General rules and recommendations

Writing and running C programs for the EdSim51 simulator:
- Each program must start with "`#include <8051.h>`".
- It is recommended that function `main` is of type `int` and returns value 0 in the end (just like in the case of the GCC compiler).
- The core of function `main` is usually an infinite loop, which executes repeatedly the same operations. Various initializations can be performed before the infinite loop, if necessary.
- It is recommended not to use many variables (either local or global), as the microcontroller memory is not very large.
- When global variables are declared, their initial values must not be set at declaration time. Instead, initialization should be done explicitly at the beginning of function `main`.
- The four general-purpose ports of the microcontroller can be accessed, either in reading or writing, by using the predefined names `P0`, `P1`, `P2`, `P3`.
- Individual bits of the ports can also be accessed; e.g., `P0_3` denotes bit 3 of port `P0`.
- The ports mentioned above are meant for communication purposes only and should not be treated as memory locations. In other words, for storage and processing operations (arithmetic, logic, etc.) you declare and use memory variables, which are sent to/read from the ports only when communication is necessary. Never write code such as `P1=P1+5;` (i.e., directly processing the value of a port). Instead, use variables and write something like `a=a+5;P1=a;` (processing is made upon variable `a`, which can later be sent to the appropriate port when necessary).

Lesson 1

Example
Write a C program that continuously alternates the LEDs which are turned on: the odd-indexed ones (i.e., 1, 3, 5, 7) and the even-indexed ones (i.e., 0, 2, 4, 6).

```
#include <8051.h>
int main()
{
  while(1) {
    // LEDs are connected to port P1
    // each LED is turned on when value 0 is applied to its corresponding bit in P1
    P1=0b10101010; // turn on LEDs 0, 2, 4, 6
    P1=0b01010101; // turn on LEDs 1, 3, 5, 7
  }
  return 0;
}
```

Problem 1
Write a C program that turns the LEDs on and off, based on the values of switches SW0...SW7 (inputs from the user).

The switches are connected to port P2, so the status of each switch can be determined by reading port P2. It is also possible to read individual bits in P2 (e.g., P2_0), but that is not useful for the current problem.
Each switch provides value 0 when closed and 1 when open.
In conclusion, the simplest way to solve the problem is to read the value from port P2 (the switches) and then copy it to port P1 (the LEDs). Remember, however, this must be done continuously, inside the infinite loop.

Problem 2
Write a program that turns on each LED individually, one LED at a time. The sequence of LEDs turned on is thus: 0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 0 - ...

Lesson 2

Problem 1
Write a C program that controls the motor, by making it continuously change the spinning direction. That is, it spins clockwise for a time T, then counter-clockwise for another time T, and so on.

In order to control the motion of the motor, one must write the right values to bits 1 and 0 of port P3:
- If the bits have the same value (either 0 or 1), the motor stops.
- If the bit values are different, the motor spins; the spinning direction depends on which bit has value 0 and which has value 1.
- In conclusion, the values that must be written to port P3 are 1 and 2, respectively; they correspond to combinations 01 and 10, respectively, on bits P3_1 and P3_0.

Problem 2
Write a program that controls the spinning direction based on the value of switch SW0. Remember that the switches are connected to port P2.

Problem 3
Write a program that, in addition to the previous one, also decides whether the motor stops or spins, depending on the value of switch SW1:
- If SW1 has value 0, the motor stops.
- If SW1 has value 1, the motor spins and its spinning direction is given by the value of SW0.

Lesson 3

Problem 1
Write a C program that displays a single digit (base 10) on a 7-segment display.

The display control is more complicated:
- When we want to display a certain digit, we must write the appropriate information to port P1. However, this information is not the value of the digit (in base 2), but a bit configuration that makes the shape of the desired digit to the display.
- For example, if we wish to display digit 0, then all segments must be turned on, except for the middle horizontal segment and the decimal point. Because we must send value 0 to turn on a certain segment (and 1 to turn it off), the configuration to be sent to P1 is $11000000_{(2)}$. The configurations corresponding to the other digits are determined in the same way.
- The display on which the configuration is shown is controlled by bits 3 and 4 of port P3 and, additionally, by bit 7 of port P0.
As all displays get their values from port P1, at most one of them is turned on at any moment, due to the decoder:
- If bit 7 of port P0 has value 0, all displays are turned off (the decoder has no active output).
- Otherwise, bits 3 and 4 of port P3 make a 2-bit number which indicates which decoder output is active - that is, which display is turned on. For example, if the 2-bit number is 01, which means 1 in base 10, then display 1 is turned on.
- For the current problem, because there is a single digit to be displayed, then the same display must be selected all the time. Thus, the values of P3_3 and P3_4 must be set only once, during the initialization phase.

Problem 2
Write a C program that displays a multi-digit number (positive integer) on the 7-segment displays.

In addition to the previous problem, you must first compute the base-10 digits of the number. Because there are 4 displays available in the simulator, we will consider that the number has exactly 4 digits, which you have to determine. Naturally, this is achieved through division by 10.

Problem 3
Write a program that displays a multi-digit number, as above, but ignores non-significant digits. For example, if the number is 698, display 3 is never turned on (in the previous case, it would show value 0).

Lesson 4

Problem 1
Write a C program that reads a key from the keypad and displays it on one of the 7-segment displays. We consider that a single key on the keypad has already been pressed when the program starts and the situation on the keypad is not changing during program execution.

The keypad is connected to the microcontroller in a special manner, which leads to a sophisticated read procedure. To detect that a key is pressed, one must first write value 0 on the corresponding keypad row and subsequently read the value on the corresponding keypad column.
For example, suppose we want to detect whether key '6' has been pressed or not. Key '6' is placed at the intersection between the second-highest row and the rightmost column. Thus, we must write 0 on the second-highest row (i.e., bit 2 on port P0) and test the value on the rightmost column (i.e., bit 4 on port P0). If that value is 0, then key '6' is pressed; otherwise, the key is not pressed.
As we do not know in advance which key has been pressed (if any), we have to scan the whole keypad; that is, we must test all rows and columns, as in the example above. There are 4 rows, which means that we have to send to bits P0_0 ÷ P0_3, in a sequence, all 4 combinations with a single bit 0 and the rest of the bits 1. For each such combination, we must test bits P0_4 ÷ P0_6 for value 0 (which would signal a pressed key).
Observation: Just before testing bits P0_4 ÷ P0_6, it is recommended to write value 1 to all of them.

Problem 2
Write a program that reads 4 keys from the keypad (pressed and released sequentially by the user) and then shows them on the 4 displays.

This problem does not bring anything new in terms of reading the keypad, but it requires the programmer to carefully manage the keys that have been pressed. The most important issue here is that, when the user presses a key, the program will read a sequence of identical values until the key has been released. For example, if the user presses key '8' at some point, it will take some time before the user releases that key; during this time interval, the program will repeatedly detect key '8' being pressed. It would be an error to consider that the user has pressed key '8' multiple times.
The solution is to permanently compare the current situation of the keypad with the previous situation (i.e., the situation recorded the last time when the keypad was previously scanned). The program must consider that a new key was pressed only when both conditions below are met:
- In the previous scan no key was pressed.
- In the current scan a key is pressed; this is the newly pressed key, which must be stored for later display.

Lesson 5

Problem 1
Write a C program that turns on all LEDs individually, one LED at a time, and then moves to the next etc. This is similar to one of the problems in Lesson 1. This time, however, moving from one LED to the next is done at a constant pace, controlled by an internal timer.

The most difficult problem here is measuring the time. As program loops do not provide a refined and reliable control of the elapsed time, it is necessary to use the built-in timers (see the 8051 data sheet).
In principle, a timer is a counter whose increment/decrement input comes from an infinite periodic signal, similar to the system clock.
When using the timer, the steps are:
- Write the initial value to the timer.
- Start the timer (there is a control signal through which one can decide when the timer works and when it remains unchanged).
- Wait until the timer reaches value 0; this is the moment when the time interval that we wanted to measure is over.

For example, suppose that a decrement timer is controlled by a clock signal with a frequency of 100 Hz (that is, 100 periods per second). If we want to measure a time interval of length 2.5 seconds, then we have to initialize the timer with value 250. This way, after decrementing 250 times, the timer reaches value 0; at that moment, 2.5 seconds (= 250 clock periods) have passed since we started the timer.

8051 has two built-in timers, marked as timer 0 and timer 1. For the sake of simplicity, we will only discuss timer 0 (operating timer 1 is similar).
In order to control timer 0, several internal registers must be read/written (see the 8051 data sheet):
- The timer's operating mode is set by writing register TMOD. For our purposes, we have to write value $00000001_{(2)}$ into TMOD.
- The initial value is a 16-bit number, which is stored into two 8-bit registers: TH0 (the most significant half of the 16-bit value) and TL0 (the least significant value). For example, suppose we want to initialize the timer with value 1000; this means we have to write value 3 into TH0 and value 232 into TL0, because $3\times256+232=1000$.
- Starting/stopping the timer is done by writing bit TR0 (part of register TCON); value 1 means start, value 0 means stop.
- In order to test whether the timer has reached value 0, one must read bit TF0 (also part of register TCON). When TF0 has value 1, this means that the time interval is over.

For the current problem, you must do the following:
- The initialization phase:
  - Set the timer's operating mode.
  - Load the initial value into the timer.
  - Initialize the LED configuration.
  - Start the timer.
- The infinite loop:
  - Wait (do nothing) until TF0 becomes 1.
  - Reload the initial value into the timer (to prepare it for the next iteration).
  - Reset TF0 (also to prepare the timer for the next iteration).
  - Change the LED that is displayed.

Problem 2
The same problem as above, but this time control the timer through the interrupt system.

Using interrupts is more efficient than waiting in a loop for the timer to become 0. In order to do that, the timer must be allowed to generate interrupt requests when it reaches the end of its cycle. This way, the time previously wasted by waiting in a loop until the timer to becomes 0 can now be used to do some other, independent work.

In the case of 8051, the management of interrupt requests is done through register IE (see the 8051 data sheet). It contains one bit for each possible source of interrupt requests: the two timers; two external inputs; the serial interface. Setting the corresponding bit in IE allows a source to generate interrupt requests; resetting it disables the interrupts requests from that source. For our problem, if we want to be able to generate interrupt requests by a timer, we must set bit ET0 for timer 0 (or bit ET1 if we use timer 1). Also, we must set the general enable interrupt bit EA; leaving it with value 0 disables all interrupts. These operations must be performed during the initialization phase in function `main`.
Getting back to the previous problem, much of the code written there inside the infinite loop must now be relocated into the interrupt handling routine. In fact, only the display operation (i.e., sending the proper value to P1) may remain into the infinite loop, although that may also be relocated.
In fact, the interrupt handling routine should do the following:
- Reload the initial value into the timer.
- Reset TF0.
- Change the LED that is displayed.
Notice that the inner loop, previously used for waiting until TF0 becomes 1, is no longer present, neither in the infinite loop of function `main`, nor in the interrupt handling routine. That inner loop is no longer necessary: when the timer's value becomes 0, an interrupt is automatically generated and the handling routine is started.

What you need to know about writing interrupt handling routines:
- Any variable that is accessed both in the handling routine and anywhere else (usually in function `main`) must be declared with the keyword `volatile`.
- Any interrupt handling routine is written like an ordinary function. However, we must tell the compiler that this function must actually be called when the timer generates an interrupt. The prototype of such a function is enhanced and looks as below:

```
void function_name() __interrupt (1)
{...}
```

We notice the use of the keyword `__interrupt`, which tells the compiler about the special use of the function. The parameter 1 tells that the routine must be called when timer 0 generates an interrupt. If we use timer 1, the value of the parameter must be 3.

Problem 3
Write a program that continuously displays the digits of a number (positive integer) on the 7-segment displays. The periodic action (changing the digit that is currently displayed - see Lesson 3) is performed by using timer interrupts, as above.