19. Runtime Errors And SPARK

Runtime errors are generally all those situations when the program does not function according to our expectations. In the case of Ada, however, this term does not refer to all possible bugs, but has a particular meaning and refers to those events that are identified as such by the language specification. Examples of runtime errors, as defined by the language, are an attempt to divide by zero or an attempt to overflow an array. I have used the word "attempt" to stress that as long as the Ada language is concerned, these things do *not* happen - they are supposed to be *detected and prevented* by the language implementation and reported to the user by means of exceptions.

This is where things can get tricky, as both of these actions (detection of dangerous situation and reporting them with exceptions) require that the executable code contains additional elements that do not directly correspond to the source code and are injected into the final executable by the compiler and linker - these are the language runtime elements that we have consistently tried to avoid throughout this tutorial. In practical terms, we have explicitly instructed the compiler to *not* inject any error checking code with the compiler option `-gnatp` - this allowed us to continue writing programs without linking them with language runtime libraries.

One could argue that the biggest advantage of Ada (over C, for example) is exactly the existence of these language features, as they contribute to more robust programs. This is true and there should be no doubt that it is better for the program to detect dangerous situations and perhaps handle them in *some meaningful way* than to fall victim of undefined behaviour. But this view will depend on the context and the constraints of embedded systems allow reviewing this subject. Note that by *constraints* we mean not just the memory or processing power limitations, but also the difficulty of integration (which includes the licensing concerns) as well as the ability to reason about the final system.

We will take a different approach, which is consistent with what was presented from the beginning of this tutorial - instead of relying on the language runtime we will benefit from the static language features to demonstrate the correctness of the program, which in turn will allow us to justify why the language runtime support is not needed. For example, instead of relying on exceptions when there is an attempt to run past the end of some array, we will demonstrate that such a situation never happens and the code is safe. This will not allow us to neglect error handling altogether (the focus will move from error detection to input validation), but will be sufficient to keep the integration process simple.

In short, what we want is to keep the `-gnatp` compiler option but with confidence that the correctness of the program is not compromised even if runtime checks are not present in the executable code.

The correctness of the program can be demonstrated, with varying levels of confidence, in different ways and the traditional approach is to do it by means of testing and review. Testing techniques is a broad subject that deserves separate book(s), but the idea is that if we can test the program with sufficient number of test cases (where *sufficient* refers to the exhaustion or at least well thought-out coverage of input data and code structure), then we can conclude that the program will always behave properly in normal use. This approach, especially when used with code reviews, can give high level of confidence that the program is correct and is widely used in embedded systems, including safety-critical ones. This is what we can do with all our example programs, even without modifying them.

Another approach, which is definitely worth exploring in the context of Ada, is the use of static analysis coupled with a reasonable dose of formal reasoning in order to obtain a mathematical *proof of correctness*, even without exhaustive testing. The reason why this approach is particularly related to Ada is that Ada supports this way of working much better than other programming languages thanks to its rich type system and relatively unambiguous semantics. In fact, there exists a set of tools that assist the programmer in such correctness proofs and these tools, together with some additional rules and constraints on how the language is used, form a solution known as SPARK. Most people consider SPARK to be a programming language that is based on Ada, but in reality SPARK refers to the language *and* the tools that are used to assist the programmer in correctness proofs. The proof tools are available for free from AdaCore and the final program, once analysed, can be compiled to the executable form using a regular Ada compiler. This also means that the SPARK tools are not themselves related to the ARM microcontrollers in any way, they operate on the source code alone and do not interfere with the compilation and linking for the target platform. As such, these tools can be installed on a regular desktop computer without regard to where the final executable will be installed.

SPARK does not allow all Ada language features and restricts the language significantly, but this should not be a problem for us, as we have already used a very constrained subset of the language. A much bigger problem is that even though Ada has a rich type system, it still allows the programmer to hide lots of information with operations on global variables, which are not taken into account in subprograms' signatures and such hidden operations are not allowed in SPARK as they obstruct the information flow analysis. Note that we have a significant number of such operations in relation to global objects that represent microcontroller's registers. This means that we can attempt to analyse one of our earlier programs *after* it was written, but we should be prepared for some rework so that the missing information is made explicit. This is also the reason why it is recommended to take the use of SPARK under consideration from the very beginning of the project. Still, we can try to prove the freedom from runtime errors in all of our programs and the following sections correspond to each complete example from previous chapters. We will fix all problems one by one as we find them.

Note that static analysis itself does not produce any artefacts that are themselves executable or that lead to the subsequent creation of executable code. As such, analysis is an activity that provides *feedback* on the source code and there are no strict and universal rules for how this feedback is used in the project. This makes it different from the compilation process, where the detected program error can prevent the compiler from producing any executable code at all - the static analysis, if performed by a separate tool, has no impact on the compilation process and it is only a matter of engineering judgement how the information obtained during static analysis should be used in the project. We will therefore *not* blindly fight to get 100% clean analysis results (which is a sometimes promoted approach), but will rationally decide whether the information from the tool justifies modification of the source code. This approach is also justified by the fact that the SPARK language was recently updated and its toolset was almost completely replaced with newer products - at the time of writing this tutorial these tools still exhibit some minor issues that will be likely corrected in their future versions. Note also that the tool output presented below might be different on your system, as with each subsequent release the reasoning capability of the tool is improved. Some of the messages described below might disappear in newer tool versions and on the other hand there might be some new diagnostics that were not yet implemented at the time of writing this text.

The following sections describe how the existing examples can be analysed with the SPARK tool and what modifications of the source code are required to get satisfactory analysis results and a high level of confidence that the programs are free from runtime errors.

First Program

In order to start working with SPARK we will need a *project file*, which is needed by a SPARK tool. The project file describes the project as a whole and allows bringing together the information on compiler options, linked libraries, etc. The simplest project file named `first_program.gpr` can be empty, with this minimal form:

```
project First_Program is
end First_Program;
```

We do not need to handle compilation and linking steps with this file, so we can leave it empty and assuming that all program files are in the same directory, we can run the SPARK tool like this:

```
$ gnatprove -Pfirst_program
Phase 1 of 3: frame condition computation ...
Phase 2 of 3: analysis and translation to intermediate language ...
Phase 3 of 3: generation and proof of VCs ...
warning: no bodies have been analyzed by GNATprove
enable analysis of a body using SPARK_Mode
```

No bodies have been analysed, because we did not instruct the tool which files in the whole program should be taken for analysis as SPARK source files - by default the tool assumed that all files are written in full Ada and are not appropriate for analysis. The hint in the last line above says that `SPARK_Mode` can be used to enable analysis of a body and we will do it by adding the following line at the beginning of *all* package specification (.ads) and body (.adb) files:

```
pragma SPARK_Mode;
```

When the `gnatprove` tool is run again with these pragmas in place, we can see error messages as the code does not fully conform to the SPARK language rules. For example:

```
$ gnatprove -Pfirst_program
Phase 1 of 3: frame condition computation ...
Phase 2 of 3: analysis and translation to intermediate language ...
program.ads:5:14: warning: subprogram "Run" has no effect
gprbuild: *** compilation phase failed
gnatprove: error during analysis and translation to intermediate
language, aborting.
```

This error message appears because from the SPARK point of view the `Run` procedure does nothing. Currently, SPARK does not have any means to represent the passage of time (or the *wasting* of time) and the simplest way to deal with this warning is to switch it off at the level of procedure declaration:

```
   pragma Warnings (Off);
   procedure Run;
   pragma Warnings (On);
```

Another approach is to simply keep this issue in mind every time we analyse the output of the SPARK tool. We will go with that second option, which will allow us to constantly keep some issues on the table and observe how they are changing as the project evolves. Our projects are not big enough to justify switching warnings off with the intention to only reduce the amount of diagnostic messages.

While we are analysing the `Run` procedure we can also express the fact that `Run` should never finish with the `No_Return` aspect, and write the following complete package specification:

```
pragma SPARK_Mode;

package Program
is

   procedure Run
      with No_Return;
   pragma Export (C, Run, "run");

end Program;
```

Apart from the `pragma SPARK_Mode`, the package body does not require any further modifications:

```
pragma SPARK_Mode;

package body Program
is

   procedure Run
   is
   begin
      loop
         null;
      end loop;
   end Run;

end Program;
```

If we run the SPARK tool again, this time with the `--warnings=continue` option to make sure that the tool does not stop after reporting a single warning, we will get the following output:

```
$ gnatprove -Pfirst_program --warnings=continue
Phase 1 of 3: frame condition computation ...
Phase 2 of 3: analysis and translation to intermediate language ...
program.ads:6:14: warning: subprogram "Run" has no effect
Phase 3 of 3: generation and proof of VCs ...
analyzing Program, 0 checks
analyzing Program.Run, 1 checks
```

The single check that was reported for procedure `Run` is related to the `No_Return` aspect - if you try to remove the infinite loop (for example by replacing it with a single null statement), you will see that this condition is no longer satisfied:

```
$ gnatprove -Pfirst_program
Phase 1 of 3: frame condition computation ...
program.adb:9:10: implied return after this statement would have raised
Program_Error
program.adb:9:10: procedure "Run" is marked as No_Return
gprbuild: *** compilation phase failed
gnatprove: error during frame condition computation, aborting.
```

Violation of the `No_Return` aspect is very serious and falls under the category of undefined behaviour, especially if this happens in the program's main procedure. We can use SPARK to detect such potential errors and we will use this aspect in all further examples.

Note that if you run the SPARK tool twice and there will be no hard error reported, the second run might not show any warnings at all - the SPARK tool can accumulate state from previous runs (the results are stored in files created in the `gnatprove` directory that is automatically created in the project's directory) and it makes sense to clean it to get a fresh scan of our code:

```
$ gnatprove -Pfirst_program --clean
```

Note also that the `--quiet` option can reduce the output to warnings and errors only:

```
$ gnatprove -Pfirst_program --warnings=continue --quiet
program.ads:6:14: warning: subprogram "Run" has no effect
```

We will use this last variant for brevity with further examples unless it will be beneficial to have more information for a more thorough problem analysis.

Digital Output

The Digital Output example does not require any extensive modifications apart from those that are analogous to the previous program. Note that the infinite loop, which was initially part of the main `Run` procedure, was moved to package `Utils`, to procedure `Spin_Indefinitely`. We should add the `No_Return` aspect to its declaration, so that the package specification should look like this:

```
pragma SPARK_Mode;

package Utils
is

   procedure Spin_Indefinitely
      with No_Return;

end Utils;
```

The `Run` procedure should still be marked as `No_Return` and the SPARK tool should be able to figure out that if the last statement in `Run` is (partial content of `program.adb`):

```
pragma SPARK_Mode;
-- ...
package body Program
is

   procedure Run
   is
   begin
      -- ...

      Utils.Spin_Indefinitely;
   end Run;

end Program;
```

then the `No_Return` condition for `Run` is automatically satisfied as well.

Unfortunately, some other rules implemented in the SPARK tool prevent it from accepting our code without any complaints and now we see the following report:

```
$ gnatprove -Pdigital_output --warnings=continue --quiet
utils.ads:6:14: warning: subprogram "Spin_Indefinitely" has no effect
program.adb:18:12: warning: precondition might fail
```

The first message is similar to the one that we have seen with our first program and we can treat it as expected. The second message is related to the fact that a procedure with `No_Return` aspect that does not have any output parameter is considered to be an error signaling mechanism and an implicit precondition evaluating to `False` is injected where such procedure is called. We can ignore this message based on the source code analysis, as effectively *halting* the running program is exactly what we wanted to do.

Apart from these two messages the tool does not report any problems with the code, so we can conclude that the program is free from runtime errors.

Very Simple Delays

This example program has a very similar structure to the previous one, so the modifications for SPARK conventions will not be surprising - after adding pragma `SPARK_Mode` at the beginning of each source file and marking procedures `Run` and `Spin_Indefinitely` with the `No_Return` aspect, the SPARK tool reports simply:

```
$ gnatprove -Pvery_simple_delays --warnings=continue --quiet
utils.ads:6:14: warning: subprogram "Spin_Indefinitely" has no effect
utils.ads:9:14: warning: subprogram "Waste_Some_Time" has no effect
```

The `Waste_Some_Time` procedure has no effect from the SPARK point of view, but we accept the warning message knowing that the purpose of this procedure is exactly to waste time.

Note that the previous message about failing precondition has disappeared, as the call to `Spin_Indefinitely` is not used at all in the `Run` procedure.

The results obtained for this example program gives us confidence that it is free from runtime errors.

Random Numbers

This example program will be the first where some interesting problems will start arising. The basic modifications (adding pragma `SPARK_Mode` at the beginning of all files) to enable SPARK analysis are the same as with previous examples, but when we run the SPARK tool we will see this output:

```
$ gnatprove -Prandom_numbers --warnings=continue --quiet
utils.ads:6:14: warning: subprogram "Spin_Indefinitely" has no effect
utils.ads:9:14: warning: subprogram "Waste_Some_Time" has no effect
random_numbers.adb:22:16: warning: unused assignment
random_numbers.adb:23:10: warning: statement has no effect
```

The messages about no-effect subprograms are not surprising (they deal with the passage of time, which is not easily modelled in SPARK), but what follows is more interesting - lines 22 and 23 in `random_numbers.adb` are:

```
         Ready := Registers.TRNG_ISR;
         exit when (Ready and 1) /= 0;
```

These two lines are immediately placed in the loop and there is nothing inherently wrong in them - they are supposed to poll the `TRNG_ISR` register until its first bit is set to 1. The register is declared as a volatile object to prevent the compiler from optimizing such loops and we expect it to work properly. The problem is that SPARK allows more fine-grained control over information flow with regard to external state (*external state* is what is outside of the program and volatile variables can be used to connect both internal and external sides of the program) and in particular it can recognize objects that can get modified by means that are external to the program and also recognize objects where multiple reading operations are meaningful even without intervening writes. As an example, consider this snippet:

```
   X := Some_Global_Variable;
   X := Some_Global_Variable;
```

From the SPARK point of view, the above is an error of the *information flow* category - there is no point in reading the same value twice without intervening write and the first assignment is considered to be unused (simply writing to the same object without intervening read is also a problem). But we really need to do it with regard to *some* I/O registers, as they are supposed to change outside of the program and repeated reads are exactly what has to be done in order to detect that some condition that is external to the program has changed. In order to account for this, SPARK introduces additional aspects for volatile objects:

- `Async_Readers` - to specify that the value written to the object can be consumed by some component that is external to the program,
- `Async_Writers` - to specify that the value of the object can be updated by some external component,
- `Effective_Reads` - to specify that each read should be considered to be effective,
- `Effective_Writes` - to specify that each write should be considered to be effective.

These new aspects allow to control with fine granularity our expectations with regard to the information flow to and from the external state. In theory, if these aspects are not specified, then they are all `True` for volatile objects. Unfortunately, the current version of the SPARK tool does not follow this rule and we have to explicitly define these aspects when needed, like in this example. Our modified declaration of `TRNG_ISR` (in package `Registers`) is:

```
   TRNG_ISR : Word
      with Async_Writers, Effective_Reads;
   pragma Volatile (TRNG_ISR);
   pragma Import (C, TRNG_ISR, "TRNG_ISR");
```

Now, with the `Async_Writers` and `Effective_Reads` aspects in place, SPARK understands that subsequent reads of the same register are meaningful and stops complaining about the unused assignments in the loop that polls the register waiting for the next value to be ready.

Note that the above modification should not be considered as a burden or an obstacle - with this fine-grained control over the information flow SPARK allows us to double-check our expectations with regard to how individual registers should be used. Not all registers will have the same properties and having the possibility to express our intent (and have an automated tool to verify it) is an added value in correctness analysis. Such modifications might be necessary with other registers, depending on their function and how they are supposed to be used in the program.

After this slight modification, the messages that we get for the program are:

```
$ gnatprove -Prandom_numbers --warnings=continue --quiet
```

```
utils.ads:6:14: warning: subprogram "Spin_Indefinitely" has no effect
utils.ads:9:14: warning: subprogram "Waste_Some_Time" has no effect
```

These diagnostics are already known and understood, and we can conclude that the program is free from runtime errors.

Digital Input

There are no unsatisfied verification conditions in this program and after adding pragma `SPARK_Mode` to all source files, the SPARK tool generates a clean report - the program is free from runtime errors.

Finite State Machines, Part 1

After preparing all source files for SPARK analysis, the only diagnostic messages that we see refer to `Utils` procedures having no side-effects. We can accept this and conclude that the program is free from runtime errors.

Constant Values

Similarly to the previous example, the `Run` procedure is flagged as having no side-effects, which we accept - this program was meant to demonstrate the use of constant values and does not perform any useful operations. As such it is free from runtime errors.

Finite State Machines, Part 1

No new surprises in this example, the program is free from runtime errors.

Machine Code Insertions

This example was not even a complete compilable program, but a package demonstrating the use of magic subprogram `System.Machine_Code.Asm`. The SPARK tool does not attempt to analyse the meaning of such machine code insertions and in consequence treats them as null statements. This can be seen in the following diagnostic messages:

```
$ gnatprove -Pmachine_code_insertions --warnings=continue --quiet
utils.ads:6:14: warning: subprogram "Spin_Indefinitely" has no effect
utils.ads:9:14: warning: subprogram "Waste_Some_Time" has no effect
utils.ads:11:14: warning: subprogram "Enable_Interrupts" has no effect
utils.ads:13:14: warning: subprogram "Disable_Interrupts" has no effect
utils.ads:15:14: warning: subprogram "Wait_For_Interrupt" has no effect
```

We can accept these messages, but note, ironically, that our package `Utils` seems to contain only useless utils...

Interrupts

Things become interesting again in this example, but only for the Arduino Due, where we find another problem related to the use of external state. The set of messages for the code after adding pragma `SPARK_Mode` looks like this:

```
$ gnatprove -Pinterrupts --warnings=continue --quiet
utils.ads:6:14: warning: subprogram "Spin_Indefinitely" has no effect
utils.ads:9:14: warning: subprogram "Waste_Some_Time" has no effect
utils.ads:11:14: warning: subprogram "Enable_Interrupts" has no effect
utils.ads:13:14: warning: subprogram "Disable_Interrupts" has no effect
utils.ads:15:14: warning: subprogram "Wait_For_Interrupt" has no effect
program.adb:17:11: warning: statement has no effect
pins.ads:20:14: warning: subprogram "Clear_Interrupt" has no effect
pins.ads:20:31: warning: unused initial value of "Pin"
```

We have already analysed the messages for package `Utils`, but here the tool complains about package `Pins` - unfortunately, the tool points us to the subprogram specification, claiming that the parameter `Pin` is not used. The complete body of the procedure in question is:

```
   procedure Clear_Interrupt (Pin : in Pin_ID)
   is
      Dummy : Registers.Word;
   begin
      case Pin is
         when Pin_3 | Pin_4 =>
            Dummy := Registers.PIOC_ISR;

         when others =>
            null;
      end case;
   end Clear_Interrupt;
```

We can see that the parameter `Pin` is used as an expression in the `case` statement - but we also see that the only statement inside is a read of a register object. The value that is read and assigned to the `Dummy` variable is not used and this can lead the tool to mark the whole subprogram as having no effects. From our point of view, however, the effect of this subprogram is in the fact of reading the register value, as this is the intended way to clear the interrupt flag - note again that this is a property of the microcontroller used in the Arduino Due board, where clearing the interrupt flag is done by *reading* it (the STM32 chips use *writing* for this purpose, which does not confuse the static analysis tool). The Atmel microcontroller knows that the read operation on this particular register has to be handled in a special way with such side-effects, but the SPARK tool has no such knowledge and reports what from its point of view is a data flow error.

In order to solve this problem we will use explicit values for the external state aspects (in `registers.ads`):

```
   PIOC_ISR : Word
      with Async_Writers, Effective_Reads;
   pragma Volatile (PIOC_ISR);
   pragma Import (C, PIOC_ISR, "PIOC_ISR");
```

After this modification the SPARK tool no longer complains about unused values in procedure `Clear_Interrupt`, but the following message remains (in addition to the usual ones about procedures in `Utils`):

```
program.adb:17:11: warning: statement has no effect
```

where line 17 in file `program.adb` is:

```
      Pins.Clear_Interrupt (Pins.Pin_3);
```

Curiously, the procedure `Clear_Interrups` is no longer flagged as having no effects thanks to explicit specification of external state aspects for the PIOC_ISR register, but the call to this procedure is still considered to have no effect. There is no need for us to fight the tool by introducing more aspects as the issue is straightforward to analyse manually and since we are mainly interested in possible runtime errors, we can stop the analysis at this point.

Shared State

After adding required pragmas to enable SPARK analysis, we get the following messages:

```
$ gnatprove -Pshared_state --warnings=continue --quiet
program.adb:6:04: warning: variable "X" is never read
  and never assigned
program.ads:6:14: warning: subprogram "Run" has no effect
```

These messages are not surprising, as the program was meant to only demonstrate how the shared variable is allocated in the executable image. There is no need to modify the program in any way, but we should note that SPARK performs an information flow analysis for static variables - this will be more visible in later examples.

Finite State Machines, Part 3

This example program does not bring any new issues, after enabling SPARK analysis for all source files we can see the following SPARK diagnostics:

```
$ gnatprove -Pfsm_3 --warnings=continue --quiet
utils.ads:6:14: warning: subprogram "Spin_Indefinitely" has no effect
utils.ads:9:14: warning: subprogram "Waste_Some_Time" has no effect
utils.ads:11:14: warning: subprogram "Enable_Interrupts" has no effect
utils.ads:13:14: warning: subprogram "Disable_Interrupts" has no effect
utils.ads:15:14: warning: subprogram "Wait_For_Interrupt" has no effect
program.adb:71:12: warning: statement has no effect
program.adb:72:11: warning: statement has no effect
program.adb:73:11: warning: statement has no effect
```

The last three messages refer to these statements in `program.adb`:

```
      Utils.Waste_Some_Time;
      Pins.Clear_Interrupt (Pins.Pin_2);
      Pins.Clear_Interrupt (Pins.Pin_3);
```

We can accept these diagnostics without further analysis and consider the program as safe with regard to runtime errors.

System Time

This example program is more complex than the previous one due to more involved patterns of sharing data between program and its interrupt handlers. As an effect we can see the following diagnostic messages after initial source code preparations:

```
$ gnatprove -Psystem_time --warnings=continue --quiet
utils.adb:58:22: volatile object cannot appear in this context
(SPARK RM 7.1.3(13))
utils.adb:75:14: volatile object cannot appear in this context
(SPARK RM 7.1.3(13))
utils.adb:87:13: volatile object cannot appear in this context
(SPARK RM 7.1.3(13))
gprbuild: *** compilation phase failed
gnatprove: error during frame condition computation, aborting.
```

The volatile object in question is the `System_Time` variable declared in the body of pacakge `Utils`, which is intended to keep track of the passing time. This object is declared as volatile even though it does not correspond to any state that is external to the program - the volatile property was assigned to this object to ensure proper data updates between main program and the SysTick interrupt handler. Still, from the information flow point of view, the interrupt handler that updates the variable is similar to a register that is automatically updated by purely hardware means. We should therefore make the information flow explicit with appropriate aspects:

```
   System_Time : Time
      with Async_Readers => False,
           Async_Writers => False;
   pragma Volatile (System_Time);
```

This does not cause the error messages to disappear, because according to the SPARK reference manual, such objects should be used only in restricted ways - either on the right or on the left side of assignment or as a subprogram parameter. This is one of the statements that the tool complains about in procedure `SysTick_Handler`:

```
      System_Time := System_Time + SysTick_Period_Millis;
```

Such expressions should be restructured and one possible way to do so is the following complete handler body:

```
   procedure SysTick_Handler
   is
      Old_Time_Value : Time := System_Time;
   begin
      System_Time := Old_Time_Value + SysTick_Period_Millis;
   end SysTick_Handler;
```

This makes the error message disappear. Similarly, the corrected `Clock` function:

```
   function Clock return Time
   is
      Current_Time : Time := System_Time;
   begin
      return Current_Time;
   end Clock;
```

and the modified `Delay_Until` procedure:

```
   procedure Delay_Until (T : in Time)
   is
      Current_Time : Time;
   begin
      loop
         Current_Time := System_Time;
         exit when Current_Time >= T;
         Wait_For_Interrupt;
      end loop;
   end Delay_Until;
```

allow the tool to continue and produce the following new message (in addition to usual ones):

```
utils.ads:21:13: function with output global "System_Time" is not
allowed in SPARK
```

This message refers to the `Clock` function and we can see that another restriction related to volatile objects is that they cannot be used (as global objects) by functions. This means that we need to rewrite the `Clock` function as a procedure:

```
   procedure Clock (T : out Time)
   is
   begin
      T := System_Time;
   end Clock;
```

Of course, the code that uses the `Clock` function also needs to be modified accordingly.

After all these formal changes we get the following set of diagnostic messages:

```
$ gnatprove -Psystem_time --warnings=continue --quiet
utils.adb:93:10: warning: statement has no effect
utils.ads:6:14: warning: subprogram "Spin_Indefinitely" has no effect
utils.ads:9:14: warning: subprogram "Waste_Some_Time" has no effect
utils.ads:11:14: warning: subprogram "Enable_Interrupts" has no effect
utils.ads:13:14: warning: subprogram "Disable_Interrupts" has no effect
utils.ads:15:14: warning: subprogram "Wait_For_Interrupt" has no effect
utils.adb:60:37: warning: overflow check might fail
program.adb:25:41: warning: overflow check might fail
program.adb:29:37: warning: overflow check might fail
```

The last three messages are particularly interesting as they indicate the possibility of runtime errors. The first message refers to the SysTick handler:

```
   procedure SysTick_Handler
   is
      Old_Time_Value : Time := System_Time;
   begin
      System_Time := Old_Time_Value + SysTick_Period_Millis;
   end SysTick_Handler;
```

Indeed - the incrementation of the `System_Time` variable is not protected against overflow (or at least SPARK does not have any reason to believe such a protection was put in place) and if the system is running long enough, an overflow condition will surely happen. If we try to compile the program without the `-gnatp` option, the compiler will inject appropriate range checks so that the operation will be protected at runtime. This has two problems - we need to handle the possible exception in some reasonable way (which is not easy in an embedded system, *especially* in the interrupt handler!), and we need to link in the language support runtime code that is referenced by such check. We have decided to avoid this and compile the program with the `-gnatp` option, but this leaves the above code unprotected.

There is no solution to this problem that can be described as universally good, as the *right* solution will depend on our expectations (requirements) with regard to this system. We might decide to reset the system time periodically, but this might not be satisfactory in those systems that expect the time to be always monotonic. The most important thing, however, is that SPARK makes such issues visible and that they have to be explicitly taken care of.

As a matter of exercise, consider the following modified SysTick handler:

```
   procedure SysTick_Handler
   is
      Old_Time_Value : Time := System_Time;
   begin
      if Old_Time_Value < 1_000_000 then
         System_Time := Old_Time_Value + SysTick_Period_Millis;
      else
         System_Time := 0;
      end if;
   end SysTick_Handler;
```

This code puts an arbitrary limit on the system time value and resets it when the timer approaches the limit. This variant is statically safe in the sense that no runtime errors related to overflow can occur and indeed SPARK does not complain about this code any longer. But it should be clear that such solutions are related to system requirements and cannot be applied by the programmer without regard to user expectations.

Another possible solution would be to leave the timer at its largest value when it is reached (in other words, to *saturate* it):

```
   procedure SysTick_Handler
   is
      Old_Time_Value : Time := System_Time;
   begin
      if Old_Time_Value < Time'Last - SysTick_Period_Millis then
         System_Time := Old_Time_Value + SysTick_Period_Millis;
      end if;
   end SysTick_Handler;
```

This also solves the overflow problem as it simply does not attempt to increase the timer when it *would* overflow, but again - whether this is a valid solution will depend on the particular context.

The two remaining messages about the overflow check relate to these two lines of the main program:

```
         Utils.Delay_Until (Cycle_Begin + Period / 2);
         -- ...
         Cycle_Begin := Cycle_Begin + Period;
```

This is essentially the same problem as with the system timer value - infinite loop that systematically increments some value will surely lead to overflow. We can apply the same arbitrary solutions and above, but here, at the level of complete program, we have enough context to make informed decisions and a good solution for both of these time overflow problems might be to reset the system timer with every iteration of the loop:

```
      Utils.Reset_Clock;
      loop
         Pins.Write (Pins.Pin_12, True);
         Utils.Delay_Until (Period / 2);

         Pins.Write (Pins.Pin_12, False);

         Utils.Delay_Until (Period);
         Utils.Reset_Clock;
      end loop;
```

Now, there are no values that are incremented without any limits and there are no error messages related to failing overflow checks - we can therefore conclude that this final version of the program is free from runtime errors.

Hello World!

Finally, our last example program - Hello World. This program exhibits the same type of problems as out previous system timer example and we can apply analogous solutions. The overflow check failures are reported in these places:

- in package `Utils`, where the system timer value is incremented by the SysTick handler - we can solve this by assuming saturation semantics on the timer value,
- in main procedure `Run`, where the main loop increments value for the next activation time (note that there are two loops that animate the display, both loops have the same problem) - we can solve this by resetting the timer after each loop iteration,
- in package `LCD`, in the helper procedure `Pause`, where the wake-up time is computed by adding constant value to the current time - we can solve this with saturation or by noting that this procedure is always called from the context where its execution is safe.

After these corrections we can declare this program to be free from runtime errors.

Note how different approaches can be used to solve what is essentially the same problem from the SPARK point of view - in all these cases it is the monotonic property of time that causes some values to be always incremented and these increments are not safe when considered in isolation. At the same time, each of them can be solved differently, depending on their context in the whole system. These are important observations and should be kept in mind when working with SPARK.

More From SPARK

In all our example programs with have used SPARK in its very minimal form - we have *only* verified the existing Ada code against the possibility of runtime errors. There is a lot of added value in getting this kind information, as runtime errors are very difficult to track and debug - but even the freedom from runtime errors is still not enough to claim that the program is correct with regard to its *requirements*. We have not checked any functional requirements with this tool.

Interestingly, SPARK has a lot to offer in this area as well. Thanks to the possibility to define *preconditions*, *postconditions* and *loop invariants* for some or all subprograms we can ask the tool to inject additional verification conditions to the set that it generates by default, and have the tool analyse them as well. These additional conditions do not have to describe low-level expectations, for example that the array index should not run out of the array range (this is the domain of runtime error checks), but can instead describe functional requirements about the state of the program, function result values, etc. This is the level of usage where the SPARK language becomes really involving, but also where it can bring great results in terms of program correctness.

Another area where SPARK can help in program verification is with regard to information flow, especially in relation to global objects and external state. We have already used several aspects related to external state, but there are more of them, allowing to define how subprogram are supposed to use the global state, which itself can be described with various degrees of abstraction. If these parts of SPARK are not used, the tool will *synthesise* the necessary information from the source code and use it for further analysis (we could see it for example when SPARK complained about function `Clock` that originally used the system timer volatile variable) - but we might also proactively describe our expectations with regard to information flow in package and subprogram specifications, even before the bodies are written, and let SPARK verify whether the implementations actually comply with what we expect.

Both of these areas are worth to be studied and used in real projects, where their impact on quality will be mostly visible.

Compiling SPARK programs

As a programming language, SPARK is an *extended subset* of Ada. The intent here is to allow SPARK programs to be compiled with regular Ada compilers (also, to interface with regular Ada code), but this intent is not always met in 100%. SPARK's relation to regular Ada relies on the ability of the Ada compiler to handle SPARK-specific aspects. Consider again the package specification from our first program:

```
pragma SPARK_Mode;

package Program
is

   procedure Run
      with No_Return;
   pragma Export (C, Run, "run");

end Program;
```

Two new elements were added to the original Ada program: the pragma `SPARK_Mode` and the aspect `No_Return`. These elements are meaningful to the SPARK tool, but the possibility to compile this program with a regular Ada compiler will depend on how the Ada compiler deals with these two new elements. This has changed in recent compiler versions and currently the following points have to be kept in mind:

- the Ada compilers are expected to ignore pragmas that they do not understand (that is, they are allowed to print some diagnostic message when they find an unknown pragma, but it should not prevent them from compiling the code) and we can assume that all versions of GNAT that are currently in use will behave as we expect,
- newer compilers also know that they should ignore unknown aspects and since SPARK aspects are related to correctness proofs and not to the program's observable behaviour, we can safely compile SPARK programs with newer Ada compilers,
- older compilers do not recognize aspects at all and consider them to be syntax errors - such compilers cannot be used to directly compile SPARK programs that use aspects.

Our example programs can be compiled with the GNAT compiler, version 2014 GPL or higher. Unfortunately, the GNAT FSF compiler that is (at the time of writing this tutorial) part of the Raspbian distribution for the Raspberry Pi cannot be used to directly compile SPARK programs, at least in the way they were written - in particular, our first example program fails at compile-time with these error messages:

```
$ gcc -c -mcpu=cortex-m3 -mthumb program.adb
program.ads:6:17: missing ";"
program.ads:7:07: "with" can only appear in context clause
```

That is, the compiler does not recognize the `with` keyword that introduces the `No_Return` aspect.

One possible strategy to deal with SPARK on older GNAT compilers is to remove (preferably with some automated script) all aspects from the source code. Note that only SPARK-specific annotations should be written as aspects for this to be safe. Consider, for example, one of the objects from the `Registers` package:

```
   TRNG_ISR : Word
      with Async_Writers, Effective_Reads;
   pragma Volatile (TRNG_ISR);
```

This declaration uses both aspects and a pragma - newer compilers (and the SPARK tool) will also accept the following variant:

```
   TRNG_ISR : Word
      with Volatile, Async_Writers, Effective_Reads;
```

The *volatile* property can be expressed either as an aspect or by means of a pragma. If only aspects are used for all these properties, then SPARK will handle them properly, but if these aspects are then removed in order to compile the program with an older Ada compiler, then the volatile property will disappear and it will influence code generation. For this reason, all examples in this tutorial use the rule that SPARK-specific properties are expressed as aspects (they can be removed for compiling with older compilers), whereas properties that have an impact on code generation are expressed as pragmas (which are not removed together with SPARK-specific aspects).

Fortunately, this rule is not necessary with newer versions of GNAT, which can process SPARK sources without any modifications. In particular, as a most complex example, the Hello World! program, after transforming it to SPARK, can be compiled with the same set of commands as the original program:

```
$ gcc -c -gnatp -mcpu=cortex-m3 -mthumb lcd.adb
$ gcc -c -gnatp -mcpu=cortex-m3 -mthumb pins.adb
$ gcc -c -gnatp -mcpu=cortex-m3 -mthumb program.adb
$ gcc -c -gnatp -mcpu=cortex-m3 -mthumb utils.adb
$ ld -T flash.ld -o program.elf lcd.o pins.o program.o utils.o
$ objcopy -O binary program.elf program.bin
```

The `-gnatp` option is now fully justified and can be safely used - the additional checks are not necessary in the executable code as the possibility of their respective runtime errors was excluded by automatic proofs of SPARK's verification conditions.

Previous: Mixing Ada With C and C++, next: Loose Ends.
See also Table of Contents.

Did you find this article interesting? Share it!

SHARE