

# 现代电子系统设计小学期课程报告

2015011506 金帆 自 56 班

## 目录

1 提高实验简介.....	3
1-1 提高实验 1：ARM 测温显示与上传.....	3
1-1-1 概况.....	3
1-1-2 遇到的问题及解决.....	3
1-1-3 代码注释.....	4
1-1-4 运行效果.....	5
1-1-5 实验收获.....	5
1-1-6 实验建议.....	6
1-2 提高实验 2：使用 CapSense 控制 PSoC 的 LED 亮度并上传.....	6
1-2-1 概况.....	6
1-2-2 遇到的问题及解决.....	7
1-2-3 代码注释.....	8
1-2-4 运行效果.....	9
1-2-5 实验收获.....	9
1-2-6 实验建议.....	9
1-3 提高实验 3：FPGA 和 NIOS 实现遥控音乐播放器 UI.....	10
1-3-1 概况.....	10
1-3-2 遇到的问题及解决.....	11
1-3-3 代码注释.....	13
1-3-4 运行效果.....	15
1-3-5 实验收获.....	16
1-3-6 实验建议.....	17
2 综合实验：基于 PSoC 的语音存储回放系统.....	17
2-1 概况和方案设计.....	17
2-2 硬件原理图.....	18
2-3 软件模块图、状态图.....	20
2-4 实验关键步骤说明.....	21
2-4-1 测试硬件直连.....	21
2-4-2 测试软件直连.....	21
2-4-3 编写状态机，实现功能.....	21
2-5 实验结果及效果.....	22
2-6 实验演示操作方法.....	22
2-7 遇到的问题及解决.....	23
2-7-1 音质太差.....	23
2-7-2 内存只够存储 2 秒.....	23
2-7-3 滤波器参数的确定.....	23

2-8 实验收获和建议 .....	24
2-8-1 实验收获 .....	24
2-8-2 实验建议 .....	25
2-9 软件程序注释 .....	25
2-10 参考资料 .....	26
3 创新实验：基于 Kinect 手势交互的自平衡小车 .....	26
3-0 背景介绍 .....	26
3-0-1 自平衡车 (Self-Balancing Scooter) .....	26
3-0-2 微软 Kinect 体感摄像头 .....	27
3-0-3 比例积分微分 (PID) 控制原理简介 .....	27
3-1 概况和方案设计 .....	28
3-2 硬件原理图 .....	29
3-2-1 机器人搭建 .....	30
3-2-2 硬件电路设计 .....	31
3-3 软件模块 .....	32
3-3-1 模块化设计 .....	32
3-3-2 核心 PID 代码 .....	33
3-4 实验关键步骤说明 .....	34
3-4-0 准备工作：传感器模块、舵机驱动模块、蓝牙模块 .....	34
3-4-1 任务 1：原地平衡 (Hold Still and Keep Balance) .....	34
3-4-2 任务 2：前后移动 (Totter under Control) .....	35
3-4-3 任务 3：通过 Kinect 手势操控 (Kinect in Command) .....	35
3-5 实验效果或结果 .....	36
3-6 本人工作总结 .....	36
3-6-1 蓝牙模块、舵机驱动模块 .....	36
3-6-2 电脑端的 Win32 程序 (C#) .....	36
3-7 遇到的问题及解决 .....	37
3-7-1 无法读取舵机转速 .....	37
3-7-2 电机扭矩不够 .....	38
3-7-3 调参太费时 .....	38
3-8 软件程序注释 .....	38
3-9 实验收获 .....	38
3-9-1 版本控制 .....	38
3-9-2 调参的顺序 .....	39
3-9-3 更加深入地使用 C# .....	40
3-10 参考资料 .....	40

2017 年 8 月 28 日 - 9 月 8 日

# 1 提高实验简介

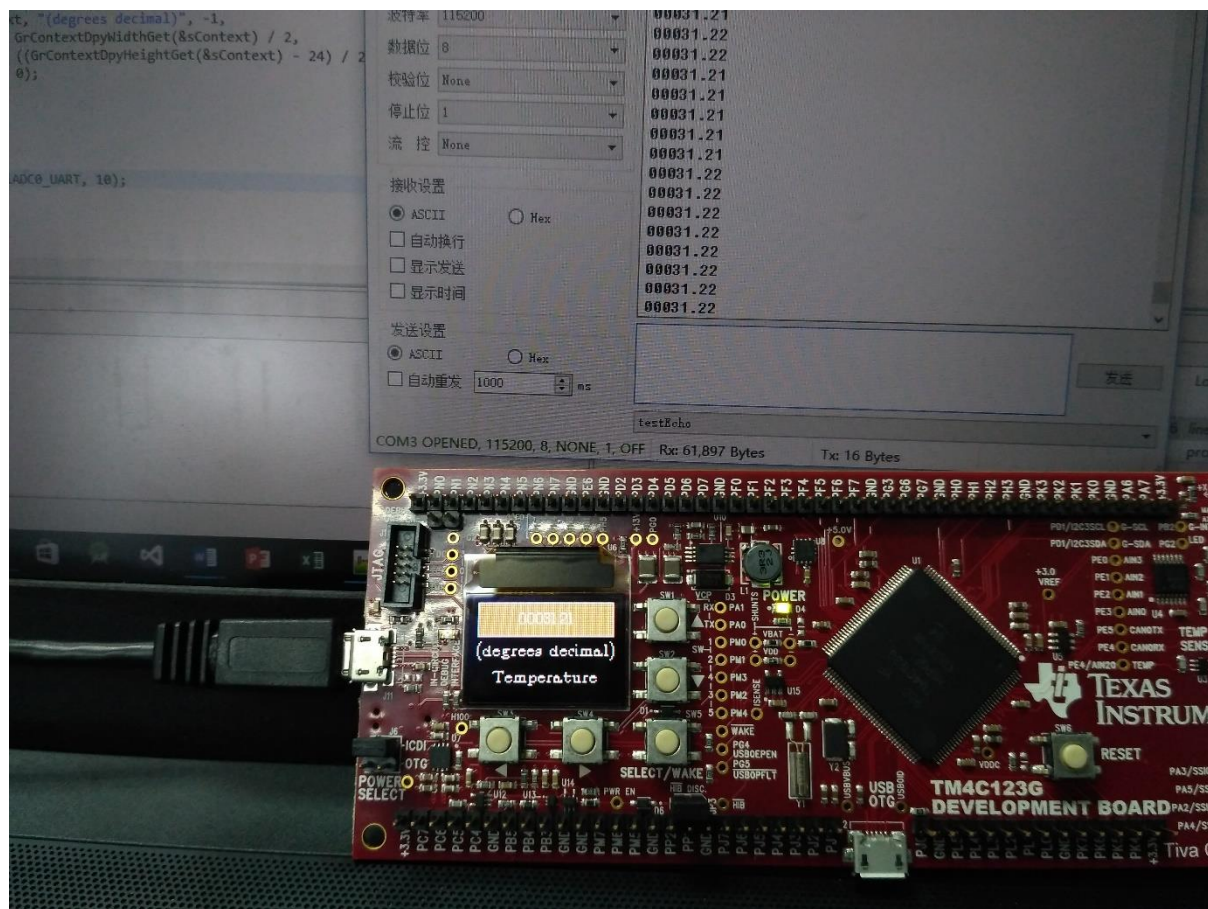
## 1-1 提高实验 1: ARM 测温显示与上传

### 1-1-1 概况

实验时间: 8 月 28 日上午

验收时间: 8 月 28 日午后

实现功能: 采集 ARM 的温度传感器的模拟电压值, 经 AD 转换, 并根据 datasheet 所列的换算公式, 换算成摄氏温度, 显示在 LCD 上, 并通过 UART 串口发送给上位机。



### 1-1-2 遇到的问题及解决

首先遇到的问题是，ARM 初始化代码比较繁杂。我在样例代码基础上进行修改，而不是从空工程开始自己搭建。当然我也仔细阅读了样例代码的初始化部分，大概了解了初始化的步骤：先使能 FPU，设置时钟，然后使能外设。在主循环中，在触发采集和 LCD 显示之前，也需要用参数进行初始化。

第二个问题是，测量的温度有较大幅值的抖动。我加入了一个简单的滤波：

$$T_t := \alpha \cdot T_{t-1} + (1 - \alpha) \cdot D_t$$

每次温度测量的显示值 $T_t$ ，都是上次的显示值 $T_{t-1}$ 和本次测量读数 $D_t$ 的加权平均。

我取权重 $\alpha = 0.95$ ，这样每次读数只占很小的权重，抖动也会减小到 $0.02^{\circ}\text{C}$

### 1-1-3 代码注释

原始代码链接：[https://github.com/kingium/ESD\\_1506/blob/master/ARM/temperature\\_1506/temperature.c](https://github.com/kingium/ESD_1506/blob/master/ARM/temperature_1506/temperature.c)（含注释）

与样例代码相同，我们将 UART 有关的代码封装在一个函数中：

```
26 // 串口接收中断服务程序
27 void UARTIntHandler(void)
28 {
29     unsigned long ulStatus;
30     // 获取中断状态
31     ulStatus = ROM_UARTIntStatus(UART0_BASE, true);
32     // 清除中断标志
33     ROM_UARTIntClear(UART0_BASE, ulStatus);
34     // 直到串口FIFO中没有数据时才退出循环
35     while(ROM_UARTCharsAvail(UART0_BASE))
36     {
37         // 读串口接收的字符并回发
38         ROM_UARTCharPutNonBlocking(UART0_BASE,
39                                     ROM_UARTCharGetNonBlocking(UART0_BASE));
40     }
41 }
42
43 // 串口发送函数
44 void UARTSend(const unsigned char *pucBuffer, unsigned long ulCount)
45 {
46     while(ulCount--)
47     {
48         // 将要发送的字符写进UART
49         ROM_UARTCharPutNonBlocking(UART0_BASE, *pucBuffer++);
50     }
51 }
```

这样做也体现了模块化的设计思想。

### 1-1-4 运行效果

实测的室温在 30℃ 左右，当将手指贴靠温度传感器时，测得的温度会缓慢上升 2~3℃，当撤去手指后，温度又会缓慢回落到正常水平。

我的程序最小分度值是 0.01℃。为便于 UART 传输，在内部处理时，我将温度值乘上 100，以整数的形式存储和传输，这样就避开了浮点数的传输（为遵循 IEEE754 标准，需对浮点数另行转换）。

### 1-1-5 实验收获

假期我借走的正好是 ARM 套件，因此预习比较充分。在一个上午里，我重复了基础实验的操作，大致理清了各个模块的初始化和主循环写法，参考了样例代码中的 AD 转换、LCD 显示和 UART 传输模块。然后我把三个工程的代码组合，就做出了提高实验的框架。

但是这个实验在 AD 转换后，还需要一个电压到温度的查表转换的过程。查阅 datasheet，我发现有两个转换公式，如下图：

在下面的等式中计算出温度结果：

$$T = -1481.96 + \sqrt{\frac{2.19262 \times 10^5 + (1.8639 - V_{OUT})}{3.88 \times 10^{-5}}} \quad (6)$$

当只关心较窄温度范围时，可计算线性转换函数。对于这些计算，请参见[±2.5℃ 低功耗，模拟输出温度传感器数据表 \(TMP20\)](#)。表 2-6 显示针对一个通常选择的温度范围的线性转换函数。

表 2-6. 针对常见温度范围的线性转换函数

温度范围		线性等式 (V)	与抛物线方程 (°C) 的最大偏离
T <sub>最小值</sub> (°C)	T <sub>最大值</sub> (°C)		
-55	130	V <sub>输出</sub> = -11.79mV/°C x T + 1.8528	±1.41
-40	110	V <sub>输出</sub> = -11.77mV/°C x T + 1.8577	±0.93
-30	100	V <sub>输出</sub> = -11.77mV/°C x T + 1.8605	±0.70
-40	85	V <sub>输出</sub> = -11.67mV/°C x T + 1.8583	±0.65
-10	65	V <sub>输出</sub> = -11.71mV/°C x T + 1.8641	±0.23
35	45	V <sub>输出</sub> = -11.81mV/°C x T + 1.8701	±0.004
20	30	V <sub>输出</sub> = -11.69mV/°C x T + 1.8663	±0.004

为了计算简便，我采用了线性近似公式，选取温度范围是 20~30℃ 的公式。

和 PSoC 提高实验比起来，ARM 的代码封装较浅，并没有采用 PSoC 那样的由设计图生成 C 代码的设计，而是预先将 C 代码封装成库，导致代码量巨大。因此我在之后的创新实验中不建议队友使用 ARM 作为开发平台。

### 1-1-6 实验建议

作为一个提高实验，这个实验的难度和任务量都适中，我感觉做下来很顺利。但我也觉得这个实验的灵活性不高，自由发挥的余地不多。我有个小建议，就是允许同学自行选择其他的传感器，使用 ARM 开发板制作一些测量程序。相比于有例程可参考的指定的温度或加速度传感器，这种做法需要同学自己根据 datasheet 编写接口程序，挑战更大，自由度也更高。

## 1-2 提高实验 2：使用 CapSense 控制 PSoC 的 LED 亮度并上传

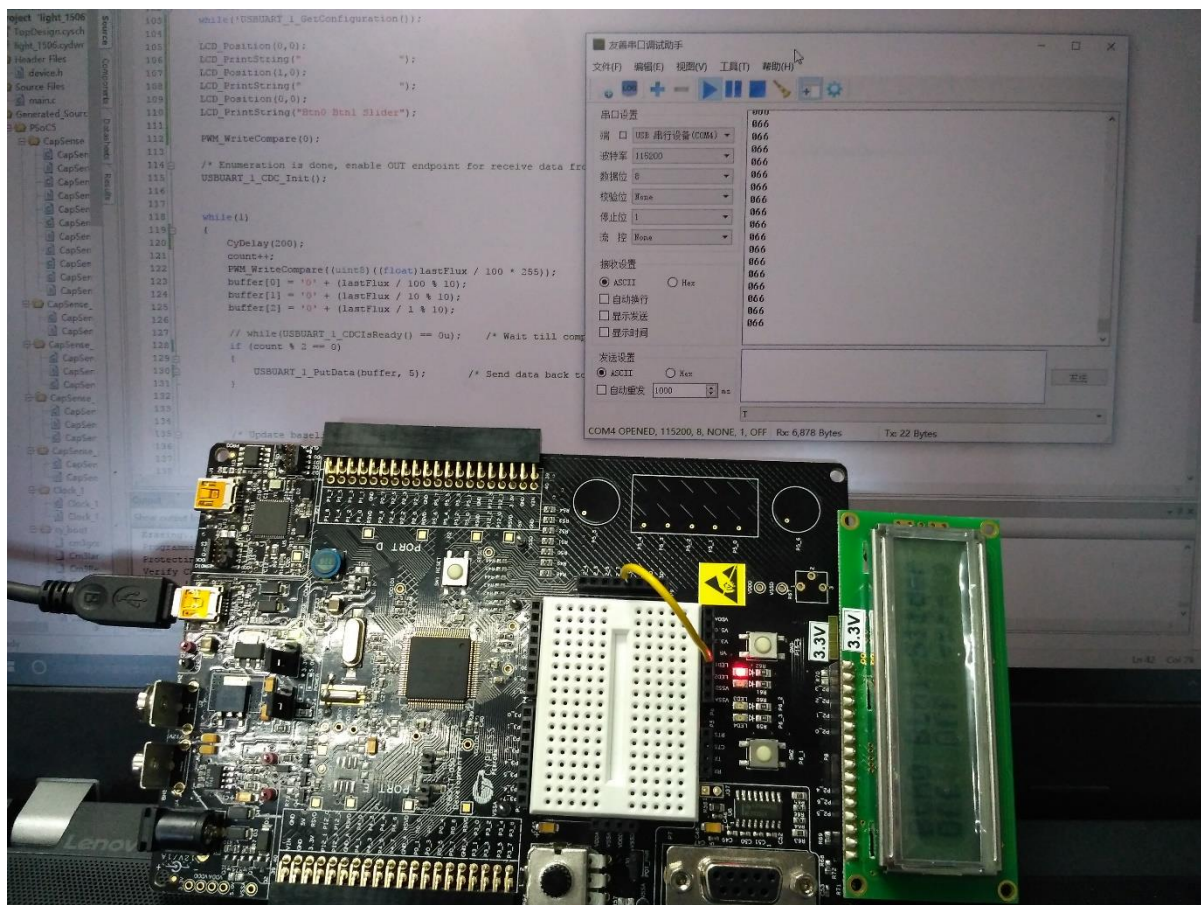
### 1-2-1 概况

实验时间：8 月 28 日下午

验收时间：8 月 29 日早晨

实现功能：在 PSoC 开发板上，使用按键和 CapSense 滑条，通过改变 PWM 占空比来控制 LED 的亮度，并实时经由 UART 上传至上位机。控制的精度可以达到 1%，两次上传之间的间隔大约为 200 毫秒。





## 1-2-2 遇到的问题及解决

同样,这次实验的初始化和主循环代码也有例程可以参考,只需将 **Breathing\_LED**、**LCD**、**UART** 三个例程的代码读懂,并组合在一起,即可完成任务。

但是我还是遇到了其他的一些问题。首先是 **LCD** 的刷新过快,导致屏幕闪烁。我将两次刷新的间隔增大至 **200 毫秒**,顺利解决了这个问题。

还有个问题是很多同学都遇到的,即 **CapSense** 不够灵敏,跳变的分度常常高达 **25%**。为此,我参考了刘江同学在网络学堂上分享的方案(谢谢):



发信人: 刘江(2015011512) 2017-08-28 15:42:55

若CapSense 示数跳变不连续,如从25->50->75->100,可将CapSense的 **Finger / Noise Threshold** 的值调小,增加灵敏度。



得益于 PSoC 对于 UART 串口的完善封装，本实验的串口数据发送和 ARM 提高实验相比，显得更加简单一些，没有遇到太大的问题。

### 1-2-3 代码注释

原始代码链接：[https://github.com/kingium/ESD\\_1506/blob/master/PSoC/Workspace/light\\_1506.cydsn/main.c](https://github.com/kingium/ESD_1506/blob/master/PSoC/Workspace/light_1506.cydsn/main.c)（含注释）

核心代码如下图：

```
118     while(1)
119     {
120         CyDelay(200);
121         count++;
122         PWM_WriteCompare((uint8)((float)lastFlux / 100 * 255));
123         buffer[0] = '0' + (lastFlux / 100 % 10);
124         buffer[1] = '0' + (lastFlux / 10 % 10);
125         buffer[2] = '0' + (lastFlux / 1 % 10);
126
127         // while(USART_1_CDCIsReady() == 0u); /* Wait till component is ready to send more data to the PC */
128         if (count % 2 == 0)
129         {
130             USART_1_PutData(buffer, 5); /* Send data back to PC */
131         }
132     }
```

由于待测位置的范围 0~100 是固定的，我采用了手动计算每一位数字的做法，而不是使用 C 语言的 `sprintf` 函数操作，以减少出 bug 的几率。

为了进一步降低 UART 的接收频率，我进行了分频处理：使用一个 `count` 变量记录循环的次数，每 2 次循环才更新一次串口（见代码 128 行），这样实际的 UART 发送间隔在 400 毫秒左右。

和之前的实验一样，我采用了模块化设计的思想。不过在 PSoC Creator 中，各个模块预先在 Top Design 中写好，然后由软件自动生成对应的 C 代码。这样就省去了自行编写模块代码的麻烦，直接在 C 语言中调用即可。

不过本实验只用到了 UART 发送数据，没用到 UART 接收数据，导致我在创新实验中使用 UART 接收数据的地方没能写对。这是后话。



## 1-2-4 运行效果

手指在 CapSense 触摸板的不同位置触摸，LED 会以相应亮度闪亮（亮度和手指的位置是线性变化的），同时显示在 LCD 屏幕上。刷新闻隔是 200 毫秒。

每个 2 个循环（400 毫秒），会通过 UART 串口向上位机以字符串的形式发送一次当前的 LED 亮度，电脑端接收后显示在串口助手中。

同时还保留了样例代码中对于 CapSense 两侧的两个按钮的用法。

## 1-2-5 实验收获

和 ARM 实验不同，这次实验中使用的 PSoC 采取的是图形化电路设计和自动生成底层代码的模式，更加易用。不过这种模式也有个问题，就是要保证硬件设计和 C 代码之间的版本同步。对于 Top Design 和引脚图的任何更改，都只有在重新生成 C 代码之后才能生效。如果忘记重新生成，那么 C 代码仍然是之前的老版本，这样很容易出现问题。

在使用串口的时候，我更加明白了 UART 串口正确的打开方式，包括需要正确设置时钟和波特率，以及合理使用缓冲区。在提高实验中这些技巧无足轻重，但是在创新实验中，我学到的这些技巧却帮我在短时间内写出了驱动舵机的代码。

在我看来，UART 串口可以类比成 C++ 中的 `iostream`，一种流式 IO 封装，只不过需要保证双方的波特率一致。

## 1-2-6 实验建议

建议在本实验中涉及一些上位机给 PSoC 发信号的内容，这样就可以提前练习使用 UART 串口接收数据的中断写法，为后面的创新实验提前做好准备。

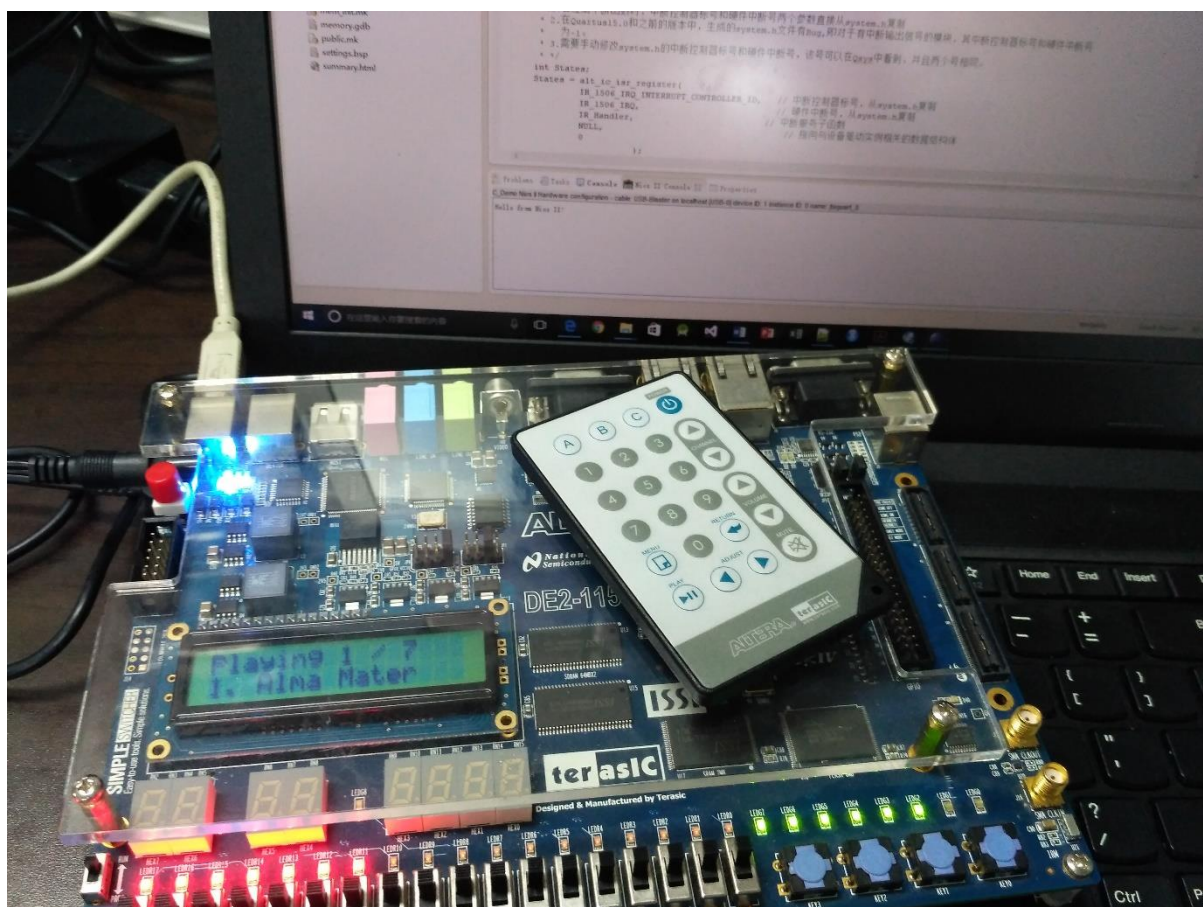
## 1-3 提高实验 3: FPGA 和 NIOS 实现遥控音乐播放器 UI

### 1-3-1 概况

实验时间: 8 月 29~30 日

验收时间: 8 月 30 日

实现功能: 在 FPGA 基础上搭建一个基于 NIOS 的软核微处理系统, 并在其上创建 C 语言工程 (在 Eclipse 中)。同时将红外模块、多个 PIO 模块 (控制 LED 亮灭)、LCD 模块加入 FPGA, 实现周围的外设, 并在 NIOS 中调用。利用红外遥控器上的大部分按键, 实现一个音乐播放器的前端界面, 功能包括歌曲信息显示、点歌、切歌、播放暂停、增大减小音量、静音、快进快退、使用多个 LED 模拟进度条和音量指示。



### 1-3-2 遇到的问题及解决

和很多同学一样，我遇到了很多问题，最大的一个莫过于“System ID 不匹配”。

我上网查找了很多资料，终于在 StackOverflow 上找到了一些答案。不过这些答案都不能解决问题。我按照教程完整重复了两三遍基础实验，仍然不能找到解决问题的办法。

在耽误了一天，我终于在深夜通过排除法找到了问题所在。虽说之前多次重复基础实验是无用功，但是我却对整个实验体系更加熟悉了，第二天我甚至能和其他同学完整说出整个体系的架构（引用我在网络学堂上的帖子）：

首先，我来谈谈我对于这个系统的理解。我们的系统有三大部分，Qsys、Verilog (Quartus)、C (Eclipse)。我觉得这三者之间的关系是，Qsys 是 parent，Verilog 和 C 是 Qsys 的两个 child。我们可以把 Qsys 看成是一个自动的代码生成工具，我们在 Qsys 里面指定各个模块的参数和接线，然后使用“Generate HDL”功能，就能将我们的设计图自动转化成 Verilog 代码（参见《基本实验指导》44 页），并放在“synthesis/\*.qip”文件中。在 Verilog 工程中包含该文件，并且按照“HDL Example”中的示例代码那样在顶层模块中，声明相应的子模块即可（《指导》58 页）。同理，我们在 Eclipse 工程中导入由 Qsys 生成的 .sopcinfo 文件，即可使用由 Qsys 自动生成的 C 代码（例如 system.h）。

因此，SystemID 和 timestamp 的作用也就出现了：上面提到，Verilog 和 C 代码必须由同一个版本的 Qsys 导出，才能保证不会不兼容。如果我们从 Qsys 中生成 Verilog 代码之后，对 Qsys 进行了修改但没有重新生成 Verilog 代码，

那么 Verilog 代码就还是基于旧版本的 Qsys；假若 Eclipse 工程由新版本的 Qsys 生成的话，那么就会版本不一致，导致 SystemID 或者 timestamp 不匹配。

我们在 Target Connection 界面（《指导》80 页）中，有个按钮叫做“System ID Properties”，在这里我们可以看到两种 System ID：Expected System ID 是 C 代码中的（位于 system.h 文件中的 #define），一般是个较小的数字；另一个则是 FPGA 中实际运行的，由 Quartus 写入。如果两者不匹配，可能的原因有：

1. Quartus 并没有将最新版的电路布局写入 FPGA 运行。排查办法是，全编译 Verilog 后，找到 .sof 和 .pof 两个文件，看看它们的修改时间是否是最新的。

如果它们的修改时间还在很早以前，说明 Quartus 在全编译的过程中，并没有更新这两个文件，自然在 FPGA 中实际运行的程序就不是最新的，甚至根本没有写入 FPGA。据老师说，如果 Quartus 没有正确破解，是可能出现编译时不刷新 .sof 文件的情况的，可以考虑换台装有正确破解的 Quartus 的电脑试试。

2. Verilog 和 C 程序不是由同一个版本的 Qsys 工程导出。解决方案是，确认 Qsys 文件设置正确后，重新由 Qsys 工程导出 Verilog 和 .sopcinfo 文件，并重新建立（注意是新建，不是重新导入原有的）Eclipse 工程。当在“System ID Properties”中看到匹配的 System ID 时，就说明我们配置成功了。

3. 在 Verilog 看来，整个 SOPC（包含 C 代码）都是一个子模块（《指导》58 页）。Verilog 只负责硬件电路的连接，而 SOPC 子模块内部则可以看成一个微型计算机，可以执行 C 语言程序，并通过定义在 system.h 中的接口，访问外设。Verilog 工程中定义的那些 wire 变量，就相当于连接线，将 SOPC 的外设端口，和 FPGA 开发板的物理端口（Pin Planner）连接起来，故需要那些

assign 语句（《指导》58 页）。至于 Verilog 和 C 内部的错误，应该在编译的时候都会报错，不至于引起 System ID 不匹配的问题。

最终我发现原因在于，由于我搞错了硬件的型号，我的工程没能正确更新 pof 和 sof 两个文件，导致最新版本的电路一直没能写入 FPGA。我重新配置了工程，正确设置了器件型号，然后就解决了这个问题。

其他的问题也有，但是在解决了上述大问题后，剩下的问题都迎刃而解。唯独需要考虑的是，如何批量控制多个 LED。理论上，GIO 总线支持 5 个地址线，最多控制 32 个 LED，而本实验仅用到 28 个，绰绰有余。但是这样需要改动已经封装好的 GIO 驱动模块。为了避免对已有代码的改动，我决定采用暴力法，在 Qsys 中声明 18 个 GIO\_PWM 以及 10 个 PIO，分别控制各个 LED。仅在上层的 C 代码中，将这么多指针收集在同一个指针数组中，实现批量管控。

```
60 unsigned int *pUser_GIO_PWM[18];  
61 unsigned int *pUser_PIO[8];
```

### 1-3-3 代码注释

原始代码链接：[https://github.com/kingium/ESD\\_1506/blob/master/Quartus/IR\\_LCD\\_1506/software/C\\_1506/main\\_1506.c](https://github.com/kingium/ESD_1506/blob/master/Quartus/IR_LCD_1506/software/C_1506/main_1506.c)（含注释）

首先是注册中断：

```

38 int IR_Init()
39 {
40     /*重要提示: 2017.07.12
41     * 1.在注册中断函数时,中断控制器标号和硬件中断号两个参数直接从system.h复制
42     * 2.在Quartus15.0和之前的版本中,生成的system.h文件有Bug,即对于有中断输出信号的模块,其中断控制器标号和硬件中断号
43     *   为-1。
44     * 3.需要手动修改system.h的中断控制器标号和硬件中断号,该号可以在Qsys中看到,并且两个号相同。
45     */
46     int States;
47     States = alt_ic_isr_register(
48         IR_1506_IRQ_INTERRUPT_CONTROLLER_ID, // 中断控制器标号,从system.h复制
49         IR_1506_IRQ, // 硬件中断号,从system.h复制
50         IR_Handler, // 中断服务子函数
51         NULL, // 指向与设备驱动实例相关的数据结构体
52         0
53     );
54     return States;
55 }
--

```

需要注意的是,由于系统的 **bug**,自动生成的代码中的中断号是错误的,需要手动更改成和 **Qsys** 中的一致。

为了代码简洁,我将所有初始化的代码封装到了一个 **Init** 函数中。

接下来的主循环是一个状态机,检测用户的按键,并做出相应的状态切换的指令:

```

89 int main()
90 {
91     int i;
92     init();
93     printf("Hello from Nios II!\n");
94
95     unsigned char KEY_CODE;
96
97     while (1)
98     {
99         if (Data_Ready == 1)
100         {
101             Data_Ready = 0;
102             KEY_CODE = (IR_Data >> 16) & 0xff;
103             switch (KEY_CODE)
104             {
105                 case 0x01:
106                     index = 0;
107                     process = 0;
108                     displayTitle(index);
109                     break;
110                 case 0x02:
111                     index = 1;
112                     process = 0;
113                     displayTitle(index);
114                     break;
115                 case 0x03:

```

后续代码从略。我使用了以下几个辅助的状态变量,表示音量、进度等:



```

60 unsigned int *pUser_GPIO_PWM[18];
61 unsigned int *pUser_PIO[8];
62
63 void init();
64
65 int process;
66 int volume;
67 int index;
68 int mute;
69 int isPlaying;
70 int powerOn;

```

由于需要操作多个 LED，我将相应指针统一放在数组里（代码 60~61 行）。

























歌曲信息和有关函数也放在一起：

```

72 unsigned char *titles[7] = {
73     "1. Alma Mater ",
74     "2. Rock and Roll",
75     "3. Rhythm Blues ",
76     "4. Classical ",
77     "5. Rap ",
78     "6. Jazz ",
79     "7. Folk "
80 };
81
82 void displayTitle(int index);
83 void displayProcess();
84 void displayVolume();

```

### 1-3-4 运行效果

Key	Key Code	Key	Key Code	Key	Key Code	Key	Key Code
	0x0F		0x13		0x10		0x12
	0x01		0x02		0x03		0x1A
	0x04		0x05		0x06		0x1E
	0x07		0x08		0x09		0x1B
	0x11		0x00		0x17		0x1F
	0x16		0x14		0x18		0x0C



按下数字 1~7 键，可切换到相应歌曲。红色 LED 将根据播放的进度，依次点亮。

按下 **channel** 上下键，可以切换歌曲。进度清零。

按下音量上下键调整音量，绿色 **LED** 亮起的个数代表音量大小。

按下左右键实现快进快退，每次进度变化 4 个红色 **LED**。

按下静音键和播放/暂停键，实现相应控制。

按下 0x17 返回键，当前歌曲回退至开头，重新播放。

### 1-3-5 实验收获

这次实验被一个 **System ID** 不匹配的问题折腾了一整天，不过我最终还是找到了解决办法，并把我的经验写成了帖子，放在网络学堂上。在排错的过程中，我对整个 **Qsys**、**NIOS**、**FPGA** 的体系越来越熟悉，甚至能不依赖于长达 90 页的教程自己完整搭建一遍。这大概就是我最大的收获吧。

我还震惊于 **NIOS** 和 **FPGA** 的精巧设计：

在 **FPGA** 看来，整个 **NIOS** 只是一个模块，需要在 **Verilog** 中使用下述语句调用：

```
149 kernel u0 (  
150     .clk_clk                (Clk_Core),           // clk.clk  
151     .new_sdram_controller_wire_addr (DRAM_ADDR),   // new_sdram_controller_wire.addr  
152     .new_sdram_controller_wire_ba   (DRAM_BA),     // .ba  
153     .new_sdram_controller_wire_cas_n (DRAM_CAS_N), // .cas_n  
154     .new_sdram_controller_wire_cke   (DRAM_CKE),   // .cke  
155     .new_sdram_controller_wire_cs_n  (DRAM_CS_N),  // .cs_n  
156     .new_sdram_controller_wire_dq    (DRAM_DQ),    // .dq  
157     .new_sdram_controller_wire_dqm   (DRAM_DQM),   // .dqm  
158     .new_sdram_controller_wire_ras_n  (DRAM_RAS_N), // .ras_n  
159     .new_sdram_controller_wire_we_n   (DRAM_WE_N), // .we_n  
160     .reset_reset_n              (reset_n),         // reset.reset_n  
161     .lcd_1506_conduit_end_0_export_data (LCD_DATA), // lcd_1506_conduit_end_0.export_data  
162     .lcd_1506_conduit_end_0_export_rw  (LCD_RW),    // .export_rw  
163     .lcd_1506_conduit_end_0_export_en  (LCD_EN),    // .export_en  
164     .lcd_1506_conduit_end_0_export_rs  (LCD_RS),    // .export_rs  
165     .lcd_1506_conduit_end_0_export_blon (LCD_BLON), // .export_blon  
166     .lcd_1506_conduit_end_0_export_on  (LCD_ON),    // .export_on  
167     .ir_1506_conduit_end_0_export      (IRDA_RXD),  // ir_1506_conduit_end_0.export_irda  
168  
169     .user_gpio_pwm_0_conduit_end_0_export (User_GPIO_PWM_0), // user_gpio_pwm_1506_0_conduit_end_0.export  
170     .user_gpio_pwm_1_conduit_end_0_export (User_GPIO_PWM_1), // user_gpio_pwm_1506_0_conduit_end_0.export  
171     .user_gpio_pwm_2_conduit_end_0_export (User_GPIO_PWM_2), // user_gpio_pwm_1506_0_conduit_end_0.export
```

但是在 NIOS 看来, FPGA 就是一个外设, 直接用 C 代码控制即可!

这样一种体系架构, 使得我们能兼顾 FPGA 在描述硬件电路方面的优点, 又能利用微处理器在执行代码方面的优点。我算是接触了硬件和软件协同工作的系统。

### 1-3-6 实验建议

首先建议助教将我们大家对于“System ID 不匹配”之类的问题的经验帖整理一下, 以供下届同学参考, 在更短时间内对于 NIOS 和 FPGA 系统有更全面的了解, 弄清 Qsys、Verilog、Eclipse 三个部分之间的版本依赖关系。

然后, 既然费了那么大劲做出这个系统, 就可以加上更多的功能。比如这次只要求做红外遥控的操作, 我们还可以加上读取超声传感器的模块。这种功能的添加不是给大家增添负担, 而是对综合实验和创新实验起到准备作用, 总体来看还是节省时间的做法。

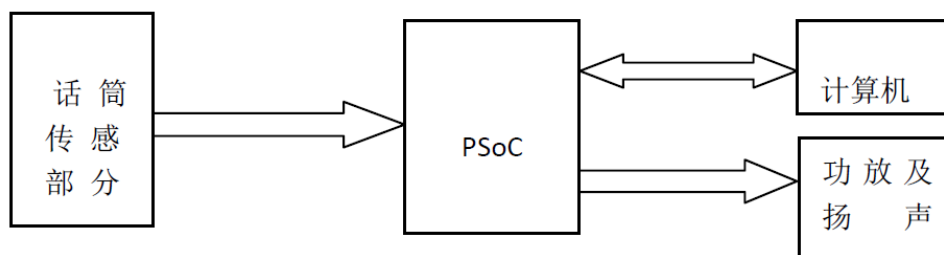
## 2 综合实验: 基于 PSoC 的语音存储回放系统

### 2-1 概况和方案设计

实验时间: 8 月 31 日、9 月 1 日

验收时间: 9 月 1 日 (周五) 下午

实现功能: 利用麦克话筒采集语音信号, 进行简单滤波后放大、A/D 转换等处理, 成数字语音信号, 并将数通过串口实时传送给计算机存储, 或者存入 SD 卡或 U 盘。当需要回放语音时, 将存储的数字通过 D/A、滤波功率放大等处理, 实现对语音信号的恢复回放。



(b) PSoC 实验方案

## 2-2 硬件原理图

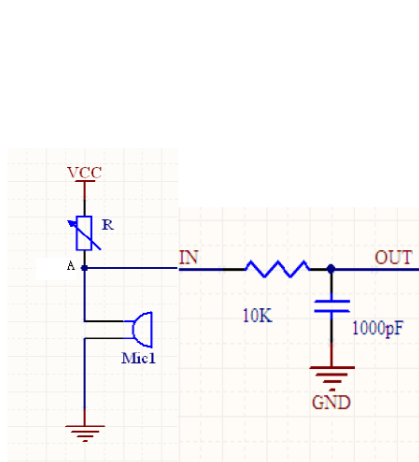


图 7 麦克风工作电路

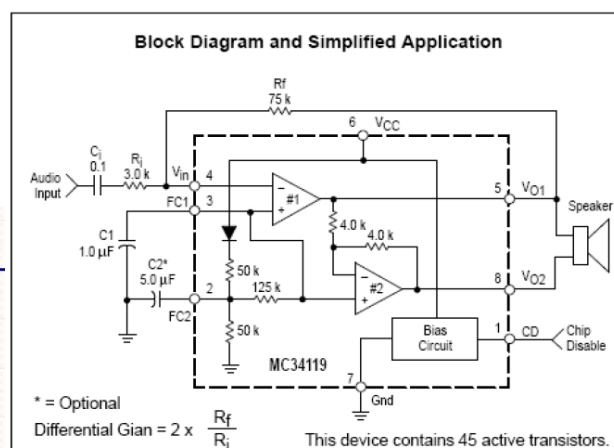
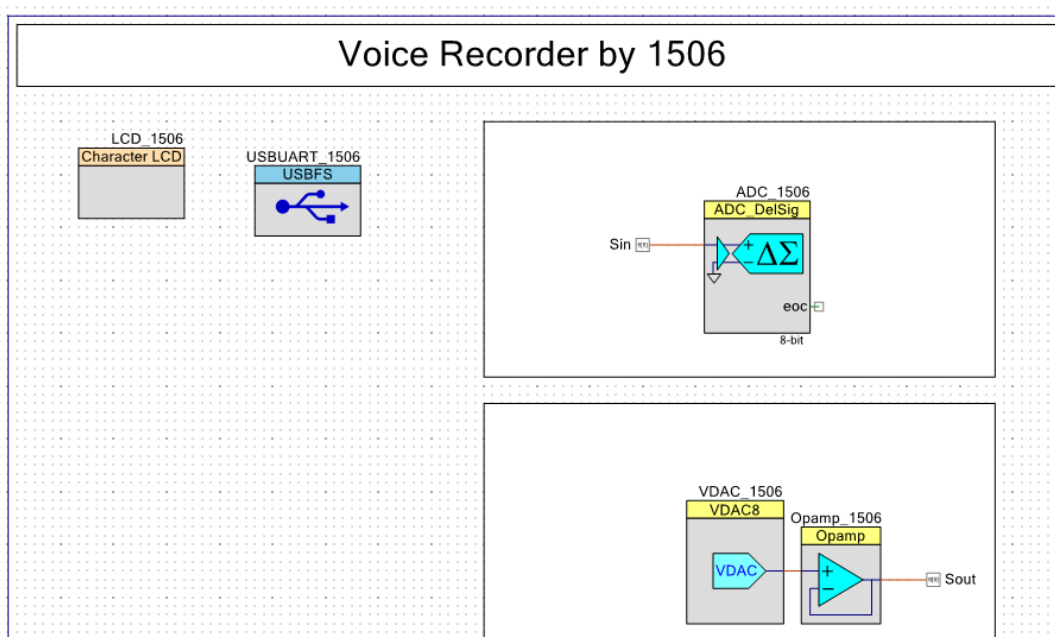


图 9 MC34119 工作电路

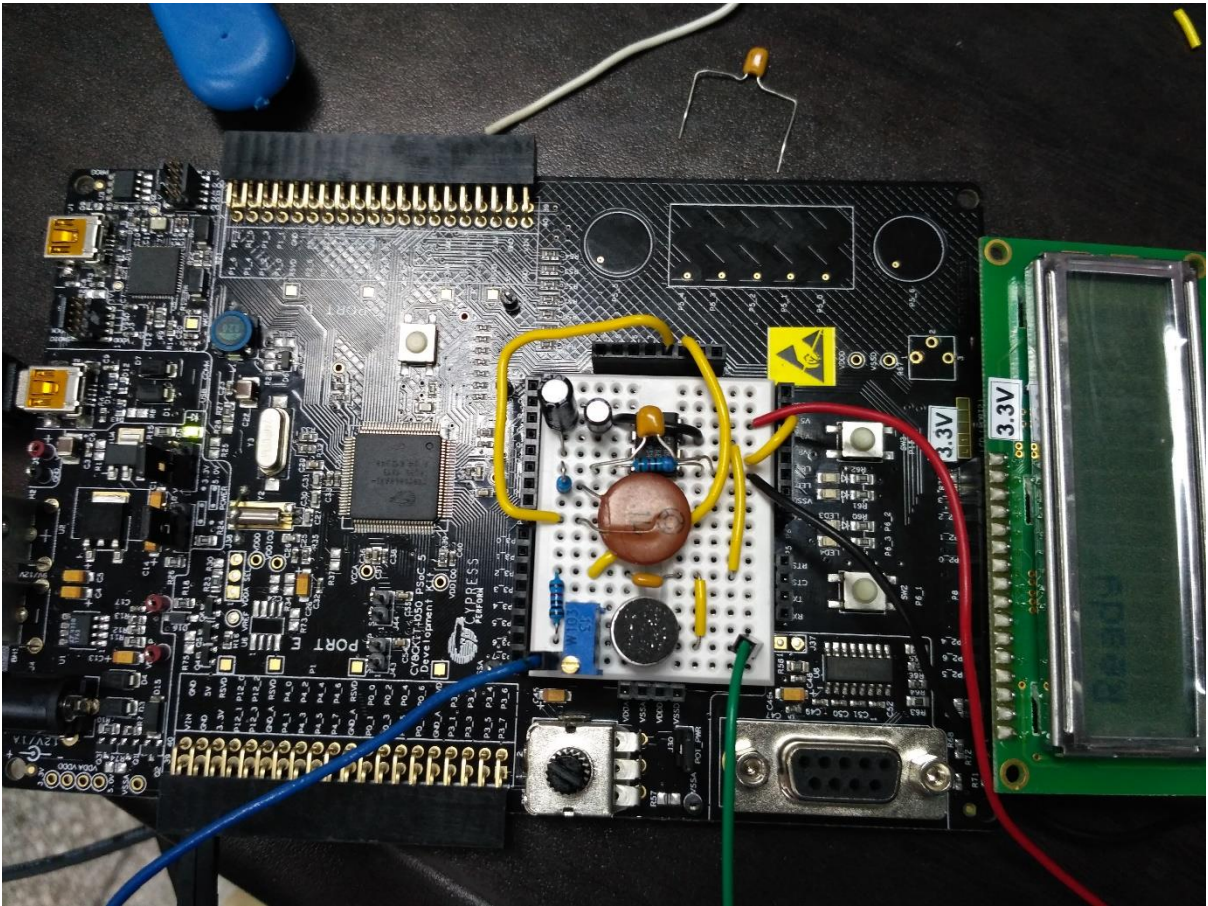


硬件连接是三部分的串联：麦克风工作电路、PSoC、MC34119 工作电路。

在 PSoC 中上, 使用一个 LCD 和一个 UART 模块, 分别用来显示工作模式信息, 以及和电脑双向通信, 收发指令和语音数据。

AD 转换使用一个简单的 DelSig 转换模块, 设置为非差分 8 位精度, 采样范围是 0~4V。之前经过示波器测试, 麦克风工作电路的输出电压一般在 2V 左右上下波动, 因此 AD 转换电路的采样范围是正常的。

DA 转换使用一个电压 DA 转换模块, 后接电压跟随器。DA 转换的精度是 8 位, 范围是 0~6V, 接入 MC34119 后可以正常工作, 音质清晰 (分模块调试)。



在上图之中, 硬件电路和 PSoC 的接口是自定义引脚 P6\_0 和 P6\_6。

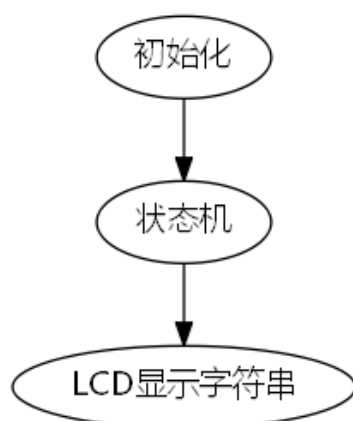
各个元件的引脚都做了剪短和弯曲处理, 以使走线精简, 减小噪声。麦克风直接连接在 PSoC 上, 扬声器则另行引出, 放在较远的位置。

## 2-3 软件模块图、状态图

软件仓库地址：[https://github.com/kingium/ESD\\_1506/tree/master/VoiceRecorder/Workspace](https://github.com/kingium/ESD_1506/tree/master/VoiceRecorder/Workspace)

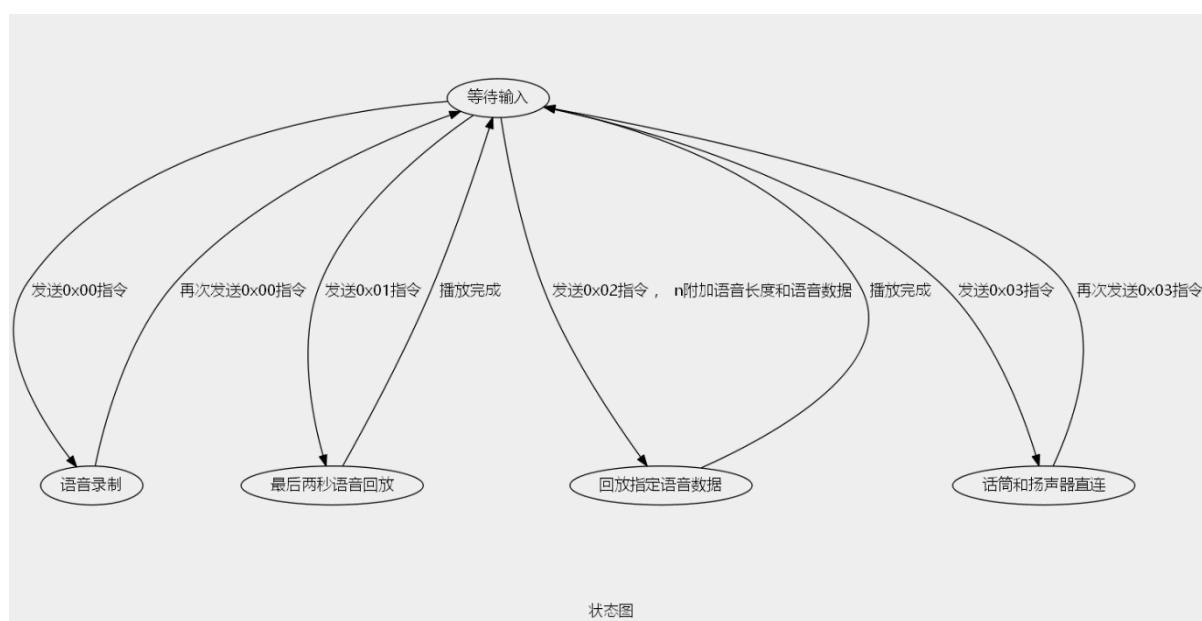
[orkspace](#)（含注释和自述文件）

软件模块图如下：



初始化和 LCD 的代码封装在各自函数中。

状态图如下：



状态机的主循环写在主函数的主循环中，状态变量是 `state`。



## 2-4 实验关键步骤说明

### 2-4-1 测试硬件直连

按照讲义的建议，首先应该测试硬件电路的正确性以及音质。我在按照讲义上的参考参数搭出麦克风工作电路和扬声器音频功放的电路后，先将两部分直接连接，测试发现音质较好，自言自语也不难分辨。唯独在扬声器靠近麦克风时，会因为反馈而出现振荡和啸叫。这时我按照讲义的建议，在音频功放的 4 号和 5 号引脚间加入了 33pF 电容，解决了啸叫的问题，且音量正常。

### 2-4-2 测试软件直连

之后我仍然在直连模式下进行测试，但是这次加入了 AD 和 DA 转换模块，在软件层面上进行直连，即将 AD 转换后的声音直接输送到 DA 转换。

（可以在 git 仓库中切换到我当时的 commit 版本，查看相应代码：[https://github.com/kingium/ESD\\_1506/commit/b73d3cb48181c8dbae1bce788d8437eb3cdc8026](https://github.com/kingium/ESD_1506/commit/b73d3cb48181c8dbae1bce788d8437eb3cdc8026)）

经测试，音质有大幅下降，出现噪音。我接上示波器进行诊断，最后成功减弱了噪音，参见后文。

### 2-4-3 编写状态机，实现功能

经过前面两步铺垫，这步就很简单了。定义好如下的编码格式：（摘自自述文件）

## 语音录制 ( 0x00 )

发送一个字节的十六进制数据 0x00，即可开始语音录制，录制时间不限，录得的数据会实时经由串口返回电脑，波特率为 8kbps，每一帧数据由一个字节的十六进制数据表示（范围是 0 ~ 255）。

录音时 LCD 显示 Recording。再次发送 0x00，即可停止录制。

## 最后 2 秒语音回放 ( 0x01 )

发送一个字节的十六进制数据 0x01，即回放最后录制的 2 秒语音。

## 回放指定语音数据 ( 0x02 )

首先发送一个字节的十六进制数据 0x02，其后紧跟 4 个字节的十六进制无符号数，代表将要发送数据的字节数（不含前述 5 字节）。随后跟着这么多个字节的无符号数，代表将要发送的语音信息，波特率为 8kbps，每一帧数据由一个字节的十六进制数据表示（范围是 0 ~ 255）。

样例：（发送 16 字节的语音数据，共计 16 帧）

```
02 00 00 00 0F 66 67 69 70 65 63 60 71 74 72 73 69 66 63 64 66
```

## 话筒和扬声器直连 ( 0x03 )

发送一个字节的十六进制数据 0x00，即可开始将话筒和扬声器的信号直连，时间不限。话筒录到的数据经过 AD 转换后，交由 DA 转换，并实时在扬声器上播放。

直连时 LCD 显示 Direct Access。

注：不要将扬声器直接对准话筒，以免引起啸叫。

然后使用一个主循环，编写状态机。

## 2-5 实验结果及效果

参见 [https://github.com/kingium/ESD\\_1506/tree/master/%E7%85%A7%E7%89%87](https://github.com/kingium/ESD_1506/tree/master/%E7%85%A7%E7%89%87)

开发板经由串口与电脑连接。在待机模式下（LCD 显示 Ready），使用串口助手发送如下格式的十六进制数据，实现语音的录制、传输、回放等功能。

## 2-6 实验演示操作方法

打开串口助手，波特率设置成 9600，然后将 PSoC 的靠中间的 USB 口连接到电脑。打开电脑的串口，输入指令（参见上述 2-4-3 节的格式约定），即可在 4 种功能之间切换。

## 2-7 遇到的问题及解决

### 2-7-1 音质太差

最大的问题是音质太差。

最开始我的电路在硬件直连时音质很好，但是软件直连时音质很差。逻辑上，这说明 AD 转换、PSoC 状态机、DA 转换之中，至少有一个模块引入了噪音。为了确定噪音来源，我使用 MATLAB 生成了一段两个频率正弦波的叠加（和声），重定向到 0~255 的范围，通过串口发送给 PSoC，发现听到的声音音质尚可。这样排除了后两个模块的问题，说明噪音主要来自录音的模块。

我在录音模块的代码中加入了如下限幅滤波算法：

$$y_{n+1}^- := \min(\max(y_{n+1}, y_n^- - 1), y_n^- + 1)$$

每两次迭代之间，幅值的改变量不能超过 1，即满幅的 1/256。经过这番修改，音质有了很大的改善，虽然还是略有噪音。

### 2-7-2 内存只够存储 2 秒

PSoC 自带的内存很小，如果按照我采用的 8000Hz 的采样率（最大采集到 4000Hz 的声音），只能存储 2s 的声音。为此，我采取的办法是，实时将声音回传到电脑上，电脑端不用担心存储容量的问题。至于开发板上，就只存储最后 2s 的声音，也只能直接播放最后 2s 的声音。播放更多的声音，必须通过电脑发送才可以。

### 2-7-3 滤波器参数的确定

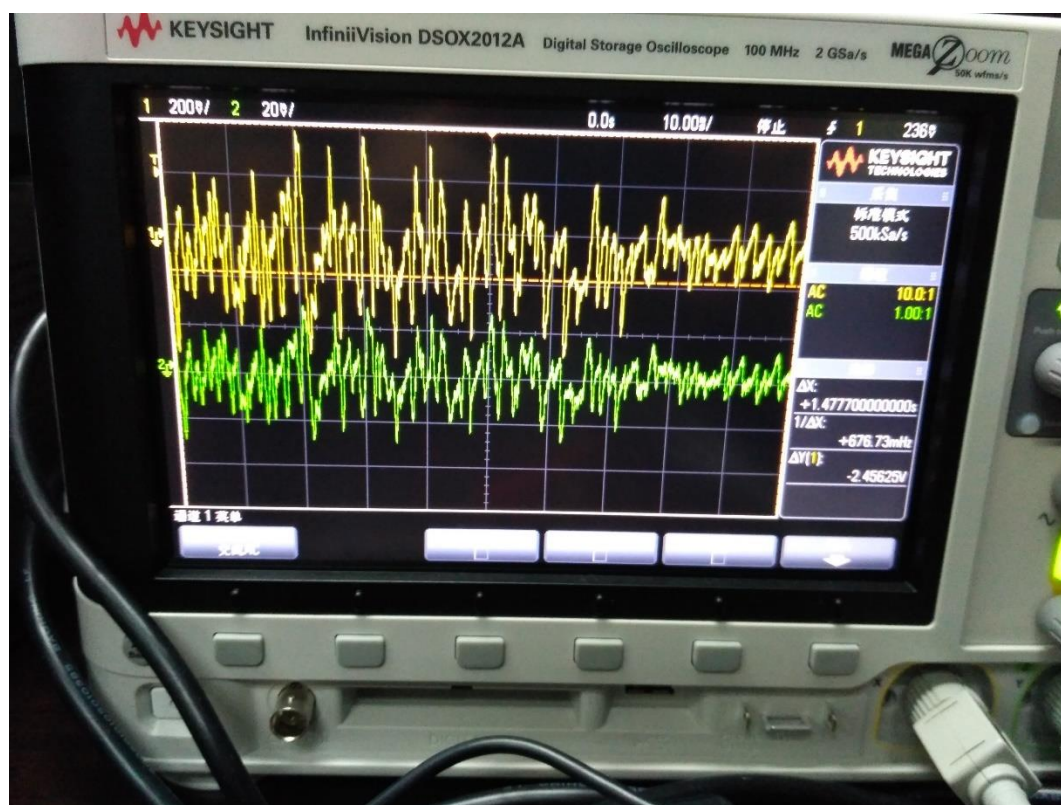
在麦克风工作电路中有高通滤波器。我把它改成了带通滤波器，并根据采样定理调整了参数，不影响人声（100~400Hz）的通过。

## 2-8 实验收获和建议

### 2-8-1 实验收获

这个实验需要实际搭接硬件电路，用到了很多模拟电子技术知识，例如滤波器、功放电路、电平匹配、静态工作点等。算是一个很综合的实验。

理论上，这个实验并不难。但是实际上，我的 2 天时间中，大部分用在了处理噪声上面。我采取了分模块调试的办法，锁定了噪声来源是录音模块，并针对性地设置了软件层面的限幅滤波。示波器的使用也给了我很大的帮助，比如：



黄色的是录音模块输出的模拟信号，绿色的是经过 AD 和 DA 转换后复原的模拟信号，两者的波形比较相似，证明了两个模块的搭建是基本正确的。而两者之间的相位差（延时），则是程序处理的耗时，这为确定主循环的周期提供了参考。

这个实验是上手容易，精通很难，提高音质的道路永无止境。

## 2-8-2 实验建议

建议老师在资料中加入一些减弱噪音的方法的名单（可不提供相应的资料），让我们自己探索。很多时候，我们找不到有用的资料，是因为我们连那些方法的名字都没听过。这个名单会对我们很有帮助的。

## 2-9 软件程序注释

主程序的代码：[https://github.com/kingium/ESD\\_1506/blob/master/VoiceRecorder/Workspace/Voice\\_Recorder\\_1506.cydsn/main\\_1506.c](https://github.com/kingium/ESD_1506/blob/master/VoiceRecorder/Workspace/Voice_Recorder_1506.cydsn/main_1506.c)（含注释）

核心代码（限幅滤波等）如下：

```
53         ADC_1506_IsEndConversion(ADC_1506_WAIT_FOR_RESULT);
54         Voltage = ADC_1506_GetResult8();
55
56         if (Voltage > lastVoltage + THRESHOLD)
57         {
58             Voltage = lastVoltage + THRESHOLD;
59         }
60         if (Voltage < lastVoltage - THRESHOLD)
61         {
62             Voltage = lastVoltage - THRESHOLD;
63         }
64         lastVoltage = Voltage;
65
66         buffer[i] = Voltage;
67         package[i % PackageSize] = Voltage;
```

定义的缓冲区解释如下：

```
9     uint8 buffer[BufferSize + 0xF];
10    uint8 package[PackageSize + 0xF];
--
```

缓冲区 `buffer` 表示音频序列，缓冲区 `package` 表示通过 UART 接收或发送的数据包，都是十六进制的 `uint8` 格式（0~255）。

## 2-10 参考资料

PSoC、ARM 综合实验说明 201608.pdf

PSoC 自带的各模块的 datasheet

<https://www.hackster.io/bmah/psoc-5lp-16-bit-and-24-bit-digital-filter-code-examples-ddcf9c>

## 3 创新实验：基于 Kinect 手势交互的自平衡小车

### 3-0 背景介绍<sup>1</sup>

#### 3-0-1 自平衡车（Self-Balancing Scooter）

自平衡车是以倒立摆控制系统为原型的一种新型交通工具，集嵌入式技术和工业设计于一身，是现代电子科学技术的产物。它可以在狭小空间中工作，也可以作为短程的代步工具，因其具有能耗低、无污染、轻便、高效等特点，愈发受到人们的喜爱和关注。

自平衡车打破传统交通工具的机械机构，由两轮甚至单轮结构组成，集单片机应用、电路设计、算法编程、信号处理、自动控制理论等内容于一身。



---

<sup>1</sup> 本小节的内容引用自同组的刘江同学的实验报告 5-1 节，不是本人的工作



### 3-0-2 微软 Kinect 体感摄像头

Kinect 是微软 2010 年起推出的一款全新的体感周边外设。

Kinect 中集成了 RGB 相机与红外景深相机，运用结构光技术（Light Coding）对测量空间进行编码，可对物体进行实时三维重建，并获取人体 20 个关节的实时状态和位置。



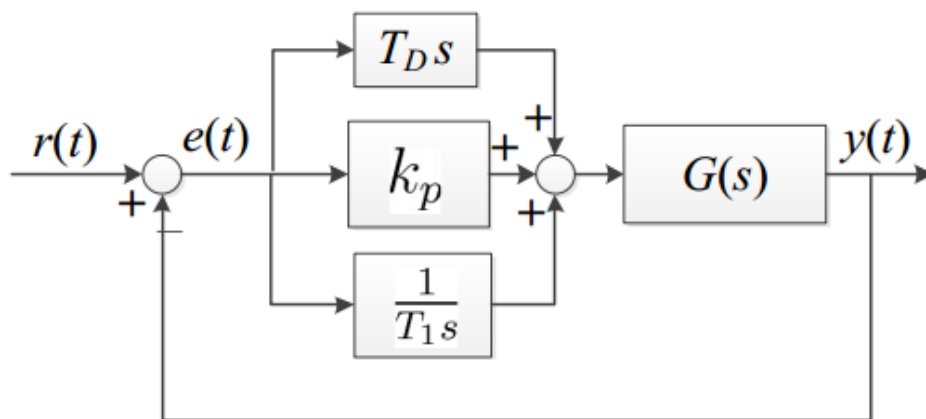
### 3-0-3 比例积分微分（PID）控制原理简介

平衡车的运动控制可由 PID 控制器实现。

在过程控制中，按偏差的比例（P）、积分（I）和微分（D）进行控制的 PID 控制器（亦称 PID 调节器）是应用最为广泛的一种自动控制器。它具有原理简单，易于实现，适用面广，控制参数相互独立，参数的选定比较简单等优点，是连续系统动态品质校正的一种有效方法。

PID 控制是一种串联校正控制器，通过积分控制克服系统静差，微分控制减小超调量、提高系统响应速度，综合了比例控制、积分控制、微分控制的优点。

PID 控制器的系统框图如下图所示：



PID 控制器的传递函数如下：

$$K(s) = k_p + T_D s + \frac{1}{T_I s}$$

PID 控制信号由下式给出：

$$u(t) = k_p e(t) + k_I \int_0^t e(\tau) d\tau + k_D \frac{de(t)}{dt}$$

PID 控制器设计的关键是根据系统参数，与所期望的系统静态和动态性能指标，得到比例项、积分项、微分项的系数 $k_p$ 、 $k_I$ 以及 $k_D$ 。

### 3-1 概况和方案设计

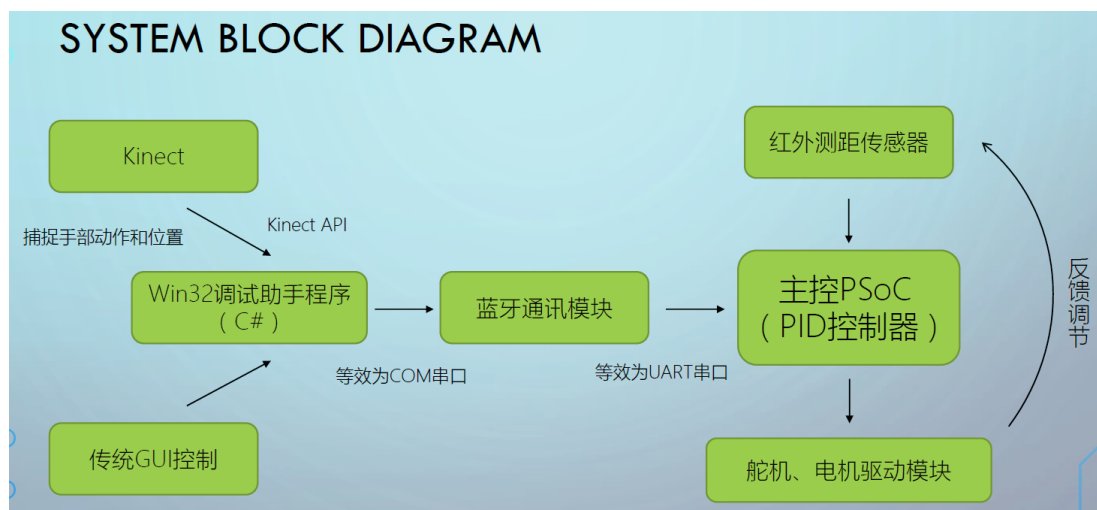
实验时间：9 月 3~8 日

验收时间：9 月 8 日上午

实现功能：主要实现了以下 3 个功能：1) 仅依靠一对红外测距传感器，实现一个原定保持动态平衡的两轮自平衡小车；2) 通过调节平衡位置的静差，在电脑上通

过蓝牙给小车发送指令，实现前进和后退。3) 将电脑连接到 Kinect 上，通过检测手势，实现对小车前后移动的控制。

总体模块图：（摘自答辩展示 PPT）本人负责下图中的 Kinect 模块、Win32 调试助手程序、蓝牙通讯模块、舵机驱动模块的搭建。



## 3-2 硬件原理图

实验开发的硬件平台是“创意之星”机器人套件：



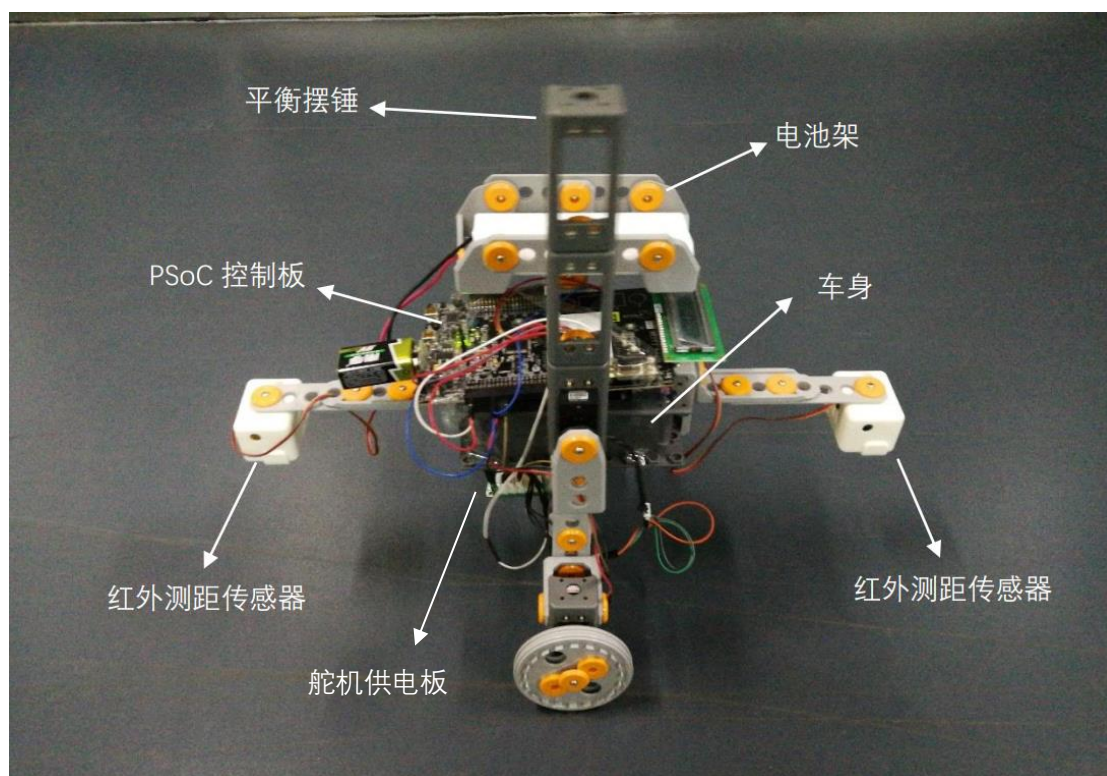
### 3-2-1 机器人搭建

机器人的搭建我们参考了“创意之星”官方附送的《InnoStar 实验指导书》的样例，搭建出一个两轮自平衡小车。为了安装红外测距传感器，我们加装了两个悬臂，置于车身前后，探测车身的倾角。同时在车体上方有一个平衡摆锤，用以辅助保持平衡，原因是车轮的电机扭矩不足以使小车保持平衡。

原本我们将电池放在靠近底盘的位置，但这样车体的固有振动周期过短，电机来不及反应，使小车难以保持平衡。最终我们参考网上的资料，认为需要抬高车体的重心，因此将电池放置在了车体的顶端。

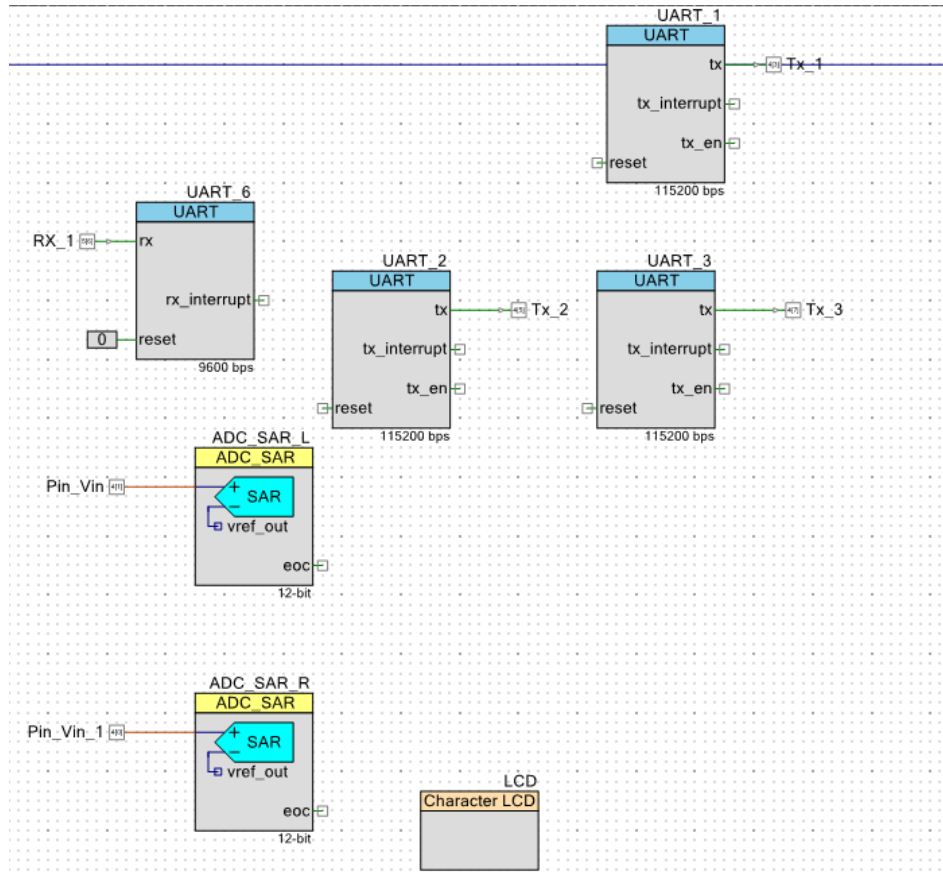
另外两块控制板（PSoC 主控、舵机供电板）则放置在车身靠近底盘的位置，它们重量不大，不会对重心造成太大影响。

最终车体设计如下。感谢孙竞耀和刘江两位队友拼出车体。



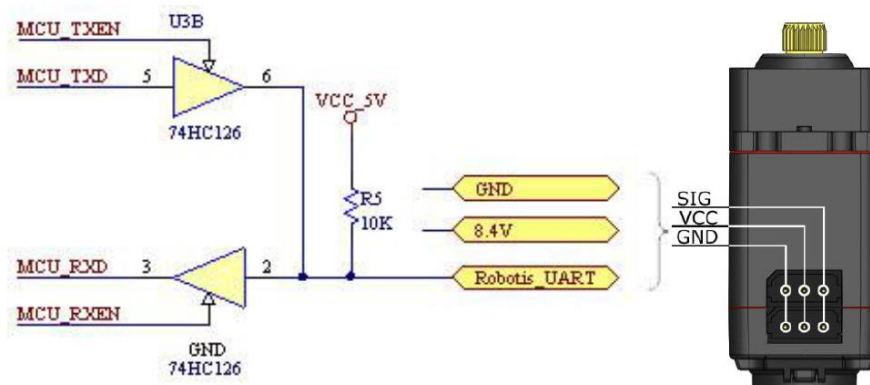
### 3-2-2 硬件电路设计

PSoC 顶层设计如下：



UART\_1、UART\_2、UART\_3 分别用来和三个舵机通信，UART\_6 用来和蓝牙模块通信，ADC\_SAR\_L、ADC\_SAR\_R 分别用来读取前后两个红外测距传感器。

注：在舵机驱动模块中，我并没有采用官方数据手册中推荐的电路：

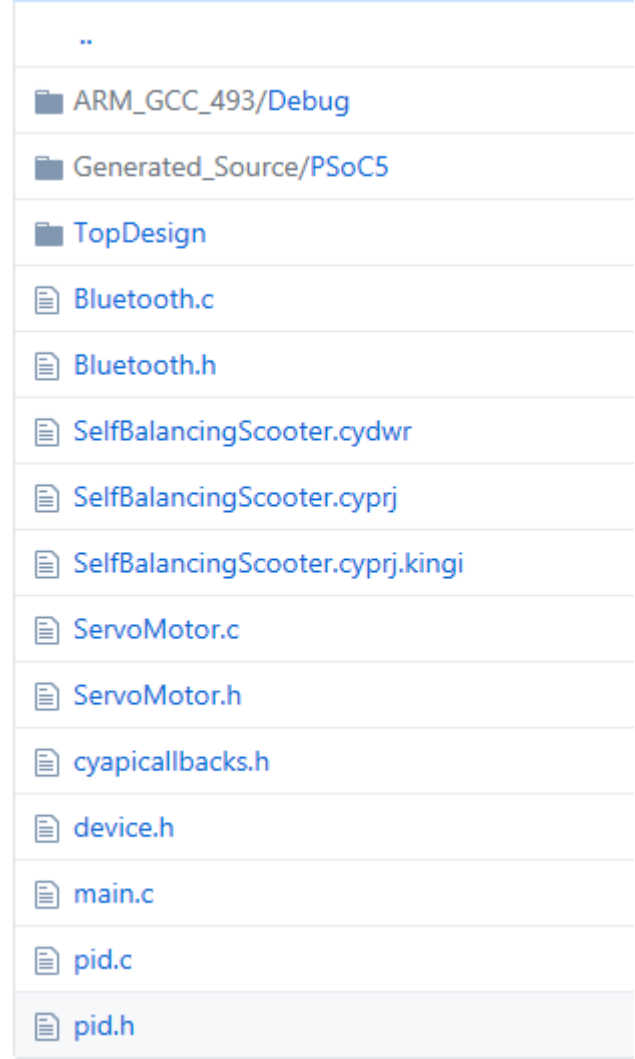


我按照助教老师编写的教程，直接将舵机的 SIG 信号引脚连接到供电板的 SIG 引脚。这样就只能实现单向的向舵机写入指令的操作，不能反过来读取电机的转速。这是一个遗憾，虽然在最后一天上午，刘江同学改用中断方式，成功实现了读取电机转速。

### 3-3 软件模块

#### 3-3-1 模块化设计

由于是团队合作，我们采用了模块化的设计思想：





各模块的信息如下：

文件	功能	作者
ServoMotor.h	舵机驱动模块	金帆
ServoMotor.c		
Bluetooth.h	蓝牙模块	金帆
Bluetooth.c		
pid.h	PID 控制器	刘江、孙竞耀
pid.c		
main.c	主函数	刘江、孙竞耀

### 3-3-2 核心 PID 代码

```
void PID()
{
    //计算当前偏差值，bias为目标倾角
    Ek = (ADC_L - ADC_R) - bias;
    //增量式PID计算
    Uk = Kp * (Ek - Ek_1 + Ki*Ek + Kd*(Ek - 2*Ek_1 + Ek_2)) + 0.95 * Ukk;
    Ukk = Uk;
    Ek_2 = Ek_1;
    Ek_1 = Ek;
    //在平衡阈值内不作调整
    if(Uk>=-range && Ukk <=range)
    {
        speed1 = speed2 = 0;
        angle3 = 512;
    }
    else
    {
        //限制输出范围
        if(Uk > 1023) Uk = 1023;
        if(Uk < -1023) Uk = -1023;
        speed1 = CompensateCoeff * Uk;
        speed2 = -CompensateCoeff * Uk;
        angle3 = 512 + (CompensateCoeff * Uk / TopMotorAngleCoeff);
    }
    //根据PID计算结果设置舵机速度
    SetMotorSpeed(_LEFT, speed1, 255);
    SetMotorSpeed(_RIGHT, speed2, 255);
    //设置摆锤角度
    SetServoPosition(_TOP, angle3, 0x03ff);
}
```

## 3-4 实验关键步骤说明

### 3-4-0 准备工作：传感器模块、舵机驱动模块、蓝牙模块

研读各自的 datasheet，编写程序，封装成 h 文件和 c 文件。

我所做的舵机驱动模块，主要参考了老师提供的《CDS5500 机器人舵机用户手册.pdf》文件，参考了其中的指令集，学会发送指令。

```
6 void SetServoPosition(uint8 id, uint16 position, uint16 speed)
7 {
8     // position ranges from 0x0000 to 0x03ff
9
10    uint8 checksum;
11
12
13    if (id == _TOP)
14    {
15        checksum = ID + 0x07 + 0x03 + 0x1e + (position & 0xFF) + (position >> 8) + (speed & 0xFF) + (speed >> 8);
16        checksum ^= 0xFF;
17
18        UART_1_PutChar(0xFF);
19        UART_1_PutChar(0xFF);
20        UART_1_PutChar(ID);
21        UART_1_PutChar(0x07);
22        UART_1_PutChar(0x03);
23        UART_1_PutChar(0x1e);
24        UART_1_PutChar(position & 0xFF);
25        UART_1_PutChar(position >> 8);
26        UART_1_PutChar(speed & 0xFF);
27        UART_1_PutChar(speed >> 8);
28        UART_1_PutChar(checksum);
29
30        CyDelay(2);
31    }
32 }
```

这份示例代码，演示了如何通过 UART\_PutChar 方法，向舵机发送指令，实现舵机位置的控制。这个函数封装起来，方便队友调用。

### 3-4-1 任务 1：原地平衡（Hold Still and Keep Balance）

视频链接：[https://github.com/kingium/ESD\\_1506/blob/master/%E7%85%A7%E7%89%87/Self%20Balance%20Scooter.mp4](https://github.com/kingium/ESD_1506/blob/master/%E7%85%A7%E7%89%87/Self%20Balance%20Scooter.mp4)

在数字控制器中，微分变为差分，积分变为求和。对于求和式做差分，得到如下的增量式 PID 算法（公式引用自刘江的报告）

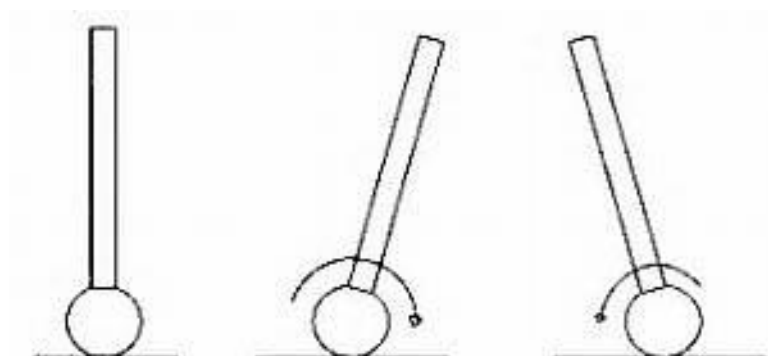
$$u(k) = K_p[e_k - e_{k-1} + K_i * e_k + K_d * (e_k - 2e_{k-1} + e_{k-2})] + u(k-1)$$

其中 $k$ 为采样序号( $k=0,1,2,\dots$ )， $u(k)$ 为第 $k$ 次采样时刻的计算机输出值，即 $e_k$ 为第 $k$ 次采样时刻输入的偏差值。 $K_p$ 为比例系数， $K_i$ 为积分系数， $K_d$ 为微分系数。

PID 调参花费了大半天，在 9 月 6 日下午成功实现原地平衡。

### 3-4-2 任务 2：前后移动 (Totter under Control)

我们的 idea 是，通过调节小车平衡时的倾角，让小车获得加速度，来控制其前后移动。如下图所示：



当小车前倾时，获得向前的加速度。当达速后，小车恢复竖直，即可保持大致的匀速前进。我们的程序中有一个 **bias** 变量，原本用来配平两侧的红外测距传感器。此处我们直接从蓝牙发送指令，改变 **bias** 的值，即可实时控制小车的前进和后退。

### 3-4-3 任务 3：通过 Kinect 手势操控 (Kinect in Command)

为便于 PID 调参，本人在电脑端使用 C# 制作了一个 PID 调参助手。而 Kinect 拥有一个 C# 的 API，可以很方便得调用：

```
Joint handRight = body.Joints[JointType.WristRight];  
Joint handLeft = body.Joints[JointType.WristLeft];  
  
if (handLeft.Position.Y > handRight.Position.Y)  
{  
    int temp = (int)((handRight.Position.Z - 0.9) * 40 + biasDefault);
```

这样就可以实现手势操纵了。我们设计的动作如下：



站立在 Kinect 前方大约 0.8 米的位置，使左手举高过右手，开始操纵。右手离 Kinect 越近，代表小车将向前加速。为保证小车不倒，小车向前加速的加速度是有限制的，因此当右手和 Kinect 的距离小于 0.5 米时，就不再起到控制的作用。反之右手离 Kinect 越远，代表小车向后加速。

### 3-5 实验效果或结果

请参见 PPT 中的视频。

### 3-6 本人工作总结

#### 3-6-1 蓝牙模块、舵机驱动模块

参见 3-4-0 节。

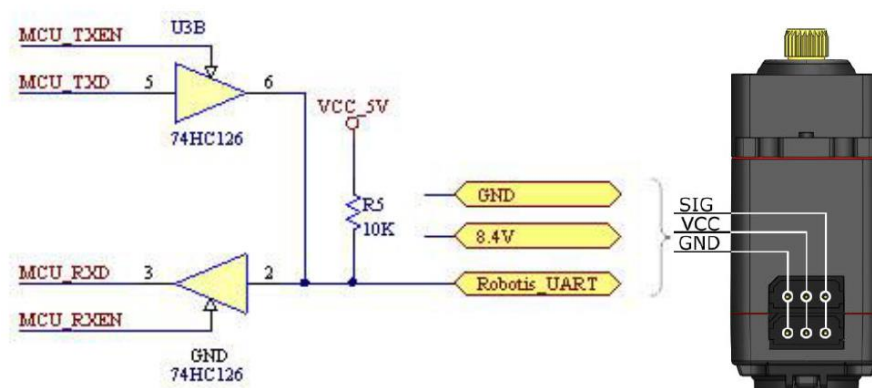
#### 3-6-2 电脑端的 Win32 程序 (C#)

[https://github.com/kingium/ESD\\_1506/tree/master/SelfBalancingScooter/Win32](https://github.com/kingium/ESD_1506/tree/master/SelfBalancingScooter/Win32)

## 3-7 遇到的问题及解决

### 3-7-1 无法读取舵机转速

按照 datasheet 要求，我们需要自行使用 74HC126 芯片搭出控制电路，将舵机的单数据口转换成双向的 UART 口，其核心是一个三态门：



如果不在硬件上制作三态门，直接在 PSoC 软件中将这个单口同时连接到 UART 的 Tx 和 Rx 上，就会出现线与的问题，报错“重定义”。

在 9 月 7 日上午，我们去购买了 74HC126 芯片，搭出了这个控制电路。仍然无法从串口读取出电机传回的信息，只能单向给舵机发送数据。

后来经过刘江同学的研究，发现问题在于，舵机在收到指令后，会立即回传相应的返回数据。必须手动检测波形图，才能捕获并读取这些数据。使用 UART 串口是会漏读这些信息，从而永远无法收到回传的数据，包含舵机位置和转速等信息。

很遗憾，由于时间不够，我们只采用了单向发指令的办法，从而失去了舵机转速这个重要的传感器。

不过我们只依靠红外距离传感器的读数，也成功实现了 PID 控制平衡！

### 3-7-2 电机扭矩不够

开始时我们的车体重心低，振动周期短，电机的反应速度跟不上。最后我们参考网上的资料，决定调高重心，解决了这个问题。

### 3-7-3 调参太费时

之前每次调参都要重新编译 PSoC 工程，小车断电，并重新写入程序，耗时至少 1 分钟。在 9 月 6 日，我写出了蓝牙模块，并应用上去。从此调参无需重新写入小车的程序，只需在电脑端更改数据，改动能即时生效。此举大大加快了调参的速度，仅不到一天，我们就调出了 PID 参数：

```
10 float32 Kp = 10.5;
11 float32 Ki = 0.09;
12 float32 Kd = 0.5; //0.48
13 float32 TopMotorAngleCoeff = 4;
14 int16 ADC_L_balance = 690;
15 int16 ADC_R_balance = 720;
16 //int16 bias = ADC_L_balance - ADC_R_balance;
17 int16 bias = -40;
```

轮子不够用？解决方案就是：自己造轮子！

## 3-8 软件程序注释

参见源代码：[https://github.com/kingium/ESD\\_1506/tree/master/SelfBalancingScooter/PSoC](https://github.com/kingium/ESD_1506/tree/master/SelfBalancingScooter/PSoC)（含注释）


## 3-9 实验收获

### 3-9-1 版本控制

我自己的代码利用 GitHub 做版本控制，这是我的提交记录（部分）：

Branch: master ▼

Commits on Sep 8, 2017

-  **ReleaseToTeachingAssistant** ...  
kingium committed 4 days ago
-  **bug fix: 自平衡车** ...  
kingium committed 5 days ago

Commits on Sep 7, 2017

-  **自平衡车 ( 蓝牙 + 低速移动 )** ...  
kingium committed 5 days ago
-  **远程PID调参 ( Bluetooth + Kinect )**  
kingium committed 5 days ago
-  **PID蓝牙远程调试助手**  
kingium committed 6 days ago

Commits on Sep 6, 2017

-  **半双工通信 ( 获取舵机位置和转速 )**  
kingium committed 6 days ago

一共有 28 次 commit，记录了我的每个改动和尝试。

### 3-9-2 调参的顺序

调参工作主要由刘江和孙竞耀同学完成，不过我也观察了他们调参的步骤。我们查找了网上的很多教程，询问上过线控的学长，得知正确的调参步骤是：先调 **P**，再调 **D**，最后调 **D**。先将参数调大，至出现低频扰动（**P**）或高频扰动（**D**），然后逐渐调小，使抖动减小。

具体的调参经验可以参考刘江和孙竞耀同学的实验报告。



### 3-9-3 更加深入地使用 C#

使用 C#自带的串口类 `SerialPort`，以及引用的 `Kinect` 接口，自己编写带有图形界面的程序，用于 `PID` 调参，及后续的蓝牙控制。代码量超过 300。



## 3-10 参考资料

StackOverflow（看得太多了）

《CDS5500 机器人舵机用户手册.pdf》

PSoC 中的串口模块自带的 datasheet

更多关于 `PID` 调参的参考资料，请参见刘江和孙竞耀的实验报告。