目录

嵌入式概念、组成

❑就是嵌入到对象体中的专用计算机系统

❑三要素：嵌入、专用、计算机

❑嵌入式系统的特点:由三要素引出

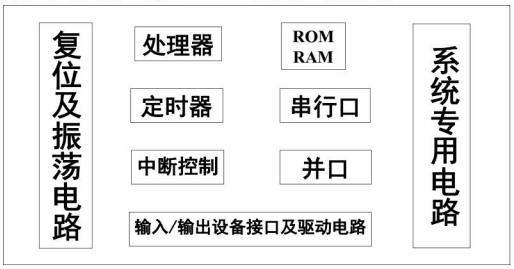
·嵌入性：嵌入到对象体系中，有对象环境要求

·专用性：软、硬件按对象要求裁减

·计算机：实现对象的智能化功能

嵌入式系统的组成

嵌入式系统一般有3个主要的组成部分：



- 硬件。处理器；存储器（ROM、RAM）；输入输出设备；其他部分辅助系统完成功能。
- 应用软件
- 实时操作系统（Real-Time Operating System，RTOS）。该系统用来管理应用软件，并提供一种机制，使得处理器分时地执行各个任务并完成一定的时限要求 。

嵌入式系统的分类

❑按表现形式分（硬件范畴）：

▲芯片级嵌入（含程序或算法的处理器）

▲模块级嵌入（系统中的某个核心模块）

▲系统级嵌入

❑按实时性要求分（软件范畴）：

▲非实时系统（PDA）

▲软实时系统（消费类产品）

▲硬实时系统（工业和军工系统）

1.4.2 嵌入式系统的开发特点

- 1．采用宿主机/目标机方式
- 2．为了保证稳定性和实时性，选用RTOS开发平台
- 3．生成代码需要固态化存储
- 4．软件代码具有高质量、高可靠性

1.4.3 嵌入式系统的开发流程

嵌入式系统的应用开发一般由5个阶段构成：

- 需求分析、
- 体系结构设计、
- 硬件/软件设计、
- 系统集成
- 代码固化。

各个阶段之间往往要求不断地反复和修改，直到完成最终完成设计目标。

1．硬件层

由嵌入式微处理器、外围电路和外设组成。

外围电路有：电源电路、复位电路、调试接口和存储器电路，就构成一个嵌入式核心控制模块。

操作系统和应用程序都可以固化在ROM或者Flash中。

有些嵌入式系统还包括：

LCD、键盘、USB接口，以及其他一些功能的扩展电路。

2．中间层

- 硬件层与软件层之间为中间层，也称为BSP（Board Support Package，板级支持包）。
- 作用：将系统软件与底层硬件部分隔离，使得系统的底层设备驱动程序与硬件无关；
- 功能：具有相关硬件的初始化、数据的输入/输出操作和硬件设备的配置等功能。BSP是主板硬件环境和操作系统的中间接口，是软件平台中具有硬件依赖性的那一部分，主要目的是为了支持操作系统，使之能够更好地运行于硬件主板上。

3．软件层

主要是操作系统，有的还包括文件系统、图形用户接口和网络系统等。操作系统是一个标准的内核，将中断、I/O、定时器等资源都封装起来，以方便用户使用。

4．功能层

由基于操作系统开发的应用程序组成，用来完成对被控对象的控制功能。功能层是面向被控对象和用户的，为了方便用户操作，往往需要具有友好的人机界面。

含义解释

FLASH：

FLASH 存储器又称闪存，它结合了 ROM 和 RAM 的长处，不仅具备电子可擦除可编程（EEPROM）的性能，还不会断电丢失数据同时可以快速读取数据（NVRAM 的优势），U 盘和 MP3 里用的就是这种存储器。在过去的 20 年里，嵌入式系统一直使用 ROM（EPROM）作为它们的存储设备，然而近年来 Flash 全面代替了 ROM（EPROM）在嵌入式系统中的地位，用作存储 Boot loader 以及操作系统或者程序代码或者直接当硬盘使用（U 盘）。目前 Flash 主要有两种 NOR Flash 和 NADN Flash

NOR Flash：

读取和常见的 SDRAM 的读取一样，用户可以直接运行装载在 NOR FLASH 里面的代码，这样可以减少 SRAM 的容量从而节约了成本

NAND Flash：

没有采取内存的随机读取技术，采用这种技术的 Flash 比较廉价。用户不能直接运 NAND Flash 上的代码，因此好多使用 NAND Flash 的开发板除了使用 NAND Flash 以外，还作上了一块小的 NOR Flash 来运行启动代码

性能比较：

NOR 的读速度比 NAND 稍快一些。

NAND 的写入速度比 NOR 快很多。

NAND 的 4ms 擦除速度远比 NOR 的 5s 快。

大多数写入操作需要先进行擦除操作。

NAND 的擦除单元更小，相应的擦除电路更少。

操作方式：

NOR Flash：

应用程序对 NOR Flash 芯片操作以字为基本单位，为方便管理通常分为 128KB 或 64KB 的逻辑块，块内分为不同扇区，读写时指定逻辑块号与片偏移

NAND Flash：

应用程序对 NAND Flash 芯片操作以块为基本单位，块大小 8KB，块内分页，一般大小 512 字节，若要修改一个字节需重写整个块

微处理器广泛的替代功能（即其他功能设置）：

UART（Universal asynchronous receiver/transmitter，通用异步收发传输器）：

是一种异步收发传输器，将要传输的资料在串行通信与并行通信之间加以转换，异步的并允许双向通信

SSI（Synchronous serial interface，串行外设接口）：

是各类 DSP 处理器中的常见接口，用于中等速度的 I/O 设备

I2C（Inter-integrated circuit，集成电路总线）：

这种总线类型是一种简单、双向、二线制、同步串行总线，主要是用来连接整体电路，用于低速外围设备

Timer（Periodic interrupts, input capture, and output compare）：input capture 和 output compare 用来创建周期性中断并且测量周期、脉冲宽度、相位和频率

PWM（Pulse width modulation，脉冲宽度调制）：

利用微处理器的数字输出来对模拟电路进行控制的一种非常有效的技术，PWM 输出用于将可变功率应用于电机接口，在典型电机控制中，输入捕捉来测量转速

ADC（Analog to digital converter, measure analog signals）：

模数转换器，测量模拟信号

用来测量模拟信号的振幅，在数据采集系统中起重要作用

Analog Comparator（Compare two analog signals）：

模拟比较器用来对比两个模拟输入，并根据更大的模拟输入来产生一个数字化输出

QEI（Quadrature encoder interface，正交编码器接口）：

用来作为与无刷直流电动机的接口

USB（Universal serial bus，通用串行总线）：

是一种高速串行通信通道

Ethernet（High-speed network）：

以太网端口可以将微控制器与因特网或一个局域网桥接起来

CAN（Controller area network，一种串行通信协议）：

用来创建一个在微控制器和机器之间的高速通信通道，常用于分布式控制系统的应用

并行端口 I/O

并行口可以作为输入口也可以作为输出口，其中的数据端口、控制端口都可以作为数字输出端口

GPIO

杂项：

引脚可配置为数字 I / O，模拟输入，定时器 I / O 或串行 I / O

一个允许软件读取外部数字信号的的输入端口只能读，输出端口可以像正常存储器一样参与读取和写入周期

A 0b000001
B 0b000010
C 0b000100
D 0b001000
E 0b010000
F 0b100000

GPIO 配置流程：

1、SYSCTL_RCGC2_R 寄存器中**激活其时钟**

SYSCTL_RCGC2_R |= 0x00000020; // 1) activate clock for Port F

delay = SYSCTL_RCGC2_R; // allow time for clock to start

2、通过向 LOCK 寄存器写入一个特殊值，然后设置 CR 寄存器中的位，来**解锁端口**

GPIO_PORTF_LOCK_R = 0x4C4F434B; // 2) unlock GPIO Port F

GPIO_PORTF_CR_R = 0x1F; // allow changes to PF4-0 0b11111,解锁对应端口

3、清除 AMSEL 寄存器中的位来**禁用模拟**功能

GPIO_PORTF_AMSEL_R = 0x00; // 3) disable analog on PF

4、**清除 PCTL 寄存器中的位**来选择 GPIO 功能 指定该引脚的替代功能（0 表示常规 I/O 端口）

GPIO_PORTF_PCTL_R = 0x00000000; // 4) PCTL GPIO on PF4-0

5、清除或设置 DIR 寄存器中的位来指定引脚是输入还是输出，**设定方向**

GPIO_PORTF_DIR_R = 0x0E; // 5) PF4,PF0 in, PF3-1 out 0 输入; 1 输出

6、**清除 AFSEL 寄存器中的相应位** （alternate function select register: 是否启用备用功能）

GPIO_PORTF_AFSEL_R = 0x00; // 6) disable alt funct on PF7-0

7、通过将 1 写入 DEN 寄存器来使能相应的 I / O 引脚，**启用数字端口**

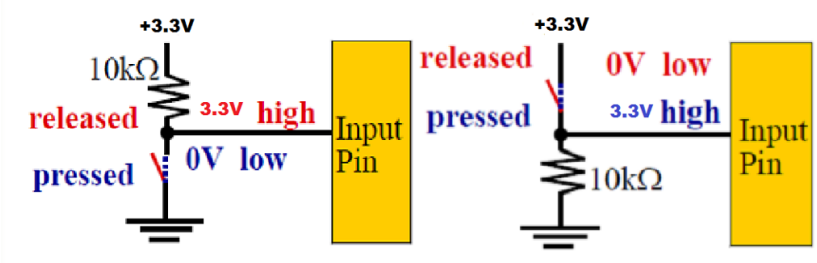
GPIO_PORTF_DEN_R = 0x1F; // 7) enable digital I/O on PF4-0

8、在 PUR 寄存器中为两个开关输入设置了一个内部上拉电阻

GPIO_PORTF_PUR_R = 0x11; // enable pull-up on PF0 and PF4 in=1，为1；in=0,为0；如果是上拉，则正好相反。

```
void PortF_Init(void){ volatile unsigned long delay;
    SYSCTL_RCGC2_R |= 0x00000020; // 1) activate clock for Port F
    delay = SYSCTL_RCGC2_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
}
```

开关设计说明：



negative – pressed = ‘0’ positive – pressed = ‘1’

SysTick

定义：SysTick 是一个简单的计数器，我们可以用它来创建时间延迟并产生周期性中断

硬件实现：SysTick 的基础是一个以总线时钟频率运行的 24 位递减计数器，**总线频率一般为80MHz**

Address	31-24	23-17	16	15-3	2	1	0	Name
$E000E010	0	0	COUNT	0	CLK_SRC	INTEN	ENABLE	NVIC_ST_CTRL_R
$E000E014	0	24-bit RELOAD value						NVIC_ST_RELOAD_R
$E000E018	0	24-bit CURRENT value of SysTick counter						NVIC_ST_CURRENT_R

初始化步骤：

1、清除 ENABLE 来停止计数器

2、指定 RELOAD 值

3、通过 NVIC_ST_CURRENT_R 清除计数器

4、设置 CLK_SRC = 1 并通过指定中断操作（INTEN 在 NVIC_ST_CTRL_R 中）

代码：

```
#define NVIC_ST_CTRL_R(*((volatile uint32_t *)0xE000E010))
#define NVIC_ST_RELOAD_R(*((volatile uint32_t *)0xE000E014))
#define NVIC_ST_CURRENT_R(*((volatile uint32_t *)0xE000E018))
void SysTick_Init(void){
    NVIC_ST_CTRL_R = 0; // 1) disable SysTick during setup
    NVIC_ST_RELOAD_R = 0x00FFFFFF; // 2) maximum reload value
    NVIC_ST_CURRENT_R = 0; // 3) any write to CURRENT clears it
    NVIC_ST_CTRL_R = 0x00000005; // 4) enable SysTick with core clock
}
// The delay parameter is in units of the 80 MHz core clock(12.5 ns)
void SysTick_Wait(uint32_t delay){
    NVIC_ST_RELOAD_R = delay-1; // number of counts
    NVIC_ST_CURRENT_R = 0; // any value written to CURRENT clears
    while((NVIC_ST_CTRL_R&0x00010000)==0){ // wait for flag
    }
}
// Call this routine to wait for delay*10ms
void SysTick_Wait10ms(uint32_t delay){
    unsigned long i;
    for(i=0; i<delay; i++){
        SysTick_Wait(800000); // wait 10ms
    }
}
```

PLL

定义：

锁相环（PLL: Phase-locked loops）是一种利用反馈（Feedback）控制原理实现的频率及相位的同步技术，其作用是将电路输出的时钟与其外部的参考时钟保持同步。当参考时钟的频率或相位发生改变时，锁相环会检测到这种 变化，并且通过其内部的反馈系统来调节输出频率，直到两者重新同步，这种同步又称为“锁相”（Phase-locked）

PLL 代码：

PLL.c:

```
void PLL_Init(void){
    // 0) Use RCC2
    SYSCTL_RCC2_R |= 0x80000000; // USERCC2
    // 1) bypass PLL while initializing
    SYSCTL_RCC2_R |= 0x00000800; // BYPASS2, PLL bypass
    // 2) select the crystal value and oscillator source
    SYSCTL_RCC_R = (SYSCTL_RCC_R &~0x000007C0) // clear XTAL field, bits 10-6
    + 0x00000540; // 10101, configure for 16 MHz crystal
    SYSCTL_RCC2_R &= ~0x00000070; // configure for main oscillator source
    // 3) activate PLL by clearing PWRDN
    SYSCTL_RCC2_R &= ~0x00002000;
    // 4) set the desired system divider
    SYSCTL_RCC2_R |= 0x40000000; // use 400 MHz PLL
    SYSCTL_RCC2_R = (SYSCTL_RCC2_R&~ 0x1FC00000) // clear system
    // 5) wait for the PLL to lock by polling PLLLRIS
    while((SYSCTL_RIS_R&0x00000040)==0){}; // wait for PLLRIS bit
    // 6) enable use of PLL by clearing BYPASS
    SYSCTL_RCC2_R &= ~0x00000800;
}
```

main.c:

```
#include "PLL.c"
#define GPIO_PORTF2             (*((volatile unsigned long *)0x40025010))
#define GPIO_PORTF_DIR_R        (*((volatile unsigned long *)0x40025400))
#define GPIO_PORTF_AFSEL_R      (*((volatile unsigned long *)0x40025420))
#define GPIO_PORTF_DEN_R        (*((volatile unsigned long *)0x4002551C))
#define GPIO_PORTF_AMSEL_R      (*((volatile unsigned long *)0x40025528))
#define GPIO_PORTF_PCTL_R       (*((volatile unsigned long *)0x4002552C))
#define SYSCTL_RCGC2_R          (*((volatile unsigned long *)0x400FE108))
#define SYSCTL_RCGC2_GPIOF      0x00000020

#ifdef __TI_COMPILER_VERSION__
void Delay(unsigned long ulCount){
    __asm ( "    subs    r0, #1\n"
            "    bne     Delay\n"
            "    bx      lr\n");
}
#else
    __asm void
    Delay(unsigned long ulCount)
    {
    subs    r0, #1
    bne     Delay
    bx      lr
    }
#endif
int main(void){  volatile unsigned long delay;
  PLL_Init();
  SYSCTL_RCGC2_R |= SYSCTL_RCGC2_GPIOF;
  delay = SYSCTL_RCGC2_R;
  GPIO_PORTF_DIR_R |= 0x04;
  GPIO_PORTF_AFSEL_R &= ~0x04;
  GPIO_PORTF_DEN_R |= 0x04;
  GPIO_PORTF_PCTL_R = (GPIO_PORTF_PCTL_R&0xFFF0FFFF)+0x00000000;
  GPIO_PORTF_AMSEL_R = 0;
  while(1){
    GPIO_PORTF2 = 0x04;
    Delay(13333333);
    GPIO_PORTF2 = 0x00;
    Delay(13333333);
  }
}
```

有限状态机

定义：

有限状态机（FSM）是一种抽象概念，它像算法一样描述问题的解决方案。与提供一系列需要遵循以实现问题解决方案的步骤不同的算法不同，FSM 将系统（该解决方案是系统行为的实现）描述为一种机器，该机器根据输入和生成而改变状态而产出适当输出

代码：

```
struct State{
    unsigned char out;      //output for the state
    unsigned short wait;    //time to wait when in this state
    unsigned char next[2];  //next state array
};

typedef const struct State StateType;

//shortcuts to refer to the various states in the FSM array
#define Even 0
#define Odd 1

//the data structure that captures the FSM state transition graph
StateType Fsm[2] = {
    {0,100,{Even,Odd}},
    {1,100,{Odd,Even}}
};

unsigned char cState;       //current state is even/odd

int main(void){
    unsigned char input;

    PortF_Init();
    SysTick_Init();

    cState = Even;
    while(1){
        //output based on current state
        GPIO_PORTF_DATA_R = Fsm[cState].out<<2;
        //wait for time relevant to state
        SysTick_Wait10ms(Fsm[cState].wait);
        //get input
        input = (~GPIO_PORTF_DATA_R & 0x10)>>4;
        //change state based on input and current state
        cState = Fsm[cState].next[input];
    }
}
```

UART

延迟时间：I / O 设备指示服务的时间与启动服务的时间之间的时间

吞吐量/带宽：系统可以处理的最大数据流量（bytes/second）

波特率：每单位时间的总位数。波特率= 1 / bit-time bit/s

带宽：每单位时间的数据。带宽=(data-bits/frame-bits)*波特率 （每单位时间帧的数量）

比特率=(总线时钟频率)/(16*分频器(divider))

如:19.2 kb/s,总线时钟是 8 MHz,比特率=8 MHz /(16*19.2 k)= 26.04167 = 11010.0000112

串行通信：

- 发送数据（TxD），接收数据（RxD）和 Signal Ground（SG）实现双工通信链路
- 两个通信设备必须以相同的比特率运行
- 最低有效位先发送

Tx操作
- 写入UART0_DR_R的数据
  - •通过16个元素的FIFO
  - •允许处理器和UART之间少量的数据速率匹配
- 移位时钟从16x时钟生成
  - •允许调整Tx和Rx时钟的差异

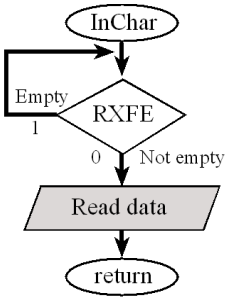
Rx操作
- 当数据可用时RXFE为0
- FIFO满时RXFF为1
- FIFO条目有四个控制位
  - •当Tx信号保持低电平超过一帧（中断）时置位
  - •当FIFO已满并且新帧已到达时，OE置位
  - •如果帧奇偶校验错误，PE设置
  - •如果停止位定时错误，FE设置

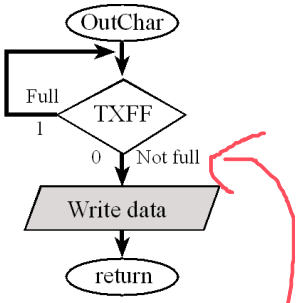
代码：

UART 初始化：

```
// Assumes a 50 MHz bus clock, creates 115200 baud rate
void UART_Init(void){
    SYSCTL_RCGCUART_R |= 0x0001; // activate UART0
    SYSCTL_RCGCGPIO_R |= 0x0001; // activate port A
    UART0_CTL_R &= ~0x0001; // disable UART
    UART0_IBRD_R = 27;
    // IBRD=int(50000000/(16*115,200)) = int(27.1267)
    UART0_FBRD_R = 8;
    // FBRD = round(0.1267 * 64) = 8
    UART0_LCRH_R = 0x0070; // 8-bit length, enable FIFO
    UART0_CTL_R = 0x0301; // enable RXE, TXE and UART
    GPIO_PORTA_AFSEL_R |= 0x03; // alt funct on PA1-0
    GPIO_PORTA_PCTL_R =
    (GPIO_PORTA_PCTL_R&0xFFFFFF00)+0x00000011;
    GPIO_PORTA_DEN_R |= 0x03; // digital I/O on PA1-0
    GPIO_PORTA_AMSEL_R &= ~0x03; // No analog on PA1-0
}
```

UART0_CTL_R=0x0001;
// just enable UART





UART Busy-Wait Send/Recv：

```
// Wait for new input,
// then return ASCII code
uint8_t UART_InChar(void) {
    while((UART0_FR_R&0x0010) != 0);
    // wait until RXFE is 0
    return((uint8_t)(UART0_DR_R&0xFF));
}
// Wait for buffer to be not full,
// then output
void UART_OutChar(uint8_t data) {
    while((UART0_FR_R&0x0020) != 0);
    // wait until TXFF is 0
    UART0_DR_R = data;
}
```

传送字符代码：

String 到 long 转换：

```
// Convert ASCII string to
// unsigned 32-bit decimal
// string is null-terminated
unsigned long Str2UDec(unsigned char string[]){
    unsigned long i = 0; // index
    unsigned long n = 0; // number
    while(string[i] != 0){
        n = 10*n +(string[i]-0x30);
        i++;
    }
}
```

输入十进制数：

```
#define CR 0x0D
    // Accept ASCII input in unsigned decimal format, up to 4294967295
    // If n> 4294967295, it will truncate without reporting the error
unsigned long UART_InUDec(void){
    unsigned long n=0; // this will be the return value
    unsigned char character; // this is the input ASCII typed
    while(1){
        character = UART_InChar(); // accepts input
        UART_OutChar(character); // echo this character
        if((character < '0') || (character > '9')){ // check for non-number
            return n; // quit if not a number
        }
        n = 10*n+(character-0x30); // overflows if above 4294967295
    }
}
```

long 到 String 转换：

```
unsigned char Data[4]; // 4-byte empty buffer
// n is the input 0 to 999
void UDec2Str(unsigned short n){
    Data[0] = n/100 + 0x30; // hundreds digit
    n = n%100; // n is now between 0 and 99
    Data[1] = n/10 + 0x30; // tens digit
    n = n%10; // n is now between 0 and 9
    Data[2] = n + 0x30; // ones digit
    Data[3] = 0; // null termination
}
// n is the input 0 to 999
void UART_OutUDec3(unsigned short n){
    UART_OutChar(0x30+n/100); // hundreds digit
    n = n%100; // 0 to 99
    UART_OutChar(0x30+n/10); // tens digit
    n = n%10; // 0 to 9
    UART_OutChar(0x30+n); // ones digit
}
```

输出电压值到外部设备：

```
//----------------------UART_OutUDec-----------------------
// Output a voltage to the UART
// Input: n is an integer from 0 to 999 meaning 0.00 to 9.99V
// Output: none
// Fixed format: for example n=8 displayed as "0.08V"
void OutVolt(unsigned long n){ // each integer means 0.01V
    UART_OutChar(0x30+n/100); // digit to the left of the decimal
    n = n%100; // 0 to 99
    UART_OutChar('.'); // decimal point
    UART_OutChar(0x30+n/10); // tenths digit
    n = n%10; // 0 to 9
    UART_OutChar(0x30+n); // hundredths digit
    UART_OutChar('V');
} // units
```

输出 long 到外部设备：

递归写法：

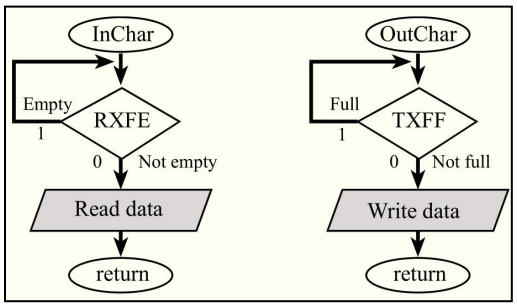
```
//---------------------UART_OutUDec-----------------------
// Output a 32-bit number in unsigned decimal format
// Input: 32-bit number to be transferred
// Output: none
// Variable format 1-10 digits with no space before or after
void UART_OutUDec(unsigned long n){
    if(n >= 10){
    UART_OutUDec(n/10);
    n = n%10;
    }
    UART_OutChar(n+'0'); /* n is between 0 and 9 */
}
```

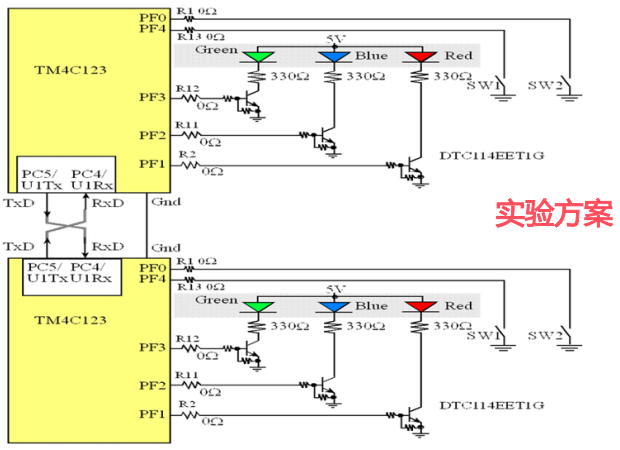
循环写法

```
// iterative method
void OutUDec(unsigned long n){
unsigned cnt=0;
unsigned char buffer[11];
  do{
    buffer[cnt] = n%10;// digit
    n = n/10;
    cnt++;
  }while(n);// repeat until n==0
  for(; cnt; cnt--){
    OutChar(buffer[cnt-1]+'0');
  }
}
```

UART 灯代码：

```
// red, yellow, green, light blue, blue, purple, white, dark
const long ColorWheel[8] = {0x02,0x0A,0x08,0x0C,0x04,0x06,0x0E,0x00};
int main(void){ unsigned long SW1,SW2;
    long prevSW1 = 0; // previous value of SW1
    long prevSW2 = 0; // previous value of SW2
    unsigned char inColor; // color value from other microcontroller
    unsigned char color = 0; // this microcontroller's color value
    PLL_Init(); // set system clock to 80 MHz
    SysTick_Init(); // initialize SysTick
    UART_Init(); // initialize UART
    PortF_Init(); // initialize buttons and LEDs on Port F
    while(1){
        SW1 = GPIO_PORTF_DATA_R&0x10; // Read SW1
    if((SW1 == 0) && prevSW1){ // falling of SW1?
        color = (color+1)&0x07; // step to next color
    }
    prevSW1 = SW1; // current value of SW1
    SW2 = GPIO_PORTF_DATA_R&0x01; // Read SW2
    if((SW2 == 0) && prevSW2){ // falling of SW2?
        UART_OutChar(color+0x30); // send color as '0' - '7'
    }
    prevSW2 = SW2; // current value of SW2
    inColor = UART_InCharNonBlocking();
    if(inColor){ // new data have come in from the UART??
        color = inColor&0x07; // update this computer's color
    }
    GPIO_PORTF_DATA_R = ColorWheel[color]; // update LEDs
    SysTick_Wait10ms(2); // debounce switch
    }
}
```





实验方案

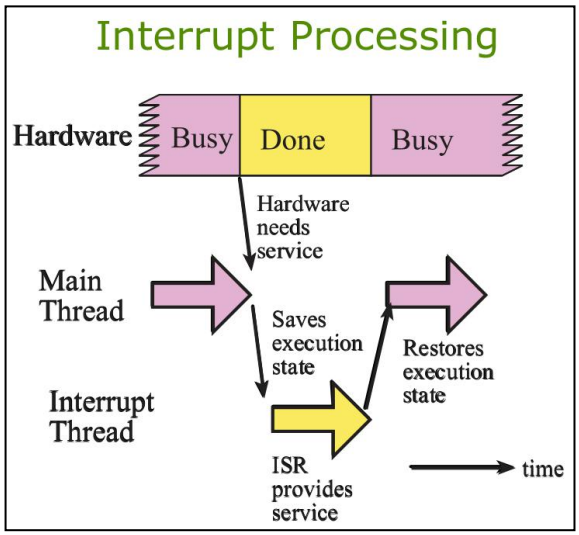
中断

定义：中断是软件执行的自动传送，以响应与当前软件执行异步的硬件事件（触发器）

产生来源/原因： 1)外部 I / O 设备（如键盘或打印机）

2)内部事件（如操作码故障或定期计时器）。

条件：当硬件需要或可以提供服务时（忙状态到完成状态） Busy→Done



中断步骤：

1、主程序的执行被暂停

1)当前指令完成

2)暂停执行并将 8 个寄存器(R0-R3,R12,LR,PC,PSR)push 进 stack

3)LR 设置为 0xFFFFFFF9(表示中断返回)

4)IPSR 设置为中断号

5)将 PC 设置为 ISR 地址(中断向量)

PSR：Program Status Register，单独访问或一次全部访问

IPSR：Interrupt Program Status Register

2、中断服务程序(ISR)被执行

清除请求中断的标志，执行必要的操作，使用全局变量进行通信

3、ISR 执行 BX LR 时，主程序恢复

从堆栈中 pop 出 8 个寄存器

中断服务程序 ISR：

定义：硬件请求中断时执行的软件模块

ISR 提供必要的服务后，它将执行 BX LR；ISR 软件必须在退出之前平衡堆栈；局部变量，共享全局内存变量，私有全局变量；ISR 应尽可能快地执行

中断向量表：

所有中断系统都必须具备计算机确定信号源的能力。向量化的中断系统为每个设备使用不同的连接，以便计算机可以自动解决问题。您可以识别矢量系统，因为每个器件都有一个单独的中断向量地址。使用轮询中断系统时，中断软件必须轮询每个设备，寻找请求中断的设备。TM4C 微控制器上的大多数中断都是向量化的，但有一些触发器共享相同的向量。对于这些中断，ISR 必须查询哪个触发器导致中断。例如，一个 GPIO 端口上的所有输入引脚都可以触发中断，但触发标志共享相同的向量。因此，如果一个 GPIO 端口上的多个引脚被布防，共享 ISR 必须进行轮询以确定哪一个请求的服务。

Vector address	Number	IRQ	ISR name in **Startup.s**	NVIC	Priority bits
0x00000038	14	-2	PendSV_Handler	NVIC_SYS_PRI3_R	23 – 21
0x0000003C	15	-1	SysTick_Handler	NVIC_SYS_PRI3_R	31 – 29
0x00000040	16	0	GPIOPortA_Handler	NVIC_PRI0_R	7 – 5
0x00000044	17	1	GPIOPortB_Handler	NVIC_PRI0_R	15 – 13
0x00000048	18	2	GPIOPortC_Handler	NVIC_PRI0_R	23 – 21
0x0000004C	19	3	GPIOPortD_Handler	NVIC_PRI0_R	31 – 29
0x00000050	20	4	GPIOPortE_Handler	NVIC_PRI1_R	7 – 5
0x00000054	21	5	UART0_Handler	NVIC_PRI1_R	15 – 13
0x00000058	22	6	UART1_Handler	NVIC_PRI1_R	23 – 21
0x0000005C	23	7	SSI0_Handler	NVIC_PRI1_R	31 – 29
0x00000060	24	8	I2C0_Handler	NVIC_PRI2_R	7 – 5
0x00000064	25	9	PWM0Fault_Handler	NVIC_PRI2_R	15 – 13
0x00000068	26	10	PWM0_Handler	NVIC_PRI2_R	23 – 21
0x0000006C	27	11	PWM1_Handler	NVIC_PRI2_R	31 – 29
0x00000070	28	12	PWM2_Handler	NVIC_PRI3_R	7 – 5
0x00000074	29	13	Quadrature0_Handler	NVIC_PRI3_R	15 – 13
0x00000078	30	14	ADC0_Handler	NVIC_PRI3_R	23 – 21
0x0000007C	31	15	ADC1_Handler	NVIC_PRI3_R	31 – 29
0x00000080	32	16	ADC2_Handler	NVIC_PRI4_R	7 – 5
0x00000084	33	17	ADC3_Handler	NVIC_PRI4_R	15 – 13
0x00000088	34	18	WDT_Handler	NVIC_PRI4_R	23 – 21
0x0000008C	35	19	Timer0A_Handler	NVIC_PRI4_R	31 – 29
0x00000090	36	20	Timer0B_Handler	NVIC_PRI5_R	7 – 5
0x00000094	37	21	Timer1A_Handler	NVIC_PRI5_R	15 – 13
0x00000098	38	22	Timer1B_Handler	NVIC_PRI5_R	23 – 21
0x0000009C	39	23	Timer2A_Handler	NVIC_PRI5_R	31 – 29
0x000000A0	40	24	Timer2B_Handler	NVIC_PRI6_R	7 – 5
0x000000A4	41	25	Comp0_Handler	NVIC_PRI6_R	15 – 13
0x000000A8	42	26	Comp1_Handler	NVIC_PRI6_R	23 – 21
0x000000AC	43	27	Comp2_Handler	NVIC_PRI6_R	31 – 29
0x000000B0	44	28	SysCtl_Handler	NVIC_PRI7_R	7 – 5
0x000000B4	45	29	FlashCtl_Handler	NVIC_PRI7_R	15 – 13
0x000000B8	46	30	GPIOPortF_Handler	NVIC_PRI7_R	23 – 21
0x000000BC	47	31	GPIOPortG_Handler	NVIC_PRI7_R	31 – 29
0x000000C0	48	32	GPIOPortH_Handler	NVIC_PRI8_R	7 – 5
0x000000C4	49	33	UART2_Handler	NVIC_PRI8_R	15 – 13
0x000000C8	50	34	SSI1_Handler	NVIC_PRI8_R	23 – 21
0x000000CC	51	35	Timer3A_Handler	NVIC_PRI8_R	31 – 29
0x000000D0	52	36	Timer3B_Handler	NVIC_PRI9_R	7 – 5
0x000000D4	53	37	I2C1_Handler	NVIC_PRI9_R	15 – 13
0x000000D8	54	38	Quadrature1_Handler	NVIC_PRI9_R	23 – 21
0x000000DC	55	39	CAN0_Handler	NVIC_PRI9_R	31 – 29
0x000000E0	56	40	CAN1_Handler	NVIC_PRI10_R	7 – 5
0x000000E4	57	41	CAN2_Handler	NVIC_PRI10_R	15 – 13
0x000000E8	58	42	Ethernet_Handler	NVIC_PRI10_R	23 – 21
0x000000EC	59	43	Hibernate_Handler	NVIC_PRI10_R	31 – 29
0x000000F0	60	44	USB0_Handler	NVIC_PRI11_R	7 – 5
0x000000F4	61	45	PWM3_Handler	NVIC_PRI11_R	15 – 13
0x000000F8	62	46	uDMA_Handler	NVIC_PRI11_R	23 – 21
0x000000FC	63	47	uDMA_Error	NVIC_PRI11_R	31 – 29

嵌套向量中断控制器：

定义：在多个来源的中断之间进行协调的硬件单元

功能： 1)定义每个中断源的优先级（NVIC_PRIx_R 寄存器）

2) 为每个中断源分配使能标志（NVIC_EN0_R 和 NVIC_EN1_R）

the priority registers on the NVIC

Address	31 – 29	23 – 21	15 – 13	7 – 5	Name
0xE000E400	GPIO Port D	GPIO Port C	GPIO Port B	GPIO Port A	NVIC_PRI0_R
0xE000E404	SSI0, Rx Tx	UART1, Rx Tx	UART0, Rx Tx	GPIO Port E	NVIC_PRI1_R
0xE000E408	PWM Gen 1	PWM Gen 0	PWM Fault	I2C0	NVIC_PRI2_R
0xE000E40C	ADC Seq 1	ADC Seq 0	Quad Encoder	PWM Gen 2	NVIC_PRI3_R
0xE000E410	Timer 0A	Watchdog	ADC Seq 3	ADC Seq 2	NVIC_PRI4_R
0xE000E414	Timer 2A	Timer 1B	Timer 1A	Timer 0B	NVIC_PRI5_R
0xE000E418	Comp 2	Comp 1	Comp 0	Timer 2B	NVIC_PRI6_R
0xE000E41C	GPIO Port G	GPIO Port F	Flash Control	System Control	NVIC_PRI7_R
0xE000E420	Timer 3A	SSI1, Rx Tx	UART2, Rx Tx	GPIO Port H	NVIC_PRI8_R
0xE000E424	CAN0	Quad Encoder 1	I2C1	Timer 3B	NVIC_PRI9_R
0xE000E428	Hibernate	Ethernet	CAN2	CAN1	NVIC_PRI10_R
0xE000E42C	uDMA Error	uDMA Soft Tfr	PWM Gen 3	USB0	NVIC_PRI11_R
0xE000ED20	SysTick	PendSV	--	Debug	NVIC_SYS_PRI3_R

DAC(数字→模拟转换)、采样

奈奎斯特采样定理：

奈奎斯特定理指出，如果信号以 fs 的频率进行采样，那么数字采样只包含 0 到 1/2 fs 的频率分量

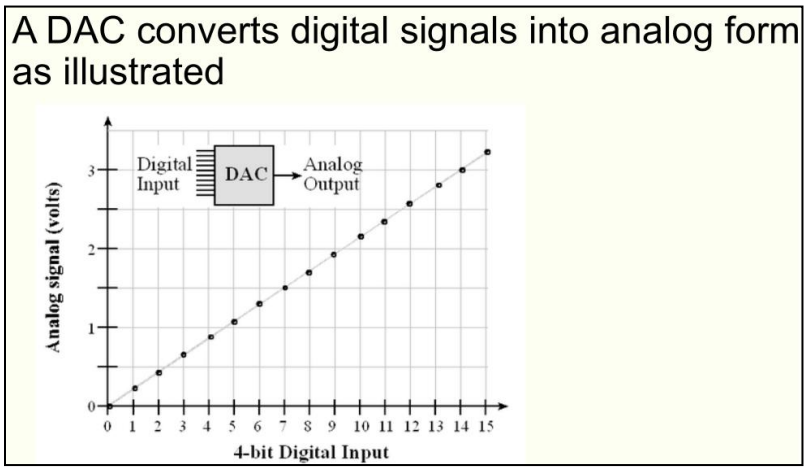
如果采样率 fs 超过每秒 2f 个最大采样点，其中 fmax 是原始信号中的最高频率，那么已经采样的带限模拟信号可以从无限采样序列完美重建；如果模拟信号确实含有大于(1/2)fs 的频率分量，则会出现混叠(数字信号出现与原始模拟信号不同的频率)误差。

表达式：C = B * log2 N (bps)

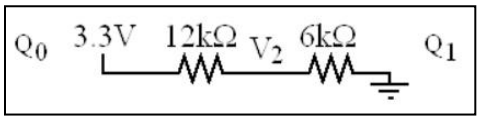
有限的采样间隔引起时间量化

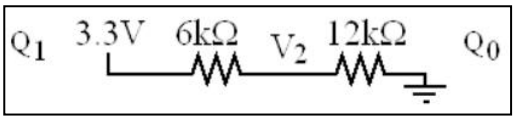
数字→模拟：

大多数数字音乐设备都依靠高速 DAC 来创建产生高质量声音所需的模拟波形



假设微控制器的输出高电压（VOH）为 3.3 V，其输出低电压（VOL）为 0.对于二进制加权 DAC，我们选择电阻比率为 2/1，因此 Q1 比特是两倍 作为 Q0 位，如图 13.3 所示。 考虑到左边的电路（没有耳机），如果 Q1 和 Q0 均为 0，则输出 Vout 为零。 如果 Q1 为 0 且 Q0 为+ 3.3V，则输出 Vout 由电阻分压网络决定





请注意，上述电路中从 3.3V 到地的总阻抗为 18kΩ。使用欧姆定律和分压器方程，我们可以计算 Vout 为 3.3V *6kΩ/18kΩ，即 1.1V。 如果 Q1 为+ 3.3V 且 Q0 为 0，则输出 Vout 由网络决定

再次注意，第二个电路中从 3.3V 到地的总阻抗为 18kΩ。但是这次我们计算 Vout 为 3.3V *12kΩ/18kΩ，这是 2.2V。如果 Q1 和 Q0 均为+ 3.3V，则输出 Vout 为+ 3.3V。该 DAC 的输出阻抗约为 12kΩ，这意味着它不能输入或吸收很多电流。

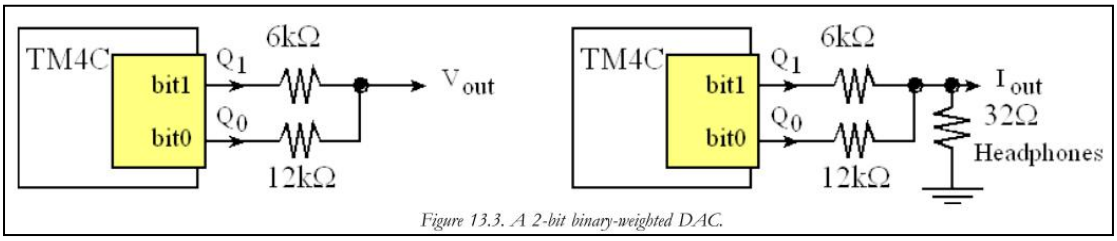


Figure 13.3. A 2-bit binary-weighted DAC.

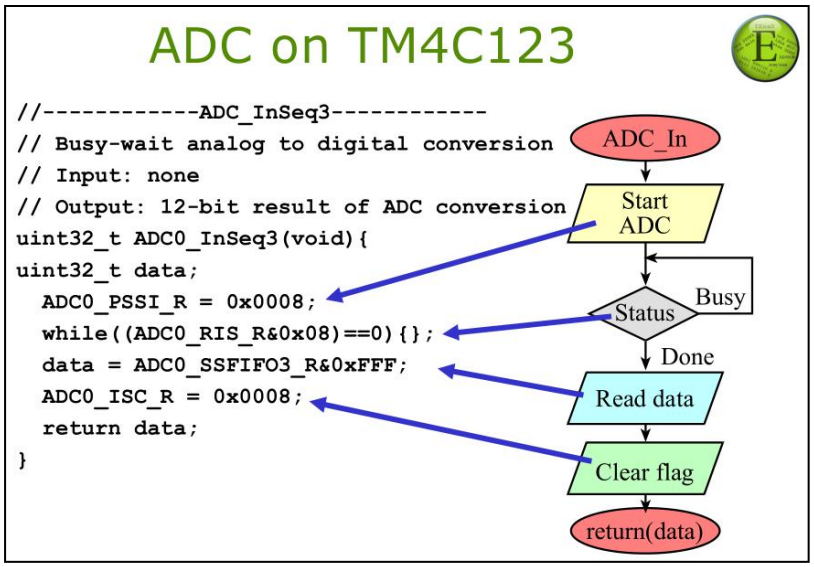
如果我们将耳机连接到该 DAC,如图 13.3 右侧所示,我们可以听到由软件将一系列数据写入 DAC 的声音.但是,由于耳机的阻抗远小于 DAC 的阻抗,因此输出电压将非常小,但我们可以计算流入耳机的电流.考虑到右边的电路（带耳机）,如果 Q1 和 Q0 均为 0,则输出电流为零.如果 Q 1 为 0 且 Q 0 为+ 3.3V,则输出电流 I out 为 3.3V 除以 12.032kΩ即 0.275mA.如果 Q 0 为 0 且 Q 1 为+ 3.3V,则输出电流 I out 为 3.3V 除以 6.032kΩ即 0.550mA.最后,如果 Q 1 和 Q 0 均为 3.3V,则 I out 为 0.275 + 0.550（Kirkhoff 电流定律）的总和,其约为 0.825mA.注意耳机中的电流与数字值线性相关.换句话说,0,1,2,3 的数字值映射到 0,0.275,0.550,0.825mA

ADC(模拟→数字转换)

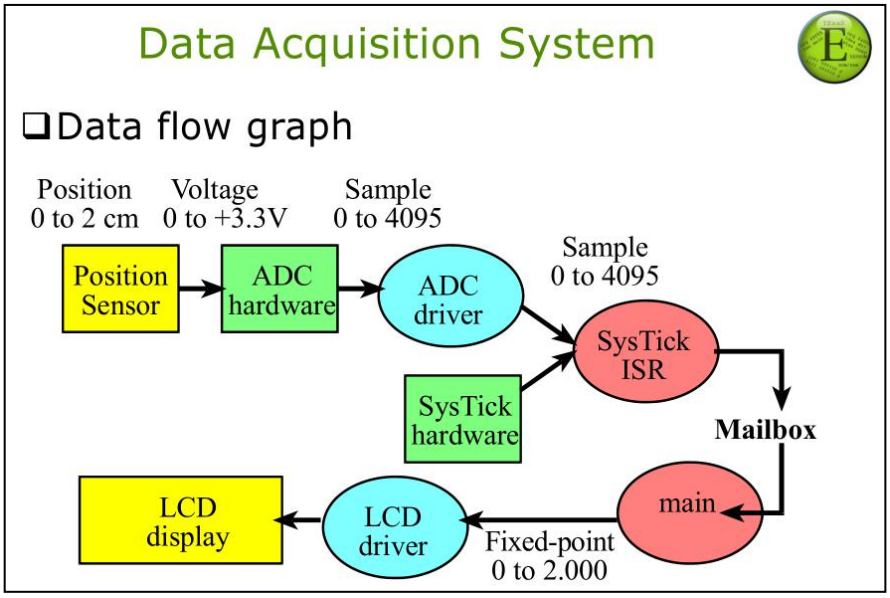
步骤：

1、启用将用于 ADC 输入的引脚的端口时钟
2、将 0 写入 DIR 寄存器来使该引脚成为输入
3、向 AFSEL 寄存器写入 1 来启用该引脚的替代功能
4、向 DEN 寄存器写入零来禁用该引脚上的数字功能
5、向 AMSEL 寄存器写入 1 来启用该引脚上的模拟功能
6、设置 SYSCTL_RCGC0_R 寄存器的第 16 位来使能 ADC 时钟
7、SYSCTL_RCGC0_R 寄存器的位 8 和 9 指定 ADC 的最大采样速率
8、在 ADC0_SSPRI_R 中设置序列器优先级：0,1,2,3
9、配置序列器之前禁用 ADC0_ACTSS_R 的 3 位
10、在 ADC_EMUX_R 寄存器中配置采样序列发生器的触发事件
11、在 ADC0_SSMUXn 寄存器中配置相应的输入源
12、在 ADC0_SSCTLn 寄存器的相应半字节中配置采样控制位
13、禁用中断：ADC0_IM_R 的右 3 位为零
14、向相应的 ASENn 写入 1 来启用采样序列发生器逻辑

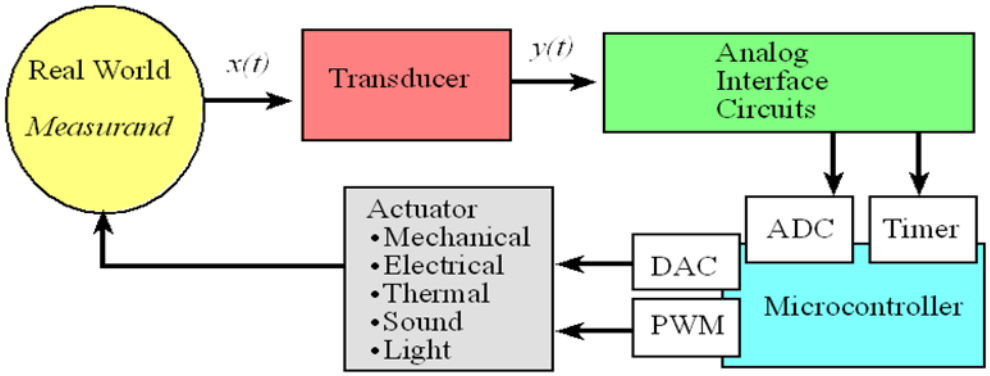
```
void ADC0_InitSWTriggerSeq3_Ch9(void){ volatile unsigned long delay;
  SYSCTL_RCGC2_R |= 0x00000010;   // 1) activate clock for Port E
  delay = SYSCTL_RCGC2_R;         //    allow time for clock to stabilize
  GPIO_PORTE_DIR_R &= ~0x04;      // 2) make PE4 input
  GPIO_PORTE_AFSEL_R |= 0x04;     // 3) enable alternate function on PE2
  GPIO_PORTE_DEN_R &= ~0x04;      // 4) disable digital I/O on PE2
  GPIO_PORTE_AMSEL_R |= 0x04;     // 5) enable analog function on PE2
  SYSCTL_RCGC0_R |= 0x00010000;   // 6) activate ADC0
  delay = SYSCTL_RCGC2_R;
  SYSCTL_RCGC0_R &= ~0x00000300; // 7) configure for 125K
  ADC0_SSPRI_R = 0x0123;          // 8) Sequencer 3 is highest priority
  ADC0_ACTSS_R &= ~0x0008;        // 9) disable sample sequencer 3
  ADC0_EMUX_R &= ~0xF000;         // 10) seq3 is software trigger
  ADC0_SSMUX3_R &= ~0x000F;       // 11) clear SS3 field
  ADC0_SSMUX3_R += 9;             //    set channel Ain9 (PE4)
  ADC0_SSCTL3_R = 0x0006;         // 12) no TS0 D0, yes IE0 END0
  ADC0_ACTSS_R |= 0x0008;         // 13) enable sample sequencer 3
}
```

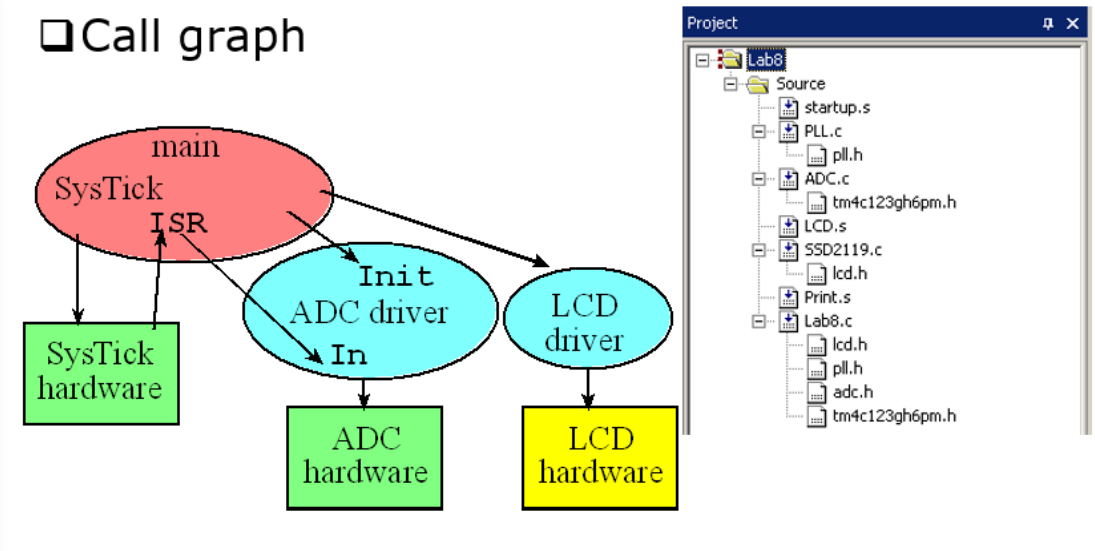


传感器 – 由一个系统的电源驱动的设备，以相同或其他形式向另一个系统供电。



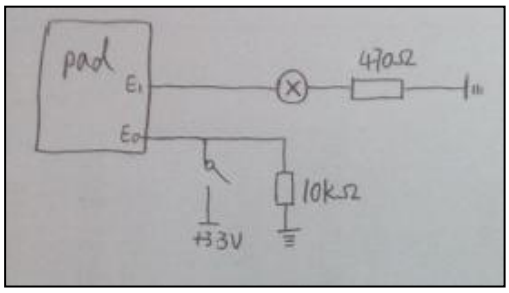
信号路径数据采集系统：





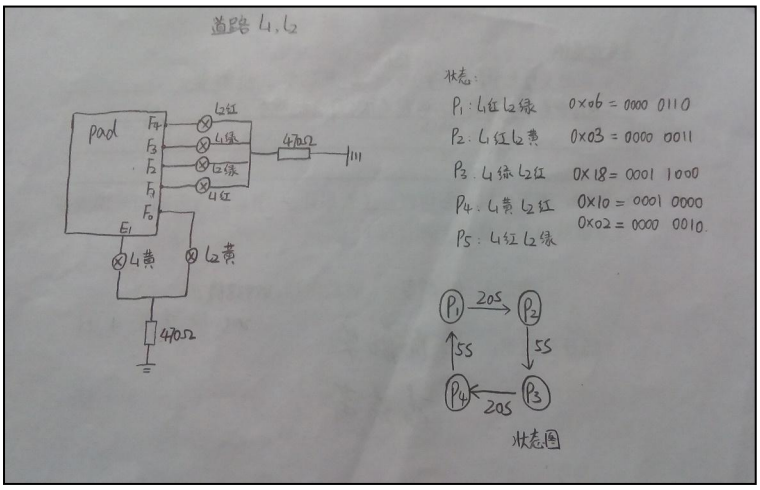
实验代码

Switch&LED 灯：




```
#define SYSCTL_RCGC2_R          (*((volatile unsigned long *)0x400FE108))
#define GPIO_PORTD_DATA_R       (*((volatile unsigned long *)0x400073FC))
#define GPIO_PORTD_DIR_R        (*((volatile unsigned long *)0x40007400))
#define GPIO_PORTD_AFSEL_R      (*((volatile unsigned long *)0x40007420))
#define GPIO_PORTD_DEN_R        (*((volatile unsigned long *)0x4000751C))
#define GPIO_PORTD_AMSEL_R      (*((volatile unsigned long *)0x40007528))
#define GPIO_PORTD_PCTL_R       (*((volatile unsigned long *)0x4000752C))
unsigned long in,out;
int main(void){ unsigned long volatile delay;
  SYSCTL_RCGC2_R |= 0x08;            // Port D clock
  delay = SYSCTL_RCGC2_R;            // wait 3-5 bus cycles
  GPIO_PORTD_DIR_R |= 0x08;          // PD3 output
  GPIO_PORTD_DIR_R &= ~0x01;         // PD0 input
  GPIO_PORTD_AFSEL_R &= ~0x09;       // not alternative
  GPIO_PORTD_AMSEL_R &= ~0x09;       // no analog
  GPIO_PORTD_PCTL_R &= ~0x0000F00F; // bits for PD3, PD0
  GPIO_PORTD_DEN_R |= 0x09;          // enable PD3, PD0
  while(1){
    in = (GPIO_PORTD_DATA_R&0x01);
    out = ~(in^0x01)<<3;
    GPIO_PORTD_DATA_R = out;
  }
}
```

交通灯：



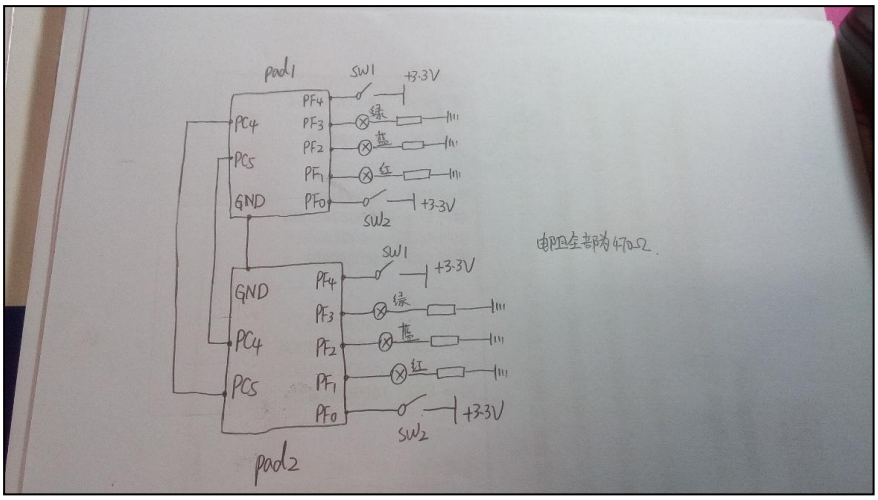

```
void Delay20s(void){unsigned long volatile time;
  time = 727240*200/91*20;  // 20s
  while(time){
      time--;
  }
}

void Delay5s(void){unsigned long volatile time;
  time = 727240*200/91*5;  // 5s
  while(time){
      time--;
  }
}
```

```
int main(void){volatile unsigned int state = 1;
    SYSCTL_RCGC2_R |= 0x09;           // Port D clock
    GPIO_PORTA_PCTL_R &= ~0x0000FF00;
    GPIO_PORTA_AMSEL_R &= ~0x0C;
    GPIO_PORTA_DIR_R |= 0x0C;
    GPIO_PORTA_AFSEL_R &= ~0x0C;
    GPIO_PORTA_DEN_R |= 0x0C;          //PA2:yellow LED1 PA3:yellow LED2
    GPIO_PORTD_PCTL_R &= ~0x0000FFFF;
    GPIO_PORTD_AMSEL_R &= ~0x0F;
    GPIO_PORTD_DIR_R |= 0x0F;
    GPIO_PORTD_AFSEL_R &= ~0x0F;       //PD0:green LED1 PD1:green LED2
    GPIO_PORTD_DEN_R |= 0x0F;          //PD2:red LED1 PD3:red LED2
    while(1){
}
```

```
while(1){
    if(state == 1){
        GPIO_PORTD_DATA_R &= 0x00;
        GPIO_PORTA_DATA_R &= 0x00;
        GPIO_PORTD_DATA_R |= 0x06;//p1
        GPIO_PORTA_DATA_R |= 0x00;
        state++;
        Delay20s();
    }
    else if(state == 2){
        GPIO_PORTD_DATA_R &= 0x00;
        GPIO_PORTA_DATA_R &= 0x00;
        GPIO_PORTD_DATA_R |= 0x04;//p2
        GPIO_PORTA_DATA_R |= 0x08;
        state++;
        Delay5s();
    }
    else if(state == 3){
        GPIO_PORTD_DATA_R &= 0x00;
        GPIO_PORTA_DATA_R &= 0x00;
        GPIO_PORTD_DATA_R |= 0x09;//p3
        GPIO_PORTA_DATA_R |= 0x00;
        state++;
        Delay20s();
    }
    else if(state == 4){
        GPIO_PORTD_DATA_R &= 0x00;
        GPIO_PORTA_DATA_R &= 0x00;
        GPIO_PORTD_DATA_R |= 0x08;//p4
        GPIO_PORTA_DATA_R |= 0x04;
        state = 1;
        Delay5s();
    }
}
```

UART：



PLL.c:

```
void PLL_Init(void){
  // 0) configure the system to use RCC2 for advanced features
  //    such as 400 MHz PLL and non-integer System Clock Divisor
  SYSCTL_RCC2_R |= SYSCTL_RCC2_USERCC2;
  // 1) bypass PLL while initializing
  SYSCTL_RCC2_R |= SYSCTL_RCC2_BYPASS2;
  // 2) select the crystal value and oscillator source
  SYSCTL_RCC_R &= ~SYSCTL_RCC_XTAL_M;   // clear XTAL field
  SYSCTL_RCC_R += SYSCTL_RCC_XTAL_16MHZ;// configure for 16 MHz crystal
  SYSCTL_RCC2_R &= ~SYSCTL_RCC2_OSCSRC2_M;// clear oscillator source field
  SYSCTL_RCC2_R += SYSCTL_RCC2_OSCSRC2_MO;// configure for main oscillator source
  // 3) activate PLL by clearing PWRDN
  SYSCTL_RCC2_R &= ~SYSCTL_RCC2_PWRDN2;
  // 4) set the desired system divider and the system divider least significant bit
  SYSCTL_RCC2_R |= SYSCTL_RCC2_DIV400;  // use 400 MHz PLL
  SYSCTL_RCC2_R = (SYSCTL_RCC2_R&~ 0x1FC00000)  // clear system clock divider
                  + (SYSDIV2<<22);      // configure for 80 MHz clock
  // 5) wait for the PLL to lock by polling PLLLRIS
  while((SYSCTL_RIS_R&SYSCTL_RIS_PLLLRIS)==0){};
  // 6) enable use of PLL by clearing BYPASS
  SYSCTL_RCC2_R &= ~SYSCTL_RCC2_BYPASS2;
}
```

```
#define NVIC_ST_CTRL_R      (*((volatile unsigned long *)0xE000E010))
#define NVIC_ST_RELOAD_R    (*((volatile unsigned long *)0xE000E014))
#define NVIC_ST_CURRENT_R   (*((volatile unsigned long *)0xE000E018))
void SysTick_Init(void){
  NVIC_ST_CTRL_R = 0;               // 在设置 SysTick时禁用
  NVIC_ST_CTRL_R = 0x00000005;      // 在 SysTick上使用系统时钟
}
// The delay parameter is in units of the 80 MHz core clock. (12.5 ns)
void SysTick_Wait(unsigned long delay){
  NVIC_ST_RELOAD_R = delay-1;  // 需要等待的时间数
  NVIC_ST_CURRENT_R = 0;       // 清空现有的值
  while((NVIC_ST_CTRL_R&0x00010000)==0){ // 等待计数位的时钟脉冲标志
  }
}
// 10000us equals 10ms
void SysTick_Wait10ms(unsigned long delay){
  unsigned long i;
  for(i=0; i<delay; i++){
    SysTick_Wait(800000);  // wait 10ms
  }
}
```

UART.c:

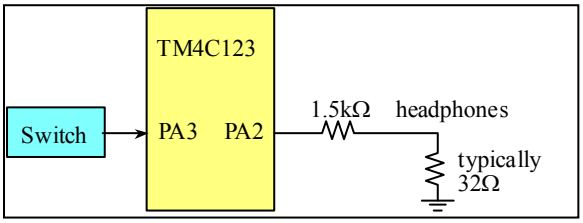
```
void UART_Init(void){
  SYSCTL_RCGC1_R |= SYSCTL_RCGC1_UART1; // 激活 UART1
  SYSCTL_RCGC2_R |= SYSCTL_RCGC2_GPIOC; // 激活 port C
  UART1_CTL_R &= ~UART_CTL_UARTEN;      // 初始化URAT时禁用 UART
  UART1_IBRD_R = 43;                    // 80,000,000/(16*115,200)) = 43.40278,  频率/（16*波特率）
  UART1_FBRD_R = 26;                    //6-bbit 小数部分, round(0.40278 * 64) = 26, 保证5%以内的误差
                                                  // 8 bit word length (没有奇偶校验位,没有停止位的FIFOs)
  UART1_LCRH_R = (UART_LCRH_WLEN_8|UART_LCRH_FEN);
  UART1_CTL_R |= UART_CTL_UARTEN;       // 初始化UART结束后启用 UART
  GPIO_PORTC_AFSEL_R |= 0x30;           // enable alt funct on PC5-4
  GPIO_PORTC_DEN_R |= 0x30;             // 在PC4与PC5端口启用I/O
                                        // 配置PC4与PC5为 UART1
  GPIO_PORTC_PCTL_R = (GPIO_PORTC_PCTL_R&0xFF00FFFF)+0x00220000;
  GPIO_PORTC_AMSEL_R &= ~0x30;          // 禁用PC4与PC5的模拟信号功能
}
```

```
//------------UART_InChar------------
// Wait for new serial port input
// Input: none
// Output: ASCII code for key typed
unsigned char UART_InChar(void){
  while((UART1_FR_R&UART_FR_RXFE) != 0);
  return((unsigned char)(UART1_DR_R&0xFF));
}

//------------UART_InCharNonBlocking------------
// Get oldest serial port input and return immediately
// if there is no data.
// Input: none
// Output: ASCII code for key typed or 0 if no character
unsigned char UART_InCharNonBlocking(void){
  if((UART1_FR_R&UART_FR_RXFE) == 0){
    return((unsigned char)(UART1_DR_R&0xFF));
  } else{
    return 0;
  }
}

//------------UART_OutChar------------
// Output 8-bit to serial port
// Input: letter is an 8-bit ASCII character to be transferred
// Output: none
void UART_OutChar(unsigned char data){
  while((UART1_FR_R&UART_FR_TXFF) != 0);
  UART1_DR_R = data;
}
```

中断产生方波：



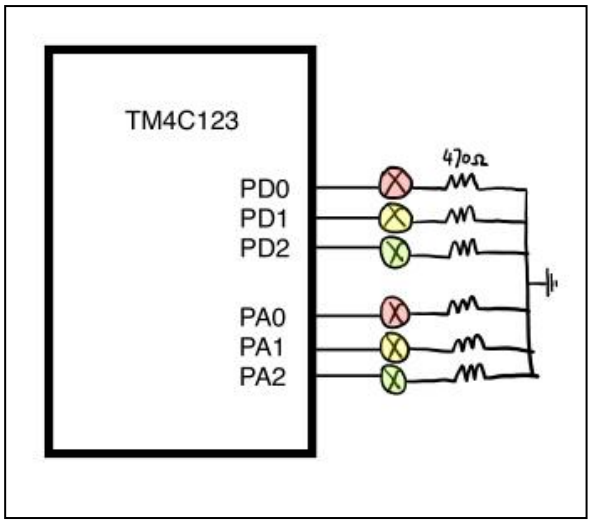

```
void Sound_Init(void){ unsigned long volatile delay;
    SYSCTL_RCGC2_R |= 0x00000001;
    delay = SYSCTL_RCGC2_R;
    GPIO_PORTA_AMSEL_R &= ~0x0C;
    GPIO_PORTA_PCTL_R &= ~0x00000F00;
    GPIO_PORTA_DIR_R |= 0x04;       //PA2 output
    GPIO_PORTA_DIR_R &= ~0x08;      //PA3 input
    GPIO_PORTA_DR8R_R |= 0x04;
    GPIO_PORTA_AFSEL_R &= ~0x0C;
    GPIO_PORTA_DEN_R |= 0x0C;
    NVIC_ST_CTRL_R = 0;
    NVIC_ST_RELOAD_R = 90908;       //80MHz/(2*440Hz)-1 = 90908.09090909091
    NVIC_ST_CURRENT_R = 0;
    NVIC_SYS_PRI3_R = NVIC_SYS_PRI3_R&0x00FFFFFF;
    NVIC_ST_CTRL_R = 0x00000007;
}

void SysTick_Handler(void){
    GPIO_PORTA_DATA_R ^= 0x04;
}
```

```
int main(void){
  TExaS_Init(SW_PIN_PA3, HEADPHONE_PIN_PA2,ScopeOn);
  Sound_Init();
  while(1){
    in = (GPIO_PORTA_DATA_R&0x08);
    if(in != 0)
        EnableInterrupts();
    else
        DisableInterrupts();
  }
}
```

$$\text{中断处理程序频率}+1=\frac{80Mhz}{2\times\text{音频}}$$

跑马灯：




```
int main(void){volatile unsigned long delay;
    volatile unsigned int state = 0;
    int time = 13333333 / 4;
    PLL_Init();
    SYSCTL_RCGC2_R |= SYSCTL_RCGC2_GPIOD;
    delay = SYSCTL_RCGC2_R;
    GPIO_PORTD_DIR_R |= 0x07;
    GPIO_PORTD_AFSEL_R &= ~0x07;
    GPIO_PORTD_DEN_R |= 0x07;
    GPIO_PORTD_PCTL_R = (GPIO_PORTD_PCTL_R&0xFFF0FFFF)+0x00000000;
    GPIO_PORTD_AMSEL_R = 0;
    SYSCTL_RCGC2_R |= SYSCTL_RCGC2_GPIOB;
    GPIO_PORTB_DIR_R |= 0x07;
    GPIO_PORTB_AFSEL_R &= ~0x07;
    GPIO_PORTB_DEN_R |= 0x07;
    GPIO_PORTB_PCTL_R = (GPIO_PORTB_PCTL_R&0xFFFFF000)+0x00000000;
    GPIO_PORTB_AMSEL_R &= ~0x07;
    while(1){
}
```

```
while(1){
    state = state % 6;
    switch(state){
        case 0:
            GPIO_PORTD_DATA_R &= 0x00;  // turn off LED
            GPIO_PORTB_DATA_R &= 0x00;
            GPIO_PORTD_DATA_R |= 0x07;  // turn on PD0,PD1,PD2
            Delay(time);                                // delay
            state ++;
        break;
        case 1:
            GPIO_PORTD_DATA_R &= 0x00;  // turn off LED
            GPIO_PORTB_DATA_R &= 0x00;
            GPIO_PORTD_DATA_R |= 0x06;  // turn on PD1,PD2
            GPIO_PORTB_DATA_R |= 0x01;  // turn on PA0
            Delay(time);                                // delay
            state ++;
        break;
        case 2:
            GPIO_PORTD_DATA_R &= 0x00;  // turn off LED
            GPIO_PORTB_DATA_R &= 0x00;
            GPIO_PORTD_DATA_R |= 0x04;  // turn on PD2
            GPIO_PORTB_DATA_R |= 0x03;  // turn on PA0,PA1
            Delay(time);                                // delay
            state ++;
        break;
        case 3:
            GPIO_PORTD_DATA_R &= 0x00;  // turn off LED
            GPIO_PORTB_DATA_R &= 0x00;
            GPIO_PORTB_DATA_R |= 0x07;  // turn on PA0,PA1,PA2
            Delay(time);                                // delay
            state ++;
        break;
        case 4:
            GPIO_PORTD_DATA_R &= 0x00;  // turn off LED
            GPIO_PORTB_DATA_R &= 0x00;
            GPIO_PORTD_DATA_R |= 0x01;  // turn on PD0
            GPIO_PORTB_DATA_R |= 0x06;  // turn on PA1,PA2
            Delay(time);                                // delay
            state ++;
        break;
        case 5:
            GPIO_PORTD_DATA_R &= 0x00;  // turn off LED
            GPIO_PORTB_DATA_R &= 0x00;
            GPIO_PORTD_DATA_R |= 0x03;  // turn on PD0,PD1
            GPIO_PORTB_DATA_R |= 0x04;  // turn on PA2
            Delay(time);                                // delay
            state ++;
        break;
    }
}
```

UML

UML 组成部分：

视图：包括逻辑视图，用例视图，组件视图，并发视图，部署视图

图：包括用例图，类图，对象图，状态图，顺序图，协作图，活动图，组件图，部署图

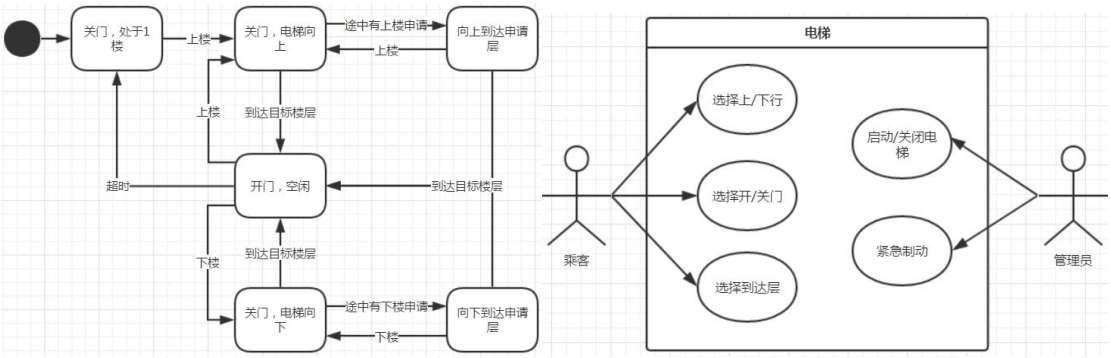
模型元素：包括类，对象，状态，节点等

通用机制：包括修饰，注解，规格说明

电梯例子：

系统需求分析：

电梯的上行下行，电梯内乘客选择到达楼层；管理员对电梯的紧急制动，对电梯的启动/关闭选择



有限状态机

有限状态机（Finite-state machine, FSM）：

又称有限状态自动机，简称状态机，是表示有限个状态以及在这些状态之间的转移和动作等行为的数学模型。

有限状态机动作类型：

进入动作：在进入状态时进行

退出动作：在退出状态时进行

输入动作：依赖于当前状态和输入条件进行

转移动作：在进行特定转移时进行

有限状态机作用：

有限状态机是指输出取决于过去输入部分和当前输入部分的时序逻辑电路。一般来说，除了输入部分和输出部分外，有限状态机还含有一组具有“记忆”功能的寄存器，这些寄存器的功能是记忆有限状态机的内部状态，它们常被称为状态寄存器。在有限状态机中，状态寄存器的的下一个状态不仅与输入信号有关，而且还与该寄存器的当前状态有关，因此有限状态机又可以认为是组合逻辑和寄存器逻辑的一种组合。其中，寄存器逻辑的功能是存储有限状态机的内部状态；而组合逻辑又可以分为次态逻辑和输出逻辑两部分，次态逻辑的功能是确定有限状态机的下一个状态，输出逻辑的功能是确定有限状态机的输出。