More Ada in Non-Ada Systems

A. Marriott, U. Maurer

White Elephant GmbH, Beckengässchen 1, 8200 Schaffhausen, Switzerland; email: software@white-elephant.ch

Abstract

This article is based on the industrial presentation "More Ada in non-Ada systems" which was given at the 2021 Ada-Europe virtual conference.

The presentation was an update to the presentation given at the Ada-Europe 2018 conference in Lisbon entitled "Using Ada in non-Ada systems" and that was subsequently published in the Ada User Journal [1].

The work used GCC 7.4.1 built by AdaCore and made available as part of their 2019 Windows hosted GNAT for ARM GPL community distribution. The GNAT sources referred to in the presentation were also obtained from this distribution.

Keywords: GCC, Modula-2, C, ZFA, ARM

1 Previously

If we restrict ourselves to a subset of Ada, the so called Zero Footprint Ada (ZFA), then it is possible to write applications that use a mixture of Ada and C. The Ada we use is not the Ada with which we are most familiar, it is a very cutback version – but even so, despite all the restrictions, it can still be beneficial to write code in Ada rather than in C.

We believe that there are many reasons why one should program in Ada rather than C. However, from a business perspective, these reasons may be insufficient to warrant the complete rewrite of a large, stable and successful system.

Rather than trying to advocate that our systems should be re-written completely in Ada we showed how existing code could be supplemented by code written in Ada.

In 2018 our work was a proof of concept. At that time our management had no intention nor desire to write any code in Ada. The existing software and expertise were in Modula-2, a language that can be considered to be a simpler yet similar language, at least in comparison with alternatives such as the dreaded C.

2 Data compression

However, all this changed when it was decided that downloads to our ARM based processors should be compressed. As products develop, they tend to grow in both complexity and size, yet the speed of the field bus used to transfer the executable images remains the same. This results in the time to download getting longer and longer.

As a consequence, it was decided that it would be an interesting salable feature if this time could be reduced by compressing the download. Because we lack the required expertise to develop this sort of software, this decision would require us to use some sort of library - either binary, or just as bad, source written in C.

Another alternative would be to use the ZipAda [2] open-source code written by our colleague Gautier de Montmollin. But, as the library's name suggests, this software is written in Ada. This presented us with the choice of either using it directly in Ada or converting it into Modula-2, which, as I've already alluded, is a very similar language. Similar in theory, but in practice it rather depends on what features of Ada one has used. Unfortunately for us, Gautier had used many features of Ada that aren't supported by Modula-2 - thereby making a translation virtually impossible. Or at least not without the risk of introducing all manner of interesting and hard to find bugs.

The solution was to use the proof of concept work we had made in 2018 and use his source code directly. I think it says a lot about the portability of software written in Ada that we could use the ZipAda software with only the most minor of changes.

The software was designed to run on a PC reading data from a file and compressing it to make what is generally referred to as a Zip File. Instead, we use a PC to read the executable to be downloaded, compress this and then transmit the compressed byte stream to the microprocessor which then uses the same software to decompress the byte stream back into the executable.

Amazingly this all worked, with almost no effort. We took software written in Ada that was designed and primarily works on PCs, recompiled it using an Ada compiler for ARM and then linked this into our product – that was otherwise written entirely in machine generated C.

This then was the first commercial use of the work I presented in 2018 as a theoretical exercise.

3 Floating-point

Up until now all the microprocessors that we have used have been without floating-point units.

Because floating-point without hardware support is incredibly inefficient, we explicitly prohibited its use by not recognizing the Modula-2 floating-point syntax.

In the Ada work I presented in 2018 we used the pragma Restrictions (No_Floating_Point) in System.ads to impose the same restriction on routines written in Ada.

However, our latest target microprocessor has a floating-point unit and so, unsurprisingly, there are plans to use it, so the dilemma has been to decide how to use it.

If we permit the floating-point syntax in Modula then we may encounter problems with our Modula to C translator – which, obviously, has never been used to translate anything concerning floating-point. Then there is the problem of what exactly is floating-point in C and then, lastly, what libraries are available, how good are they and can we obtain the sources for them?

An alternative strategy, and what we have ultimately decided to do, is to forget Modula-2 and, because only our ARM microprocessor has an FPU, write all our floating-point code in Ada.

Allowing the Ada syntax was easy – all we had to do was remove the pragma Restrictions (No_Floating_Point). The Ada compiler can be directed to generate code using the ARM floating-point instructions by including the switch "-mfloat-abi=hard" and the switch "-mfpu" to describe what sort of FPU the processor has.

Therefore it is relatively easy to support code that merely performs simple arithmetic.

The fun starts when one wants to use functions that are provided by the package Ada.Numerics. Elementary_Functions and/or its long float counterpart Ada.Numerics.Long_Elementary_Functions.

Trying to compile the Ada statement

```
X:= Ada.Numerics.Elementary_Functions.Tan(45.0);
```

will produce the error message

```
"Ada.Numerics" is not a predefined library unit
```

This means that GNAT cannot find the specification for Ada.Numerics in the source file search path. This specification file must be provided even though it only contains two constants: Pi and e.

Moreover, GNAT requires that the specification is provided in a file with the crunched file name a-numeri.ads.

Once this file is made available, GNAT will then require the file a-nuelfu.ads and once this is provided, it will start requiring various system definitions, starting with System.Generic_C_Math_Interface contained in

s-gcmain.ads

To cut a long story short, to use Ada's floating point elementary functions requires that the GNAT compiler be able to find 14 specification packages in files with crunched filenames.

Helpfully the compiler tells you what is missing and therefore these files are relatively easy to find and place in the source path.

The 14 files are:

- a-numeri.ads
- a-ndelfu.ads
- a-nuelfu.ads
- a-nlelfu.ads
- s-gcmain.ads
- s-libsin.ads
- s-libpre.ads
- s-libdou.ads
- s-exnllf.ads
- s-fatflt.ads
- s-fatlfl.ads
- s-fatgen.ads
- s-fatgen.adb
- s-lisisq.ads

Once GNAT can find these files it will compile. However, it will then fail at link time.

In my tangent example it will fail with

```
undefined reference to
`ada__numerics__elementary_functions__tan'
```

because although we have provided the specification of the tangent function, we haven't provided the implementation.

Unfortunately, if we take the source of the implementation and try to compile a-nuelfu.adb, GNAT produces the error message

```
user-defined descendants of package Ada are not allowed
```

The work-around to this is to rename the sources and to export the functions with the name they would have had, had they been compiled as descendants of Ada. If we place the objects into a static library (object archive) and provide this library explicitly in the linker script, the GCC can resolve the references, and all is well.

Our Ada is used in conjunction with C machine generated from source written in Modula-2. We called this mixture of Ada and Modula, Adam. For this reason, we chose to base our library on the root package Adam.

For example, for the Tangent function we copied the source file a-nuelfu.ads and renamed it to Adam-Numerics-Elementary_Functions.ads.

We then modified the contents to rename the package name to Adam.Numerics.Elementary_Functions and exported every function with the External name prefixed by "ada__numerics__elementary_functions__"

```
package Adam.Numerics.Elementary_Functions with
Preelaborate is

   Prefix : constant String :=
   "ada__numerics__elementary_functions__";

   function Tan (X : Float) return Float
    with Inline,
         Export,
         External_Name => Prefix & "tan";

end Adam.Numerics.Elementary_Functions;
```

For the implementation we renamed the file a-nuelfi.adb to Adam-Numerics-Elementary_Functions.adb and modified

the contents so that the package name became Adam.Numerics.Elementary_Functions.

We had to do this for the following 21 files:

- Adam-Numerics.ads
- Adam-Numerics-Elementary_Functions.ads
- Adam-Numerics-Elementary_Functions.adb
- (a-nuelfu.adb)
- Adam-Numerics-Long_Elementary_Functions.ads
- Adam-Numerics-Long_Elementary_Functions.adb (a-nlelfu.adb)
- Adam-Generic_C_Math_Interface.ads
- Adam-Generic_C_Math_Interface.adb
- (s-gcmain.adb)
- Adam-Libm.ads
- Adam-Libm.adb (s-libm.adb)
- Adam-Libm-Single.ads
- Adam-Libm-Single.adb (s-libsin.adb, s-lisisq.adb)
- Adam-Libm-Double.ads
- Adam-Libm-Double.adb (s-libdou.adb)
- Adam-Exn_LLF.ads
- Adam-Exn_LLF.adb (s-exnllf.adb)
- Adam-Generic_Attributes.ads
- Adam-Generic_Attributes.adb (s-fatgen.adb)
- Adam-Attributes.ads
- Adam-Attributes.adb (s-fatflt.adb)
- Adam-Long_Attributes.ads
- Adam-Long_Attributes.adb (s-fatlfl.adb)

Other than renaming we did not have to modify the code very much, but some minor changes had to be made:

3.1 Exception handling

In ZFA, exceptions cannot be propagated outside the procedure, so either they must be caught and handled (usually by an SVC instruction) or suppressed if the exception is thought to be impossible to raise. I refer you to my previous presentation [1] for a description on how to do this and why it has to be done.

3.2 Square-Root

In Adam-Libm-Single.adb and Adam-Libm-Double.adb we implemented the square root function as the built-in intrinsic so that the FPU VSQRT instruction is used rather than the acres of code provided in s-lisisq.adb and s-lidosq.adb respectively.

3.3 Copy_Sign

The function Copy_Sign in s-fatgen.adb didn't compile using GCC v7.4.1.

GNAT issued the error message "Incorrect context for intrinsic convention" on the pragma import for the function Is_Negative.

Our solution was simply to replace this call with a comparison with 0.0 and the presentation described the code this produced and questioned whether or not it was actually correct.

All this was rendered moot by AdaCore releasing GPL 2021

In this release GNAT (GCC v10.3) no longer recognizes the intrinsic function Is_Negative and, as a result, the procedure Copy_Sign has been totally reworked.

Because the new coding is backward compatible the problem described in the presentation has now effectively been fixed.

3.4 Static library

When all the sources have been modified, the static library can be built. To build the library we simply compiled all the .adb files that start with the prefix Adam- and then used the GCC archive program Ar to place the objects into a static library. This library is then cited as an input in the GCC linker script.

4 Returning unconstrained types

String handling in Modula-2 is very primitive. Strings are simply arrays of characters and because Modula-2 functions cannot return variable sized objects, they cannot therefore return strings.

Up until now we have had very little need for string manipulation. This is probably because our user interface is PC based rather than executing on any of the microprocessors. However, this would change if, for example, we implemented a stand-alone product that communicated with the user using TCP Telnet or as a web server using HTML. In such a project we would need improved string handling – such as that provided by Ada.

However, in order that Ada functions are able to return unconstrained types such as strings, we need to implement a secondary stack. This is a per-task area of memory from which the compiler can allocate space to return unconstrained types.

First the pragma Restrictions (No_Secondary_Stack) has to be removed from System.ads otherwise any function that attempts to return an unconstrained type will cause GNAT to issue the error message violation of implicit restriction "No_Secondary_Stack".

Once this restriction pragma has been removed from System.ads, GNAT will then issue the rather cryptic message

construct not allowed in configurable run-time mode

What this means is that GNAT has been unable to find the specification packages that enables returning unconstrained types. Just like with the enabling of floating-point these specifications need to be placed in the source search path and be contained within files with crunched names.

The required specification files are:

- s-parame.ads
- s-secsta.ads

- s-stoele.ads

Once these have been made available, code can be compiled.

However, linking will fail because of unresolved references.

A program that uses the secondary stack to return unconstrained types will need the routines:

- system__secondary_stack__ss_allocate
- system__secondary_stack__ss_mark
- system__secondary_stack__ss_release

We provide these routines in the same way as we did the floating-point routines, namely, in a static library. These routines are wrappers around our kernel functions to allocate memory from the calling task's secondary stack, to read its secondary stack pointer and to set its secondary stack pointer respectively.

In our Kernel, when a task is created, the size of its primary stack is given and optionally the size of a second stack. The primary stack is accessed via the processor's stack pointer register, and the secondary stack via access routines provided by the Kernel. Both stacks are protected by the Memory Protection Unit (MPU) and are private to the task. I.e. a memory fault will be raised should another task attempt to access them.

The Kernel provides three routines:

- AllocateFromSecondaryStack
  Takes the required size as a parameter and returns the start address of the memory area that has been allocated.
- GetSecondaryStackPointer
  Returns the secondary stack pointer of the current task.
- SetSecondaryStackPointer
  Sets the task's secondary stack pointer to the address passed as its parameter if the address is between the task's top of stack and the current secondary stack pointer.

These routines conveniently directly map onto the routines required by Ada.

5 String attributes

The attribute 'image, its short-hand form 'img and the attribute 'value are immensely useful when dealing with strings. To enable these, we need to provide GNAT with 12 specification packages all contained in files with crunched filenames.

We need 6 for the image attribute

- s-imgboo.ads
- s-imgllu.ads
- s-imgrea.ads
- s-imguns.ads
- s-imgint.ads
- s-imglli.ads

and a further six for the value attribute

- s-valboo.ads
- s-valint.ads
- s-vallli.ads
- s-valllu.ads
- s-valrea.ads
- s-valuns.ads

At link time we need to resolve unsatisfied references, once again by use of a static library. Just as we did to support floating-point, we need to rename the Ada source files to the parent package Adam, compile them, and place the resultant objects into a static library.

This time there are 37 files:

- Adam-Image_LLU.ads
- Adam-Image_LLU.adb
- Adam-Image_Uns.ads
- Adam-Image_Uns.adb
- Adam-Img_Bool.ads
- Adam-Img_Bool.adb
- Adam-Img_Int.ads
- Adam-Img_Int.adb
- Adam-Img_LLI.ads
- Adam-Img_LLI.adb
- Adam-Img_LLU.ads
- Adam-Img_LLU.adb
- Adam-Img_Real.ads
- Adam-Img_Real.adb
- Adam-Img_Uns.ads
- Adam-Img_Uns.adb
- Adam-Float_Control.ads
- Adam-Val_Bool.ads
- Adam-Val_Bool.adb
- Adam-Val_Int.ads
- Adam-Val_Int.adb
- Adam-Val_LLI.ads
- Adam-Val_LLI.adb
- Adam-Val_LLU.ads
- Adam-Val_LLU.adb
- Adam-Val_Real.ads
- Adam-Val_Real.adb
- Adam-Val_Uns.ads
- Adam-Val_Uns.adb
- Adam-Val_Util.ads
- Adam-Val_Util.adb
- Adam-Value_LLU.ads
- Adam-Value_LLU.adb
- Adam-Value_Uns.ads

- Adam-Value_Uns.adb
- Adam-Case_Util.ads
- Adam-Case_Util.adb

6 Images of Enumerations

However this isn't quite the whole story.

Left like this, obtaining the string representation of an enumeration literal does not bring back the uppercase version of the declaration but the string representation of its position within the enumeration.

For example:

```
type Color is (Blue, Green, Red);
C  : Color := Red;
CI : constant String := C'image;
```

In the above example we would expect CI to be the 3 byte string "RED" whereas, in fact, it is the two byte string " 2"!

I.e. GNAT has silently, and without any warning whatsoever, implemented

```
CI : constant String:= Color'pos(C)'image;
```

rather than what was written!

For the correct implementation the name strings have to be retained. GNAT can be instructed to do this by commenting out or otherwise removing the pragma Discard_Names in System.ads.

The comment in System.ads for this pragma is

"Disable explicitly the generation of names associated with entities in order to reduce the amount of storage used. These names are not used anyway."

However, the last sentence in this comment is no longer true, the names, if they are there, can be used.

But just providing the names is not sufficient. Two more crunched specifications need to be provided:

- s-imenne.ads
- s-valenu.ads

along with their implementations:

- Adam-Img_Enum_New
- Adam-Val_Enum

Then, and only then, does GNAT compile the correct code and the program work correctly and according to the Ada standard.

7 Protected Objects

As I mentioned in the description of the secondary stack, our system is multi-tasking, but we cannot use Ada's tasking model because ZFA precludes the use of Ada's run-time. However, our Kernel does support multi-tasking and therefore we need to protect certain data against simultaneous access.

In Modula-2, one way of doing this is by using a semaphore for each group of variables that need to be protected. Unfortunately, this approach is somewhat error prone, it is all too easy to forget to write the code to wait on the group's semaphore, or even sometimes to use the wrong semaphore.

Ada makes this easier and less error prone with its protected object construct. Fortunately, although we don't use Ada's tasking model, ZFA supports the implementation of protected objects, albeit restricted to procedural operations. Protected entries are not supported because these would not work with our state machine model.

The following is an example of how we can use a protected type to ensure that the increment of a quadword is made task safe on a processor where 64-bit operations are not atomic.

```
type Value is mod 2**64;

protected Data is
  procedure Increment;
  function Actual_Value return Value;
private
  The_Value : Value := 0;
end Data;

protected body Data is
  procedure Increment is
  begin
    The_Value := The_Value + 1;
  end Increment;

  function Actual_Value return Value is
  begin
    return The_Value;
  end Actual_Value;
end Data;
```

If we try to compile this, GNAT will issue the by now familiar message

```
construct not allowed in configurable run-time mode
```

indicating that an Ada feature needs to be enabled by supplying a specific specification package within a file with a crunched filename.

To enable protected objects, we need to supply:

- s-taprob.ads
- s-taskin.ads

At link-time we will need to provide three routines

- system__tasking__protected_objects__initialize_protection
- system__tasking__protected_objects__lock
- system__tasking__protected_objects__unlock

Which we implement simply as wrapper routines around our Kernel's semaphore routines Initialize, Wait_For and Signal respectively.

8 Dynamic memory allocation

For example

```
C := new Character ('x');
```

In order that we can dynamically allocate memory using the Ada "new" construct we need to enable the feature by including the specification file s-memory.ads

At link time we must then satisfy the reference to ___Gnat_Malloc with an implementation that simply calls our Kernel's Allocate function that takes the desired amount of memory as a parameter and returns the address of the allocated memory.

9 Summary

In my first presentation [1] I described how Ada could be linked into an application predominantly written in C, how Ada routines could call C routines and C routines call Ada and how the Ada packages could be correctly elaborated.

This presentation goes further and shows that even more Ada features can be used if they are enabled by the simple expedient of providing the required specification package in a file with the expected crunched filename. I also explained how the implementation could be provided by a static library and how this could be built. Together, these two techniques have allowed us to expand our use of Ada to include floating-point, functions returning strings, protected objects and memory allocation thereby making Zero Footprint Ada even more useful.

References

[1] A. Marriott and U. Maurer, "Using Ada in non-Ada Systems", *Ada User Journal*, vol 39 no 3, pp 180-187, 2018.

[2] Zip-Ada is a free, open-source programming library for dealing with the Zip compressed archive file format. https://sourceforge.net/projects/unzip-ada or ssh://git@github.com/zertovitch/zip-ada