

1. 实验题目

- 板级验证
- 逻辑分析仪

2. 实验目的

- 初步了解TM4C123开发板的使用
- 学习TM4C123的基础编程
- 学习使用Keil仿真
- 学习TM4C123的debug和使用仿真进行debug

3. 实验内容

- 理解TM4C123中自带InputOutput project的代码内容，并修改程序改变按键对应的驱动灯颜色，做个前后对比，并作调试分析。
- 理解给出的实验代码，并修改输出位到portd.2后，与之前的portd.3截图做对比，然后修改程序，使得脉宽有所变化，也把修改前后截图做对比，并做程序修改分析对比。

4. 实验过程

- InputOutput板级验证实验
 - 代码理解

```
#include <stdint.h>
#include "inc/tm4c123gh6pm.h"
#define GPIO_LOCK_KEY      0x4C4F434B // Unlocks the GPIO_CR register
#define PF0                (*((volatile uint32_t *)0x40025004))
#define PF4                (*((volatile uint32_t *)0x40025040))
#define SWITCHES           (*((volatile uint32_t *)0x40025044))
#define SW1                0x10 // on the left side of the Launchpad board
#define SW2                0x01 // on the right side of the Launchpad board
#define SYSCTL_RCGC2_GPIOF 0x00000020 // port F Clock Gating Control
#define RED                0x02 // PF1
#define BLUE               0x04 // PF2
#define GREEN              0x08 // PF3
```

这部分主要是一些宏定义，包括一些端口的地址，开关和LED的值

```

void PortF_Init(void){ volatile uint32_t delay;
    SYSCTL_RCGCGPIO_R |= 0x00000020; // 1) activate clock for Port F
    delay = SYSCTL_RCGCGPIO_R;        // allow time for clock to start
    GPIO_PORTF_LOCK_R = 0x4C4F434B;   // 2) unlock GPIO Port F
    GPIO_PORTF_CR_R = 0x1F;           // allow changes to PF4-0
    // only PF0 needs to be unlocked, other bits can't be locked
    GPIO_PORTF_AMSEL_R = 0x00;        // 3) disable analog on PF
    GPIO_PORTF_PCTL_R = 0x00000000;    // 4) PCTL GPIO on PF4-0
    GPIO_PORTF_DIR_R = 0x0E;          // 5) PF4,PF0 in, PF3-1 out
    GPIO_PORTF_AFSEL_R = 0x00;        // 6) disable alt funct on PF7-0
    GPIO_PORTF_PUR_R = 0x11;          // enable pull-up on PF0 and PF4
    GPIO_PORTF_DEN_R = 0x1F;          // 7) enable digital I/O on PF4-0
}

```

此函数为Port F的注册函数，一共做了如下7步：

1. 激活端口F的时钟，并启动时钟
2. 解锁Port F，此处只需要使用到PF0-4，但是其中只有PF0需要解锁，其他并不需要。
3. 在整个PF关闭其模拟信号
4. 设置GPIO Port Control
5. 设置PF4(左边按钮)，PF0(右边按钮)为输入，PF3-1(依次为绿蓝红灯)为输出
6. 禁用PF所有端口的ALT,允许输入端PF0,PF4上拉
7. 设置PF0-4为数字I/O。

```

uint32_t PortF_Input(void){
    return (GPIO_PORTF_DATA_R&0x11); // read PF4,PF0 inputs
}

void PortF_Output(uint32_t data){ // write PF3-PF1 outputs
    GPIO_PORTF_DATA_R = data;
}

```

第一个为读入输入，此时&0x11是为了只保留第5位和第1位的输入。

第二个为输出函数，将数据放到PF端口上。

```

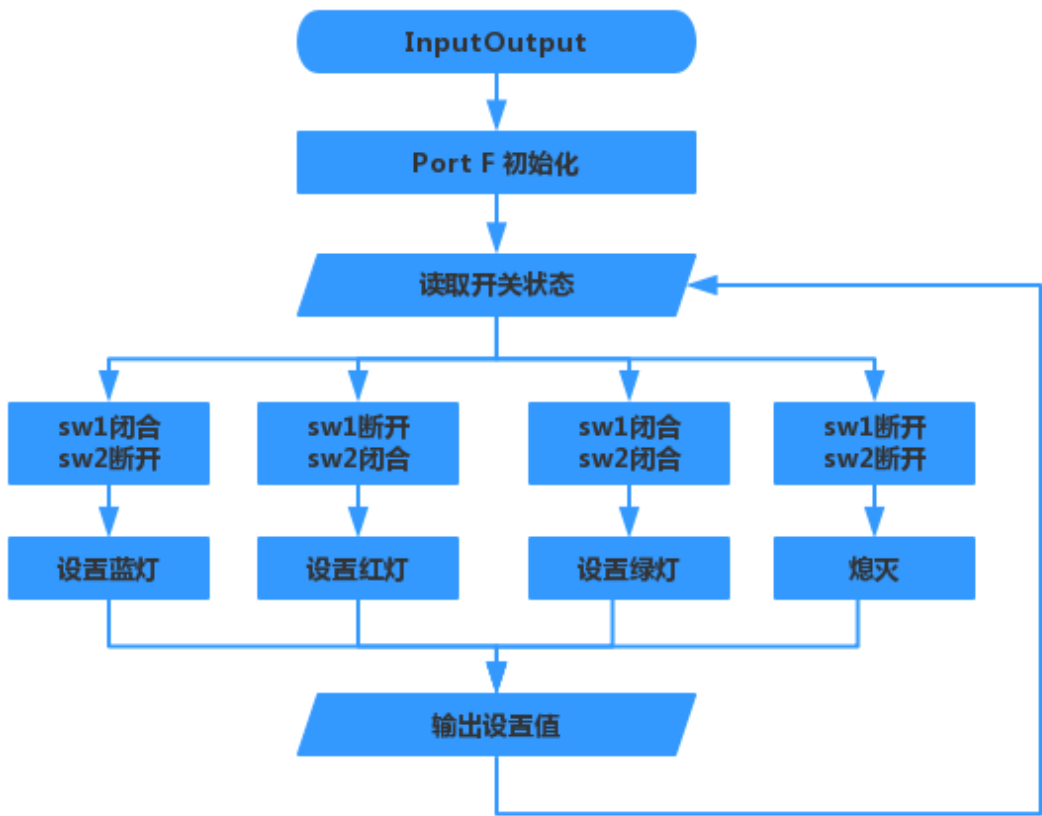
int main(void){ uint32_t status;
    PortF_Init();           // initialize PF0 and PF4 and make them inputs
                           // make PF3-1 out (PF3-1 built-in LEDs)
    while(1){
        status = PortF_Input();
        switch(status){ // switches are negative logic on PF0 and PF4
            case 0x01: PortF_Output(BLUE); break; // SW1 pressed
            case 0x10: PortF_Output(RED); break;  // SW2 pressed
            case 0x00: PortF_Output(GREEN); break; // both switches pressed
            case 0x11: PortF_Output(0); break;     // neither switch pressed
        }
    }
}

```

此处为主函数，根据开关按下状态修改输出的值。

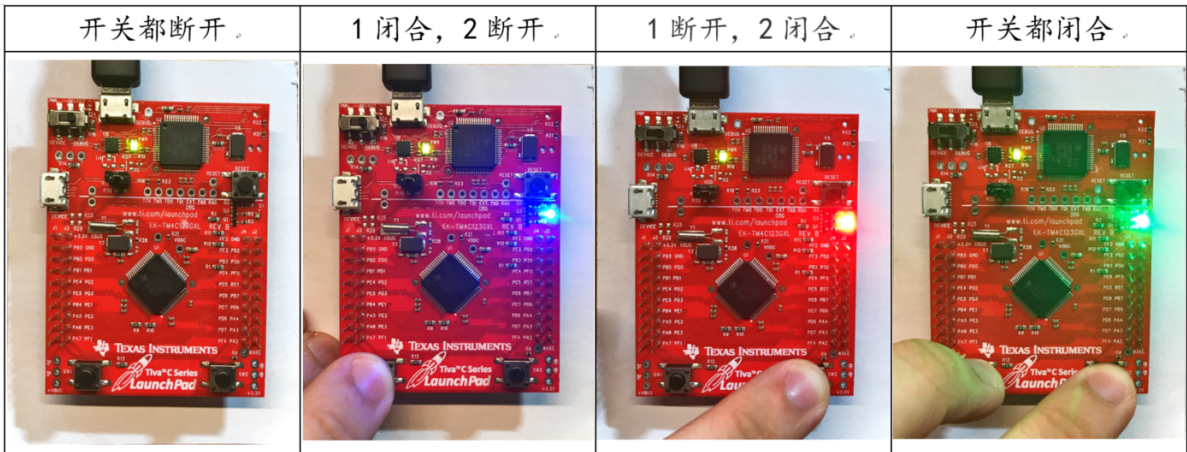
- 开关1闭合，开关2断开，为蓝灯。
- 开关1断开，开关2闭合，为红灯。
- 开关都闭合，为绿灯。
- 开关都断开，所有灯不亮。

代码流程：



○ 修改程序改变按键对应的驱动灯颜色

修改前：



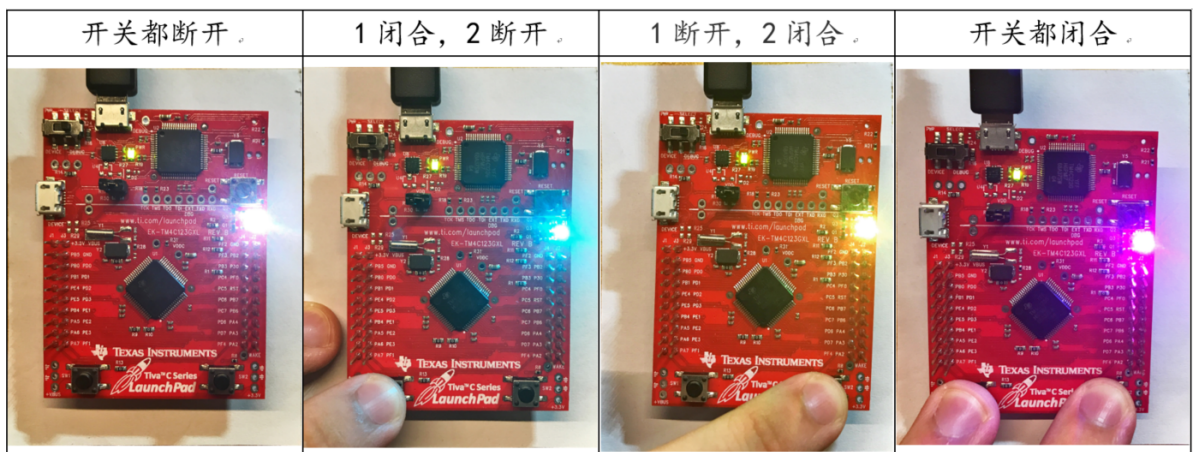
这个部分就比较简单了，直接修改一下main函数里面的传给输出的data，颜色见注释

```

int main(void){ uint32_t status;
PortF_Init();           // initialize PF0 and PF4 and make them inputs
                        // make PF3-1 out (PF3-1 built-in LEDs)

while(1){
    status = PortF_Input();
    switch(status){ // switches are negative logic on PF0 and PF4
        case 0x01: PortF_Output(GREEN+BLUE); break; // SW1 cyan
        case 0x10: PortF_Output(RED+GREEN); break;   // SW2 yellow
        case 0x00: PortF_Output(RED+BLUE); break;    // both magenta
        case 0x11: PortF_Output(BLUE+RED+GREEN); break; // neither white
    }
}
}

```



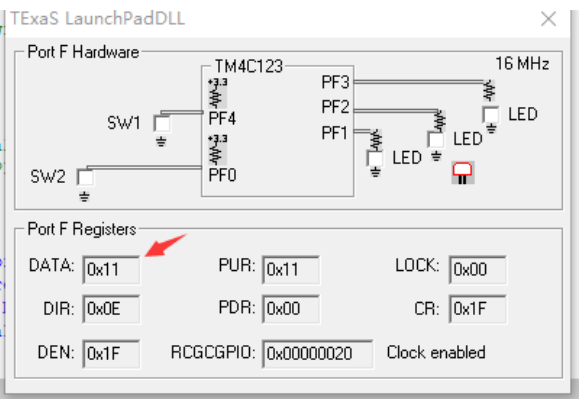
分析:

- 未修改前(此处就分析开关全部断开的情况其他时候类似,此处直接使用仿真比较好结合代码)

```

70 void PortF_Output(uint32_t data){ // w
71     GPIO_PORTF_DATA_R = data;
72 }
73
74 int main(void){ uint32_t status;
75     PortF_Init();           // initia
76                             // make P
77
78     while(1){
79         status = PortF_Input();
80         switch(status){
81             case 0x01: PortF_Output(BLUE); b
82             case 0x10: PortF_Output(RED); br
83             case 0x00: PortF_Output(GREEN);
84             case 0x11: PortF_Output(0); brea
85         }
86     }
87 }

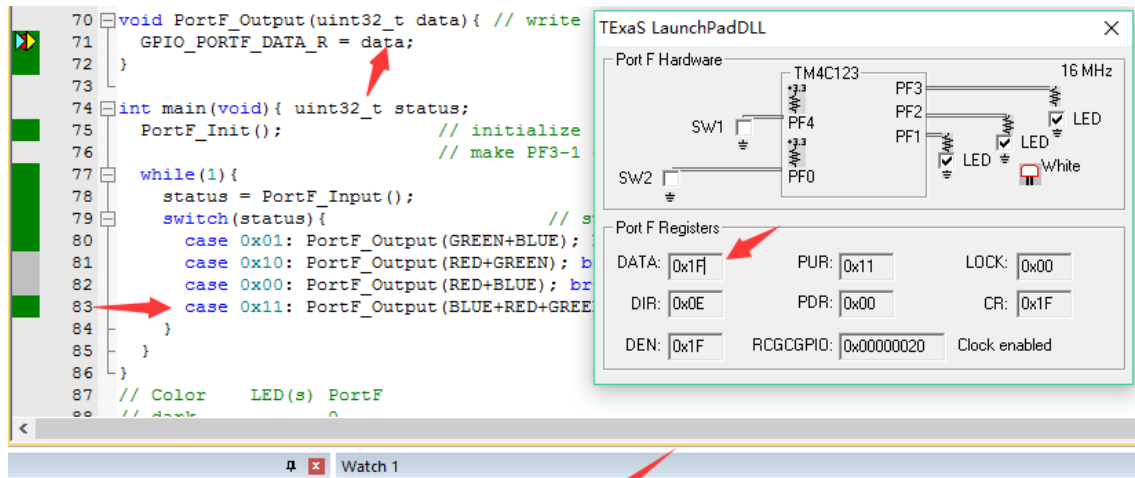
```



Name	Value	Type
data	0x00000000	unsigned int

此时因为开关全部没有按下所以input得到的值为0x11(开关没按下为1),此时传递给output值为0,所以PF1-3端口没有一个置1,那么就不会有灯亮,此时DATA的值为0x11(0001 0001)也验证了这一点

- 修改后(此处就分析开关全部断开的情况其他时候类似)



此时因为开关全部没有按下所以input得到的值为0x11(开关没按下为1),此时传递给output值为BLUE+RED+GREEN 也就是0x0E(0000 1110), 也就是PF1-3端口全部置1, 所以三盏灯全亮。此时我们从仿真中看到的DATA为0x1F(0001 1111)主要是算进去了两个按钮的值。

• NOTGate逻辑分析仪实验

◦ 代码理解

```

GPIO_PORTD_DATA_R EQU 0x400073FC
GPIO_PORTD_DIR_R EQU 0x40007400
GPIO_PORTD_LOCK_R EQU 0x4C4F434B
GPIO_PORTD_AFSEL_R EQU 0x40007420
GPIO_PORTD_DEN_R EQU 0x4000751C

SYSCTL_RCGCGPIO_R EQU 0x400FE108

AREA |.text|, CODE, READONLY, ALIGN=2
THUMB
EXPORT Start

GPIO_Init
; 1) activate clock for Port D

LDR R1, =SYSCTL_RCGCGPIO_R
LDR R0, [R1] ; R0 = [R1]
ORR R0, R0, #0x08 ; R0 = R0|0x08
STR R0, [R1] ; [R1] = R0
NOP
NOP
NOP
NOP ; allow time to finish activating
;unlock PD0,PD3
;LDR R1, =GPIO_PORTD_LOCK_R
;LDR R0, [R1] ; R0 = [R1]
;ORR R0, R0, #0x09 ; R0 = R0|0x08
;STR R0, [R1] ; [R1] = R0
; 3) set direction register
LDR R1, =GPIO_PORTD_DIR_R ; R1 = &GPIO_PORTD_DIR_R

```

```

LDR R0, [R1] ; R0 = [R1]
ORR R0, R0, #0x08 ; R0 = R0|0x08 (make PD3 output)
BIC R0, R0, #0x01 ; R0 = R0 & NOT(0x01) (make PD0 input)
STR R0, [R1] ; [R1] = R0
; 4) regular port function
LDR R1, =GPIO_PORTD_AFSEL_R ; R1 = &GPIO_PORTD_AFSEL_R
LDR R0, [R1] ; R0 = [R1]
BIC R0, R0, #0x09 ; R0 = R0&~0x09 (disable alt funct on PD3,PD0)
STR R0, [R1] ; [R1] = R0
; 5) enable digital port
LDR R1, =GPIO_PORTD_DEN_R ; R1 = &GPIO_PORTD_DEN_R
LDR R0, [R1] ; R0 = [R1]
ORR R0, R0, #0x09 ; R0 = R0|0x09 (enable digital I/O on PD3,PD0)
STR R0, [R1] ; [R1] = R0

BX LR

```

首先这部分是端口的定义和注册：

1. 激活端口D的时钟，并启动时钟
2. 设置PD3为输出，PD0为输入
3. 禁用PD3和PD0的ALT
4. 设置PD3和PD0为数字I/O。

```

Start
BL GPIO_Init
LDR R0, =GPIO_PORTD_DATA_R

loop
LDR R1, [R0]
AND R1, #0x01 ; Isolate PD0
EOR R1, #0x01 ; NOT state of PD0 read into R1
STR R1, [R0]
nop
nop
LSL R1, #3 ; SHIFT left negated state of PD0 read into R1
STR R1, [R0] ; Write to PortD DATA register to update LED on PD3
B loop ; unconditional branch to 'loop'

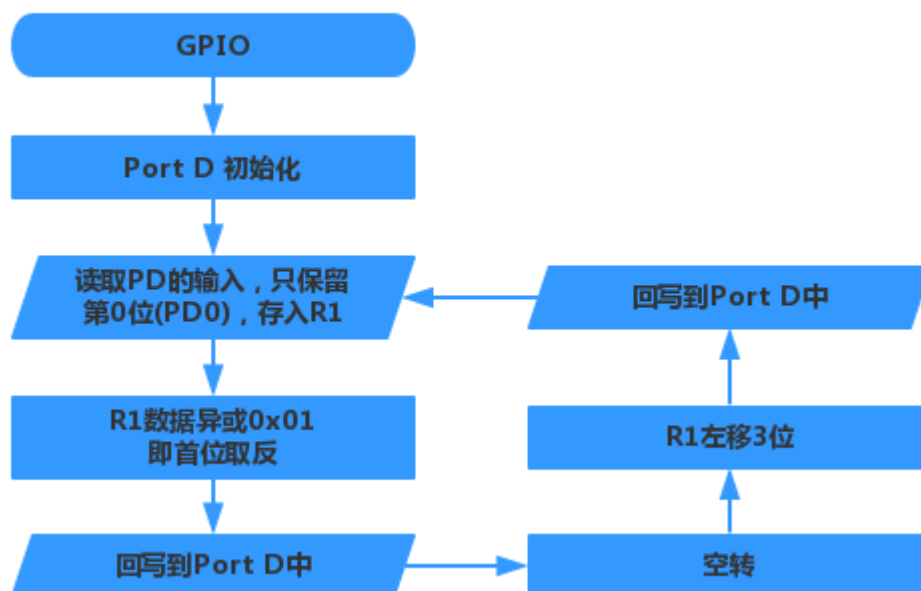
ALIGN ; make sure the end of this section is aligned
END ; end of file

```

这部分为函数主体，其调用之前的端口注册，然后将R0设置为Port D 的DATA地址。

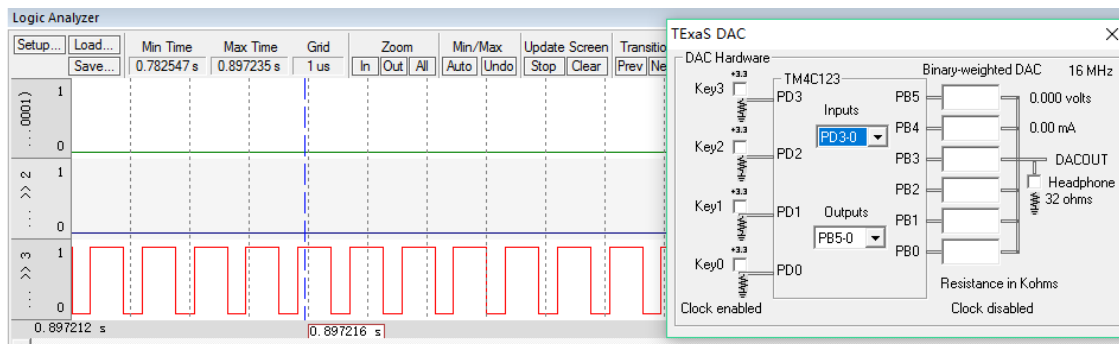
之后进入循环，每次从地址中读取数据的第1位,即PD0的值存入R1，然后R1与0x01异或(即首位取反)，再将R1数据写回Port D，之后使用nop空转一会，在将R1数据左移3位，之后再次写回。

函数流程如下：

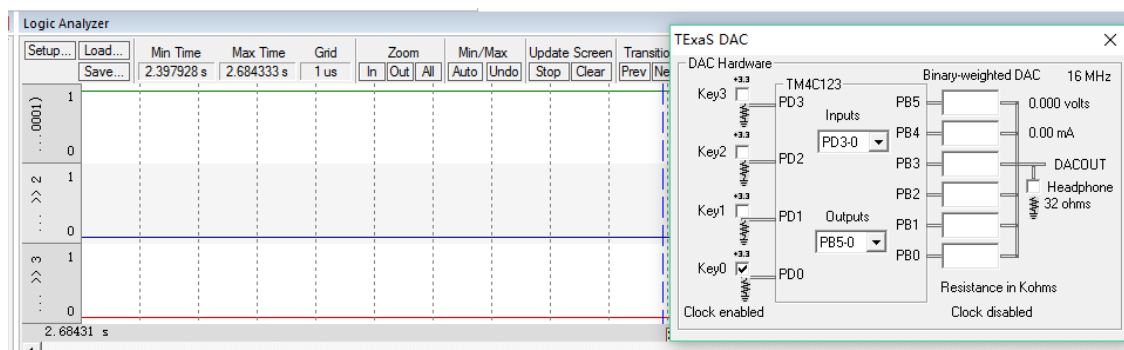


具体分析：两种情况

- PD0输入为0：此时无论如何从PD0读出来的数据都是0,也就是读出数据为(0000 0000),此时首位取反为(0000 0001),写回PortD中那么此时PD0应该为1,但是PD0没有定义为输入而不是为输出,所以PD0依然为0,也就是整个Port D 的DATA为(0000 0000),意味着唯一输出口PD3输出为0。之后空转一会。然后R1数据左移3位写入Port D 此时Port D的数据应该为(0000 1000),此时PD3输出为1,如此循环我们就能看到PD3输出方波。



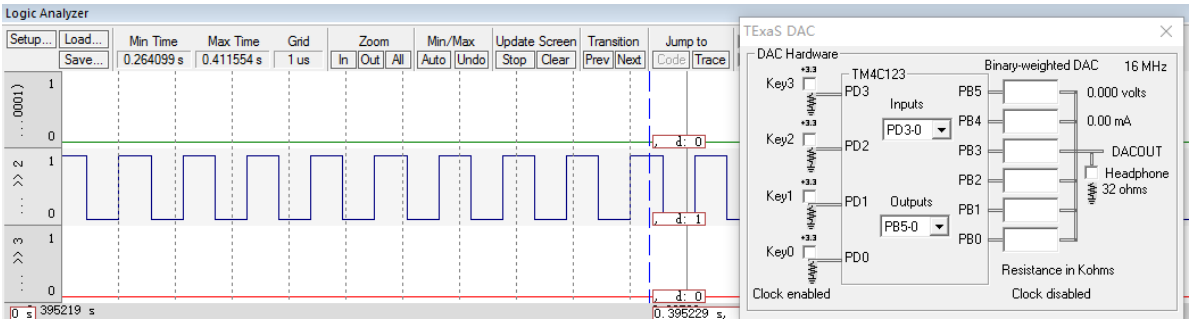
- PD0输入为1：此时无论如何从PD0读出来的数据都是1,也就是读出数据为(0000 0001),此时首位取反为(0000 0000),意味着PD3输出为0。空转一会之后左移3位，依然为(0000 0000),在写入PortD,此时PD3输出还是0,也就是此时PD3一直输出低电平。



- 修改程序使得输出口改为PD2

所以如果我们想要延长高电平时间只要在第一个STR R1,[R0]之前插入nop,如果想要延长低电平时间只要在第一个和第二个STR R1,[R0]之间插入nop即可。

此处展示延长低电平时间，使其变成一个近似50%占空比的方波，此处第一个和第二个STR R1,[R0]之间插入4个nop，效果如下：



5. 实验心得

这次实验主要还是熟悉一下开发板和IDE的使用，修改部分也是显而易见的，对于代码的理解C语言的代码是比较好懂，但是对于汇编语言ARM的还没学过，需要去看看指令意思，而且注释都有，不算什么困难的，修改时代码改动也不是很大，就是改改端口看懂了代码的话和C改起来差不多。