12. Machine Code Insertions

According to the official documentation, the ARM Cortex-M microcontrollers were designed to make programming easier - in particular by not forcing programmers to use any assembly code. Indeed, to a large extent this objective was achieved and as you have seen in this tutorial we had no need to use assembly language in any of the previous examples. It was instructive to analyse assembly output from the compiler, but there was no need to implement any functionality in assembly code.

The no-assembly objective was not met in 100% and there are several CPU instructions that have special meaning and which are not directly generated by the compiler from any high-level construct. These special instructions are related to:

- explicit breakpoints,
- enabling and disabling interrupts,
- memory access and synchronization barriers,
- communicating events and supervisor invocations,
- possibly some even more obscure issues.

Some of the things above become useful only in implementation of operating systems, but even in single-task programs there are a couple things that allow the program to manage interrupts - these are enabling, disabling, and suspending execution until interrupt. That is, we need a way to invoke one or two magic assembly instructions from our Ada programs in order to benefit from interrupts. Fortunately, these instructions have no arguments and this makes them easier to integrate with the rest of our code.

There are two ways to achieve this - one of them relies on the language facility for inserting machine code into the program, which is provided by the standard package `System.Machine_Code`. If this package is visible, then arbitrary assembly instruction can be inserted in code like this:

```
System.Machine_Code.Asm (Template => "some_machine_instruction",
   Volatile => True);
```

There are three machine instructions that we will need in later chapters, these are:

```
cpsie i
```

for enabling interrupts,

```
cpsid i
```

for disabling interrupts, and

```
wfi
```

for suspending execution of the program until some interrupt arrives (this single instruction allows to reduce the power consumption of the system that apart from waiting for various events has nothing else to do).

We will wrap the above instructions into Ada parameterless procedures that will be conveniently defined in package `Utils`. That is, we will put these declarations in the package specification:

```
procedure Enable_Interrupts;

procedure Disable_Interrupts;

procedure Wait_For_Interrupt;
```

and the following implementations in the body of `Utils`:

```
   procedure Enable_Interrupts is
   begin
      System.Machine_Code.Asm
        (Template => "cpsie i", Volatile => True);
   end Enable_Interrupts;

   procedure Disable_Interrupts is
   begin
      System.Machine_Code.Asm
        (Template => "cpsid i", Volatile => True);
   end Disable_Interrupts;

   procedure Wait_For_Interrupt is
   begin
      System.Machine_Code.Asm
        (Template => "wfi", Volatile => True);
   end Wait_For_Interrupt;
```

To verify that indeed these instructions are inserted in code, we can check the assembly output when compiling the whole package with `-S` option - the resulting `utils.s` file should contain sequences similar to this:

```
utils__enable_interrupts:
        @ args = 0, pretend = 0, frame = 0
        @ frame_needed = 1, uses_anonymous_args = 0
        @ link register save eliminated.
        push    {r7}
        add     r7, sp, #0
#APP
@ 22 "utils.adb" 1
        cpsie i
@ 0 "" 2
        .thumb
        mov     sp, r7
        pop     {r7}
        bx      lr
```

Indeed, the `cpsie i` instruction was inserted in the code of the otherwise empty procedure `Enable_Interrupts`. Any other parameterless instruction can be inserted in the same way.

You might find that many existing run-time libraries prefer to define such functions in a way that encourages the compiler to inline them. You can experiment with `pragma Inline` and `-O2` compiler option, but for the sake of simplicity we will not make these examples more obscure than necessary.

From now on, the program that for example needs to enable interrupts can do so by calling our wrapper procedure:

```
      Utils.Enable_Interrupts;
```

Another way to achieve a similar effect is to implement such wrapper procedures entirely in assembly (you can use assembly output from compiling empty procedures as a convenient starting point for such work), compile the assembly file to the object file and link it with the rest of the program that can then declare such procedures with `pragma Import`. The final effect will be more or less the same, but the build process will involve additional files with assembly code. The standard package `System.Machine_Code` allows to keep everything within Ada source files and for simple code insertions seems to be a better choice.

Previous: Finite State Machines, Part 2, next: Interrupts.
See also Table of Contents.