14. Shared State

We have been quite lucky to go so far in this tutorial without any need to maintain program state across procedure invocations. In other words, the programs that we have written up to now relied on stack variables to manage their state - before introducing interrupts, this was possible, because the whole program was executing in the context of some top-level procedure (in our case it was the `Run` procedure) that used its variables for the whole program's execution, perhaps sharing them by means of passing parameters to other called procedures.

Introducing interrupts into the design means that this approach is no longer valid if some state needs to be shared between different handlers, because interrupt handlers do not have any visible common ancestor on the call stack. In other words, if there is a need to share state between different interrupt handlers, this state needs to be implemented in terms of other constructs.

We have already used shared *constants*, like in this example from one of the previous chapters:

```
package body Program is

   X : constant Integer := 123;

   procedure Run is
   -- ...

end Program;
```

Above, `X` is a constant object that exists *outside* of the `Run` procedure - and in fact outside of any other procedure in the same package, which means that all such procedures can see this object in the same way. In other words, `X` is *shared*.

Unfortunately, it is also constant, and we have verified that with our current linker script such objects end up in the address range of the flash memory.

We need a way to create shared variable objects like this one:

```
package body Program is

   X : Integer;

   procedure Run is
   begin
      loop
         null;
      end loop;
   end Run;

end Program;
```

and provide appropriate instructions to the linker to have such objects placed in RAM instead.

If you compile the above file with the `-S` option for assembly output, you will notice that symbol `X` was marked as belonging to section ".bss":

```
        .bss
        .align  2
program__x:
        .space  4
```

The name ".bss" has a long history, but nowadays it refers to the program section that contains *uninitialized static data*. Objects from this section are not included in the linked image (that is, they do not take space in the loadable program image), but information about their addresses and sizes is available to the program loader. Most runtime systems make additional arrangements to clear all such objects by overwriting them with zeros before the user-implemented part of the program begins, but in this tutorial we will consciously decide not to do it in order to keep our programs clear from any hidden or implicit code, and will simply use the section mark for managing address ranges. This can be done with this additional section definition in the linker script:

```
    .bss 0x20070000 :
    {
        *(.bss)
    }
```

The address 0x20070000 is the example *beginning* of the RAM block in the microcontroller (this of course might be different for each chip and should be checked in the memory mapping sections of the vendor documentation). Note that similarly to the top-of-stack address, it is not strictly necessary to use RAM from its boundary addresses and there might be valid reasons to set up up different ranges for both stack and shared data, but setting them on opposite boundaries (that is, shared data at the lower end and stack at the higher end) is a common approach.

After linking this simple program we can see how individual symbols were assigned:

```
$ nm program.elf
20070000 b program__x
0008010c D program_E
00080100 T run
```

As you can see, the symbol denoting variable `X` was assigned address at the beginning of the RAM region, just as we need. If there are more similar objects in the program, they will have addresses assigned consecutively from the beginning of RAM.

Note that such variables do not take space in the linked program and after converting the ELF file to raw binary image, the size of image file is not affected by the presence of such objects in the program. Literally, these objects do not exist until the program starts executing and referring to appropriate memory addresses in RAM.

The above approach is very simple and for our purposes very effective, but the simplified treatment of the `.bss` section marker has its consequences: such objects are not initialized when the program starts executing, even if Ada language rules clearly define initial values for the given type. In a full Ada runtime (this is true for C and C++ as well) there would be an automatically injected code that takes care of the initialization for such variables, but in this tutorial we have decided not to rely on existing runtimes - as a consequence, we need to keep in mind that objects created this way are indeed not initialized. The program has to explicitly write initial values before first use, for example like here:

```
package body Program is

   X : Integer;

   procedure Run is
   begin
      X := 123; -- initial value

      -- ...
   end Run;

end Program;
```

Note that it is *not* possible to provide initial values in this way:

```
package body Program is

   X : Integer := 123; -- do NOT do this

   procedure Run is
   -- ...

end Program;
```

because as soon as you provide initial value together with the declaration, such object will be marked as `.rodata` and will be placed in the flash memory. Full runtimes make sure that such initialization values are copied to RAM at the program startup, but we will not rely on this (again - in order to keep our program free from hidden and implicit code).

The following example is a cheat-sheet summary of what works in our simplified setup:

```
package body Program is

   -- uninitialized shared variable in RAM:
   X : Integer;

   -- do NOT do this:
   Y : Integer := 123;

   -- constant (read-only) object in flash:
   Z : constant Integer := 123;
```

```
   procedure Run is
   begin
      X := 123; -- properly initialized variable in RAM:

      -- ...
   end Run;
end Program;
```

Note that depending on how shared variables are used, it might be beneficial to declare them as volatile objects, so that all accesses to them are effective at the level of memory operations. We will point it out when such a situation arises in the next chapters.

Now, having a way to create variable objects that can be shared between procedures and that can retain their state between subsequent invocations, we can create stateful event-based systems.

Arduino M0 and STM32 Nucleo-32

As you remember from the earlier chapter about random numbers, the simpler boards do not have any hardware support for random number generation and we had to rely on software algorithms to produce pseudo-random sequences - these, however, required the possibility to retain some state between invocations to actually produce *new* values in the sequence.

We were not able to finish these examples before, but once the subject of sharing state was sorted out, we can attempt to write some more reasonable implementation of the software-based random number package.

There are many software algorithms for generating pseudo-random numbers and this turorial is not intended to explain them (and definitely not to analyse their mathematical properties), but the general idea is to compute new pseudo-random value based on the previously computed value, which therefore has to be retained between generator invocations. We can try the following scheme as a sketch of our already specified package `Random_Numbers`:

```
package body Random_Numbers is

   State : -- shared state of some type

   procedure Enable_Generator is
   begin
      State := -- initialize state
   end Enable_Generator;

   procedure Read_Next_Value (V : out Registers.Word) is
   begin
      State := -- compute new state based on the previous value
      V := -- compute the output value from the new state
   end Read_Next_Value;

end Random_Numbers;
```

As you can see, `State` is a shared object that should be kept in RAM so that it can retain its value between consecutive invocations of procedure `Read_Next_Value`.

There are many possible ways to implement this scheme, but for demonstration purposes we can use something that resembles the structure of the linear congruential generator, where we will rely on the modulo arithmetic on the `Word` type:

```
with Registers;

package body Random_Numbers is

   Initial_State : Registers.Word;
   State : Registers.Word;

   procedure Enable_Generator is
   begin
      Initial_State := 16#12345678#;
      State := Initial_State;
   end Enable_Generator;

   procedure Read_Next_Value (V : out Registers.Word) is
      New_State : Registers.Word;
      use type Registers.Word;
   begin
      New_State := State * 16#42# + 7;
      if New_State = State then
         Initial_State := Initial_State + 1;
         New_State := Initial_State;
      end if;

      State := New_State;
      V := State;
   end Read_Next_Value;
```

```
end Random_Numbers;
```

This generator probably would not pass the cryptography robustness test (as one of its many shortcomings, note that it always generates the same sequence), but is sufficient for the blinking LED kind of projects and demonstrates how a single package can be implemented in two different ways depending on the capability of the underlying hardware - the application code can use the `Random_Numbers` package independently of the target board, but will get better results on more capable boards that are equipped with real hardware generators.

Previous: Interrupts, next: Finite State Machines, Part 3.
See also Table of Contents.

Did you find this article interesting? Share it!