Ministerul Educatiei al Republicii Moldova
Universitatea Tehnica a Moldovei
Filiera Anglofona

Report

Embedded Systems

Laboratory Work #5

Performed by: Vlad Croitoru

Verified by: Andrei Bragarenco

Chisinau 2017

Topic

Task runner. Timers. Interrupts.

Scope:

- Implement a task runner (scheduler)
- Implement descriptor for task
- Use timers for controlling tasks in time.

Task:

Write a C program and schematics for task runner(scheduler) which will run a **task** in defined interval of time with defined run properties. This will look like multi tasking.

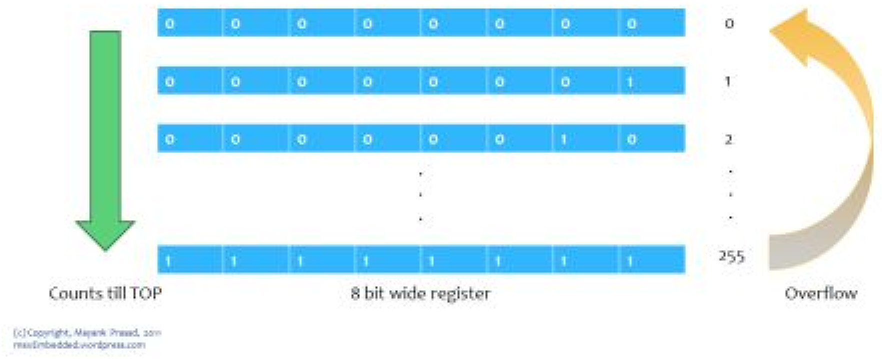
Overview:

Timers

Timers are used everywhere. Without timers, you would end up nowhere! The range of timers vary from a few microseconds (like the ticks of a processor) to many hours, and AVR is suitable for the whole range! AVR boasts of having a very accurate timer, accurate to the resolution of microseconds! This feature makes them suitable for timer applications. Let’s see how.

Timers as registers

So basically, a timer is a register! But not a normal one. The value of this register increases/decreases automatically. In AVR, timers are of two types: 8-bit and 16-bit timers. In an 8-bit timer, the register used is 8-bit wide whereas in 16-bit timer, the register width is of 16 bits. This means that the 8-bit timer is capable of counting 2^8=256 steps from 0 to 255 as demonstrated below.



8 bit Counter

Similarly a 16 bit timer is capable of counting 2^16=65536 steps from 0 to 65535. Due to this feature, timers are also known as counters. Now what happens once they reach their MAX? Does the program stop executing? Well, the answer is quite simple. It returns to its initial value of zero. We say that the timer/counter overflows.

In ATMEGA32, we have three different kinds of timers:

- TIMER0 – 8-bit timer
- TIMER1 – 16-bit timer
- TIMER2 – 8-bit timer

The best part is that the timer is totally independent of the CPU. Thus, it runs parallel to the CPU and there is no CPU's intervention, which makes the timer quite accurate.

Apart from normal operation, these three timers can be either operated in normal mode, CTC mode or PWM mode.

Timer Concepts

Basic Concepts

Since childhood, we have been coming across the following formula:

$$Time\ Period = \frac{1}{Frequency}$$

Now suppose, we need to flash an LED every 10 ms. This implies that its frequency is 1/10ms = 100 Hz. Now let's assume that we have an external crystal XTAL of 4 MHz. Hence, the CPU clock frequency is 4 MHz. Now, as I said that the timer counts from 0 to TOP. For an 8-bit timer, it counts from 0 to 255 whereas for a 16-bit timer it counts from 0 to 65535. After that, they overflow. This value changes at every clock pulse.

Let's say the timer's value is zero now. To go from 0 to 1, it takes one clock pulse. To go from 1 to 2, it takes another clock pulse. To go from 2 to 3, it takes one more clock pulse. And so on. For F_CPU = 4 MHz, time period T = 1/4M = 0.00025 ms. Thus for every transition (0 to 1, 1 to 2, etc), it takes only 0.00025 ms!

Now, as stated above, we need a delay of 10 ms. This maybe a very short delay, but for the microcontroller which has a resolution of 0.00025 ms, its quite a long delay! To get an idea of how long it takes, let's calculate the timer count from the following formula:

$$Timer\ Count = \frac{Required\ Delay}{Clock\ Time\ Period} - 1$$

Substitute Required Delay = 10 ms and Clock Time Period = 0.00025 ms, and you get Timer Count = 39999. Can you imagine that? The clock has already ticked 39999 times to give a delay of only 10 ms!

Now, to achieve this, we definitely cannot use an 8-bit timer (as it has an upper limit of 255, after which it overflows). Hence, we use a 16-bit timer (which is capable of counting up to 65535) to achieve this delay.

The Prescaler

Assuming F_CPU = 4 MHz and a 16-bit timer (MAX = 65535), and substituting in the above formula, we can get a maximum delay of 16.384 ms. Now what if we need a greater delay, say 20 ms? We are stuck?!

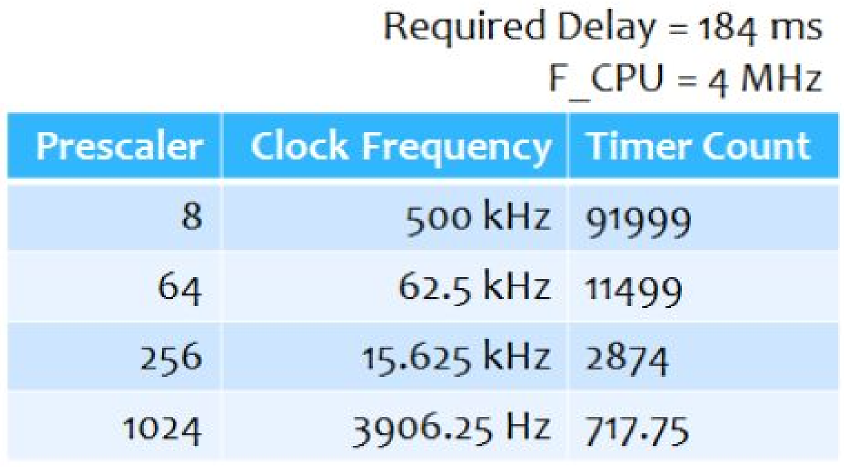
Well hopefully, there lies a solution to this. Suppose if we decrease the F_CPU from 4 MHz to 0.5 MHz (i.e. 500 kHz), then the clock time period increases to 1/500k = 0.002 ms. *Now* if we substitute **Required Delay = 20 ms** and **Clock Time Period = 0.002 ms**, we get **Timer Count = 9999**. As we can see, this can easily be achieved using a 16-bit timer. At this frequency, a maximum delay of 131.072 ms can be achieved.

Now, the question is how do we actually reduce the frequency? This technique of frequency division is called prescaling. We do not reduce the actual F_CPU. The actual F_CPU remains the same (at 4 MHz in this case). So basically, we derive a frequency from it to run the timer. Thus, while doing so, we divide the frequency and use it. There is a provision to do so in AVR by setting some bits which we will discuss later.

But don't think that you can use prescaler freely. It comes at a cost. There is a trade-off between resolution and duration. As you must have seen above, the overall duration of measurement has increased from a mere 16.384 ms to 131.072 ms. So has the resolution. The resolution has also increased from 0.00025 ms to 0.002 ms (technically the resolution has actually decreased). This means each tick will take 0.002 ms. So, what's the problem with this? The problem is that the accuracy has decreased. Earlier, you were able to measure duration like 0.1125 ms accurately (0.1125/0.00025 = 450), but now you cannot (0.1125/0.002 = 56.25). The new timer can measure 0.112 ms and then 0.114 ms. No other value in between.

Choosing Prescalers

Let's take an example. We need a delay of 184 ms (I have chosen any random number). We have F_CPU = 4 **MHz**. The AVR offers us the following prescaler values to choose from: 8, 64, 256 and 1024. A prescaler of 8 means the effective clock frequency will be F_CPU/8. Now substituting each of these values into the above formula, we get different values of timer value. The results are summarized as below:



Required Delay = 184 ms
F_CPU = 4 MHz

Prescaler	Clock Frequency	Timer Count
8	500 kHz	91999
64	62.5 kHz	11499
256	15.625 kHz	2874
1024	3906.25 Hz	717.75

Choosing Prescaler

Now out of these four prescalers, 8 cannot be used as the timer value exceeds the limit of 65535. Also, since the timer always takes up integer values, we cannot choose 1024 as the timer count is a decimal digit. Hence, we see that prescaler values of 64 and 256 are feasible. But out of these two, we choose 64 as it provides us with greater resolution. We can choose 256 if we need the timer for a greater duration elsewhere.

Thus, we always choose prescaler which gives the counter value within the feasible limit (255 or 65535) and the counter value should always be an integer.

We will discuss how to implement it in later posts.

Interrupts

Well, this is not exclusively related to timers, but I thought of discussing it as it is used in a variety of places. Let me explain it using an analogy. Say now you are reading my post. It's dinner time and your mom (only if you live with your mom ;)) calls you for dinner. What do you do (if she gets too creepy)? You save your work and attend to your mom's call, then return and resume reading. This is an example of interrupt.

In most microcontrollers, there is something called interrupt. This interrupt can be fired whenever certain conditions are met. Now whenever an interrupt is fired, the AVR stops and saves its execution of the main routine, attends to the interrupt call (by executing a special routine, called the **Interrupt Service Routine**, **ISR**) and once it is done with it, returns to the main routine and continues executing it.

For example, in the condition of counter overflow, we can set up a bit to fire an interrupt whenever an overflow occurs. Now, during execution of the program, whenever an overflow occurs, an interrupt is fired and the CPU attends to the corresponding ISR. Now it's up to us what do we want to do inside the ISR. We can toggle the value of a pin, or increment a counter, etc etc.

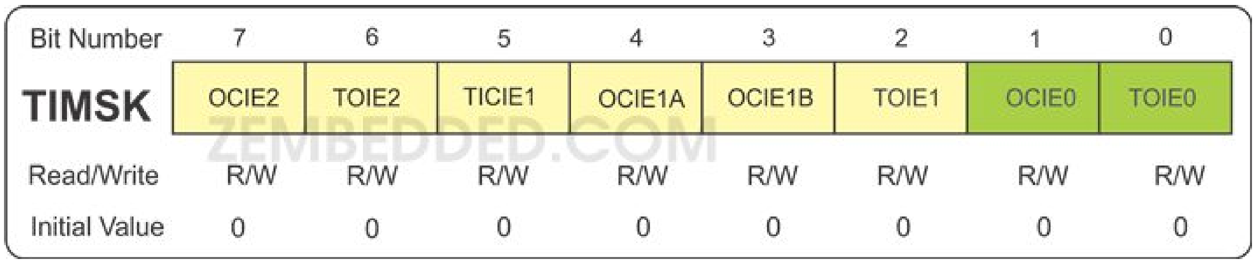
Used Resources

Using Interrupts with CTC Mode

The AVR will compare TCNT1 with OCR1A. Whenever a match occurs, it sets the flag bit OCF1A, and also fires an interrupt! We just need to attend to that interrupt, that's it! No other headache of comparing and stuffs! There are three kinds of interrupts in AVR – overflow, compare and capture. We need to enable the compare interrupt. The following register is used to enable interrupts.

TIMSK Register

The Timer/Counter Interrupt Mask Register– TIMSK Register is as follows. It is a common register to all the timers. The greyed out bits correspond to other timers.



Bit Number	7	6	5	4	3	2	1	0
TIMSK	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

We have already come across TOIE1 bit. Now, the **Bit 4:3 – OCIE1A:B – Timer/Counter1, Output Compare A/B Match Interrupt Enable bits** are of our interest here. Enabling it ensures that an interrupt is fired whenever a match occurs. Since there are two CTC channels, we have two different bits OCIE1A and OCIE1B for them.

Thus, to summarize, whenever a match occurs (TCNT1 becomes equal to OCR1A = 24999), an interrupt is fired (as OCIE1A is set) and the OCF1A flag is set. Now since an interrupt is fired, we need an Interrupt Service Routine (ISR) to attend to the interrupt. Executing the ISR clears the OCF1A flag bit automatically and the timer value (TCNT1) is reset.

Interrupt Service Routine (ISR)

Now let’s proceed to write an ISR for this. The ISR is defined as follows:

```
ISR (TIMER1_COMPA_vect)

{

   // code here

{
```

Solution

GPIO Descriptor

GPIO descriptor stands for GP I/O operations and manipulation. It has possibility to read or write, set mode to input or output.

```
typedef struct GPIO {
        uint8_t volatile *ddr;
        uint8_t volatile *port;
        uint8_t volatile *pin;
        uint8_t id;
} GPIO;

typedef enum GPIO_Mode {
        GPIO_MODE_OUTPUT,
        GPIO_MODE_INPUT
} GPIO_Mode;

typedef enum GPIO_Value {
        GPIO_LOW = 0,
        GPIO_HIGH = 1
} GPIO_Value;

GPIO* GPIO_create(uint8_t volatile *ddr,uint8_t volatile *port,uint8_t volatile *pin,uint8_t id);

void GPIO_set_mode(GPIO *descriptor,GPIO_Mode mode);

void GPIO_write(GPIO *descriptor,GPIO_Value value);
```

GPIO_Value GPIO_read**(GPIO *descriptor);**

Task

Task – descriptor for a task. It has properties like **delay** of start, repeat **interval**,enable flag and of course handler.

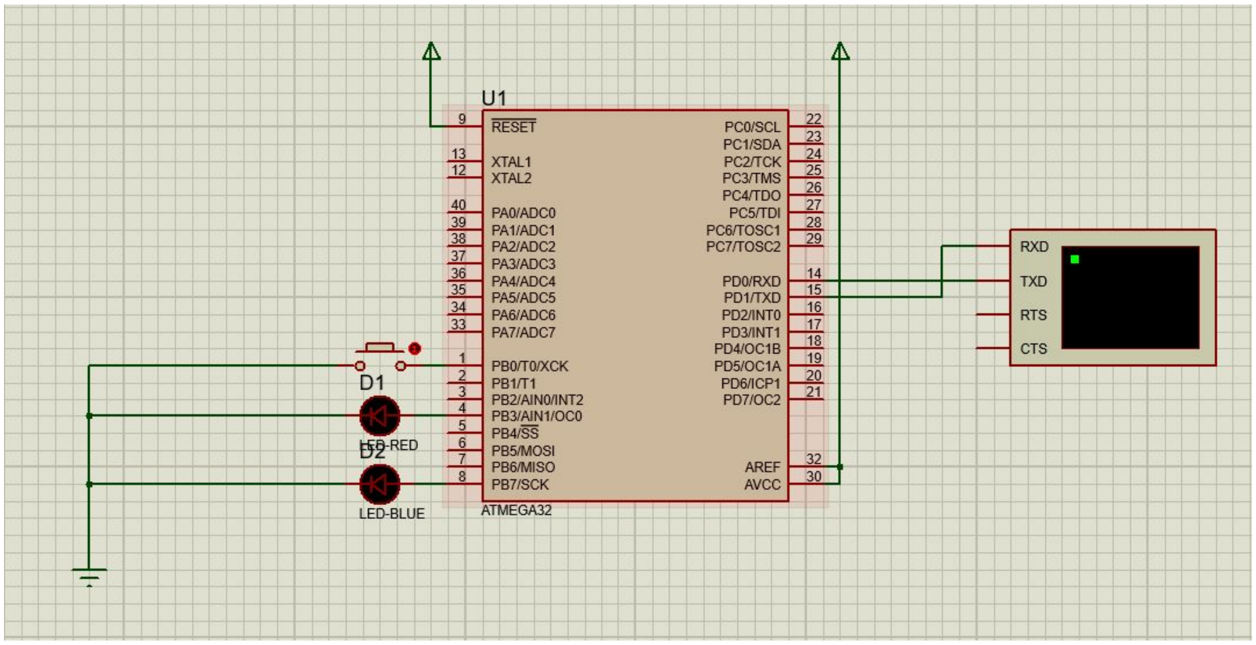
```
typedef struct Task {
        uint32_t delay;
        uint32_t interval;
        uint8_t enabled;
        void (*handler)();
} Task;
```

Task Scheduler

Task scheduler has biggest responsibility, to invoke registered task in desired interval of time. Task scheduler works on 16 bit **TIMER1**, using **CTC Mode** with **Interrupts Service Routine.**

```
void TASK_SCHEDULER_start();
void TASK_SCHEDULER_add(Task *task);
```

Schematics



Used elements:

1. Push button
2. Red led
3. Blue led

This model has 3 tasks:

1. Switches **red led** output bit
2. Switches **blue led** output bit
3. Button press checker

Initially blue led is toggling. When pressing push button, then this toggle task is disabled, and red toggle task is enabled.

Conclusion

In order to implement task runner I had used 16 bit timer, because that timer, on prescaller 8 and CTC Mode with breaking 1250. Which exactly gives us interval of 0.01 s or 10 ms.

$$\frac{1}{10^6} * 1250 * 8 = 0.01\ s$$

Having runner with 10 milliseconds interval, we can schedule task with interval of 10 or more milliseconds.

That’s enough for initial purpose. Next thing what I had to do, is to implement t mechanism for registering task and check in each 10 miliseconds if task should be run.