16. System Timer

Up to now we have carefully avoided any serious handling of one of the most important system concepts: the passage of time. We have used artificial and experimentally configured delays whenever they were needed for interaction with the external world (blinking LEDs or basic debouncing of inputs), but still our programs were not aware of the time that is passing outside of the device, in the environment.

The ARM Cortex-M microcontrollers have several standard features that are related to the management of time and many implementations have additional, non-standard built-in peripherals that can be used for this purpose as well. The most basic timing feature in Cortex-M is the SysTick subsystem, which is able to automatically generate events in programmable intervals. These events can be visible as interrupts and we will rely on this to build a simple system timer.

SysTick can be seen as an *internal peripheral*, although it operates somewhat outside of the NVIC controller and has simpler rules for handling interrupts. The operation of SysTick can be shortly described in these points:

- there is a counter that counts down at the pace of main processor clock (either fed directly or slowed down by some configurable factor),
- when the counter reaches 0, an event (interrupt) is generated and the counter can be automatically reloaded with its initial value, thus ensuring continuous and periodic operation,
- the user can program the initial counter value to influence the counter period.

The possibility to generate interrupts in a periodic way is a basis for implementing simple system timers, although it should be noted that the accuracy of such timers will depend on the accuracy of the main system clock signal. This precision can be sufficient in many systems, especially those that need to operate with short periods, but is certainly not sufficient for tracking real calendar time across long time-scales. Independently on the precision, building the system timer in relation to the main processor clock requires complete control or at least understanding of its parameters - that is, we need to know how the main clock operates if we want to build our time management on top of it. The microcontrollers used in our boards have several ways to configure their main clocks, with a wide range of operational frequencies. We will not explore all these possibilities in this tutorial, but we will at least need to know what is the default main clock frequency. These are different for each microcontroller.

Arduino M0

The microcontroller documentation, in the chapter about *Clock System*, explains that by default the internal oscillator called OSC8M is active and feeds the rest of the system. This is a 8MHz clock that is divided by 8 before being routed to the system timer.

Arduino Due

According to Atmel documentation:

By default, at startup the chip runs out of the Master Clock using the Fast RC oscillator running at 4 MHz.

STM32 Nucleo-32

The STM32F03x chips by default (after reset) use internal 8MHz clock, which, after dividing by 8, is fed to the system clock.

STM32 Nucleo-144

The STM32F746x microcontroller by default uses internal 16MHz clock, which is used directly as a system clock, but the Cortex system timer is fed with the system clock divided by 8.

We will rely on these values when configuring our own timing system.

Apart from the actual frequencies of the underlying clock sources, the SysTick subsystem is a standard component of the Cortex-M core and can be controlled by means of the following registers:

- CTRL - *SysTick Control and Status Register* (at address 0xE000E010), which can be used to enable/disable SysTick and to control the generation of interrupts when the counter reaches 0,
- LOAD - *SysTick Reload Value Register* (at address 0xE000E014), containing the initial, automatically reloaded value for

the counter,

- VAL - *SysTick Current Value Register* (at address 0xE000E018), with the current value of the counter (the current value can be read or written),
- CALIB - *SysTick Calibration Value Register* (at address 0xE000E01C), which was intended to contain the standardized calibration value for the reload register.

We only need to set bits 0 and 1 of the CTRL register in order to enable the counter and to switch on the generation of interrupts. The LOAD and VAL registers just have a single 24-bit field and do not require any special handling.

The last of these registers, CALIB, is a source of some confusion in both the actual implementation and in the official vendor documentation as well. The intent of this register was to provide a reload value that will ensure 10ms of SysTick period (in other words, 100Hz of operation frequency). This was described in the official ARM architecture documentation and 24 of bits of this register form a field that was named TENMS ("ten milliseconds") to reflect this intent. The idea was that a programmer can just copy a value from CALIB to LOAD and achieve standardized 10ms of period independently on the actual Cortex-M implementation. Unfortunately this promise does not hold in some of the chips (not even counting the different source frequencies) and the calibration value that is hardwired there was chosen for 1ms (1kHz) in the case of Atmel chips and *only* when the microcontroller runs at its maximum speed.

This is unfortunate and certainly confusing, because it invalidates the portability intent of this register - if we just copy a value from CALIB to LOAD, we will not get the same period on another Cortex-M device, or even on the same device with different clock source. For this reason we will ignore this feature altogether and instead we will create a proper value for 10ms period ourselves, putting appropriate definitions in the body of package `Utils`.

Arduino M0

For Arduino Due the following definitions will be used:

```
   Master_Clock_Frequency_Hz : constant := 8_000_000;
   SysTick_Divider : constant := 8;
   SysTick_Clock_Frequency_Hz : constant :=
      Master_Clock_Frequency_Hz / SysTick_Divider;
   SysTick_Period_Millis : constant := 10;

   SysTick_Reload_Value : constant :=
      (SysTick_Clock_Frequency_Hz / 1_000) * SysTick_Period_Millis;
```

Arduino Due

For Arduino Due the following definitions will be used:

```
   Master_Clock_Frequency_Hz : constant := 4_000_000;
   SysTick_Divider : constant := 8;
   SysTick_Clock_Frequency_Hz : constant :=
      Master_Clock_Frequency_Hz / SysTick_Divider;

   SysTick_Period_Millis : constant := 10;

   SysTick_Reload_Value : constant :=
      (SysTick_Clock_Frequency_Hz / 1_000) * SysTick_Period_Millis;
```

STM32 Nucleo-32

For Nucleo-32 the following definitions will be used:

```
   Master_Clock_Frequency_Hz : constant := 8_000_000;
   SysTick_Divider : constant := 8;
   SysTick_Clock_Frequency_Hz : constant :=
      Master_Clock_Frequency_Hz / SysTick_Divider;

   SysTick_Period_Millis : constant := 10;

   SysTick_Reload_Value : constant :=
      (SysTick_Clock_Frequency_Hz / 1_000) * SysTick_Period_Millis;
```

STM32 Nucleo-144

For Nucleo-32 the following definitions will be used:

```
   Master_Clock_Frequency_Hz : constant := 16_000_000;
   SysTick_Divider : constant := 8;
   SysTick_Clock_Frequency_Hz : constant :=
      Master_Clock_Frequency_Hz / SysTick_Divider;
   SysTick_Period_Millis : constant := 10;

   SysTick_Reload_Value : constant :=
      (SysTick_Clock_Frequency_Hz / 1_000) * SysTick_Period_Millis;
```

The last constant above will ensure 10ms of interrupt period and this can be changed when necessary (for example, when the microcontroller is configured to run with a different master clock) by just changing earlier definitions above.

Our strategy for handling system time will be based on a simple interrupt handler that will periodically increment a shared variable acting as our own system clock. This clock will operate in terms of milliseconds, although physically its resolution will be based on the interrupt period. By decoupling the clock value from its update period we will be able to modify the SysTick frequency if necessary without changing the rest of the program - for this we will assume that millisecond precision is enough for our purposes.

As usual, we need to introduce proper symbol definitions for the new registers (CTRL, LOAD and VAL) in the linker script and also declare appropriate register objects in package `Registers`. With these basics in place, we can try to write the scaffolding for our system time handling. All definitions will be put in package `Utils`.

The shared variable that will keep the system time can look like:

```
   System_Time : Time;
   pragma Volatile (System_Time);
```

where type `Time` should be defined earlier (in the package specification) as:

```
   type Time is new Natural;
```

The type `Natural` is an integer type that can handle non-negative values and is based on the 32-bit `Integer` type. This is enough for our needs, as we can assume that the device will reset its system clock to 0 when initialized and the capacity of this type allows handling the time span of 24 days. If this is not sufficient, some other solution will be needed, but note that keeping the system time within a single word of memory guarantees atomic accesses and greatly simplifies the whole implementation. On the other hand, the lack of precision of the main clock has to be taken into account and if it is not a problem even on such a long time scale, then very likely the clock can be reset in the meantime without regard to amount of time that has already passed - this will be typical for those designs that have to operate periodically, but need not be aware of the actual time value.

Note that if we define this shared variable in the *body* of package `Utils`, it will not be directly accessible by the main program - this way of encapsulating state is very useful.

As described above, the strategy was to have the periodic interrupt handler updating this shared variable - the straightforward implementation of the handler is:

```
   procedure SysTick_Handler is
   begin
      System_Time := System_Time + SysTick_Period_Millis;
   end SysTick_Handler;
```

This procedure has to be given some external name and wired in the linker script, in the exception vector at offset 0x3c - the rules here are exactly the same as for our earlier interrupt handlers, but this time the slot to be used is a standard part of the Cortex-M architecture. At run time, this procedure will periodically update the `System_Time` variable, so that when the user program reads for example value 123450, it will mean that the last initialization took place 2 minutes, 3 seconds and 450 milliseconds ago, *and* possibly several milliseconds more if we happen to be somewhere in the middle of the system timer update period. Whether this remaining error is a problem will of course depend on the actual purpose of the whole system - but it depends on the update period and we can configure it as needed.

Apart from the shared variable and an interrupt handler, we will need some procedure to initialize the whole SysTick subsystem. Note that SysTick operates somewhat outside of the NVIC and as such does not need to be enabled at the NVIC level. Similarly, the reception of the interrupt need not be *acknowledged* as there is no flag that needs to be cleared by the handler, as was the case for I/O interrupts. The SysTick initialization needs to ensure that a proper value is in the LOAD register, and only when this is done the operation of the counter can be enabled:

```
   procedure Enable_System_Time is
      use type Registers.Word;
   begin
      System_Time := 0;

      Registers.SysTick_VAL := 0;
      Registers.SysTick_LOAD := SysTick_Reload_Value;
      Registers.SysTick_CTRL := Registers.Bit_0 or Registers.Bit_1;
   end Enable_System_Time;
```

Note that this procedure alone is not enough to initialize the system time handling; the interrupts have to be enabled globally as well with the call to procedure `Enable_Interrupts` that we have implemented already. The *use type* clause in the declarative part of this procedure brings visibility of the operators of type `Word`, this allows us to use the `or` operator in the last line without any additional qualification.

We will need several other subprograms that will be visible to other packages and that will provide a high-level interface to our timing subsystem:

```
   function Clock return Time is
   begin
      return System_Time;
   end Clock;

   procedure Reset_Clock (T : in Time := 0) is
   begin
      System_Time := T;
   end Reset_Clock;

   procedure Delay_Until (T : in Time) is
   begin
      while System_Time < T loop
         Wait_For_Interrupt;
      end loop;
   end Delay_Until;
```

Function `Clock` will be used by the main program (or any other package) to check the current time (that is, the time value that was established during the last SysTick interrupt, which might have happened several milliseconds ago, depending on the update period) and `Reset_Clock` will be used to reset the time, which can prove useful in long-running systems. The most interesting is the last procedure, `Delay_Until` as it allows the program to suspend execution until system time reaches the given value. Note that the loop uses call to `Wait_For_Interrupt`, which makes it very likely that the delay will not waste cycles and energy while waiting until the given time. The `Delay_Until` procedure is not strictly necessary (the user can write such a loop himself), but allows to raise the level of abstraction when handling delays. It should be also clear why it was important to declare `System_Time` as volatile object - it is periodically updated by the SysTick handler *in background* and has to be re-checked in every loop iteration, for this reason physical accesses to this variable should not be optimized out by the compiler.

Now, let's try some basic usage patterns - professional blinking LED will look like this:

```
with Pins;
with Utils;

package body Program is

   procedure Run is
      Period : constant := 1_000;
      Cycle_Begin : Utils.Time;
      use type Utils.Time;
   begin
      -- initialize output pin
      Pins.Enable_Output (Pins.Pin_12);

      -- enable interrupts and set up system time
      Utils.Enable_Interrupts;
      Utils.Enable_System_Time;

      Cycle_Begin := Utils.Clock;
      loop
         Pins.Write (Pins.Pin_12, True);
         Utils.Delay_Until (Cycle_Begin + Period / 2);

         Pins.Write (Pins.Pin_12, False);

         Cycle_Begin := Cycle_Begin + Period;
         Utils.Delay_Until (Cycle_Begin);
      end loop;
   end Run;

end Program;
```

The program above blinks the LED (assuming that a LED is connected to pin 12, of course) with 1s of period and it shows a canonical way to implement periodic activity:

```
      Cycle_Begin := Utils.Clock;
      loop
```

```
         -- some periodic activity here:
         -- ...

         -- wait until next cycle:
         Cycle_Begin := Cycle_Begin + Period;
         Utils.Delay_Until (Cycle_Begin);
      end loop;
```

We have an additional delay in the "periodic activity" part above just to keep the LED lit up, but the general structure will be similar for other periodic systems as well. The advantage of this approach is that as long as the activity fits within the given time window, the consecutive periods will always start at right times, with quite good precision, even if the duration of the activity is not regular. The whole iteration structure will work correctly for up to 24 days, when the system timer will reach the upper limit of the `Natural` data type. If this is a problem, the following modification can help:

```
      Utils.Reset_Clock;
      loop
         -- some periodic activity here:
         -- ...

         -- wait until next cycle:
         Utils.Delay_Until (Period);
         Utils.Reset_Clock;
      end loop;
```

Now the clock will be reset after every iteration and the device will never know that it works longer than one operational period (one second in our case). If this is a problem, the program might maintain its own counter of iterations to keep track of the total time that has passed, of whatever capacity is needed. One thing that is worth to consider in this approach is whether there exists a possibility of missing a time update just before resetting the clock - in this simple system there is no such risk, but in the presence of multiple interrupts with heavy computational load such a possibility should be taken into account.

The above system timer design allows handling both time-stamping of events and absolute delays and can be treated as a solid foundation block for building more complex time-aware systems.

Previous: Finite-State Machines, Part 3, next: Hello World!.
See also Table of Contents.

Did you find this article interesting? Share it!

SHARE