State Machine

ADuC834, black board

Mateusz Szumilas, PhD Eng.

1. Goal

To implement a countdown timer whose timeout can be adjusted from 1 to 20 seconds.

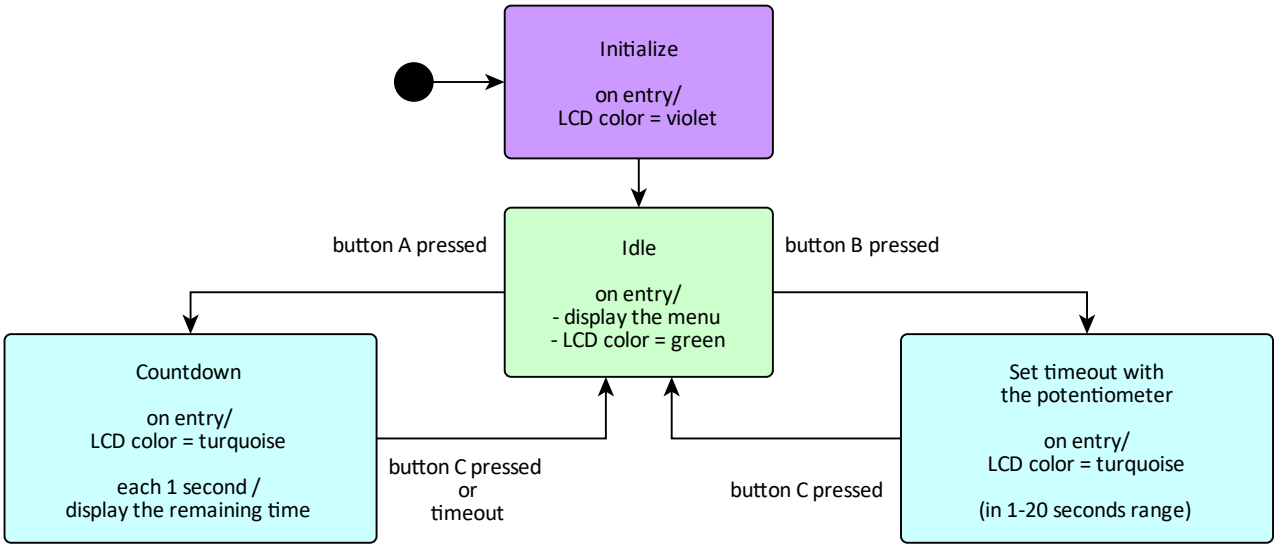
The solution should be based on the state machine design pattern.

2. Description

A state machine is an implementation of a state diagram. The state diagram presents the operating **states** of the system and possible **transitions** between them. The transitions may be unconditional or conditional.

There are multiple possible implementations of the state diagram. This exercise builds upon a solution based on a set of functions. Each function implements the operation of the respective state. A simple mechanism based on an **enumerated variable** listing all possible states and an **array of function** pointers is introduced for continuous execution of the state machine and transitioning between states.

A state diagram to be implemented is shown below. The execution begins from the black circle. You can freely decide which physical buttons on the development board will be used as the A, B, and C buttons for transitioning between the states.



There are four states of operation:

- **initialization** – the system is configured here;

  a fixed delay of 3 seconds for simulation of the initial system workload;

- **idle** – the state in which the user interface may be handled;

  information about the function of the A and B buttons should be displayed (according to the state diagram);

- **timeout setting** – when the application is in this state, the timeout should be modifiable by the user in the range from 1 to 20 seconds, based on the results of analog-to-digital conversion of the voltage at the potentiometer; the current timeout setting should be displayed along with information about the function of the C button;

- **countdown** – in this state, the countdown is performed; after each 1 second, the remaining countdown time is displayed; information about the function of the C button should be displayed.

Set the LCD backlight color according to the state diagram when entering each state.

3. Implementation

Note: Use of Keil µVision 5 IDE (with Cx51 compiler) is assumed in this handout.

3.1. Provided modules

Use the following files in your solution:

- LCD communication module (*lcd_comm.c*, *lcd_comm.h*);
- a header with type definitions for fixed-width integers (*intx_t.h*);
- a header with hardware-specific definitions for the lab board (*lab2_board.h*);
- a state machine template *sm_main.c* (described in the subsequent subsections).

Place these in the project directory, and remember to add the .c files to your project tree.

The LCD module requires a millisecond-delay function **LCD_DelayMs()** to be defined in your project. Its declaration is provided in the *lcd_comm.h* header and starts with the **extern** keyword, which means that it is defined outside this source file:

```
extern void LCD_DelayMs (uint16_t ms_cnt);
```

The definition of this function should appear, e.g., in sm_*main.c* (or any other file that belongs to your project).

3.2. System timer

You will use a millisecond system timer to measure time in this application. In the template, the **system_timer** variable is used as a system counter incremented every millisecond. To ensure that the **system_timer** variable is incremented every 1 ms, one can use an interrupt triggered by an overflow of the Timer 2, configured in the mode with automatic reloading of the initial value.

A function that handles the Timer 2 interrupt increments the **system_timer** variable and resets the counter overflow flag (which requires software reset).

In the case of the Cx51 compiler used in the µVision environment, to use a function as a handler for a specific interrupt, the **interrupt attribute** is added at the end of the function header, also specifying the interrupt number (interrupts are numbered from 0, according to their address sequence in the interrupt vector; for Timer 2 counter it is interrupt number 5 - see the table on the right).

The implementation of the function that will be used as the Timer 2 interrupt service routine is shown in Listing 1.

Interrupt Number	Address
0	0003h
1	000Bh
2	0013h
3	001Bh
4	0023h
5	002Bh
6	0033h
7	003Bh
8	0043h
9	004Bh
10	0053h
11	005Bh
12	0063h
13	006Bh

Keil C51 Help

Listing 1

```
1.  /* a millisecond system timer */
2.  volatile long int system_timer;
3.
4.  void IsrTimer2 (void) interrupt 5
5.  {
6.    system_timer++;
7.    TF2 = 0;
8.  }
```

The **system_timer** variable of `long int` type is created by the Cx51 compiler as a 32-bit one. Therefore, our 8-bit MCU won't be able to read its value in one operation. This may lead to a situation in which the readout becomes corrupted due to being interrupted by Timer 2 overflow, which results in modification of **system_timer** in the **IsrTimer2()** handler. To avoid this risk, you will use a function that disables interrupts while reading the variable (Listing 2).

Listing 2

```
1.  long int SysTimerGet(void)
2.  {
3.       long int current_timer;
4.       EA = 0;
5.       current_timer = system_timer;
6.       EA = 1;
7.       return current_timer;
8.  }
```

In the template, the **Timer2Init()** and **InterruptsInit()** functions are defined. You will use these to initialize the Timer 2 and the interrupt system (Listing 3).

Listing 3

```
void Timer2Init (void)
{
/*****************************************
TODO:
    configure the Timer 2 to overflow each 1 ms,
    use the mode with automatic reload
*****************************************/
}

void InterruptsInit (void)
{
/*****************************************
TODO:
    configure the interrupt system – required for
    the system timer
*****************************************/
}
```

3.3. State machine

The state machine implementation in *sm_main.c* starts with the state functions prototypes (Listing 4). For readability, these are named according to the names of the states.

Listing 4

```
void StateInitialize(void);
void StateIdle(void);
void StateSetTimeout(void);
void StateCountdown(void);
```

Next, an array of state function pointers (a *state array*) is defined. Also, an enumerated type *state_name_t* is defined and subsequently used to define variables for indexing the state functions array. It is essential to check if the order of the elements in the *enum* type agrees with the order of the respective functions in the state functions array (Listing 5). The **current_state** variable sets the currently running state.

Listing 5

```
/* an array of pointers to state functions */
void (*state_array[])() = {StateInitialize, StateIdle,
StateSetTimeout, StateCountdown};

/* a definition of an enumerated type
   used for indexing the state_array */
typedef enum {ST_INITIALIZE, ST_IDLE, ST_SET_TIMEOUT, ST_COUNTDOWN}
state_name_t;

/* a variable specifying the current state */
state_name_t current_state;
```

The definitions of the state functions and their partial implementations are shown in Listing 6.

Listing 6

```
/* timeout for the countdown value in milliseconds */
static long int timeout_set;

void StateInitialize (void)
{
     /* set the LCD backlight to violet */
     PORT_LCD_LED &= ~((1 << LCD_RED) | (1 << LCD_BLUE));
     Timer2Init();
     InterruptsInit();
     LCD_Init();

     LCD_Byte(0, LCD_CLEAR);
     LCD_SendString("Init...");

/*****************************************
TODO:
1.  call the ADC initialization function
2.  wait 3 seconds
*****************************************/

    /* transition to the next state occurs after calling
       its function for the first time */
    StateIdle();
}

void StateIdle (void)
{
    /* check for entry */
    if (current_state != ST_IDLE)
    {
        current_state = ST_IDLE;
/*****************************************
TODO:
1.  display the menu
2.  set the LCD backlight to green
*****************************************/
        return;
    }
```

Listing 6

```
41.
42.     if (BUTTON1_CHK)
43.     {
44.         StateSetTimeout();
45.
46.     } else if (BUTTON2_CHK)
47.     {
48.         StateCountdown();
49.     }
50. }
51.
52. void StateCountdown (void)
53. {
54.     static long int time_start, time_1s_ref;
55.
56.     /* check for entry */
57.     if (current_state != ST_COUNTDOWN)
58.     {
59.         current_state = ST_COUNTDOWN;
60.
61.         /* store the value of the system timer
62.            from which the countdown begins */
63.         time_start = SysTimerGet();
64.         /* a reference for 1-sec timeouts */
65.         time_1s_ref = time_start;
66.
67.         LCD_Byte(0, LCD_CLEAR);
68.         LCD_SendString("counting..." );
69.         LCD_Byte(0, LCD_LINE_FOUR);
70.         LCD_SendString("Btn C: idle");
71.
72.         return;
73.
74. /*****************************************
75. TODO:
76. 1.  conditionally transit to the idle state
77.     when the button C is pressed
78. 2.  set the LCD backlight to turquoise
79. *****************************************/
80.
81.     /* check if a 1-sec timeout expired
82.        (used for 1 Hz display update) */
83.     if (SysTimerGet() - time_1s_ref > 1000)
84.     {
85.         time_1s_ref += 1000;
86. /*****************************************
87. TODO:
88.     display the remaining time (after each 1 second)
89. *****************************************/
90.     }
91.
92.     /* check if the timeout expired */
93.     if (SysTimerGet() - time_start > timeout_set)
94.     {
95.         StateIdle();
96.     }
97. }
98.
99.
100.
101.
102.
```

Listing 6

```
103. void StateSetTimeout (void)
104. {
105. /*****************************************
106. TODO:
107. Implement the timeout adjustment:
108. 1.  on entry, set the current state to ST_SET_TIMEOUT
109. 2.  change the „timeout_set” variable value in the range
110.     from 1000 to 20000 milliseconds based on the readings
111.     from the ADC channel connected to the potentiometer
112. 3.  when the user presses the C button,
113.     transit to the idle state
114. *****************************************/
     }
```

In the code shown above, the state machine is implemented only partially. You should complete the implementation. Things to be done are summarized in the comments, which start with TODO headings. Try to adhere to the description, but if you find it reasonable to introduce some modifications, please feel encouraged to do so.

When the state functions are completed, the state machine is easy to run in the application. Only two operations are necessary: set the initial state and continuously execute the current state from the state function array. Implementation in the body of the **main()** function is shown in Listing 7, in lines 3 and 9.

Listing 7

```
1.  int main (void)
2.  {
3.    current_state = ST_INITIALIZE;
4.
5.    while (1)
6.    {
7.      /* current state function is executed from the
8.         state function array */
9.      state_array[current_state]();
10.   }
11. }
```

What should the lab report include?

- The final version of your code with appropriate comments.

End of Exercise