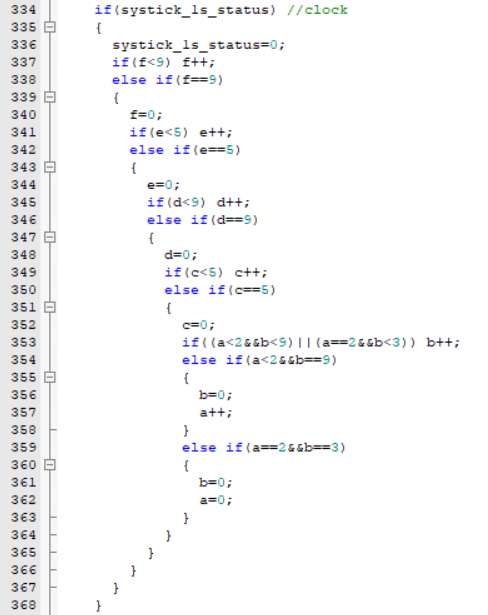
最后分类判断a的值，输回相应的月份即可。

实验3.6：

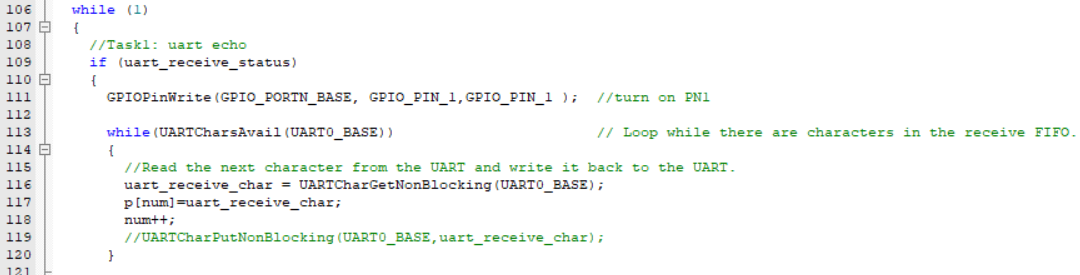




数码管显示模块。相比实验二，多做了388行的去残影，另外加入了小时，扩展为8位显示。由于显示位数增多，3ms显得长了，改为1ms。为了配合reset后是12：00：00，重新设定了各位初值。因为reset后systick\_1s\_couter是初始值0，会立即触发一次systick中断中关于1秒定时的语句使运行秒数立即加一，所以故意设置为11：59：59。

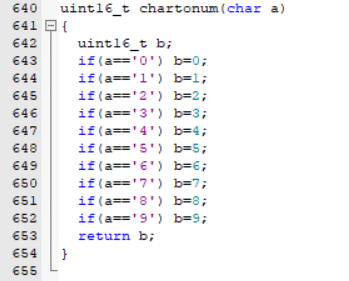


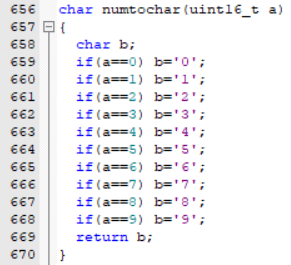
正常的时钟模块，每隔1s加1秒。依然是实验二中的逐位判断法，由于小时是24进制，有特殊性，因此从353行开始做了一点小调整。

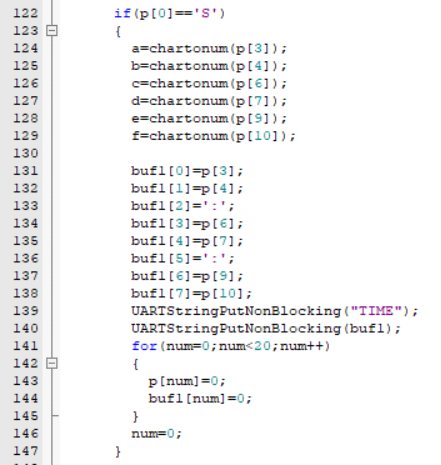




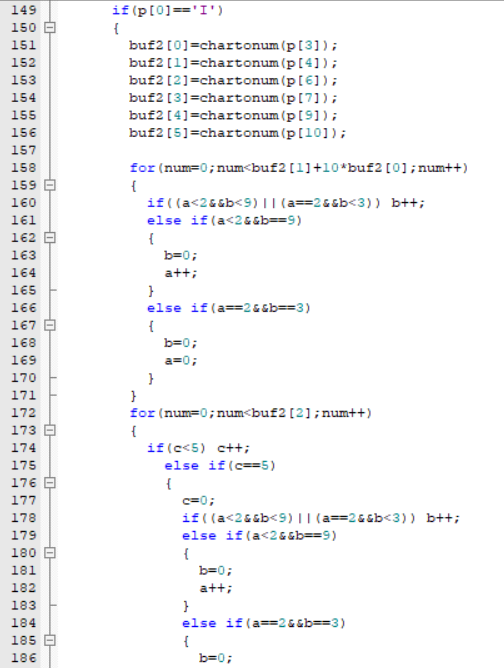
利用p数组接收字符，这和之前是一样的。由于设置的触发深度是14位，超过了指令可能出现的最大长度，因此不必担心分多次送达的问题。

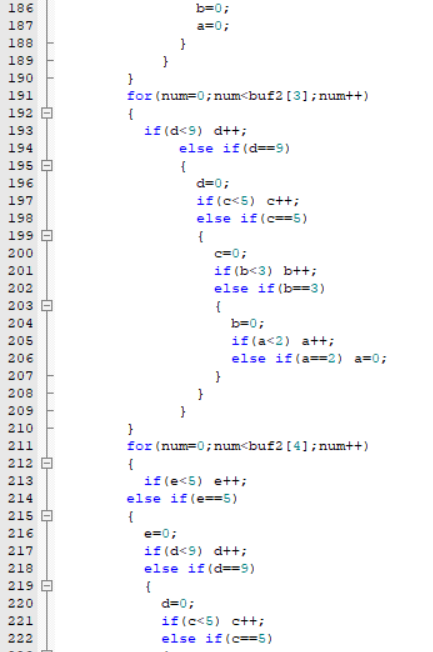


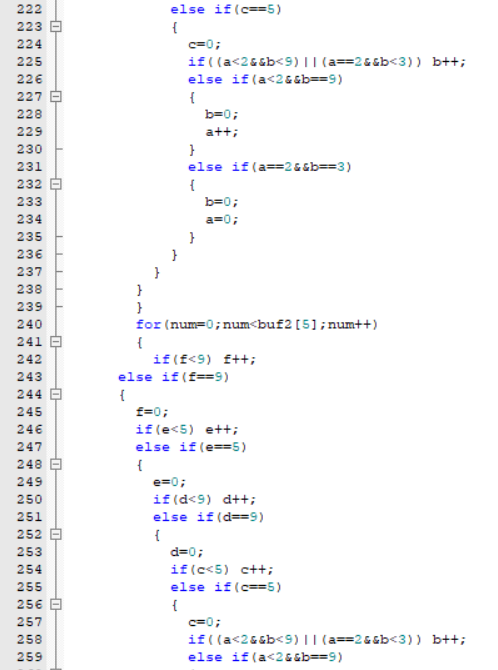


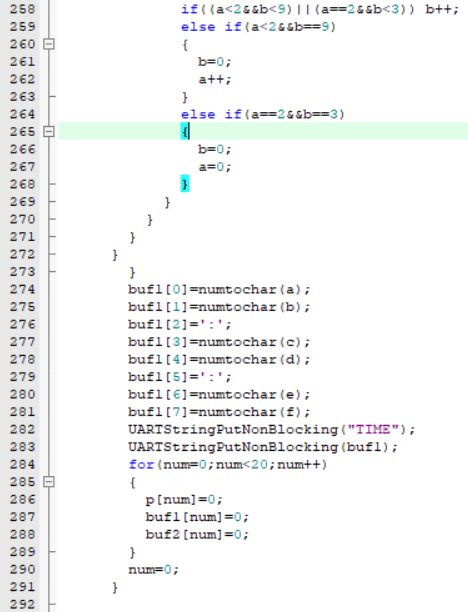


为了方便将数字从整型变量和字符型变量之间转化，我另外写了两个转换函数。利用p数组第一位就可以识别出SET命令。直接将相应的位转化为整形输入给a~f就可以改变时钟显示了。将要输回的时间放入char数组buf1内。输回时间前先输回“TIME”即可。最后清空用到的中间数组和变量，为下一次接收命令和处理做准备。

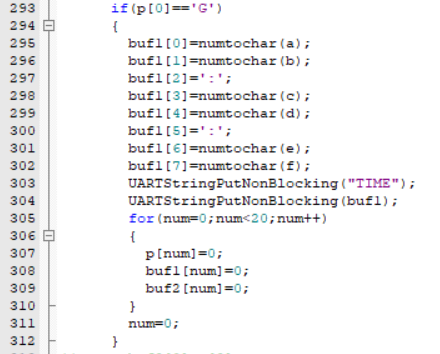








同样，判断p数组第一位就可以识别INC命令。输回和清空代码都是类似的，不再赘述。关键是如何实现时间加法。主要思路是：可以利用循环，某一位加多少，就对该位循环加1多少次。这样做的好处是可以大批使用先前的正常clock代码，减少开发工作量。buf2存放整型后的命令，作为循环次数。



同样，判断p数组第一位就可以识别GETTIME命令。回复时间只需要将a~f字符化后输出即可。

1. **讨论题**
   1. 实验3-1中，if (UARTCharsAvail(UART0\_BASE))此行程序的作用。如果没有此行，会导致什么问题？

答：3-1采用的是阻塞接收方法。题干中if语句用来判断接收FIFO内有没有字符，使只有在接收FIFO收到字符后才进行接收。如果没有该语句，由于阻塞接收的特性，若迟迟没有字符发来，程序将一直停在uart\_receive\_char = toupper(UARTCharGet(UART0\_BASE));这句中等待接收，如果主函数有其他功能的代码，那么这些代码的执行将受到影响。

* 1. 实验3-2中，采用中断方式进行UART接收时，为什么要使能“接收超时”中断条件？若设RxFIFO深度分别为4/8和7/8，则当PC端一次性发送17个字符时，至少会产生几次UART中断请求？

答：如果不设置超时中断，那当发来的字符数量不是预设深度的整数倍时，必定最后会剩下少量字符留在接收FIFO里却不能触发中断使cpu将其读走，导致信息丢失。如果设置了超时中断就可以应付这种情况。RxFIFO深度为4/8时一次可以接收8个字符，将产生3次中断；深度为7/8时一次可以接收14个字符，将产生2次中断。

* 1. 实验3-2中，void UART0\_Handler(void)为什么没有在主函数前声明？ 为什么UART和GPIO的中断处理程序需要读取并清除中断状态，而SYSTICK不需要？

答：因为该函数是中断服务函数。中断服务函数默认已经在程序里声明了，是弱函数，执行时会选择字节数量较多的执行，因此只需要直接重写一遍。Systick是单源中断，硬件会自动清除中断状态，而另两个是多源中断，需要手动清除中断状态。

* 1. 实验3-4中，执行语句IntPrioritySet(INT\_UART0,0x3); 后，Uart0的实际优先级为多少？

答：实际优先级为b000.00011，因为已经设置为全抢占。实际优先级是最高的抢占优先级下的第三号子优先级。