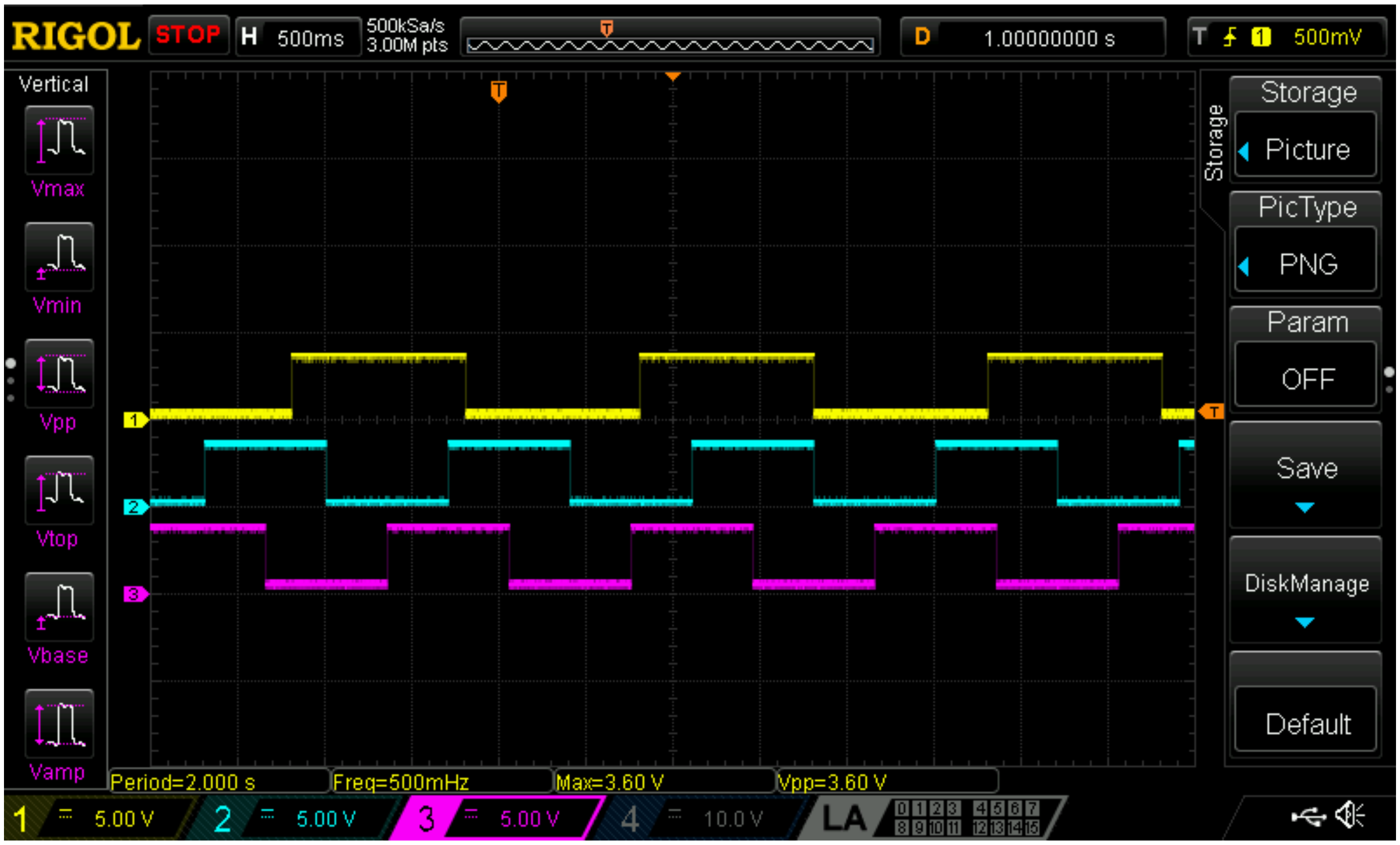
Very Simple Scheduler

This article is an extension of the Ada on ARM Cortex-M book and presents a very simple programming solution to scheduling independent activities in an embedded system. The Nucleo-32 prototyping board is used as a realistic target, but the essential part of this article does not rely on it and can be reused with other targets as well. The lower-level hardware abstraction layer, dealing with system timer and digital I/O, is not explained and is just taken from the book examples - refer to the book for details.

Imagine a system where several independent activities have to be managed concurrently. The notion of concurrency is of course limited by the very nature of a single-core processing, but can have realistic impact at the higher level, for example in interactions with different peripherals. For the sake of example, imagine a system that needs to periodically blink three LEDs, with independent periods or phase shifts. Let's say that the blinking periods are specified as follows:

- first LED blinks with the period of 2000ms,
- second LED blinks with the period of 1400ms,
- third LED blinks with the period of 1400ms, too, but its transitions are delayed by 350ms with regard to the second LED.

In other words, we want our system to generate the control signals as in this oscilloscope screen capture:



On this screen the time base is 500ms/div, so indeed the timing of each blinking LED is as specified.

Such a system can be designed in terms of state machines, where a blinking LED obviously has just two states - "On" and "Off" - and where a single state machine manages a single LED.

It is possible to implement this system within a single loop by carefully juggling with states, but it is better to decouple timing specification from the transition logic, so that the system can be easily extended or modified if the number of tasks (blinking LEDs) or their timing needs to be changed.

First, let's express the application logic in terms of a state machine, without regarding the specified timing. The blinking LED has just two states, which is easy to describe in Ada:

```
   type States is (Off, On);
```

The state machine transition logic can be implemented like here:

```
   procedure Blink (State : in out States; P : in Pins.Pin_ID) is
   begin
      if State = Off then
         State := On;
      else
         State := Off;
      end if;

      Pins.Write (P, State = On);
   end Blink;
```

The procedure `Blink` above needs to operate on a state that survives between procedure's invocations, so the state is passed as a modifiable parameter. The second parameter tells which I/O pin is responsible for interactions with the external world (the LED) and allows us to reuse this single procedure for all three LEDs. That is, since all three LEDs blink by following the same "application logic", it makes sense to implement that logic in a separate, reusable procedure.

Let's assume that we will use pins 10, 11 and 12 to control the LEDs and the pin numbers will be used to name those program entities that need to be separate for each task. For example, separate states are:

```
   State_10 : States;
   State_11 : States;
   State_12 : States;
```

and we will also have separate *parameterless* procedures that are top-level activation points for each task:

```
   procedure Blink_10 is
   begin
      Blink (State_10, Pins.D10);
   end Blink_10;

   procedure Blink_11 is
   begin
      Blink (State_11, Pins.D11);
   end Blink_11;

   procedure Blink_12 is
   begin
      Blink (State_12, Pins.D12);
   end Blink_12;
```

Note that these procedures are very similar and it would be possible to refactor them by means of generics, but this is not the focus of this example - these procedures are similar, because three LEDs are similar, but in a more realistic example the independent tasks will be more complex - and more different. What is most important, though, is that these procedures deal with *state transitions* and as such are not supposed to take any significant amount of time to execute. This simplified assumption will make it easier to implement the scheduler, as there is no need to interrupt short-executing actions.

The structure above achieves the separation of transition logic from the timing specification. In other words, time can be managed separately and, in particular, the time management of such tasks can be implemented in terms of a reusable library component.

The idea is to describe each independent task in a small amount of data that can be processed by a general-purpose algorithm. Such *task descriptor* can look like here:

```
   type Task_Description is record
      Action : access procedure;
      Activation_Period : Time;

      Time_Since_Last_Due_Active : Time;
   end record;
```

where `Time` is some convenient arithmetic type (in the book examples it was defined simply as `new Natural`, with 1ms as an underlying physical unit, which is sufficient for our purposes.

In this task descriptor:

- `Action` points to the top-level transition procedure for the given task; note that this is a *parameterless procedure*, so we assume that the given procedure has sufficient information to execute its chunk of work, as is the case with the three separate blink procedures above,
- `Activation_Period` is the period, in `Time` units, of subsequent intended invocations of the `Action` procedure,
- `Time_Since_Last_Due_Active` is the time that already has passed since the last time when the `Access` procedure *was intended* to run; note that it is *not* the same as the actual time of last activation, which for various reasons might have been a bit later than intended - this distinction will allow us to deal with occasional slips, as will be explained later on.

With this scaffolding in place, the *Very Simple Scheduler* idea can be described in more details:

- make an array of task descriptors with descriptions of all managed tasks,
- *every so often* update (increase) the counter `Time_Since_Last_Due_Active` for each task in the array and check if it has achieved the value of the intended activation period for this task - in which case the task should be activated and its counter should be reset.

The array of tasks can be defined simply as:

```
type Task_ID is new Positive;
type Task_Descriptions is array (Task_ID range <>) of Task_Description;
```

and the whole scheduling logic can be implemented in terms of the `Tick` procedure:

```
procedure Scheduler_Tick (Tasks : in out Task_Descriptions; Time_Passed : in Time) is
begin
   for T in Tasks'Range loop
      if Tasks (T).Action /= null then
         Tasks (T).Time_Since_Last_Due_Active := Tasks (T).Time_Since_Last_Due_Active + Time_Passed;
         if Tasks (T).Time_Since_Last_Due_Active >= Tasks (T).Activation_Period then

            Tasks (T).Action.all;

            Tasks (T).Time_Since_Last_Due_Active := Tasks (T).Time_Since_Last_Due_Active - Tasks (T).Activation_Period;
         end if;
      end if;
   end loop;
end Scheduler_Tick;
```

The procedure above operates on the whole task array and processes all tasks in a loop, according to the logic described above. The most natural usage pattern for this procedure is to call it repeatedly in the main program loop, thus forcing progress in all task counters. Note that the `Time_Passed` is an argument to this procedure, which opens up several interesting alternative usage patterns. Typically, this procedure will be invoked by the main program with higher average frequency than the intended frequency of task executions, otherwise the `Time_Since_Last_Due_Active` counters will not have a chance to reset and will keep growing until overflow - and the tasks will not be executed as they should.

The example application of this scheduler with regard to our three example LED blinking tasks can be seen in the main procedure:

```
Tasks : Utils.Task_Descriptions (1 .. 3);

procedure Run is
   Next_Time : Utils.Time;

   use type Utils.Time;
begin
   Pins.Enable_Output (Pins.D10);
   Pins.Enable_Output (Pins.D11);
   Pins.Enable_Output (Pins.D12);

   Utils.Enable_Interrupts;
   Utils.Enable_System_Time;

   State_10 := Off;
   State_11 := Off;
   State_12 := Off;

   Tasks :=
     (1 => (Action => Blink_10'Access, Activation_Period => 1000, Time_Since_Last_Due_Active => 0),
      2 => (Action => Blink_11'Access, Activation_Period => 700, Time_Since_Last_Due_Active => 0),
      3 => (Action => Blink_12'Access, Activation_Period => 700, Time_Since_Last_Due_Active => 350));

   Next_Time := Utils.Clock;
   loop
      Utils.Scheduler_Tick (Tasks, Utils.Base_Tick);
```

```
         Next_Time := Next_Time + Utils.Base_Tick;
         Utils.Delay_Until (Next_Time);
      end loop;
   end Run;
```

Above, the `Tasks` array is declared outside of the main procedure `Run` as a matter of convention, it might as well be declared within the `Run` procedure. In any case, this array holds the complete scheduler state and, as we will see shortly, there is no special reason to have only one such state.

The initial part of `Run` initializes the digital I/O pins that will be used to control blinking LEDs and sets up the system time utility. After this is done, the finite state machines are initialized by setting their states to initial `Off` values and the scheduler is configured by setting timing parameter for all three tasks. Note that activation periods are set up to be half of the LED's blinking periods, as there are two transitions (two actions of the state machine) needed per one full blinking cycle. Thus, the first task has activation period of 1000ms, and the other two are activated every 700ms. Interestingly, the `Time_Since_Last_Due_Active` value of the scheduler state for each task, normally used to keep track of the time that the task needs to wait for its next activation, can be used to introduce the initial phase shift with regard to its normal execution cycle - here we use it to achieve the effect of a phase shift between two blinking LEDs.

After everything is initialized, the main procedure enters the inifinite loop, where the scheduler is "ticked" regularly with some frequency that is higher than that of the most busy task. It is convenient to use activation periods that are multiples of the ticking period (which here is 10ms) but it is not necessary, as the scheduler can handle occasional waiting overruns - it is the design tradeoff, as the inevitable jitter might or might not be acceptable in the context of the complete system.

The above example is complete and indeed generates the controlling sequence as shown in the oscilloscope screenshot at the beginning of the article, but it is still useful to highlight some interesting properties of this very simple scheduler:

- **Time is not the only useful application domain.** Even though the scheduler uses parameter and field names that refer to timing concepts, it can be used to operate on other values that have any reasonable notion of progress. For example, a car engine needs to have its oil changed every 20.000 kilometers - the same scheduler can be used to remind about the oil change if instead of time it is regularly fed the distance that was traveled.
- **There can be many schedulers in a single program.** This is a simple consequence of the fact that the scheduling algorithm takes the scheduler table as a parameter. A single program can use multiple scheduling tables and invoke the `Scheduler_Tick` procedure on different tables - not necessarily in the same intervals. This might be useful if a set of tasks does not have an easy to find common divisor for their activation periods. Instead of wasting computing resources on pointless "ticks" with very low granularity, it might make sense to partition the system into separate timing domains with their own, separate "ticking", chosen to be most convenient for each domain.
- **Schedulers can be nested.** Expanding on the observation above, if there can be many schedulers in a single program, there is no reason to "tick" them all from a single place (like from the main loop). It might make sense to arrange schedulers hierarchically, for example to reflect a hierarchy of nested state machines.
- **Scheduler configuration table does not have to be constant.** Well, it cannot be constant for a simple reason that the `Time_Since_Last_Due_Active` fields need to be updated with every "tick", but every single field can be updated while the scheduler is running, either from outside of it (like from the main procedure), or even from inside the controlled tasks. The activation period can be changed if different states of a single task demand different timing granularity. More interestingly, tasks can *influence each other* by meddling with their activation periods or the accounted time from the last activation - whether such interactions are safe from the design perspective is another story, but the scheduler itself does not introduce any limits on such solutions if they are justified.

As you have seen, just a few lines of code were enough to set up a very simple scheduling solution for basic control of independent tasks and even though very simple, the scheme allows some interesting alternative usage patterns. This scheduler is useful not only for blinking LEDs - actually, it can handle any set of independent state machines, as long as their actions are sufficiently short to neglect them. If the invocation jitter is not acceptable, it might be useful to think about the ordering of tasks in the configuration array - or, as was already said, to partition the system into multiple schedulers.

The interesting aspect of multi-task designs that that of information sharing - our assumption so far was that the tasks are *independent*, which is not always the case with regard to data sharing. Stay tuned for further practical Ada examples.

The complete source code with build instructions for this example can be downloaded from here: very_simple_scheduler.zip.

See the complete Ada on Cortex-M book.

Did you find this article interesting? Share it!

SHARE