Using 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 "Using Ada in non-Ada systems" which was given at the 2018 Ada-Europe conference in Lisbon.

The presentation was an experience report on our use of Ada packages within existing non-Ada embedded microprocessor based systems.

Keywords: GCC, Modula-2, C, ZFA

1 History

In the late eighties, when we first started replacing discrete electronics with embedded microprocessors, there were very few Ada compilers available, especially for microprocessors, and those that did exist were slow, required vast resources and were very expensive.

As a consequence, and because we wanted to use a highly typed language rather than use the ubiquitous C, we decided to implement our systems in ISO 10514 Modula-2. Modula-2 is a programming language invented by Niklaus Wirth that has many features in common with Ada. For example, it's verbose non-ambiguous syntax and the separation of specification and implementation into separately compiled units.

Originally we used cross compilers, compiling Modula-2 source directly into the target machine code. However, over time it became increasingly difficult to find Modula-2 compilers for the new microprocessors that were coming onto the market.

For this reason we switched to using a Modula-2 translator that translates Modula-2 into C. This machine generated C is then compiled into the target machine code. We rarely look at this machine-generated C code – preferring to treat it as some form of intermediary "binary".

None the less our Modula-2 is translated into C and it is this C that is compiled and linked to form our embedded hard real-time systems. Later in this article, when I present how and why we have started using Ada in our systems, it should be noted that we are effectively talking about using Ada in a predominantly C environment. The fact that we ourselves don't actually program in C, or even know how to program in C, is a luxury we have been afforded but that shouldn't distract from the usefulness or relevancy of this article.

We have a large amount of well-established code that executes on a multitude of platforms and that uses our own proprietary multitasking run-time. Management is unlikely to sanction the conversion of this code base into Ada - if only because the risk of introducing errors would far out way any perceived benefit of coding exclusively in Ada.

However this is not to say that new features or features that have to be substantially modified couldn't be written in Ada, provided that an affordable Ada compiler exists for the target microprocessor architecture and if the code can be integrated into the existing system.

Until recently, we have been using the Wind River Diab tool chain to build our executables (in ELF format with DWARF debugging information) for Motorola M68332 and Coldfire microprocessors. We have no intention of touching these systems. However our most recent hardware is ARM based and we have also switched C compiler and now use the Gnu Compiler Collection (GCC).

In fact we use GCC version 6.3.1 to compile our C code for ARM which is the same version of the GCC that AdaCore releases under GPL 2017 for compiling Ada for ARM. So the challenge has been to write code in Ada and then use GNAT to compile it and link it together with our existing C code.

An important caveat is that we are not talking about using full Ada. A lot of the power of Ada comes from language features that depend on its runtime. However we already have a runtime. Rather than modify the Ada runtime to use our runtime or modify our runtime to use Ada's, we decided, at least for now, that the simplest course of action is to restrict ourselves to a subset of Ada that doesn't require a runtime.

This is what is known as the Zero Footprint profile for Ada.

Exactly what Zero Footprint Ada means for any particular system depends on which pragma restrictions are declared in the file System.ads

For example a typical ZFA could be defined as:

```
pragma Restrictions (No_Exception_Propagation);
pragma Restrictions (No_Implicit_Dynamic_Code);
pragma Restrictions (No_Finalization);
pragma Restrictions (No_Tasking);
pragma Restrictions (No_Delay);
pragma Discard_Names;
```

These restrictions mean we lose a lot of nice features of Ada. Features such as:

- Tasks

- Protected objects
- Controlled types
- The delay statement
- Dynamic storage allocation using new
- Exception propagation

In addition to the above we also voluntarily imposed additional restrictions to reduce Ada down to the level we wanted to support.

For example our target microprocessor has no fixed point hardware therefore any code that uses floating point will be exceedingly slow. To prevent the accidental use of floating point we added

```
pragma Restrictions (No_Floating_Point);
```

into System.ads.

Another restriction, at least initially, is to forego Ada functions that return unconstrained types. This is because variable length results are returned to the caller using what GNAT terms the secondary stack. However the microprocessors we are currently using have very little RAM, therefore we can ill afford the luxury of having a second stack for each and every task.

Consequently we added

```
pragma Restrictions (No_Secondary_Stack);
```

into System.ads.

The main consequence of this decision is that we can't write Ada functions that return strings nor can we use attributes such as 'image or 'img.

We also initially decided to restrict ourselves to writing pre-elaborated packages. By declaring all our packages "*with preelaborate*" or "*with pure*" and including

```
pragma Restrictions (No_Elaboration_Code);
```

in system.ads we forego elaboration.

Without elaboration:

1. Global variables can only be initialised to values evaluated at compile time
2. Packages may not have a body, i.e. code between the *begin* and *end* of the package implementation.
3. Pre-elaborated packages may only call packages that are themselves pre-elaborated or pure.

However, even with all these restrictions we believe that there is still enough left of Ada to make integration attractive. We have always considered Modula-2 to be a poor man's Ada. However, in our opinion, even a severely cut back Ada is better than programming in Modula-2 and we can only imagine how much of an improvement it must be over writing in C.

Ada is obviously syntactically superior to C and even though they share the same roots, Ada offers many advantages over Modula-2

For example:

- Named parameters
- Named fields in constructors
- Private types, functions and procedures.
- Bit level specification in representation clauses.

Representation clauses are extremely useful when interfacing to hardware and third party protocols. An enumeration that is not represented as a complete byte is accessed in most computer programming languages by a combination of bit masks and shifting – a typically error prone endeavour that is handled automatically by Ada.

2 Getting Started

The simplest form of integration is when a program written in Modula-2 calls a parameter-less procedure written in Ada.

To make procedures accessible from other modules, Modula-2 mangles the procedure names by prefixing them with the name of the module together with a separating underscore. Thus procedure Y defined in the definition of module X would be called X_Y.

In Ada a similar thing happens. The global procedure name is composed of the package name followed by a double underscore followed by the name of the procedure, and the whole name rendered to lowercase. Thus procedure Y defined in the specification of package X would be called x__y

Therefore to access an Ada procedure from C you first need to declare the Ada procedure as an external procedure

```
extern void adaunit__adaprocedure (void);
```

and then call it using its full mangled name

```
adaunit__adaprocedure();
```

To do this in Modula-2 we have to import the package and then call the procedure in the same way we would for a procedure written in Modula-2

```
IMPORT AdaUnit;
AdaUnit.AdaProcedure;
```

As this is written in the same way that a Modula-2 procedure would be called we need to inform the translator that the procedure being called is an Ada procedure rather than one written in Modula-2.

This is achieved by creating a foreign definition module that tells the Modula-2 translator which language the procedures within the module are written in.

```
DEFINITION MODULE ["Ada"] AdaUnit;
  PROCEDURE AdaProcedure;
END AdaUnit.
```

The above informs the translator that the procedure AdaProcedure in the module AdaUnit is written in Ada and therefore will have its global name mangled to adaunit__adaprocedure.

We then have to write and compile the procedure in Ada

```
package body AdaUnit is
  procedure AdaProcedure is
  begin
    null;
  end AdaProcedure;
end AdaUnit;
```

and then make a specification so that it is exported.

```
package AdaUnit is
  procedure AdaProcedure;
end AdaUnit;
```

The Ada package has to be compiled using GNAT and the Modula-2 translated into C which is then compiled by the GCC. The resultant objects then have to be linked together to produce an executable.

In order that certain Ada features are made available, the compiler requires a number of ads files to be found somewhere in the source search path.

A package implementation is not required because the implementation is intrinsic (built into) the compiler.

For example, using Ada.Unchecked_Conversion requires that the file a-unccon.ads to be found in the source file search path.

Unfortunately GNAT has the very strange restriction that these specification files MUST have the "crunched" file names listed below. It does not support their being named according to the more usual convention derived from the full name of the package they contain. This is presumably some sort of historical left over, which is a pity, because these names are both ugly and unreadable.

- a-except.ads (ada.exceptions)
- a-unccon.ads (ada.unchecked_conversions)
- interfac.ads (interfaces)
- s-maccod.ads(system.machine_code)
- s-stoele.ads (system.storage_elements)
- s-unstyp.ads (system.unsigned_types)

3 Debugging

If the executable had been written entirely in Ada and ran on a machine sitting on a nearby desktop, we could have used something like GPS for debugging. However this is not the case. Our code is a mixture of Modula-2, C and now Ada. Moreover the machines are physically remote and not easily accessible.

So when something goes wrong our machines generate a memory dump and then, sometime later, we use a static dump analyser. The analyser uses the debug information that is stored in the executable file by the compiler and linker. It expects this information to be written according to the DWARF standard.

Fortunately for us GNAT is based on the GCC, which accepts the switch -gdwarf-3. This switch causes GNAT to supply debugging information according to version 3 of the DWARF standard and to place this information into the ELF executable.

Our challenge has been to enhance our analyser to better support bit fields and sub-ranges – something it never had to deal with when the executables were built purely from C.

Another debugging problem concerns the GCC's link time optimisation feature. This feature is enabled using the -lto switch and is required for the in-lining explained later in section 7.

Entries within the DWARF debugging information are contained within compilation units. These compilation units are Ada packages or Modula Modules. The full global name of an entity can normally be derived by prefixing the name of the compilation unit with the name of the entity. Unfortunately a side effect of using the lto feature is that the compilation units are all renamed <artificial>!

To solve this problem all our Modula-2 & C variables and procedures have to be name mangled in order that we can differentiate and know in which module the entity was defined. We have to do this even if the entity is not exported, i.e. is only used locally and therefore, from the linker's perspective, does not have to have a globally unique name.

Fortunately for us, Ada also mangles all its names - even if they are not exported. So this is not a problem and we can therefore use Link Time Optimization.

4 Functions

To make our example a little more useful we can replace the parameter-less procedure with a function that increments a global variable and returns its new value.

```
package AdaUnit is
  function AdaFunction return Integer;
end AdaUnit;

package body AdaUnit is

  TheGlobal : Integer;

  function AdaFunction return Integer is
  begin
    TheGlobal := TheGlobal + 1;
    return TheGlobal;
  end AdaFunction;

end AdaUnit;
```

However when we try to link a program that calls AdaFunction the linker complains that it is missing a last chance handler for the function.

This is because the function will raise an exception when TheGlobal reaches Integer'last. If this situation is not explicitly handled, the Ada compiler will insert a call to the last chance handler **__gnat_last_chance_handler**.

Of course, one could define a last chance handler and then link this into the final program. However we chose not to. Instead we chose to always explicitly handle Ada implicit exceptions.

For example by rewriting the code so that the error situation cannot arise:

```
function AdaFunction return Integer is
begin
  if TheGlobal < Integer'last then
    TheGlobal := TheGlobal + 1;
    return TheGlobal;
  else
    return Integer'last;
  end if;
end AdaFunction;
```

or by catching the exception

```
function AdaFunction return Integer is
begin
  TheGlobal := TheGlobal + 1;
  return TheGlobal;
exception
when Constraint_Error =>
  return Integer'last;
end AdaFunction;
```

By adding the switch -gnatw.x the Ada compiler will generate a warning if an implicit or explicit exception is not covered by a local handler.

Unfortunately Ada doesn't always get it right and we often have false positives – occasions when Ada warns that an exception may be raised when in fact this is not possible.

In the following example Ada complains that *Constaint_Error* might be raised when calling The_Handler.all even though the explicit check for a null pointer precludes this.

```
type Handler is access procedure;
The_Handler : Handler;

procedure Test is
begin
  if The_Handler /= null then
    The_Handler.all;
  end if;
end Test;
```

Interestingly, if we switch off warnings for the duration of the code in question, the program still links. This therefore shows that the compiler was, in fact, smart enough to realise that the exception could not be raised.

Rather than disable and then re-enable warnings we prefer to use the pragma *Suppress* to remove the specific check.

Suppressing checks can be selective. Typically we place the code that is causing the problem within a declaration block and add the appropriate **pragma suppress** between the **declare** and **begin** statements.

For example:

```
declare
  pragma suppress (Access_Checks);
begin
```

We consider this less error prone than messing around with warnings but we also hope that, as the compiler is improved, it might one day warn us that these pragmas are no longer necessary.

5 Initialising Globals

Global variables can be initialised using the standard Ada syntax. In our example the global variable can be initialised to forty two by declaring it as:

```
TheGlobal : Integer := 42;
```

Initialising variables in this manner does not work without a runtime to initialise the variable. Zero footprint Ada does not have a runtime and so if used purely by itself it would require an alternative mechanism to initialise global variables. However we are using Ada within an existing system, the runtime of which will initialise **ALL** global variables, irrespective of the compiler used, provided that all the compilers adhere to a few conventions.

Fortunately for us, GNAT adheres to these conventions and so its global variables are initialised in the same way that global variables written in ether Modula-2 or C are.

How does this work?

Quite simply, global variables are placed in a section called .bss if they are initialised to zero or in a section called .data if they are initialised to anything else.

The following GCC linker script groups all the .bss variables along will all uninitialized variables (COMMON) together and sets two linker symbols to the start and end addresses of the area of memory they have been allocated. The same script groups all initialised data together, assigns another pair of linker variables to their start and end addresses and instructs the linker to place their initialization values into ROM.

```
.mdata :
{
  __Data_Start = ABSOLUTE(.);
  *(.data*)
  __Data_End = ABSOLUTE(.);
} > Ram AT > Rom
.bss :
{
  __Bss_Start = ABSOLUTE(.);
  *(.bss)
  *(COMMON)
  __Bss_End = ABSOLUTE(.);
} > Ram
__Data_Rom  = LOADADDR(.mdata);
__Bss_Size  = __Bss_End - __Bss_Start;
__Data_Size = __Data_End - __Data_Start;
```

The runtime has access to the linker defined global symbols __Data_Start, __Data_Rom, __Data_Size and Bss_Size. Using these symbols the runtime can initialise memory thus:

```
MOVE (DataRom(),DataStart(),DataSize());
FILL (BssStart(), 0, BssSize());
```

The first instruction copies the initial values of initialised variables into the space occupied by the variables. The second instruction initialises to zero all remaining variables.

6 Ada calling Modula-2

The examples so far have shown how code written in Modula-2 or C can call routines written in Ada however these Ada routines would be severely restricted if they were not able to communicate with portions of the application written in languages other than Ada.

To be useful our Ada routines need to be able to call routines written in Modula-2. This is achieved by declaring the function as an import using the C calling convention and by specifying its external name. In the case of Modula-2 the external name is the name of the module followed by an underscore followed by the name of the procedure.

For example, the Modula-2 module ModulaUnit could be defined as:

```
DEFINITION MODULE ModulaUnit;
  PROCEDURE ModulaFunction () : INTEGER;
END ModulaUnit.
```

And implemented as:

```
IMPLEMENTATION MODULE ModulaUnit;
  PROCEDURE ModulaFunction () : INTEGER;
  BEGIN
    RETURN 42;
  END ModulaFunction;
END ModulaUnit.
```

And then the function called from Ada as:

```
 procedure Example is
  function ModulaFunction  return Integer
  with Import, Convention => C,
       External_Name => "ModulaUnit_ModulaFunction";
begin
  TheGlobal := ModulaFunction;
end Example;
```

This is all very well provided that the types are base types that both Modula-2 and Ada agree are the same. Problems arise when the types are represented differently. In these cases a wrapper is required.

For example, in Modula-2 (and C) a Boolean is defined to be eight bits wide. In Ada the Boolean type is defined to be only one bit wide however the compiler is generally free to allocate more than this for objects of type Boolean – how much isn't defined by the language. Therefore when Ada calls a Modula-2 function that returns a Boolean we need to do this via a wrapper function.

For example:

If our Ada code wants to call the Modula-2 function Hardware_Is_Available from module Ip we first define the specification in Ip.ads as

```
function Hardware_Is_Available return Boolean;
```

and then define the wrapper in Ip.adb as

```
type Modula_BOOLEAN is new Standard.Boolean
with Size => 8;

 function Hardware_Is_Available return Boolean is

  function Ip_Hardware_Is_Available return
Modula_BOOLEAN
   with Inline, Import, Convention => C,
   External_Name => "Ip_HardwareIsAvailable";

begin
  return Boolean(Ip_Hardware_Is_Available);
end Hardware_Is_Available;
```

The astute will notice that the Modula-2 function that the wrapper calls is declared as Inline. Which brings us nicely onto the subject of in-lining.

7 In-lining

The relatively weak microprocessors we use cannot afford the overhead of superfluous calls. Certain time critical portions of our code must be in-lined for efficiency reasons. The GCC is very good at in-lining provided the –flto option is specified on the command line when compiling C and – Winline when linking. In addition, in order that Ada in-lines in the same way, we need to specify the switch –gnatn2 when compiling our Ada source code.

The result is very impressive; in-lining is possible between units as well as between languages. The example of the Boolean wrapper produces absolutely no extra code - the wrapper keeps Ada happy without any additional overhead.

8 Enumerations

In C, the amount of storage allocated to enumeration types defaults to the word size of the target machine. In our case this is 32 bits. However reserving 32 bits for every enumeration is extremely wasteful for microprocessors that are memory challenged, so we compile using the switch --short-enums which directs the GCC to use the least number of bytes possible to store any given enumeration. This turns out to have been a very fortunate decision because enumerations in Ada use the same storage strategy, and so by using this switch we make enumerations compatible between Ada and C.

9 Strings

Strings are another occasion when wrapper functions are required.

In the following example, the Modula-2 procedure takes a string as its parameter. Strings in Modula-2 are unconstrained arrays of character and so the procedure DefineComputerNameAs is defined as follows.

```
PROCEDURE DefineComputerNameAs (TheName :
ARRAY OF CHAR);
```

This translates into C as

```
extern void Nbns_DefineComputerNameAs(const
char [], unsigned long);
```

Where the unconstrained array of characters has been translated into two parameters, the first being the start address of the array and the second the number of elements in the array.

To call this from Ada we need to provide a wrapper. For example:

```
procedure Define_Computer_Name_As
(The_Name : String) is

  procedure Nbns_Define_Computer_Name_As
      (Name_Address : ADDRESS;
       Name_Size    : CARD32)
   with Inline, Import, Convention => C,
   External_Name =>
       "Nbns_DefineComputerNameAs";

begin
  Nbns_Define_Computer_Name_As
       (The_Name'address, The_Name'length);
end Define_Computer_Name_As;
```

10 Exception Handling

The above example is not quite right. We shouldn't pass the address of the String but the address of the first character of the string. However if we code this then we need to check that the string has at least one character and decide what to do if it doesn't.

Ideally we would raise an exception. Unfortunately zero foot print Ada precludes the propagation of exceptions, however this does not mean that we cannot define exceptions provided we catch them locally or use them for other purposes.

Note however that the –gnatwh compiler switch to detect declaration hiding does not detect the hiding of standard exceptions. The Standard exceptions

- Constraint_Error
- Program_Error
- Storage_Error
- Tasking_Error

are implicitly raised by compiler checks. Therefore, to avoid confusion, it is highly recommended not to declare exceptions with these names.

Our existing Modula-2 system has an exception concept. Our Modula-2 exceptions can be raised but not caught and are always fatal. They stop the machine and produce a memory dump for later analysis.

In the previous example, if we correct the code to pass the address of the first character, Ada will complain that this might raise an exception. So we need to include additional code that explicitly handles that situation.

```
Empty_Name : exception;
begin
  Nbns_Define_Computer_Name_As
    (The_Name(The_Name'first)'address,
     The_Name'length);
exception
when Constraint_Error =>
  HALT (Empty_Name'identity);
end Define_Computer_Name_As;
```

The procedure HALT saves the exception identity and stops the system. Our analyser can retrieve this identity, which is nothing more than the address of the exception, and convert it into its symbolic name.

11 Elaboration

Unlike C, Modula-2 has the concept of elaboration. It is not as powerful as Ada – global variables cannot be elaborated – but modules can have body code that is executed at start-up before any of the exported procedures can be called.

However our Ada packages only link to specific named routines and there is no concept of using "with" to import units written in anything other than Ada. Consequently there is no Ada syntax or mechanism whereby Ada can be instructed to elaborate a specific foreign unit.

And even if there were, we decided that all our Ada packages are either pure or pre-elaborate.

However this decision turns out to be too much of a restriction. Too much of our existing code requires the Modula-2 module bodies to be executed prior to their exported routines being made available. Not being able to elaborate our Ada packages was also inconvenient.

Therefore we changed our strategy and decided to implement and support elaboration.

The first problem was establishing the elaboration order. If unit A calls unit B then unit B must be elaborated before unit A is elaborated. If unit B calls other units then these must be elaborated before unit B and so on. If any unit calls a unit that has to be elaborated before itself, then this is a cyclic dependency and must be regarded as an error.

Because Modula-2 has the concept of elaboration our IDE already had a mechanism for determining the elaboration

order of Modula-2 modules. So all we had to do was extend this mechanism to include units written in Ada.

The IDE has to parse the Modula-2 source files and process the *IMPORT* statements and parse the Ada source files and process the *with* statements. Whilst the Modula-2 *IMPORT*s indicate a unit dependency, irrespective of language, the Ada *with* is restricted to indicating the package's dependency only on other Ada packages and does not include any dependency on units written in other languages.

We were therefore obliged to enhance our IDE to recognise a new pragma.

By default GNAT issues a warning when it encounters an unrecognised pragma. This warning can be switched off using the -gnatwG switch. Using this switch is potentially dangerous and contrary to the Ada Reference Manual specification that a warning be generated whenever an unrecognised pragma is encountered. Therefore we had to enhance our IDE to verify pragma names and issue our own error message if it detected any unrecognised pragmas, i.e. unrecognised by GNAT and not an extension implemented by our own IDE.

So solve the elaboration problem we recognised the new pragma *Modula_Import*. The pragma takes as its parameter the name of a Modula-2 module.

For example: **pragma** Modula_Import (ModulaUnit);

Note: It isn't quite that simple because Modula-2 module names can have names that aren't Ada identifiers. However how we handled this anomaly is a detail beyond the scope of this short article.

By processing the *IMPORT*s, *with*s and *pragma Modula_Import* statements, our IDE can build the dependency tree. Provided that there are no cyclic dependencies it can then generate a table of procedures that must be called at start-up before the main program is entered.

For example:

```
extern void
__attribute__((weak)) ModulaUnit_BEGIN(void);

extern void
__attribute__((weak)) adaunit___elabb(void);

typedef void (*Unit_List[ 1])(void);

static const Unit_List Unit_Body_the_list = {
  ModulaUnit_BEGIN,
  adaunit___elabb};
```

The name of the elaboration routine for a Modula-2 module is the name of the module followed by _BEGIN whilst the name of the elaboration routine for an Ada package is the name of the package followed by elabb if the implementation is being elaborated or elabs if the specification requires elaboration.

There is no easy way to detect whether or not an Ada package requires elaboration, so our IDE needs to assume that all Ada packages might be elaborated unless directed otherwise. This is implemented by the IDE building a table of weak links to possible elaboration routines.

The use of weak external links means that if the unit did not require elaboration and consequently the expected elaboration routine was not generated, the linker would not complain but instead leave the default null pointer in the table. These null entries obviously have to be skipped when processing the table.

```
int the_index;
 for (the_index = 0; the_index < 7ul; the_index++) {
   if (Unit_Body_the_list[the_index]!=0)
    Unit_Body_the_list[the_index]();
 };
```

To avoid possible cyclic dependencies it is sometimes necessary that Ada (and the IDE) be told that the package does not require elaboration. This is achieved using the aspect "*with preelaboration*" or "*with pure*".

12 Interrupt routines

Our applications require that we write interrupt routines. On ARM microprocessors, interrupt routines are nothing other than parameter-less procedures whose addresses are placed into the vector table.

Using standard Ada the address of the procedure is placed into the vector table using the *pragma Attach_Handler*. Unfortunately when we use this, GNAT complains that the argument of *pragma Attach_Handler* must be a protected procedure.

However protected types and procedures require a run-time and are therefore not allowed in the Zero Footprint Profile.

In any case, even if *Attach_Handler* was allowed, it is unlikely that it would of any use because we need a mechanism that allows a vector table to be generated that has entries of procedures written in a mixture of languages – not just Ada.

For this reason, our IDE builds the vector table. The IDE is instructed to add a procedure into the vector table by special constructs within the source.

In Modula-2 this is achieved by using the direct language specification "Vector"

For example:

```
PROCEDURE ["Export", "Vector=36"] InterruptHandler;
```

In order that a similar thing could be achieved from sources written in Ada, we further enhanced our IDE to recognise an additional **pragma** Use_Vector

For example:

```
procedure Interrupt_Handler with Export;
pragma Use_Vector (36);
```

In the above example, the pragma Use_Vector instructs the IDE to place the address of the exported, parameter-less procedure Interrupt_Handler into interrupt vector position 36.

13 The use of assembler

We haven't had cause to write much assembler but there will always be occasions when this is necessary. Fortunately this is possible. The GNAT package System.Machine_Code provides the procedure Asm which behaves in a similar and recognisable manner to that of the standard GCC embedded assembler but with the rather tiresome restriction that parameters can only be reference by position rather than by name.

In the following example, written in C, the procedure DisableInterrupts places the constant 1 into a register of its choice that we symbolically call Mask which the MSR instruction then loads into the Primask register.

```
__attribute__ ((always_inline)) inline
static void DisableInterrupts(void)
{
  asm volatile (
      "MSR primask, %[Mask];"
      :
      :[Mask] "r" (1)
      :"memory");
}
```

Unfortunately GNAT does not support the use of named parameters and therefore in Ada the register used to house the constant has to be referred to by its position in the list of inputs (starting at zero)!

```
with System.Machine_Code; use
System.Machine_Code;

procedure Disable_Interrupts with Inline is
begin
  Asm ("msr primask, %0;",
      Inputs   => Integer'asm_input ("r", 1),
      Clobber  => "memory",
      Volatile => True);
end Disable_Interrupts;
```

Referring to parameters by their numeric position rather than by name seems like a step back into the stone-age.

14 Results

In the guise of conducting a feasibility study, we did exactly what we originally stated we wouldn't do. Rather than wait until an opportunity arose that would benefit from being written in Ada we decided to convert two ARM based applications that already existed and had already been written in Modula-2.

We didn't convert the whole application; we left the run-time and a lot of low level interfaces written in Modula-2 but we did convert all the application specific modules into Ada packages.

These included interrupt routines, interfaces to hardware and, of course, interfaces to our proprietary multitasking runtime.

So the port wasn't exactly trivial but on the other hand because of the similarities between Modula-2 and Ada it wasn't that difficult either.

We are pleased to report that the conversions were very successful and we now have two of our ARM specific applications written in Ada.

This is not to say that the conversion didn’t have any problems. Unfortunately we did introduce a few errors as part of the conversion process. These occurred when the conversion was more complex than a simple syntax change

We identified four areas where conversion errors were likely to occur:

1. Ada has no syntax to increment or decrement a variable so it is impossible to implement the Modula-2 procedures INC and DEC without resorting to generics.
2. Modifying the code to replace INC and DEC statements with X:=X+1 and X:=X-1 respectively presented an opportunity to accidently decrement when we should have incremented and vice versa.
3. The Ada attribute ‘size returns the size of an object in bits whereas the equivalent Modula-2 SIZE procedure returns the size in bytes. Therefore one must remember to divide ‘size by the number of bits in a byte.
4. Expressions in Modula-2 are evaluated left to right and so there is no need for the Ada constructs *and then* and *or else*. Care is therefore required when converting Modula-2 Boolean expressions.
5. In Modula-2 *in* parameters can be modified – thereby saving a local variable. In Ada this is not allowed and so a local variable must be explicitly created, initialised and then used instead of the original *in* parameter. This complicated code modification is another opportunity to introduce subtle conversion errors.

15 Conclusion

This article is an experience report. It does not present anything particularly clever or original. Far from it. Our goal in writing this article was to illustrate how easy it is to integrate Ada into an existing non-Ada system and thereby perhaps animate others in a similar situation to use Ada where previously it would not have been considered.