Inspirel

7. Random Numbers

As a small diversion from handling of GPIO lines, it might be interesting to have a look at other peripheral submodules that are available on the example boards. These of course differ between microcontrollers, but there is one that is relatively easy to use and can even deliver new ideas for GPIO applications - the random number generator.

There are several ways to generate random or pseudo-random numbers and the choice of any given method can depend on the availability of dedicated hardware modules or software algorithms. If the hardware can generate random numbers (for example based on the analogue noise) that are ready to be consumed by software without any further processing, then the software layer can be much simpler (and, arguably, analogue noise sources can have very good properties with regard to the actual randomness of the generated sequence) - and conversely, without hardware support there is more stress on software to produce high quality pseudo-random values.

Our example boards differ in that only two of them (Arduino Due and Nucleo-144) have dedicated peripherals for generating random numbers - these peripherals are very simple to operate (and actually quite similar in use) and yet, reportedly, provide very good statistical properties of their output data. The smaller boards do not have such peripherals, but we can still try to implement a portable random-number package that will deliver values either from hardware where it is available or from one of the known software algorithms otherwise.

The specification of the package that will be common to all our boards can look like this:

```
with Registers;

package Random_Numbers is

   procedure Enable_Generator;

   procedure Read_Next_Value (V : out Registers.Word);

end Random_Numbers;
```

where the meaning of the two procedures should be easy to understand.

Note that we have reused the definition of `Word` data type from package `Registers` and this is why the specification of `Random_Numbers` has to `with Registers` at the beginning. You can also see that in the signature of `Read_Next_Value`, `V` is an *output* parameter, which means that the procedure will write a new value to the object named by the caller - in C++ similar effect can be achieved with references and in C this could be done with a pointer parameter.

The body of this package will depend on what is actually available in the target microcontroller.

Arduino M0 and STM32 Nucleo-32

The Arduino M0 and Nucleo-32 boards have very simple microcontrollers that do not include dedicated random number generators and we will need to implement the generation of pseudo-random values in terms of software algorithms. Interestingly, any such algorithm relies on the possibility to retain the internal state of the generator between invocations, and at this stage we do not yet have the proper means to do it. For this reason we will defer the actual implementation until the chapter about shared state and will instead implement simple mock-ups:

```
package body Random_Numbers is

   procedure Enable_Generator is
   begin
      null;
   end Enable_Generator;

   procedure Read_Next_Value (V : out Registers.Word) is
   begin
      V := 42;
   end Read_Next_Value;

end Random_Numbers;
```

Arguably, the sequence generated by this algorithm is not very random, but we will replace it with something more reasonable in later chapters.

Arduino Due

The generator available in Arduino Due is named *True Random Number Generator (TRNG)* and it has a separate, dedicated chapter in the Atmel documentation.

The generator works in background and produces new values in regular intervals, but in order to do so it needs to be fed with the clock signal. That is, we need to properly configure the microcontroller before we can use the generator - in particular, we need to ensure that a clock signal is delivered to the TRNG submodule.

The Power Management Controller (PMC) is fully described in its dedicated chapter of the Atmel documentation, but for the purpose of random number generation we are only concerned with *Peripheral Clock Controller*, which points at two registers, PMC_PCER0 and PMC_PCER1, that are responsible for enabling peripheral clocks. The actual register and bit position depends on the Peripheral Identifier, which can be found in *Peripheral Identifiers* - where we can see that peripheral identifier for TRNG is 41. This information in turn allows us to find (in *PMC Peripheral Clock Enable Register 1*) that it is the bit 10 in PMC_PCER1 that enables peripheral clock for TRNG.

Once the peripheral clock is enabled, the random number generator itself is easy to operate. The registers that are of interest to us are TRNG_CR (control register to enable the generator), TRNG_ISR (flag to signal that a new random value was produced) and TRNG_ODATA (the random value itself). The only curiosity is that setting of the control register requires a specific "key" value as part of the whole written word and that key value has to be 0x524e47, left-shifted to the most-significant bytes of the word.

All the above can be summarized shortly - in order to configure the random number generator we need to:

- enable the peripheral clock by setting bit 10 of PMC_PCER1,
- enable the generator by setting bit 0 (combined with the required key) of TRNG_CR

and in order to read each new random value we need to:

- wait for bit 0 of TRNG_ISR to ensure that the new value was produced, and
- read the new value from TRNG_ODATA.

We will implement the above rules within the structure that was already established in previous examples. The extension of the linker script will allow us to map register names to proper memory locations:

```
    PMC_PCER1 = 0x400E0700;
    TRNG_CR = 0x400BC000;
    TRNG_ISR = 0x400BC01C;
    TRNG_ODATA = 0x400BC050;
```

The `Registers` package specification should declare appropriate objects for those symbols:

```
package Registers is

   -- the rest as before

   PMC_PCER1 : Word;
   pragma Volatile (PMC_PCER1);
   pragma Import (C, PMC_PCER1, "PMC_PCER1");

   TRNG_CR : Word;
   pragma Volatile (TRNG_CR);
   pragma Import (C, TRNG_CR, "TRNG_CR");

   TRNG_ISR : Word;
   pragma Volatile (TRNG_ISR);
   pragma Import (C, TRNG_ISR, "TRNG_ISR");

   TRNG_ODATA : Word;
   pragma Volatile (TRNG_ODATA);
   pragma Import (C, TRNG_ODATA, "TRNG_ODATA");

end Registers;
```

The implementation of package `Random_Numbers` is then straightforward.

```
package body Random_Numbers is

   use type Registers.Word;

   procedure Enable_Generator is
      Key : constant := 16#524e4700#;
   begin
      Registers.PMC_PCER1 := 2#10_0000_0000#; -- 10th bit
      Registers.TRNG_CR := Key or 2#1#;
   end Enable_Generator;

   procedure Read_Next_Value (V : out Registers.Word) is
      Ready : Registers.Word;
   begin
      -- make sure new value is available:
      loop
         Ready := Registers.TRNG_ISR;
         exit when (Ready and 2#1#) /= 0;
      end loop;

      V := Registers.TRNG_ODATA;
   end Read_Next_Value;

end Random_Numbers;
```

STM32 Nucleo-144

The generator available in Nucleo-144 is named simply *Random Number Generator (RNG)* and it has a separate, dedicated chapter in the documentation. Curiously, the one in Arduino Due is called a *True Random Number Generator* - let's hope that the difference in names has no meaning other than pure marketing...

The generator works in background and produces new values in regular intervals, but in order to do so it needs to be fed with the clock signal. That is, we need to properly configure the microcontroller before we can use the generator - in particular, we need to ensure that a clock signal is delivered to the RNG submodule.

The Reset and clock control (RCC) module is responsible for enabling clock signals to all peripherals and we have already used it in relation to GPIO. In the case of RNG the relevant RCC register is named RCC_AHB2ENR and RNG can be enabled by setting bit 6 of this register. This is not enough, however, since RNG is clocked with a dedicated clock line that is meant to be independent from the main system clock (the USB subsystem is also connected to that dedicated clock) - as you can see from the *Clock tree* diagram in the documentation, that dedicated clock can be sourced from the PLL module, which itself needs to be separately enabled by means of bit 24 in register RCC_CR. This is the additional step that was not necessary with GPIO.

Once the peripheral clock is enabled, the random number generator itself is easy to operate. The registers that are of interest to us are RNG_CR (control register to enable the generator), RNG_SR (status register with a flag to signal that a new random value was produced) and RNG_DR, the random value itself.

All the above can be summarized shortly - in order to configure the random number generator we need to:

- enable the PLL clock source by setting bit 24 of RCC_CR,
- enable the RNG peripheral clock by setting bit 6 of RCC_AHB2ENR,
- enable the generator itself by setting bit 2 of RNG_CR

and in order to read each new random value we need to:

- wait for bit 0 of RNG_SR to ensure that the new value was produced, and
- read the new value from RNG_DR.

We will implement the above rules within the structure that was already established in previous examples. The extension of the linker script will allow us to map register names to proper memory locations:

```
    RCC_CR = 0x40023800;
    RCC_AHB2ENR = 0x40023834;
    RNG_CR = 0x50060800;
    RNG_SR = 0x50060804;
    RNG_DR = 0x50060808;
```

The `Registers` package specification should declare appropriate objects for those symbols:

```
package Registers is

   -- the rest as before

   RCC_CR : Word;
   pragma Volatile (RCC_CR);
   pragma Import (C, RCC_CR, "RCC_CR");

   RCC_AHB2ENR : Word;
   pragma Volatile (RCC_AHB2ENR);
   pragma Import (C, RCC_AHB2ENR, "RCC_AHB2ENR");

   RNG_CR : Word;
   pragma Volatile (RNG_CR);
   pragma Import (C, RNG_CR, "RNG_CR");

   RNG_SR : Word;
   pragma Volatile (RNG_SR);
   pragma Import (C, RNG_SR, "RNG_SR");

   RNG_DR : Word;
   pragma Volatile (RNG_DR);
   pragma Import (C, RNG_DR, "RNG_DR");

end Registers;
```

The implementation of package `Random_Numbers` is then straightforward.

```
package body Random_Numbers is

   use type Registers.Word;

   procedure Enable_Generator is
   begin
      Registers.RCC_CR := Registers.RCC_CR
         or 2#1_0000_0000_0000_0000_0000_0000#; -- bit 24
      Registers.RCC_AHB2ENR := Registers.RCC_AHB2ENR
         or 2#100_0000#; -- bit 6
      Registers.RNG_CR := 2#100#; -- bit 2
   end Enable_Generator;

   procedure Read_Next_Value (V : out Registers.Word) is
      Ready : Registers.Word;
   begin
      -- make sure new value is available:
      loop
         Ready := Registers.RNG_SR;
         exit when (Ready and 2#1#) /= 0;
      end loop;

      V := Registers.RNG_DR;
   end Read_Next_Value;

end Random_Numbers;
```

Note that it is possible to use logical operators (`and`, `or`) on values of modular types like `Word` - they allow to manipulate individual bits and their patterns. However, since these operators come from the `Registers` package, they are not directly visible by default - the `use type` construct allows them to be used in their natural form in this file. Interestingly, it is not possible to use such bitwise operations on regular integer types. The `exit when` construct allows to break the loop if some given condition is met - this construct can be used in other loop forms as well.

With the `Pins` package already supporting digital output control and our new package for random numbers, it is now possible to implement more animated projects - the following main program will randomly blink two pins that we have already used in previous examples:

```
with Pins;
with Random_Numbers;
with Registers;
with Utils;

package body Program is

   use type Registers.Word;

   procedure Run is
      R : Registers.Word;
      Bit_0 : Boolean;
      Bit_1 : Boolean;
   begin
      Pins.Enable_Output (Pins.Pin_11);
      Pins.Enable_Output (Pins.Pin_12);

      Random_Numbers.Enable_Generator;

      loop
         Random_Numbers.Read_Next_Value (R);
         Bit_0 := (R and 2#01#) /= 0;
         Bit_1 := (R and 2#10#) /= 0;

         Pins.Write (Pins.Pin_11, Bit_0);
         Pins.Write (Pins.Pin_12, Bit_1);

         Utils.Waste_Some_Time;
      end loop;
   end Run;
end Program;
```

Note that variable `R` in the main program is used as an actual for output parameter `V` when `Read_Next_Value` is called - the effect is that new random values are written to variable `R`.

The complete compilation command sequence (common for both Arduino Due and Nucleo-144) for this program is:

```
$ gcc -c -gnatp -mcpu=cortex-m3 -mthumb pins.adb
$ gcc -c -mcpu=cortex-m3 -mthumb program.adb
$ gcc -c -mcpu=cortex-m3 -mthumb random_numbers.adb
$ gcc -c -mcpu=cortex-m3 -mthumb utils.adb
$ ld -T flash.ld -o program.elf pins.o program.o random_numbers.o
        utils.o
$ objcopy -O binary program.elf program.bin
```

And of course the `program.bin` is the binary image that needs to be loaded to the flash memory of the microcontroller.

The "Christmas Tree" category of embedded systems is not the only one that can benefit from the random number generator - the one provided in Arduino Due can produce new values with good statistical properties as often as every 84 clock cycles and the one in Nucleo-144 every 40 cycles, which allows to produce noise within a wide signal band. This makes it a good candidate for use in many real-time signal processing systems, for example in the audio domain.

Previous: Very Simple Delays, next: Digital Input.
See also Table of Contents.

Did you find this article interesting? Share it!

SHARE