20. Loose Ends

Congratulations!

We have arrived at the last chapter of this tutorial and we have managed to solve a number of fundamental problems on the way. In particular, the last example program, Hello World!, was already a pretty complete demonstration of how Ada can be used in embedded systems based on ARM Cortex-M microcontrollers.

Still, there are several subjects that were not covered but that deserve at least a bit of awareness (and a thorough exploration when needed), as they can emerge in real embedded projects.

Clocks

In this tutorial we have basically ignored the subject of clock configurations.

Apart from enabling clock signals for selected peripherals, we have always assumed that the microcontroller has the main system clock *just working* by default. This assumption is true, because the chip designers make sure to provide some reasonable clock configuration out of the box by means of internal oscillators that become active right after power on - this allows to use these microcontrollers even in systems that do not have any other (external) clock sources. By relying on such safe defaults we could keep the initialization code very simple, but with the disadvantage of not using the full capability of the example boards, which have complete and sophisticated clock systems already built in. In particular, our programs do not use the full clock frequency potential of the underlying hardware.

This omission was not a problem in any of our systems - note that even the most "advanced" example program, Hello World!, relied on artificial delays in order to fit the timing expectations of the LCD display, so increasing the processing speed would not provide any advantage. Still, the unused potential of these microcontrollers should be kept in mind when designing systems that are more demanding with regard to the processing speed (note also that more elaborate clock configurations might be needed by some of the peripherals like USB or Ethernet, independently on the data processing load) and the reference documentation for each chip always provides very precise information on what are the options in this area.

Floating-Point Arithmetic

Floating-point arithmetic is of course supported by Ada and the type system allows programmers to define types with required range and accuracy. What is interesting here is that only some of ARM Cortex-M microcontrollers support such operations *natively* in the sense that they understand a dedicated subset of instructions for handling floating-point values. The floating-point instructions are supported by Cortex-M4 microcontrollers (or higher), but not by Cortex-M3, which is installed in Arduino Due and definitely not by Cortex-M0, which is used in the smallest of our boards. It is still possible to compile and link programs that use floating-point types and operations, but on microcontrollers that do not have a hardware FPU, the compiler will implement floating-point operations in terms of calls to functions from the `fplib` library, which has to be linked into the final program.

Consider this simple package specification and body:

```
 package Test is

   type F is digits 8 range 0.0 .. 1_000.0;

   function Compute (X : in F; Y : in F) return F;

end Test;

package body Test is

   function Compute (X : in F; Y : in F) return F is
   begin
      return 2.34 * X + Y;
   end Compute;

end Test;
```

If we compile this package using the same set of compiler options as before:

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

then we will get the assembly output similar to this:

```
        ...
        .type   test__compute, %function
test__compute:
        @ args = 0, pretend = 0, frame = 16
        @ frame_needed = 1, uses_anonymous_args = 0
        push    {r7, lr}
        sub     sp, sp, #16
        add     r7, sp, #0
        strd    r0, [r7, #8]
        strd    r2, [r7]
        ldrd    r0, [r7, #8]
        adr     r3, .L3
        ldrd    r2, [r3]
        bl      __aeabi_dmul
        mov     r2, r0
        mov     r3, r1
        mov     r0, r2
        mov     r1, r3
        ldrd    r2, [r7]
        bl      __aeabi_dadd
        mov     r2, r0
        mov     r3, r1
        nop
        mov     r0, r2
        mov     r1, r3
        add     r7, r7, #16
        mov     sp, r7
        pop     {r7, pc}
        ...
```

The `__aeabi_dmul` and `__aeabi_dadd` symbols denote functions that are implemented in the `fplib` library. Without this library the complete program will not link.

If, however, we compile this package for the Cortex-M4 microcontroller, which has a hardware FPU unit and we will instruct the compiler to use it like here:

```
$ gcc -S -gnatp -mcpu=cortex-m4 -mthumb -mfloat-abi=hard test.adb
```

then the assembly output will look differently:

```
 test__compute:
       @ args = 0, pretend = 0, frame = 16
       @ frame_needed = 1, uses_anonymous_args = 0
       @ link register save eliminated.
       push    {r7}
       sub     sp, sp, #20
       add     r7, sp, #0
       fstd    d0, [r7, #8]
       fstd    d1, [r7, #0]
       fldd    d6, [r7, #8]
       fldd    d7, .L3
       fmuld   d6, d6, d7
       fldd    d7, [r7, #0]
       faddd   d7, d6, d7
       nop
       fcpyd   d0, d7
       add     r7, r7, #20
       mov     sp, r7
       pop     {r7}
       bx      lr
```

As you see, in the above snippet there are no calls to `fplib` functions, and `fxxx` instructions are used directly to execute operations on floating-point values. Of course, if the target microcontroller supports floating-point operations natively, this approach will be the most effective (and convenient), but code compiled for use with `fplib` will also work properly - this has to be taken under consideration if portability of executable code is required.

The `fplib` library can be found in Arduino IDE and other development environments.

Last, but important note: proving any properties about programs containing floating-point operations with SPARK can be much more involving than with integral types.

Direct Memory Access

Direct Memory Access, or DMA for short, is a hardware-level mechanism that allows to transfer data without involvement from the program code. That is, the program can configure and set up the parameters of the data transfer, which is then automated. DMA can be used with some peripherals to automate processing of consecutive data items.

There are no benefits of using DMA with relation to single reads and writes with digital I/O pins - such operations are best executed as regular register accesses, just like in our example programs. The real value of DMA becomes apparent when the data transfer involves a whole block of data. The more capable chips that are used in Arduino Due and Nucleo-144 were designed to be very versatile communication interfaces and have hardware support for many communication protocols, from simple serial communication to USB and Ethernet. This is where the DMA can be used to pass the responsibility for data transfer to the microcontroller and leave the main program free to do other tasks - or even to do nothing at all.

Note that in order to set up DMA transfers it is always necessary to provide addresses of memory blocks that are supposed to be sources or receivers of data transfer. We have managed to avoid using addresses and pointers directly in the source code (all symbols and memory allocations were handled at the level of linker script), but DMA might require that in order to properly configure data transfer parameters, addresses of involved memory blocks are directly written to some registers. The address of any given variable can be obtained with the `'Address` attribute, and constrained (fixed) array objects happen to have addresses correctly pointing at their first elements. The following example shows how the address of some variable can be obtained:

```
with System;

package body Program is

   X : Integer;

   X_Ptr : System.Address := X'Address;

   type Byte is mod 2**8;
   type Byte_Array is array (Integer range <>) of Byte;

   Buffer : Byte_Array (1 .. 100);

   Buffer_Ptr : System.Address := Buffer'Address;

   -- ...
```

Above, `X_Ptr` contains the address of shared variable `X` and `Buffer_Ptr` contains the address of the beginning of the shared object `Buffer`, which is a byte array of fixed size - that latter example can be particularly useful for DMA transfers of longer data sequences. For this we can also safely assume that the GNAT compiler does not introduce *unnecessary* padding between array elements.

Note that `System.Address` is *not* an integer type and unlike in C, arithmetic operations on such values are not directly supported (although addresses can be *compared* to each other). If arithmetic with buffer offset is needed, the necessary operations and conversion functions can be found in the predefined package `System.Storage_Elements`.

Analog I/O

We have also ignored the fact that our boards have whole rows of pins for analog input and output. Arguably, processing analog signals is also a place where floating-point data types and operations might prove useful, but this will depend on the particular domain and algorithm preferences.

Managing both digital-analog and analog-digital converters is possible from Ada, but the microcontroller documentation should be read with care in this area - even though there are many physical pins that can be used for the purpose of analog I/O, internally they are processed as multiple channels of a single hardware component and the conversion is not handled independently for each channel.

Interestingly, analog I/O is also a very nice candidate for DMA, which can handle a continuous stream of samples with little jitter and without involving any processing at the software level. The complete configuration of this setup can be complex, but might be worth the effort in audio or similar real-time signal processing applications.

Dynamic Memory Allocation

Dynamic memory allocation requires extensive support from the language runtime, especially if the intent is to retain the natural language syntax and predefined operators. Ada's support for dynamically allocated objects is very sophisticated and apart from simple creation of objects on the free store allows creating and redefining hierarchies and pools of allocators, even separately for each access type.

Such extensive support is usually not needed in embedded systems and for this reason it is not justified to reinvent (or reimplement) all possible mechanisms in this area - in fact, even a simple general-purpose memory allocator can turn to be quite complex and difficult to implement correctly with regard to all possible corner-cases. Interestingly, many process guidelines and coding standards that were written with embedded systems in mind define severe limits on how dynamic

memory can be used in such systems, or even forbid this feature altogether, proposing some reasonable alternatives for constrained systems.

A very effective alternative (if the notion of dynamic allocation is needed at all) that should be taken into account is a *pool of objects* - whenever a new object is needed, the next object from the pool is assigned for the given task and deallocation means simply marking the object as ready for further reuse. The size of the pool should be big enough to allow allocation of all objects that will be ever needed at run time in the given system. This is relatively easy to do with simple arrays, but has the disadvantage of requiring a *pessimistic sizing*: if many different pools are needed for objects of different types, then the sum of their sizes might not fit into available memory, even if the program as a whole will never use all objects at the same time. Still, even with this constraint, this approach can be very useful in embedded systems, as it is quite easy to implement without any support from the language runtime library.

Object-Oriented Programming

Object-oriented programming (OOP) is not frequently used in embedded systems and many design and coding standards are not prepared with this paradigm in mind, but with the growing complexity of such systems (enabled by the growing computing power of embedded devices) we should expect also a growing pressure from programmers to allow object-orientation as a valid design tool. There is nothing wrong with that and even safety-critical processes and coding standards were updated with appropriate guidelines in this area. From the perspective of this tutorial, however, the most important part is to discover the issues related to the compilation of programs that use language features related to OOP.

Ada supports OOP in the way that is physically similar to the object model known from C++, with analogous language features:

- there is a concept of a *class* as a record-like type that binds together data and operations on that data - in Ada this is achieved with packages and record types,
- classes can form the *inheritance hierarchies* to allow extension and composition of types,
- there is a notion of *virtual operations* that can be *overriden* in derived classes - in Ada these are called *primitive operations* of the given type, and are implemented as functions and procedures that involve the given type as one of their parameters,
- there exists a run-time mechanism that allows a *dynamic dispatch* of operations based on the dynamic types of objects - in C++ this is what happens when a virtual function is called via a pointer or reference to the base class.

It is this last element that is the biggest challenge in embedded systems, as it involves some run-time support from the language. In the case of GCC, dynamic dispatch is implemented for both Ada and C++ in terms of a *v-table*, which is an array of addresses of actual method bodies for the given type - this array is bound to each object, but can have different content in each type in the hierarchy. The calls to operations are performed indirectly via this array of addresses and this is how the proper operation is selected at the call site. We have tried to avoid any additional run-time features in this tutorial, but fortunately the compiler generates all code and data that are needed for this mechanism and no language run-time support libraries are needed to link the complete program.

Consider this simple example that in the embedded system could be used for communication over various channels with serial channel being one particular specialization of general communication channel concept:

```
-- comm_channels.ads:
package Comm_Channels is

   pragma Pure;

   type Byte is mod 2**8;

   -- this is a base type for all possible channels:
   type Channel is interface;

   -- primitive abstract operation of the channel type:
   procedure Put (Ch : in Channel; V : in Byte) is abstract;

end Comm_Channels;

-- serial_channels.ads:
with Comm_Channels;

package Serial_Channels is

   -- this is the derived type, specialization for serial channels:
   type Channel is new Comm_Channels.Channel with null record;

   -- overriding operation for serial channels:
   procedure Put (Ch : in Channel; V : in Comm_Channels.Byte);

end Serial_Channels;

-- serial_channels.adb:
package body Serial_Channels is
```

```
   -- implementation of data transfer over serial channel:
   procedure Put (Ch : in Channel; V : in Comm_Channels.Byte) is
   begin
      -- whatever is needed here...
      null;
   end Put;

end Serial_Channels;

-- test.ads:
package Test is
   -- our typical main procedure:
   procedure Run;
   pragma Export (C, Run, "run");
end Test;

-- test.adb:
with Comm_Channels;
with Serial_Channels;

package body Test is

   -- dispatching procedure that takes base class as a parameter:
   procedure Put_Byte (Ch : in Comm_Channels.Channel'Class;
      B : in Comm_Channels.Byte) is
   begin
      -- dynamic dispatch happens here,
      -- based on the actual object type:
      Ch.Put (B);
   end Put_Byte;

   -- object of derived type:
   Serial : Serial_Channels.Channel;

   procedure Run is
   begin
      -- here we use the object of derived type
      -- with the procedure that accepts base type as a parameter:
      Put_Byte (Serial, 123);

      loop
         null;
      end loop;
   end Run;

end Test;
```

The above complete example is more-or-less equivalent to the following C++ code sketch:

```
// this is a base type for all possible communication channels:
class comm_channel
{
public:
    // abstract operation:
    virtual void put(uint8_t v) = 0;
};

// this is the derived type, specialization for serial channels:
class serial_channel : public comm_channel
{
public:
    // overriding operation for serial channels,
    // implementation of data transfer over serial channel:
    virtual void put(uint8_t v)
    {
        // whatever is needed here...
    }
};

// dispatching procedure that takes base class as a parameter:
void put_byte(comm_channel & ch, uint8_t b)
{
    // dynamic dispatch happens here, based on the actual object type:
    ch.put(b);
}

void run()
{
    // object of derived type:
    serial_channel serial;

    // here we use the object of derived type
    // with the procedure that accepts base type as a parameter:
    put_byte(serial, 123);

    while (true)
```

```
    {
    }
}
```

If we try to compile and link the Ada implementation files the usual way, we will see that the linker is not able to resolve all symbols:

```
$ gcc -c -gnatp serial_channels.adb
$ gcc -c -gnatp test.adb
$ ld -T flash.ld -o test.elf serial_channels.o test.o
serial_channels.o:(.rodata+0x40): undefined reference to
`comm_channels__channelT'
```

The missing symbol `comm_channels__channelT` names a data item that contains the v-table that is needed for the dynamic dispatch.

Interestingly, this symbol cannot be resolved by just compiling the package bodies (`.adb` files), but the missing parts can be generated by compiling the package specification for the base class - even though seemingly there is no executable code defined there:

```
$ gcc -c -gnatp comm_channels.ads
```

Curiously, if you try to compile other package specification files, you will see compiler errors like this one:

```
$ gcc -c -gnatp serial_channels.ads
cannot generate code for file serial_channels.ads (package spec)
```

This curiosity has to be kept in mind when compiling complete programs. In any case, now we have complete set of files to resolve all symbols:

```
$ ld -T flash.ld -o test.elf comm_channels.o serial_channels.o test.o
```

Note that even though object-oriented techniques do not necessarily force the programmer to use dynamic memory allocation, these two are very often used together and only the most simple design patterns can be reasonably implemented without support from the dynamic memory. Whether this constraint is acceptable in the context of embedded systems is a matter of engineering judgement. See also the earlier note about dynamic allocation, where an alternative approach with object pools is described - this technique can be used with object hierarchies, with separate pools created for objects of those concrete types in the inheritance hierarchy that need dynamic allocation support.

Multithreading

Multithreading is a feature that is typically organized at the level of the operating system and even though there are implementations of Ada (and other languages) that support high-level multitasking even within the context of a single thread of execution, the GNAT compiler refers to primitives from the underlying operating system. This works fine in the desktop or server environment. In the embedded system where a user program is the only loaded software entity, there is no underlying runtime layer that would provide necessary support for software-level multitasking. A possible solution could be to integrate the user program within the environment managed by a Real-Time Operating System (RTOS) and there are several products that can be used with ARM Cortex-M microcontrollers - still, such products are mostly designed with C programs in mind and most likely they will not provide any support for the language-level features that Ada offers with regard to multitasking. The integration of Ada program will therefore rely on calls to system-provided functions (see another chapter on interfacing between Ada and C) and on the architecture and conventions that the system imposes on user programs.

Interestingly, this is not the only possible approach. On one side, the execution environment provided by the microcontroller is much simpler than what we can get from the full-featured desktop operating system, but on the other hand it offers access to features that are normally hidden from programmers working with desktop systems - the microcontroller is not just a computing resource, but includes also a range of hardware modules that are able to perform complex operations without constant assistance from the user program. That is, there is already a lot of parallelism built-in that can be reused without involving any special language features. Several timers with interrupts can provide independent logical execution drivers that can be used to process independent state machines. Peripherals can react to external events without involving user code. Communication modules can transmit blocks of data without any support from the CPU, and so on. What this means is that it is possible to achieve the physical independence of parallel streams of events (we could call them *processes* without any ties to how this concept is understood in a full-featured operating system) with just a basic set of language elements. Obviously, not every design can be implemented by such means, but we can as well design the system with such means in mind. In this context, the lack of multitasking at the language level is not necessarily a show-stopping problem.

We should also take into account the possibility of dividing the whole system into separate physical modules, with separate microcontrollers that will specialize in some particular tasks. This is also a way to introduce parallelism (in ver

terms) into the design and a low price of microcontrollers makes it potentially viable. After all, multithreading was invented to help in allocation of expensive physical resources, but if those resources themselves are relatively cheap, then using multiple CPUs can be more attractive than complicating the single system with multi-layered software. This can be particularly relevant with critical systems that require correctness analysis - it might be easier (and cheaper!) to prove correctness of several simple systems that are physically partitioned, than a single but complex one.

Thank You!

The only last words that could summarize this tutorial are big *Thank You* for reaching that far and for contributing valuable comments in the mean time. I hope that the information presented here will allow you to continue the path of exploration and that this material will motivate you to choose Ada for your next embedded project.

Good luck!

Previous: Runtime Errors And SPARK.
See also Table of Contents.