Digital Clock - Instructions 数字钟说明文档

> by 江天航 521021911101

File directory 文件目录

```
Digital Clock
|   README.md
|   readme.txt
└———Source Group 1
|   |   main.c
|   |   initialize.c
|
└———Source Group 2
    |   headers.h
    |   initialize.h
```

headers. h 包含各种所需要的库以及自定义的宏

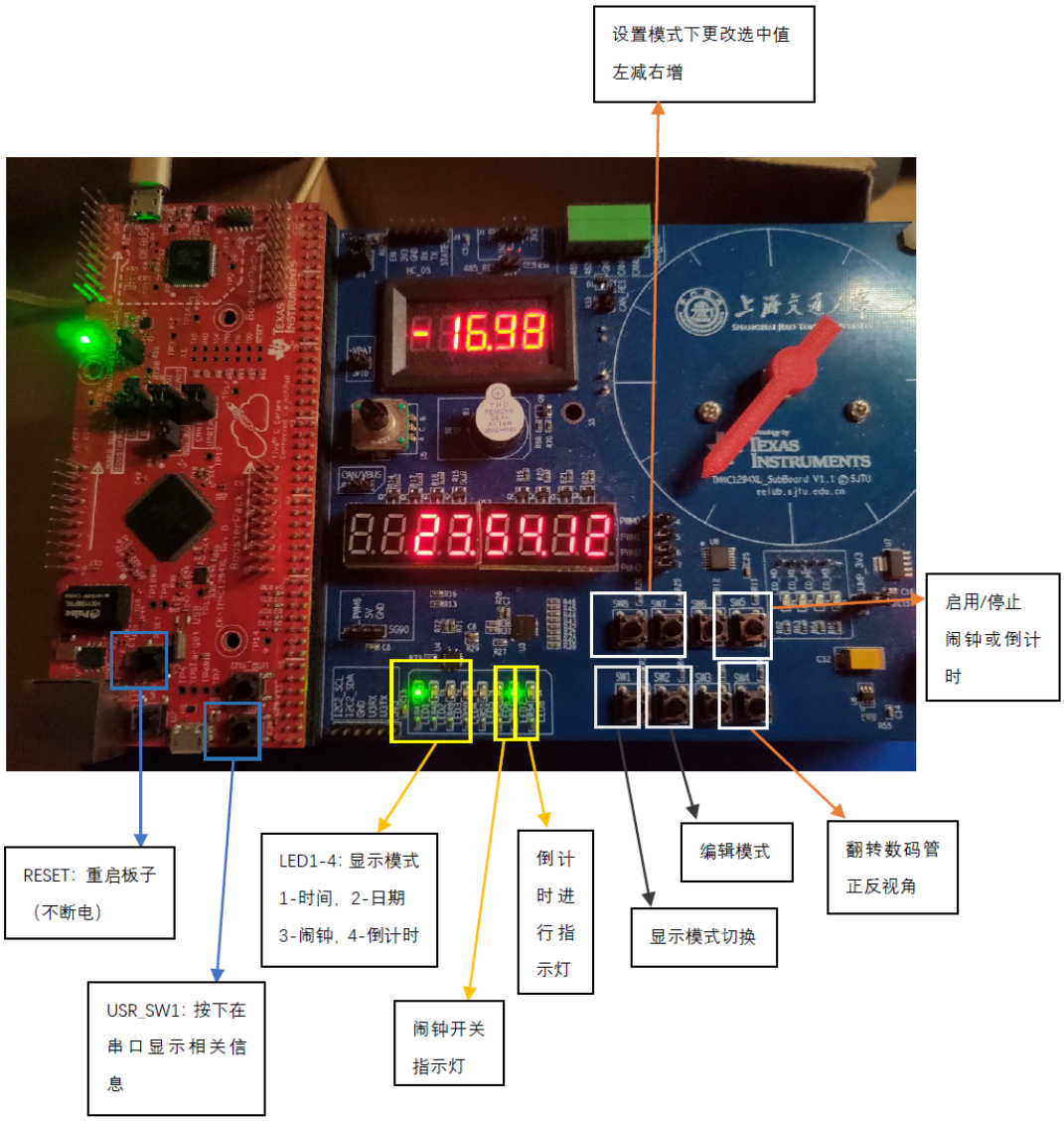
initialize. h, initialize. c 包含各个IO初始化函数

main. c 自编部分

Function list 功能列表

- 时间的显示及按键设置
- 日期的显示及按键设置
- 闹钟时间的显示及按键设置和使能
- 倒计时的显示及按键设置和使能
- 闹钟到点播放音乐铃声，数码管闪烁，可手动止闹
- 倒计时结束播放音乐铃声，数码管闪烁，可手动停止
- 开机画面：包括开机音乐、学号显示、LED闪烁和串口ASCII ART字符画
- RESET按键重启后（不断电），读取休眠模块内存中备份的时间日期闹钟和倒计时
- 红板USR_SW1：每次按键按下时，均在串口输出按下时及松开后的两次时间，以及按键持续时间，显示内容为秒及毫秒。
- 按键切换数码管正反视角
- 串口通信，详见 $README.md$ 或 $readme.txt$

Buttons & LEDs 按键及LED说明



- RESET键

不断电重启板子，重新播放开机画面。重启后会读取备份时间和当前RTC并进行设置，每次误差在0-2秒

- USR_SW1键

每次按键按下时，均在串口输出按下时及松开后的两次时间，以及按键持续时间，显示内容为秒及毫秒。

- SW1键

按下循环切换显示模式，如在设置模式中则退出设置

- SW2键

按下进入当前显示类的编辑模式。编辑模式下，被选中的部分数码管会闪烁，再次按下则切换选中下一部分

- SW4键

按下切换数码管正反视角，不更改任何其他状态

- SW5键

在闹钟或者计时器显示模式下，按下可开启/关闭闹钟或倒计时

- SW7键

在编辑模式下，选中部分的值+1，边界自动处理，内部关联数据实时更新（如更改2020-2-29中的2020至　2021，则自动跳变到2021-3-1）

- SW8键

在编辑模式下，选中部分的值-1，处理同上

Implementation 实现

基于模块化开发的思想，我的大致开发思路如下：先定义内部各个类，实现其类函数，再实现与外部的通信模块——分为串口协议通信和硬件交互通信，最后将这些模块整合到主函数中，逐一实现需求功能。这样构建的工程逻辑清晰，调试方便，代码可读性、可扩展性强（ps：如果没有“除必要的底层共用函数，其他的自编处理部分请放在main.c 文件中”这一规定，则可以将各模块划分到不同文件中，进一步厘清代码逻辑）

核心类

我首先定义并实现了三个核心类及其类函数

dgtclock_t 类

用以存储时间和日期

```
/* Digital clock type */
typedef struct
{
    int sec;      /* second, range 0~59 */
    int min;      /* minute, range 0~59 */
    int hour;     /* hour, range 0~23*/
    int mday;     /* which day in this month, range 1~31 */
    int month;    /* month range 0~11 */
    int year;     /* year */
    int yday;     /* which day in this year, 0~365 */
    int isleap;   /* whether leap year or not */
    int days[12]; /* built-in calendar */
} dgtclock_t;

/* Clock methods */
void clock_init(dgtclock_t *clock, int ss, int mm, int hh, int mday, int month,
int year);
void clock_update(dgtclock_t *clock);
int clock_set_date(dgtclock_t *clock, int mday, int month, int year);
int clock_set_time(dgtclock_t *clock, int sec, int min, int hour);
```

```
void clock_get_date(dgtclock_t *clock, char *buf);
void clock_get_time(dgtclock_t *clock, char *buf);
void clock_display_date(dgtclock_t *clock);
void clock_display_time(dgtclock_t *clock);
void clock_button_increase(dgtclock_t *clock, int incr, int ptr);
```

alarm_t 类

用以存储闹钟时间和状态

```
/* Alarm type*/
typedef struct
{
    int sec;
    int min;
    int hour;
    bool enable;
} alarm_t;

/* Alarm methods */
void alarm_init(alarm_t *alarm, int sec, int min, int hour);
int alarm_set(alarm_t *alarm, int sec, int min, int hour);
void alarm_get(alarm_t *alarm, char *buf);
void alarm_display(alarm_t *alarm);
void alarm_go_off(alarm_t *alarm, dgtclock_t *clock);
void alarm_button_increase(alarm_t *alarm, int incr, int ptr);
```

timer_t 类

用以存储倒计时时间和状态

```
/* Countdown type */
typedef struct
{
    int millisec;
    int sec;
    int min;
    bool enable;
} timer_t;

/* Timer methods */
void timer_init(timer_t *timer, int millisec, int sec, int min);
void timer_update(timer_t *timer);
void timer_display(timer_t *timer);
void timer_go_off(timer_t *timer);
void timer_button_increase(timer_t *timer, int incr, int ptr);
```

串口通信

接着我着手串口通信部分。第一个遇到的问题就是指令接收遗漏，检查发现是在UART0_Handle()中字符接受部分

```
while (UARTCharsAvail(UART0_BASE)) // Loop while there are chars in the receive FIFO.
```

如果板子从RX中读取字符比串口发送字符要快，那么会直接返回，导致后续命令丢失，于是我加了个延迟等待

```
while (UARTCharsAvail(UART0_BASE)) // Loop while there are chars in the receive FIFO.
{
    /* Read the next character from the UART and write it back to the UART. */
    c[0] = UARTCharGet(UART0_BASE);
    strcat(buf, c);
    if (c[0] == '\r')
        break;
    /* wait for the line to come to the end */
    if (!UARTCharsAvail(UART0_BASE))
        delay_ms(5);
}
```

解决了这个问题，我又发现在解析参数时用malloc分配空间会导致板子卡死。合理怀疑是 $Segmentation\ fault$ 导致的，于是我检查了项目自动生成的 $\mathrm{startup\_TM4C129.s}$ 汇编文件，在开头发现默认堆大小竟然是零

```
Stack_Size      EQU     0x00000200
                AREA    STACK, NOINIT, READWRITE, ALIGN=3
Stack_Mem       SPACE   Stack_Size
__initial_sp

Heap_Size       EQU     0x00000000
                AREA    HEAP, NOINIT, READWRITE, ALIGN=3
__heap_base
Heap_Mem        SPACE   Heap_Size
__heap_limit
```

将Heap_Size改至0x00010000，能够正常malloc分配空间

随后我通过两个串口函数实现了串口通信功能

```
/* Serial command functions */
int parse_command(char *cmd, int *argc, char *argv[]);
int execute_command(int argc, char *argv[]);
```

硬件交互

数码管的显示在核心类的各个display函数里已经实现，还剩令人头疼的按键交互

我参考了寝室和家里用过的数字钟，发现一个特点，就是它们的按键都比较少，复用性很强，并且都具备时间、日期、闹钟和闹铃等设置

为了处理方便，我抽象了十几个**按钮事件**，以及事件捕获和事件清空方法

```
/* Global button events */
volatile int BUTTON_EVENT_TOGGLE, BUTTON_EVENT_MODIFY, BUTTON_EVENT_CONFIRM;
volatile int BUTTON_EVENT_ADD, BUTTON_EVENT_DEC, BUTTON_EVENT_ENABLE;
volatile int BUTTON_EVENT_USR0_PRESSED, BUTTON_EVENT_FLIP;

/*  Event functions */
void events_catch(void);
void events_clear(void);

// macros in headers.h
/* Button ids corresponding to respective button events */
#define BUTTON_ID_TOGGLE 0
#define BUTTON_ID_MODIFY 1
#define BUTTON_ID_CONFIRM 1
#define BUTTON_ID_ADD 6
#define BUTTON_ID_DEC 7
#define BUTTON_ID_ENABLE 4
#define BUTTON_ID_FLIP 3
```

这样的事件抽象简化了开发，一方面，事件捕获函数只需要根据硬件IO来更新事件而不用关心事件处理，另一方面，主函数只需要把触发的事件拿来更新状态而不用关心事件是如何捕获的，

全局状态

为了记录当前显示模式、编辑模式、编辑指针、闪烁掩码等状态，我定义了一些全局变量，来给数字钟提供当前的状态指示

```
/* Global display mode
 * 0 - time
 * 1 - date
 * 2 - alarm
 * 3 - countdown
 * */
volatile int global_display_mode = 0;

/* Global modification mode
 * 0 - display mode
 * 1 - modification mode  */
volatile int global_modify_mode = 0;

/* Global modification pointer
 *   0 - undefined
```

```
 * 1~3 - correspond to respective position */
volatile int global_modify_ptr = 0;

/* Global blink mask vector*/
volatile uint8_t global_blink_mask = 0xff;

/* Global already indicator */
volatile int global_already = 0;

/* Global flip indicator */
volatile int global_flip = 0;

/* Buzzer enable indicator*/
volatile int buzzer_enable;
```

用户按下某个按键，事件被捕获，主函数就会改变其中某些状态，让数字钟在下一时刻根据改变行为

音乐播放

参考了样例PWN_Init()以及查阅了pwm相关资料，我了解到可以通过调周期来调节蜂鸣器频率，调节占空比来

调整响度，于是我写了几个函数来控制蜂鸣器播放

```
/* Buzzer functions */
void buzzer_on(int freq, int time_ms);
void buzzer_off(void);
void buzzer_music_nonblocking(int len, pitch_t notes[], int time[], bool set);
```

其中 buzzer_music_nonblocking() 可以接收由一个音高(Hz)数组和时值(ms)数组编码的乐谱来播放音乐，而且还利用了Systick_Handler实现了并发播放，即音乐播放时数字钟仍然可以进行时间显示修改等操作。

（比如我将开机音乐时长设得比开机闪烁动画要长，那么在动画结束但音乐还未结束的这段时间，是可以进行模式切换和编辑的）

休眠存储

查阅官方文档后发现板子有Hibernation module，可以实现断电后数据保存，但可惜的是，在询问学长和查阅资料后发现这个模块需要在板子的VBAT 引脚上接上后备电池才能保证数据休眠存储，否则只能在板子上电情况下按RESET来达到这个效果

我封装了两个休眠相关的函数，实现了RESET后重读以及往休眠模块里更新当前时间日期等值

```
/* Hibernation functions */
void hibernation_wakeup_init(dgtclock_t *clock, alarm_t *alarm, timer_t *timer);
void hibernation_data_store(dgtclock_t *clock, alarm_t *alarm, timer_t *timer);
```

其中 hibernation_data_store() 每隔两秒进行一次，因为调试发现官方库提供的 HibernateDateSet() 会花费较长时间将数据从内存写入休眠模块，如果每秒实时更新，会导致来不及等待上一次完成写入

Experiences 心得体会

整个项目相对比较简单，写下来没有什么大的阻碍，主要bug还是出在硬件调试方面，因为没用gdb进行单步调试，于是我通过在怀疑出错的地方用串口返回信息，以此来精确定位错误位置，这是很朴素的调试思路

因为实现功能较多，代码量相当大，非常考验模块化开发技巧和对工程逻辑的整体把握，才不至于写出一堆难以维护的屎山