18. Mixing Ada with C and C++

Hopefully this tutorial has managed to achieve one of its major goals: to demonstrate that Ada is a valid choice for the development of embedded systems that are based on ARM Cortex-M microcontrollers. We have dealt with several development boards for the sake of having some practical exploration platform, but it should be clear that the same or just slightly adapted techniques (and the same compiler technology) can be used with other boards as well.

The prospect of writing embedded systems in Ada alone is very attractive and it has the potential to bring high readability and quality to the source code. In addition, Ada naturally supports and promotes good engineering practices (modular designs, separation of concerns, carefully defined type systems, etc.) and this is also what contributes to the high quality of final projects. Unfortunately this ideal is difficult to achieve and there are several reasons for it.

One important factor to consider when planning a more complex project is the availability of libraries that can handle various peripherals or communication protocols. These libraries can be very simple or can have a form of frameworks that provide complete program skeletons. Another factor that is frequently taken into account is the existing experience of engineers who will be involved in the project. These reasons will have an impact on the choice of technologies and programming languages, and it is wise to assume that more substantial projects will need to handle a mix of programming languages instead of just one, independently on its technical advantages.

This chapter outlines the basic concepts of mixing Ada with C and C++, but intentionally does not present any compilable example - the realistic problems that are related to mixing of bigger source packages are impossible to predict and a small example would not be representative - instead, this chapter will discuss issues that should be addressed in such mixed projects, where depending on what is the *main* programming language in use (as an arbitrary criterion we can consider which language was used for the startup and initialization sequence or the main control loop, if these elements exist in the program) we can distinguish two major scenarios:

- adding C or C++ code to the Ada program, or
- adding Ada code to the C or C++ program.

In both cases we can benefit from the fact that a single toolset can handle compilation and linking of all program elements (this is true even if in addition to these high-level languages we also need to involve source files written in assembler), although depending on particular package, the C++ programming language might not be supported - for example, the ARM package from the AdaCore website does not include C++, but packages available on ARM-based Linux systems (like Raspbian on Raspberry Pi) do support it. It might be also possible to link together object files that were generated by different compilers, although this can induce additional issues that we will not address here.

The general approach is based on the fact that both Ada and C allow to create subprograms with the "C" calling convention - the C compiler does it naturally and Ada compiler can be asked for this with appropriate pragmas or aspects, which we have used already to provide symbols for the linker script. In other words, the function that has known C *signature* can be implemented both in C and Ada and the resulting object file can be later used by the linker without regard to which language was used at the level of source code.

The pragmas in Ada that we will need are `pragma Import` and `pragma Export` and both of them were used already for the purpose of handling names defined or used by the linker script. These pragmas instruct the compiler about the external names that should be used for entities that are defined somewhere outside of Ada and used from Ada, or the other way round - defined in Ada, but intended for use outside of it. These two pragmas are sufficient to handle simple objects (we have done it already for registers) or subprograms that do not have any parameters or return values (these were used for interrupt handlers). In other words, we already have enough tools to implement some interrupt handlers in Ada, some others in C and link them together using a single linker script - but these tools are not sufficient for linking together subprograms and functions that use any types in their parameters.

In order to allow Ada programs to interface with C code (in both directions), Ada provides a standard package called `Interfaces.C` with definitions of types that correspond to primitive types in C and with some basic operations on null-terminated strings. Another package `Interfaces.C.Strings` provides more complex operations related to C-style strings. The types defined in these packages are supposed to be *compatible* with their corresponding C types, *provided that* compatible compilers are used. This condition is met when GCC is used for both programming languages, but even then special care should be taken when various compiler options are used on the C side which can affect the sizes or layout of primitive types, and there are no guarantees whatsoever (apart from reasonable engineering practice) if different compilers are used for both languages. Still, assuming that primitive types defined in `Interfaces.C` are compatible on both Ada and C sides, we can implement the example function with this "C" signature:

```
extern "C" int sum(int a, int b);
```

in both languages, for example in C as (obviously):

```
int sum(int a, int b)
{
    return a + b;
}
```

and in Ada as:

```
with Interfaces.C;
-- ...

   function Sum (
      A : in Interfaces.C.int; B : in Interfaces.C.int)
          return Interfaces.C.int is
   begin
      return A + B;
   end Sum;
```

If the above looks too verbose, the `use type` clause can be used to make the `int` type visible without additional qualification:

```
with Interfaces.C;
use type Interfaces.C.int;
-- ...

   function Sum (A : in int; B : in int) return int is
   begin
      return A + B;
   end Sum;
```

Interestingly, this:

```
   function Sum (A : in Integer; B : in Integer) return Integer is
   begin
      return A + B;
   end Sum;
```

will also *very likely* work properly. That is, the layout of types for parameters and return value will be the same due to the fact that `int` in C and `Integer` in Ada are both intended to reflect the "native" integer type on the given architecture and the GNAT user documentation even contains such examples, but for reasons of readability it is a good programming practice to explicitly show that inter-language interfacing is taking place here and use types from `Interfaces.C` as a matter of code documentation.

Both the C and Ada functions above are interchangeable in the sense that they can be declared in one language and implemented in another and the whole can be linked together to form a consistent program. It will be necessary to use pragmas on the Ada side to enforce external name and a calling convention - for example, if the function is defined in C and Ada will use (call) it, then the function should be declared in Ada as:

```
with Interfaces.C;
-- ...

   function Sum (
      A : in Interfaces.C.int; B : in Interfaces.C.int)
          return Interfaces.C.int;

   pragma Import (C, Sum, "sum");
```

and for the reverse case, when the function is implemented in Ada and called from C, the Ada declaration should be:

```
with Interfaces.C;
-- ...

   function Sum (
      A : in Interfaces.C.int; B : in Interfaces.C.int)
          return Interfaces.C.int;

   pragma Export (C, Sum, "sum");
```

(we have used such declarations already for interrupt handlers) - such implementations are not distinguishable at the linker level and object files that result from compiling C and Ada sources can be used together.

Note that the above declarations need not be placed in the Ada package specification (the C code will not be able to use them anyway and will have to declare this function using the C syntax), but it is still a good practice as it documents the services that are used from *outside* of the package where the subprogram is implemented.

Using basic primitive types in subprograms that are used across Ada and C is quite straightforward and of course some type conversions will be needed between types from `Interfaces.C` and other types used in the system, but these conversions should not be treated as a nuisance - these conversions are actually explicit opportunities to check whether the values conform to our expectations and show where that might not be the case.

The interfacing between Ada and C becomes more difficult where complex data structures or records are used, which involves passing pointers as parameters at the C level - this also applies to the case of interfacing with C++, if classes and object-oriented interfaces are used. Navigating such structures or creating them in Ada is possible, but very difficult and error-prone - in practice the most robust solution is to prepare a wrapper interface that exposes *flattened* or opaque view of the data so that only primitive data types are visible at the border between the two languages. Typically such wrapper interfaces will be very simple translators that do not significantly contribute to the size or time cost of the whole program.

Mixing Ada with C or C++ raises the issue of existing language runtime, as there can be more than one that has to be supported. Incidentally, throughout this tutorial we have managed to steer away from the Ada language features that require any runtime support, which is now very convenient. The object files that result from compiling Ada source code can be linked together with other object files without any additional elements and the only thing that might need some merging is the linker script, which contains symbol definitions for registers and which (in our approach) wires interrupt handlers. This is where the following different cases can be considered:

- C library is used in the Ada program - very likely the C library will not have any runtime dependencies and the Ada's linker script will be used without any significant changes, perhaps with the exception of new symbol definitions if they are used by the C library.
- C++ library is used in the Ada program - similarly as above, but the C++ runtime library might need to be added to the linking process if the C++ code used language features that require such support; in addition to that, the linker script will have to be extended to cover the conventions of C++ related to startup and initialization of static objects.
- Ada packages are used in the C or C++ program (in particular, when Ada is used with some existing operating system) - if the Ada code avoids the use of features that require language support (like in all examples in this tutorial), then no additional integration code is needed, but the linker script will have to be merged into the script that is used for linking the main C program. This may require some investigation, especially if the C development process is managed by an integrated environment, as the details of linking might not be easily accessible.

Note that in all these cases there is a huge benefit from simple codebase that does not rely on any sophisticated language features, so that integration between languages is not obstructed by their dependencies. Even though you might not have complete control on how the C and C++ libraries and frameworks are designed and developed, keeping Ada code simple will always help when the two languages are mixed together.

Finally, one of the most important advices in the context of mixing programming languages is that it should be done based on sound estimation of costs. Solving problems that arise at the border between languages can be expensive and error-prone, especially when data conversion or flattening of interfaces are used, as this is where neither of the two languages is able to verify the type safety at that point - which is slightly ironic, if at the same time we try to justify such a mix based on the good type safety of Ada. A good dose of engineering judgement is necessary to avoid the trap where new and unnecessary risks are introduced instead of just bringing together advantages of two different languages. Still, creating such systems properly is possible and the ability to address potential issues will be a serious point in your embedded programming toolbox.

Previous: Hello World!, next: Runtime Errors And SPARK.
See also Table of Contents.