1. Introduction

Who should read this tutorial?

This tutorial was written for all those who are interested in embedded systems and who would like to play with one of the popular prototyping boards and learn a bit of Ada in the meantime. The boards that were selected for use in this tutorial are:

- Arduino M0,
- Arduino Due,
- STM32 Nucleo-32,
- STM32 Nucleo-144.

No previous experience with the example boards is required, and in particular no Ada expertise is required to use this tutorial - as such, this tutorial can be treated as an introduction to programming embedded systems in Ada.
Still, I assume that you have some programming knowledge in any of the mainstream programming languages, so that basic concepts need not be explained. I will occasionally show examples in C to explain some Ada constructs.

Why Ada?

The world of embedded systems seems to be dominated (or even monopolized) by the C programming language. C++ is also present in this market niche as a very close relative to C, but at the level of their use in embedded systems we can consider them to represent the same programming mindset. If you try to find some on-line information about Arduino, Nucleo or a similar prototyping board, you will surely see that every discussion forum is focused on C or C++. The Arduino ecosystem in particular and its associated tools are also based on C++ subset and apparently all microcontroller vendors offer (or at least suggest) C and C++ compilers and libraries for their products. Not surprisingly, books, journals and most tutorials take this for granted as well, presenting code snippets exclusively in C and C++ - the final impression is that if you want to program embedded systems, you have to do it in C. This is a false impression and there are very reasonable arguments that C and languages that are derived from it are not even the most adequate tools for implementing such systems.

This tutorial intends to bring some fresh view on this subject and present the alternative approach. What is important, that alternative should not be considered to be an "innovation" (in the wrong sense of this word), but rather a reminder that a proper technology exists for a long time already - Ada was created to target embedded systems about 30 years ago and by virtue of its maturity has a lot to offer in the area of embedded systems.

Ada should not be considered to be a drop-in replacement for C, either. The potential of Ada comes from its rich type system and expressive syntax that allows to embed (sic!) a significant amount of business knowledge about what we expect the program to do (in addition to what we instruct it to do) and that additional information can be used to reason about program correctness even before the program is actually executed. Ada is known for its ability to deliver low defect rates - and considering that testing and debugging embedded systems is a very expensive and difficult activity, such language features bring a great added value to the development process.

Considering the positive properties that Ada has with regard to programming embedded systems, its relative absence on the market and even the persistent lack of support from microcontroller vendors is a curiosity that deserves correction and this tutorial is not even alone nor the first attempt to fix it - the AdaCore company, which is a leading Ada compiler vendor, tries to animate the Ada community with their project examples for the STM32Fx Discovery family of boards. Following this, the community answered with cross-compiler packages that allow to compile and build Ada programs for ARM processors on every major operating system. These initiatives were very well received, but somehow the Arduino ecosystem (with which the Nucleo boards are pin-compatible) was not explicitly targeted. This tutorial was written to cover that gap.

Why Arduino and Nucleo?

Arduino is a family of prototyping boards that got very active community traction thanks to the basic Arduino Uno model. Even though this basic board is very limited in terms of computing power, the market for Arduino books and hardware extensions (so-called shields that can be easily connected to the main board) simply exploded.

The family of Arduino boards also includes the Zero/M0 and Due models (depending on the vendor), which are based on powerful ARM Cortex-M microcontrollers. These boards differ in their capabilities, but share the ARM core as a hardware platform. Both of them retain the layout of basic connectors from the Uno model and are therefore compatible with a large number of shields that are available on the market.

The STM32 Nucleo boards come from a different vendor, but are meant to be compatible with Arduino at the level of their pin and connector layouts, which allows some portability with regard to the shields that can be connected to them. The STM32 Nucleo-32 board is a small-factor board, but since it was meant to be compatible with another small Arduino product, we can consider them to represent the general family of Arduino products.

All of these development boards seem to provide a very nice combination of the built-in technology and available accessories and as such they seem to be a nice vehicle for learning Ada in the context of embedded programming. In addition, since the range of microcontrollers covered by these four boards is very wide - from Cortex-M0 to Cortex-M7 cores - it is also possible to demonstrate some basic strategies in building a small HAL (Hardware Abstraction Layer) library that allows to write portable embedded software.

What you will learn?

I would like to propose an approach that is a bit different from a typical tutorial. Even though written for Ada beginners, it will not focus on giving you ready answers ("what should I type to do this or that?") - rather, it will describe in detail the exploration process itself ("how can I find the solution?"). The reason for this is that this tutorial cannot be exhaustive with regard to everything that can be done with Arduino and Nucleo boards or with their microcontrollers and it would be a pity to leave you without answers if you want to do something more. By focusing on the exploration rather than on solutions, I hope to give you tools which you will be able to reuse even outside of what this tutorial can cover.

You will learn how to find relevant information in the technical documentation from Atmel and STMicroelectronics. There is no need to read the whole documentation (the ARM documentation together with processor-specific information is daunting), but I will show you, with practical examples, how to quickly find the necessary details.

You will learn how to use the basic features of Ada, starting from the bare-metal environment - that is, targeting the raw processor, without any glue or startup code that is typically injected by linkers to final programs. You will also learn how language features interact with the environment and whether they need any additional support - by doing it you will also learn how to provide such supporting infrastructure for those language features that you will decide to use.

Learning how to work with the language in a raw environment has an interesting added value - by retaining complete control on the executable code, we can also retain control over its licensing properties. There is a lot of misunderstanding related to licensing and with two variants of the GNAT compiler that are available for use (each with a different approach to the licensing of final executables), the controversies are not likely to finish any time soon. We will solve such problems by avoiding them in the first place. In addition, you will learn how to take complete responsibility for the program - that is, for every single byte of it - without relying on various bits and pieces that are "provided as is and with no warranty of any kind", which is a typical licensing assertion that is present even in tools that are supposed to be used by professionals. This can become valuable if at some point you will get involved in a commercial or safety critical embedded project.

You will also learn how to use basic linker scripts in order to control the association of your program elements with memory regions and addresses expected by the processor. The linker is a tool that is always used with compiled languages like Ada, C or C++, yet it is almost always treated like a black box, leading to a common perception that this final development phase is governed by magic. Even though such an approach seems to be adequate for typical desktop and server programs, in embedded systems this would be very limiting. By learning how to control the linking process you will gain the ability to target not only Arduino and Nucleo boards, but in fact any other embedded system based on ARM, even those that use unusual memory configurations.

We will also have a look at the process of mixing Ada and C together in a single system. The reason for this is that, as explained above, C and C++ are well supported on any embedded platform and this can be seen also in the availability of libraries. In the Arduino ecosystem many shields have their own libraries that make using all the shield features easier. Even though writing "pure Ada" system sounds attractive (and is certainly possible), it is not always practical. By learning how to mix Ada with C you will be able to find the most appropriate trade-off between pros and cons of each language.

Last but not least, we will see how we can use a bit of *formal methods* with the SPARK language and its toolset in order to *prove* that our Ada programs are free from runtime errors. Such proofs can be very valuable, especially in the context of embedded systems, and especially in the environment where we decide not to rely on the Ada language runtime. We will leave that exercise to the end of this tutorial, even though in real projects it would be preferable to introduce such techniques from the very beginning - but since learning to walk is necessary before learning to run, this tutorial will benefit from leaving those most effective techniques to the last chapters.

And, by the way, the *last* example program in this tutorial is... Hello World!

Other learning sources

This tutorial is not the only attempt to promote and demonstrate how to use Ada with ARM Cortex-M microcontrollers and the prominent sources of other related information are AdaCore blogs as well as the SourceForge's *Cortex GNAT Run Time Systems* project, which has the goal to provide various Ada Run Time Systems (RTSs) for several Cortex-powered boards. Both of them are regularly updated.

It should be stressed, however, that this tutorial does not duplicate content from those other sources and is not even intended to directly compete with them - their goal is to provide Run Time Systems that are as complete as possible with regard to supported language features and available standard and peripheral library packages; that is, their goal is to provide the programming experience of *full Ada* in embedded systems. This tutorial, on the other hand, fills the niche on the other side of the spectrum, where a selected language subset (coupled with a dose of rigorous verification) is used to retain full control over the engineering process and where the lack of any language run-time dependencies allow easier integration with other software components. As such, this tutorial, together with the sources mentioned above, are complementary and demonstrate a wide set of possibilities for use of Ada with ARM Cortex-M embedded systems.

What this tutorial is not?

This tutorial is not a source of ready to use libraries or complete run-time systems, even though in further chapters you will find complete working projects that are ready to build and deploy. This is in line with the motivation described above - this tutorial is about detailed exploration of the interactions between different elements, so preparing complete abstraction layer that hides the details would not help to achieve this goal.

This tutorial is also not a complete handbook on what you can do with the Arduino and Nucleo boards (the raw microcontroller documentation has typically more than 1000 pages and this tutorial touches only a small fraction of what you can find there) - rather, it is a description of the discovery process, which can be performed for any other functionality that is supported by the board, even if not explicitly covered here. I believe that this approach has a significant educational value, and can be applied on any other ARM-based board - and not only there are many such boards on the market, but the market itself seems to be accelerating. There is a high chance that before you will finish reading this tutorial you will already want to use this knowledge on some other board.

This tutorial is also not a complete description of the Ada programming language. Ada is a multi-paradigm and a very mature programming language, and a small embedded system is not even a proper place to apply every single language feature (as a joke, consider the fact that the full Ada Reference Manual would not even fit in the memory available on these boards!). This tutorial presents a small, but very functional subset of the language, which is appropriate for embedded microcontroller applications. At the same time, the tutorial contains enough problem-solving examples to guide you when you find out that some language feature that you decided to use relies on foundations that are not yet there.

In short, this tutorial will not give you a fish to feed you for a single day - instead, by focusing on the exploration process, it will teach you to fish, and it will give you enough expertise to apply on future ARM-based prototyping boards. Good luck!

How to read this tutorial?

This tutorial was prepared with four example boards in mind. These boards share the ARM Cortex-Mx architecture and most of the concepts can be described without relating to any particular target. Still, in it unavoidable to tackle many practical differences between the involved microcontrollers and assuming that the reader might not be interested in learning the gory details for all four boards, sections devoted to each particular board are marked separately. Apart from these discontinuities, each consecutive chapter builds on top of what was discovered in previous chapters (starting from base concepts) and as such the tutorial was intended for reading from the beginning to the end.

Good luck!

Next: Documentation and Tools.
See also Table of Contents.

Did you find this article interesting? Share it!

SHARE

Copyright © 2007-2017, Inspirel