10. Constant Values

We have already used constant values in previous chapters, but we did not give them much attention. The reason why there is a separate chapter devoted to them is that there are several kinds of constant values in Ada and they can be used to solve different problems - all these details deserve some explanation.

The simplest kind of constant value in Ada is a *literal*, which is an unnamed entity that can be used directly in expressions or to initialize other entities. Some spectacular examples include lengthy bit patterns that we have used in the `Pins` package to operate on individual register bits, like here:

```
when Pin_4 =>
   Registers.PIOC_PER :=
      2#100_0000_0000_0000_0000_0000_0000#; -- bit 26
   Registers.PIOC_OER :=
      2#100_0000_0000_0000_0000_0000_0000#;
```

The literal values above are written with generous use of underscores, which are ignored in numbers, but help to visually separate groups of digits. Note that even though the above literals are clearly numbers, there is no need to think about them as having any specific type. They are assigned to objects of some specific type and this is when the typing gets sorted out. What is also important is that they do not exist as entities in RAM (consequently, they do not take up any space in RAM), they are simply encoded in the stream of instructions that form the program's executable code. This also means that they do not have addresses.

The above example is acceptably readable, but obviously it could benefit from some refactoring and this is where *named numbers* can be useful. We have already used a named number in the `Random_Numbers` package for the Arduino Due, where a magic key value was needed for configuration of the random number generator. It was used, although not explained, like this:

```
      Key : constant := 16#524e4700#;
```

and later on the `Key` name was used in expression that also involved other, unnamed literal:

```
      Registers.TRNG_CR := Key or 1;
```

Named literals can make the code more readable and we can use similar strategy for bit patterns in the `Pins` package, for example:

```
   Bit_0  : constant :=                                  2#1#;
   Bit_1  : constant :=                                 2#10#;
   Bit_2  : constant :=                                2#100#;
   -- ...
   Bit_26 : constant :=  2#100_0000_0000_0000_0000_0000_0000#;
   -- ...
```

and then:

```
         when Pin_4 =>
            Registers.PIOC_PER := Bit_26;
            Registers.PIOC_OER := Bit_26;
```

There is still room for improvement, as named numbers can be initialized with *static expressions*, so instead of typing all these bit patterns we can express the arithmetic nature of the whole series like here:

```
   Bit_0  : constant := 1;
   Bit_1  : constant := 2 * Bit_0;
   Bit_2  : constant := 2 * Bit_1;
   -- ...
   Bit_26 : constant := 2 * Bit_25;
   -- ...
```

This is much less tedious to write, but at the same time we have lost the graphical representation of relations between all bit patterns and since these constants have to be defined once and will never be changed, it makes sense to invest a bit of time in more readable format - the choice between these alternatives is a matter of engineering judgment. In any case, such bit patterns can be useful not only in the `Pins` package, but also in other packages that need to operate on hardware registers, so we will put these definitions in the `Registers` package instead.

Note that named numbers are similar to unnamed literals in that we can consider them to have effectively no type (the Ada Reference Manual has a precise answer to this and that becomes important when such a constant is used in a call to an overloaded subprogram, for example - but we can ignore such issues in this tutorial) and that they do not exist as run-time entities in memory. They are basically compile-time constructs and end up encoded within the instruction stream. Consequently, they do not have any addresses.

These are not the only possibilities and there is a third form that will be of interest to us, as it allows to create constants of our own types. A simple example is:

```
   X : constant Integer := 123;
```

Above, `X` is a constant *object*. We can only read it, but since it is a real object, it exists as a run-time entity as well. It has a type and one of the properties of the type is the amount of space that the object needs in memory. Consequently (and unlike named numbers above), such constant objects have addresses.

But what addresses, exactly?

That depends on where in the program such constant is defined. Consider the following simple program example:

```
package body Program is

   X : constant Integer := 123;

   procedure Run is
      Y : constant Integer := 124;
   begin
      loop
         null;
      end loop;
   end Run;

end Program;
```

In this example, `X` was defined at package level and is not related to any subprogram, whereas `Y` was defined as a local entity in the `Run` subprogram. If we compile this program and ask for the assembler output:

```
$ gcc -S -mcpu=cortex-m3 -mthumb program.adb
```

we will find the following part in the `program.s` output file:

```
        .section        .rodata
        .align  2
        .type   program__x, %object
        .size   program__x, 4
program__x:
        .word   123
        .text
        .align  2
        .global run
        .thumb
        .thumb_func
        .type   run, %function
run:
        @ args = 0, pretend = 0, frame = 8
        @ frame_needed = 1, uses_anonymous_args = 0
        @ link register save eliminated.
        push    {r7}
        sub     sp, sp, #12
        add     r7, sp, #0
        mov     r3, #124
        str     r3, [r7, #4]
.L4:
        nop
        nop
        b       .L4
```

and, after actually compiling, linking and analyzing the executable ELF file with `nm` we get:

```
$ gcc -c -mcpu=cortex-m3 -mthumb program.adb
```

```
$ ld -T flash.ld -o program.elf program.o
$ nm -S program.elf
...
00080114 00000004 r program__x
00080118 00000002 D program_E
00080100 00000012 T run
...
```

What it means is that `X` exists as a real symbol and was allocated 4 bytes in the linked file. It was also marked as read-only (note the `.rodata` part in assembler and the `r` flag in the output of `nm`) and automatically included in the binary output file with symbol address 0x00080114. We can expect to see it at the offset of 0x114 bytes from the beginning of the binary image after running objcopy on the ELF file. Indeed, this is what we can see:

```
$ objcopy -O binary program.elf program.bin
$ od -A x -t x4 program.bin
...
000110 bf00e7fc 0000007b 00000000
000119
```

The word 0x0000007b is 123 in decimal, which is exactly the initialization value for `X`.

What all this means is that constant objects like `X` are properly linked and are included in the binary image that we would normally load into flash when programming the board. There is nothing else that we need to do in order for this to happen and curiously, even the linker script did not require any modifications - the constant objects are included right after the `.text` section that keeps the executable code for all subprograms. This is satisfactory - in particular note carefully that such objects take up the flash space (which is read-only from the program's point of view, but there is a lot of it), but do not take up any RAM, which has to be used sparingly. Some embedded programming environments prefer to copy such objects from flash into RAM as part of the program initialization, but from our point of view there is no benefit in doing it. We will return to this subject in later chapters, but at this stage it is important to remember that the above analysis is true only for constants that have simple, but complete static initializers (that is, they are not initialized by function calls or default values).

What about the `Y` object? It does not exist as a symbol in the compiled files and it only takes up the stack space, although the exact size is buried in the overall handling of the stack pointer. Still, we can see in the assembly output that the procedure `Run` properly initializes this constant every time it is executed (note the `mov` and `str` pair of instructions). This is satisfactory and there is nothing else that we need to do to make it work.

The above explorations allow us to expand our programming toolset and we will use constant values extensively in later programs.

Previous: Finite State Machines, Part 1, next: Finite State Machines, Part 2.
See also Table of Contents.