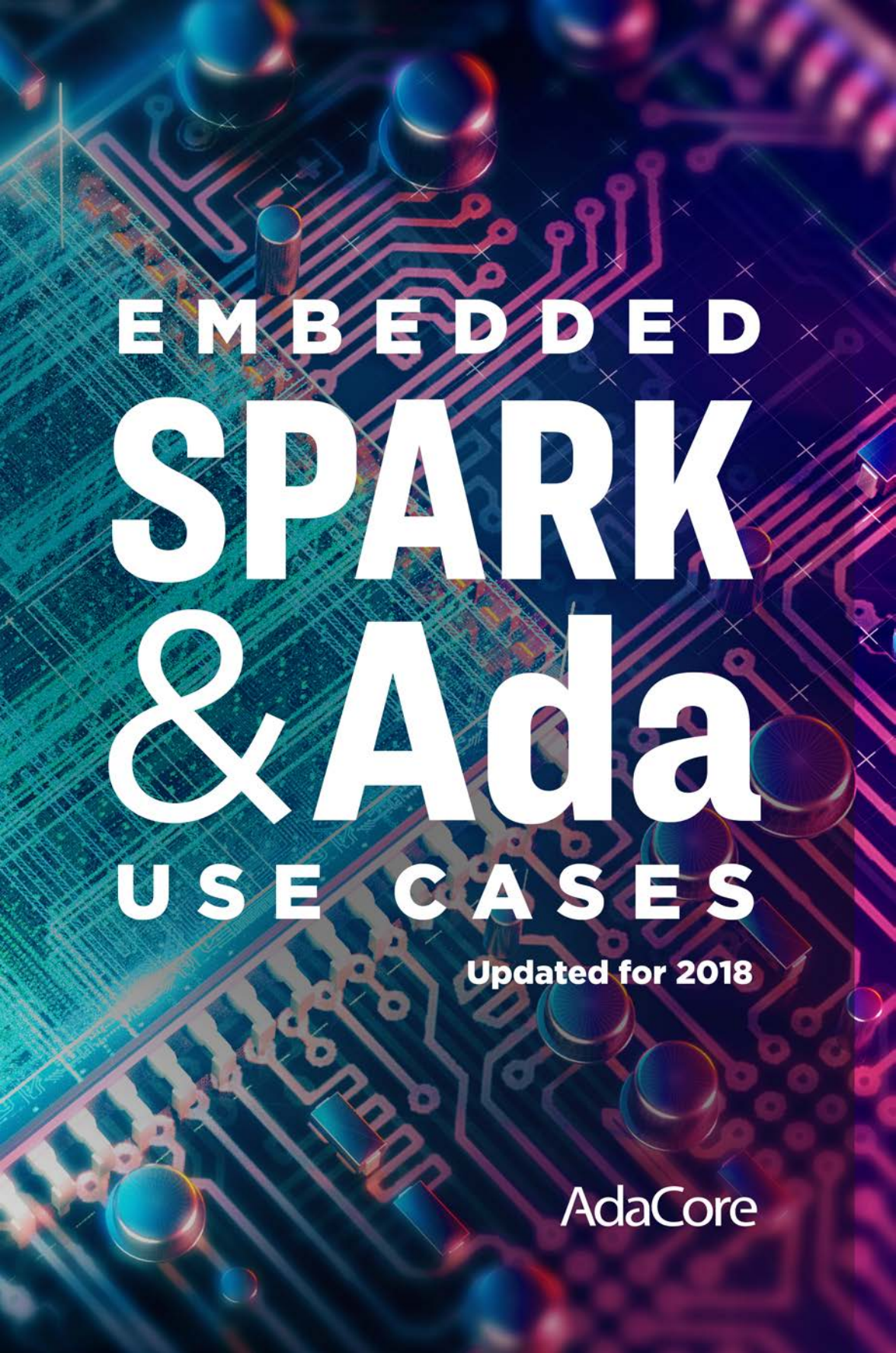

EMBEDDED
SPARK
&Ada
USE CASES
Updated for 2018
AdaCore

EMBEDDED SPARK & Ada USE CASES

Introduction

This booklet is a sampling of AdaCore blogs, including some of our engineers' ARM project creations! They illustrate how embedded system developers can take advantage of Ada's benefits in software reliability, early error detection, code readability, and maintainability while still satisfying performance requirements.

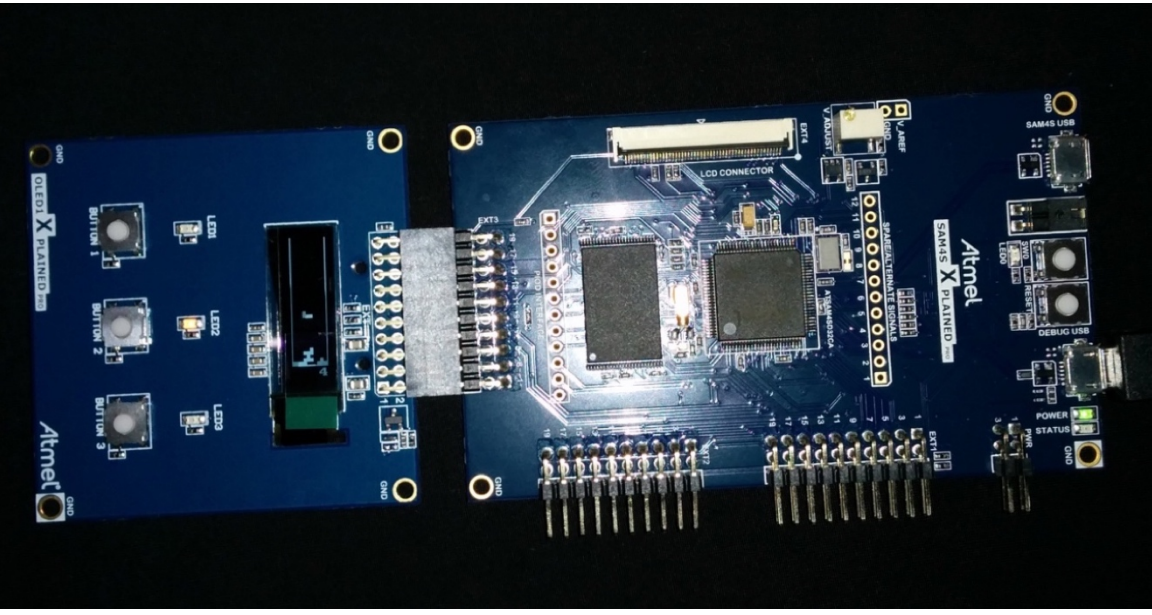
The blogs were written by Raphaël Amiard, Jonas Attertun, Arnaud Charlet, Fabien Chouteau, Tristan Gingold, Anthony Leonardo Gracio, Johannes Kanig, Jérôme Lambourg, Yannick Moy, Jorge Real, J. German Rivera, Pat Rogers and Rob Tice.

For more blogs, visit our AdaCore Blog Page http://blog.adacore.com .

Table of Contents

Tetris in SPARK on ARM Cortex M4

By Tristan Gingold, Yannick Moy
Jan 07, 2015



Tetris is a well-known game from the 80's, which has been ported in many versions to all game platforms since then. There are even versions of Tetris written in Ada. But there was no version of Tetris written in SPARK, so we've repaired that injustice. Also, there was no version of Tetris for the Atmel SAM4S ARM processor, another injustice we've repaired.

The truth is that our colleague Quentin Ochem was looking for a flashy demo for GNAT using SPARK on ARM, to run on the SAM4S Xplained Pro Evaluation Kit of our partner Atmel. Luckily, this kit has an extension with a small rectangular display that made us think immediately of Tetris. Add to that the 5 buttons overall between the main card and the extension, and we had all the necessary hardware.

Now, how do you make a Tetris in SPARK for ARM? First, the ingredients:

- a SAM4S Xplained Pro Evaluation Kit + OLED1 Xplained Pro extension + Atmel manuals
- GNAT GPL 2014 for ARM
- a recent wavefront of SPARK Pro 15.1 (*)
- the WikiPedia page describing Tetris rules
- a webpage describing the Super Rotation System (SRS) for Tetris
- an engineer who knows SPARK
- an engineer who knows ARM

(*) If you don't have access to SPARK Pro 15.1, you can use SPARK GPL 2014, but expect some differences with the verification results presented here.

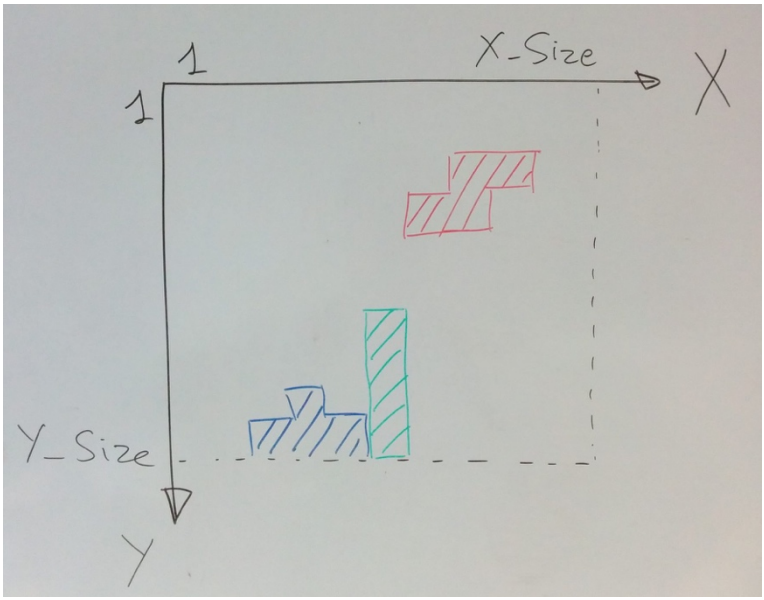
Count 2 days for designing, coding and proving the logic of the game in SPARK, another 2 days for developing the BSP for the board, and 0.5 day for putting it all together. Now the detailed instructions.

The whole sources can be downloaded in the tetris.tgz archive at http://blog.adacore.com/uploads/attachments/tetris.tgz.

A Core Game Logic in SPARK

SPARK is a subset of Ada that can be analyzed very precisely for checking global data usage, data initialization, program integrity and functional correctness. Mostly, it excludes pointers and tasking, which is not a problem for our Tetris game.

We modeled the display (the board) as an array of Y_Size lines, where each line is an array of X_Size cells, and the origin is at the top left corner:



A cell is either empty, or occupied by a piece in I, O, J, L, S, T, Z (the name of the piece hints to what form it takes...), and the piece falling as a record with the following components:

- a shape in I, O, J, L, S, T, Z
- a direction (North, East, South or West) according to SRS rules
- a pair of (X,Y) coordinates in the board, corresponding to the coordinate of the top-left cell of the square box enclosing the piece in SRS

Two global variables Cur_Board and Cur_Piece store the current value of the board and falling piece. Finally, SRS rules are encoded in Boolean matrices defining the masks for oriented shapes (where True stands for an occupied cell, and False for an empty cell).

All the above can be expressed straightforwardly in SPARK as follows:

```
--  possible content of the board cells
type Cell is (Empty, I, O, J, L, S, T, Z);

--  subset of cells that correspond to a shape
subtype Shape is Cell range I .. Z;

--  subset of shapes that fits in a 3 x 3 box, that is, all expect I and O
subtype Three_Shape is Cell range J .. Z;

--  the board is a matrix of X_Size x Y_Size cells, where the origin (1,1)
--  is at the top left corner

X_Size : constant := 10;
Y_Size : constant := 50;
```

```
subtype X_Coord is Integer range 1 .. X_Size;
subtype Y_Coord is Integer range 1 .. Y_Size;

type Line is array (X_Coord) of Cell;
type Board is array (Y_Coord) of Line;

Cur_Board : Board;

--  the current piece has a shape, a direction, and a coordinate for the
--  top left corner of the square box enclosing the piece:
--    a 2 x 2 box for shape O
--    a 3 x 3 box for all shapes except I and O
--    a 4 x 4 box for shape I

subtype PX_Coord is Integer range -1 .. X_Size - 1;
subtype PY_Coord is Integer range -1 .. Y_Size - 1;

type Direction is (North, East, South, West);

type Piece is record
   S : Shape;
   D : Direction;
   X : PX_Coord;
   Y : PY_Coord;
end record;

Cur_Piece : Piece;

--  orientations of shapes are taken from the Super Rotation System at
--  http://tetris.wikia.com/wiki/SRS
--    shape O has no orientation
--    shape I has 4 orientations, which all fit in the 4 x 4 box
--    shapes except I and O have 4 orientations, which all fit in the 3 x 3 box

--  Note that Possible_I_Shapes and Possible_Three_Shapes should be accessed
--  with Y component first, then X component, so that higher value for
--  Direction correspond indeed to clockwise movement.

subtype I_Delta is Integer range 0 .. 3;
type Oriented_I_Shape is array (I_Delta, I_Delta) of Boolean;
subtype Three_Delta is Integer range 0 .. 2;
type Oriented_Three_Shape is array (Three_Delta, Three_Delta) of Boolean;

--  orientations for I

Possible_I_Shapes : constant array (Direction) of Oriented_I_Shape :=
  (((False, False, False, False), (True,  True,  True,  True),  (False, False, False,
False), (False, False, False, False)),
   ((False, False, True,  False), (False, False, True,  False), (False, False, True,
False), (False, False, True,  False)),
   ((False, False, False, False), (False, False, False, False), (True,  True,  True,
True),  (False, False, False, False)),
   ((False, True,  False, False), (False, True,  False, False), (False, True,  False,
False), (False, True,  False, False)));

Possible_Three_Shapes : constant array (Three_Shape, Direction) of
Oriented_Three_Shape :=
  (--  orientations for J
   (((True, False, False), (True,  True,  True),  (False, False, False)),
    ((False, True, True), (False,  True,  False),  (False, True, False)),
    ((False, False, False), (True,  True,  True),  (False, False, True)),
    ((False, True, False), (False,  True,  False),  (True, True, False))),

   --  orientations for L
   (((False, False, True), (True,  True,  True),  (False, False, False)),
    ((False, True, False), (False,  True,  False),  (False, True, True)),
    ((False, False, False), (True,  True,  True),  (True, False, False)),
    ((True, True, False), (False,  True,  False),  (False, True, False))),

   --  orientations for S
```

```
   (((False, True, True), (True,  True,  False),  (False, False, False)),
    ((False, True, False), (False,  True,  True),  (False, False, True)),
    ((False, False, False), (False,  True,  True),  (True, True, False)),
    ((True, False, False), (True,  True,  False),  (False, True, False))),

   --  orientations for T
   (((False, True, False), (True,  True,  True),  (False, False, False)),
    ((False, True, False), (False,  True,  True),  (False, True, False)),
    ((False, False, False), (True,  True,  True),  (False, True, False)),
    ((False, True, False), (True,  True,  False),  (False, True, False))),

   --  orientations for Z
   (((True, True, False), (False,  True,  True),  (False, False, False)),
    ((False, False, True), (False,  True,  True),  (False, True, False)),
    ((False, False, False), (True,  True,  False),  (False, True, True)),
    ((False, True, False), (True,  True,  False),  (True, False, False))));
```

Both the user (by punching buttons) and the main loop of the game (by moving the falling piece down), can perform one of 5 elementary actions, whose names are self explanatory: Move_Left, Move_Right, Move_Down, Turn_Counter_Clockwise, Turn_Clockwise. Dropping the piece is not an elementary action, as it can be obtained by rapidly moving the piece down.

The game logic provides the following API:

- Do_Action: attempts an action and returns whether it was successfully applied or not
- Include_Piece_In_Board: transition from state where a piece is falling to its integration in the board when it cannot fall anymore
- Delete_Complete_Lines: remove all complete lines from the board

Note that all 3 services are implemented as procedures in SPARK, even Do_Action which could be implemented as a function in full Ada, because functions are not allowed to write to global variables in SPARK (that is, functions cannot perform side-effects in SPARK).

There are a few additional functions to return the new value of the falling piece after a move (Move), to know whether a line in the board is empty (Is_Empty_Line) and whether a piece occupies only empty cells of the board (No_Overlap), that is, it fits in the board and does not conflict with already occupied cells.

The complete game logic is only 165 sloc according to GNATmetrics (see tetris_initial.adx in the archive attached). The tool GNATprove in the SPARK toolset can check that this is valid SPARK by using switch –mode=check or equivalently menu SPARK➤Examine File in GPS. See files tetris_initial.ad? in the tetris_core.tgz archive attached.

Note that we use expression functions to implement most small query functions. This allows both to define these functions in a spec file, and to prepare for proof, as the body of these functions acts as an implicit postcondition.

Proving Absence of Run-Time Errors in Tetris Code

Of course, the benefit of writing the core game logic in SPARK is that we can now apply the SPARK analysis tools to demonstrate that the implementation is free from certain classes or errors, and that it complies with its specification.

The most immediate analysis is obtained by running GNATprove with switch –mode=flow or equivalently menu SPARK➤Examine File in GPS. Here, it returns without any message, which means that there are no possible reads of uninitialized data. Note that, as we did not provide data dependencies on subprograms (global variables that are input and output of subprograms), GNATprove generates them from the implementation.

The next step is to add data dependencies or flow dependencies on subprograms, so that GNATprove checks that the implementation of subprograms does not read other global variables than specified (which may not be initialized), does not write other global variables than specified, and derives output values from input values as specified. Here, we settled for specifying only data dependencies, as flow dependencies would not give more information (as all outputs depend on all inputs):

```
procedure Include_Piece_In_Board with
  Global => (Input => Cur_Piece, In_Out => Cur_Board);
--  transition from state where a piece is falling to its integration in the
--  board when it cannot fall anymore.

procedure Delete_Complete_Lines with
  Global => (In_Out => Cur_Board);
--  remove all complete lines from the board
```

Running again GNATprove in flow analysis mode, it returns without any message, which means that procedures Include_Piece_In_Board and Delete_Complete_Lines access global variables as specified. See files tetris_flow.ad? in the tetris_core.tgz archive attached.

The next step if to check whether the program may raise a run-time error. First, we need to specify with preconditions the correct calling

context for subprograms. Here, we only add a precondition to Include_Piece_In_Board to state that the piece should not overlap with the board:

```
procedure Include_Piece_In_Board with
  Global => (Input => Cur_Piece, In_Out => Cur_Board),
  Pre    => No_Overlap (Cur_Board, Cur_Piece);
```

This time, we run GNATprove in proof mode, either with switch –mode=prove or equivalently menu SPARK➤Prove File in GPS. GNATprove returns with 7 messages: 3 possible array accesses out of bounds in procedure Include_Piece_In_Board, 3 possible violations of scalar variable range in function Move, and 1 similar violation in procedure Delete_Complete_Lines.

The message on Delete_Complete_Lines points to a possible range check failure when decrementing variable To_Line. This is a false alarm, as To_Line is decremented at most the number of times the loop is run, which is To_Line - 1. As usual when applying GNATprove to a subprogram with a loop, we must provide some information about the variables modified in the loop in the form of a loop invariant:

```
pragma Loop_Invariant (From_Line < To_Line);
```

With this loop invariant, GNATprove proves there is no possible range check failure.

Although the possible array index messages in Include_Piece_In_Board also occur inside loops, the situation is different here: the indexes used to access array Cur_Board are not modified in the loop, so no loop invariant is needed. Instead, the relevant part of the No_Overlap precondition that is needed to prove each case needs to be asserted as follows:

```
when I =>
      --  intermediate assertion needed for proof
      pragma Assert
        (for all YY in I_Delta =>
           (for all XX in I_Delta =>
              (if Possible_I_Shapes (Cur_Piece.D) (YY, XX) then
                 Is_Empty (Cur_Board, Cur_Piece.Y + YY, Cur_Piece.X + XX))));

      for Y in I_Delta loop
         for X in I_Delta loop
            if Possible_I_Shapes (Cur_Piece.D) (Y, X) then
               Cur_Board (Cur_Piece.Y + Y) (Cur_Piece.X + X) := Cur_Piece.S;
            end if;
```

```
         end loop;
      end loop;

   when Three_Shape =>
      --  intermediate assertion needed for proof
      pragma Assert
        (for all YY in Three_Delta =>
           (for all XX in Three_Delta =>
              (if Possible_Three_Shapes (Cur_Piece.S, Cur_Piece.D) (YY, XX) then
                 Is_Empty (Cur_Board, Cur_Piece.Y + YY, Cur_Piece.X + XX))));

      for Y in Three_Delta loop
         for X in Three_Delta loop
            if Possible_Three_Shapes (Cur_Piece.S, Cur_Piece.D) (Y, X) then
               Cur_Board (Cur_Piece.Y + Y) (Cur_Piece.X + X) := Cur_Piece.S;
            end if;
         end loop;
      end loop;
end case;
```

With these intermediate assertions, GNATprove proves there is no possible array index out of bounds.

The situation is again different in Move, as there is no loop here. In fact, the decrements and increments in Move may indeed raise an exception at run time, if Move is called on a piece that is too close to the borders of the board. We need to prevent such errors by adding a precondition to Move that does not allow such inputs:

```
function Move_Is_Possible (P : Piece; A : Action) return Boolean is
   (case A is
      when Move_Left   => P.X - 1 in PX_Coord,
      when Move_Right  => P.X + 1 in PX_Coord,
      when Move_Down   => P.Y + 1 in PY_Coord,
      when Turn_Action => True);

function Move (P : Piece; A : Action) return Piece is
   (case A is
      when Move_Left   => P'Update (X => P.X - 1),
      when Move_Right  => P'Update (X => P.X + 1),
      when Move_Down   => P'Update (Y => P.Y + 1),
      when Turn_Action => P'Update (D => Turn_Direction (P.D, A)))
with
  Pre => Move_Is_Possible (P, A);
```

With this precondition, GNATprove proves there is no possible range check failure in Move, but it issues a message about a possible precondition failure when calling Move in Do_Action. We have effectively pushed the problem to Move's caller! We need to prevent calling Move in an invalid context by adding a suitable test:

```
procedure Do_Action (A : Action; Success : out Boolean) is
   Candidate : Piece;
begin
   if Move_Is_Possible (Cur_Piece, A) then
      Candidate := Move (Cur_Piece, A);
```

```
      if No_Overlap (Cur_Board, Candidate) then
         Cur_Piece := Candidate;
         Success := True;
         return;
      end if;
   end if;

   Success := False;
end Do_Action;
```

The program is now up to 181 sloc, a relatively modest increase. With this code modification, GNATprove proves in 28s that the integrity of the program is preserved: there are no possible run-time errors, and no precondition violations. See files tetris_integrity.ad? in the tetris_core.tgz archive attached. All timings on GNATprove are given with switches -j0 –prover=cvc4,altergo –timeout=20 on a 2.7 GHz Core i7 with 16 GB RAM.

Proving Functional Properties of Tetris Code

We would like to express and prove that the code of Tetris maintains the board in one of 4 valid states:

- Piece_Falling: a piece is falling, in which case Cur_Piece is set to this piece
- Piece_Blocked: the piece Cur_Piece is blocked by previous fallen pieces in the board Cur_Board
- Board_Before_Clean: the piece has been included in the board, which may contain complete lines that need to be deleted
- Board_After_Clean: complete lines have been deleted from the board

We define a type State that can have these 4 values, and a global variable Cur_State denoting the current state. We can now express the invariant that Tetris code should maintain:

```
function Valid_Configuration return Boolean is
   (case Cur_State is
      when Piece_Falling | Piece_Blocked => No_Overlap (Cur_Board, Cur_Piece),
      when Board_Before_Clean => True,
      when Board_After_Clean => No_Complete_Lines (Cur_Board))
with Ghost;
```

where No_Complete_Lines returns True if there are no complete lines in the board (that is, they have been removed and counted in the player's score):

```
function No_Complete_Lines (B : Board) return Boolean is
   (for all Y in Y_Coord => not Is_Complete_Line (B(Y)))
with Ghost;
```

Note the aspect Ghost on both functions, which indicates that these functions should only be called in assertions and contracts. Ghost code has been introduced for SPARK, but it can also be used independently of formal verification. The idea is that ghost code (it can be a variable, a type, a function, a procedure, etc.) should not influence the behavior of the program, and be used only to verify properties (either dynamically or statically), so the compiler can remove it when compiling without assertions. As you can expect, we also declared type State and variable Cur_State as Ghost:

```
type State is (Piece_Falling, Piece_Blocked, Board_Before_Clean, Board_After_Clean)
with Ghost;

Cur_State : State with Ghost;
```

We add preconditions and postconditions to the API of Tetris, to express that they should maintain this invariant:

```
procedure Do_Action (A : Action; Success : out Boolean) with
  Pre  => Valid_Configuration,
  Post => Valid_Configuration;

procedure Include_Piece_In_Board with
  Global => (Input => Cur_Piece, In_Out => (Cur_State, Cur_Board)),
  Pre    => Cur_State = Piece_Blocked and then
            Valid_Configuration,
  Post   => Cur_State = Board_Before_Clean and then
            Valid_Configuration;
--  transition from state where a piece is falling to its integration in the
--  board when it cannot fall anymore.

procedure Delete_Complete_Lines with
  Global => (Proof_In => Cur_Piece, In_Out => (Cur_State, Cur_Board)),
  Pre    => Cur_State = Board_Before_Clean and then
            Valid_Configuration,
  Post   => Cur_State = Board_After_Clean and then
            Valid_Configuration;
--  remove all complete lines from the board
```

Note the presence of Valid_Configuration in precondition and in postcondition of every procedure. We also specify in which states the procedures should be called and should return. Finally, we add code that performs the state transition in the body of Include_Piece_In_Board:

```
Cur_State := Board_Before_Clean;
```

and Delete_Complete_Lines:

```
Cur_State := Board_After_Clean;
```

Although we're introducing here code to be able to express and prove the property of interest, it is identified as ghost code, so that the GNAT compiler can discard it during compilation when assertions are disabled.

GNATprove proves that Do_Action and Include_Piece_In_Board implement their contract, but it does not prove the postcondition of Delete_Complete_Lines. This is expected, as this subprogram contains two loops whose detailed behavior should be expressed in a loop invariant for GNATprove to complete the proof. The first loop in Delete_Complete_Lines deletes all complete lines, replacing them by empty lines. It also identifies the first such line from the bottom (that is, with the highest value in the Y axis) for the subsequent loop. Therefore, the loop invariant needs to express that none of the lines in the range already treated by the loop is complete:

```
for Del_Line in Y_Coord loop
   if Is_Complete_Line (Cur_Board (Del_Line)) then
      Cur_Board (Del_Line) := Empty_Line;
      Has_Complete_Lines := True;
      To_Line := Del_Line;
      pragma Assert (Cur_Board (Del_Line)(X_Coord'First) = Empty);
   end if;
   pragma Loop_Invariant
     (for all Y in Y_Coord'First .. Del_Line => not Is_Complete_Line (Cur_Board
(Y)));
end loop;
```

The second loop in Delete_Complete_Lines shifts non-empty lines to the bottom of the board, starting from the deleted line identified in the previous loop (the line with the highest value in the Y axis). Therefore, the loop invariant needs to express that this loop maintains the property established in the first loop, that no line in the board is complete:

```
if Has_Complete_Lines then
   for From_Line in reverse Y_Coord'First .. To_Line - 1 loop
      pragma Loop_Invariant (No_Complete_Lines (Cur_Board));
      pragma Loop_Invariant (From_Line < To_Line);
      if not Is_Empty_Line (Cur_Board (From_Line)) then
         Cur_Board (To_Line) := Cur_Board (From_Line);
         Cur_Board (From_Line) := Empty_Line;
```

```
         To_Line := To_Line - 1;
         pragma Assert (Cur_Board (From_Line)(X_Coord'First) = Empty);
      end if;
   end loop;
end if;
```

Note that we added also two intermediate assertions to help the proof. GNATprove now proves the program completely (doing both flow analysis and proof with switch –mode=all) in 50s. See files tetris_functional.ad? in the tetris_core.tgz archive attached.

Obviously, proving that complete lines are deleted is only an example of what could be proved on this program. We could also prove that empty lines are all located at the top of the board, or that non-complete lines are preserved when deleting the complete ones. These and other properties of Tetris are left to you reader as an exercise! Now that the core game logic is developed, we can switch to running this game on the SAM4S Xplained Pro Evaluation Kit.

Developing a Board Support Package (BSP) for the Atmel SAM4S - Startup code

We already had a compiler targeting the ARM line of processors, including the variant present in the SAM4S - the Cortex-M4. But like many modern processors, the Atmel SAM4S is not simply a processor but a SOC (System On Chip). A SOC is a set of peripherals like memory controller, timers or serial controller around a core. Thus we needed to develop a Board Support Package to be able to run our Tetris program on the SAM4S.

In order to run a program, some of the devices must be initialized. The first device to initialize (and it should be initialized early) is the clock controller. Like all integrated circuits, the SAM4S needs a clock. To build cheap applications, the SAM4S provide an internal oscillator that deliver a 4MHz clock. But that clock is neither very precise nor stable, and has a low frequency. So we use an external 12MHz crystal on the main board and setup the internal clock multiplier (a PLL) to deliver a 120MHz clock, which provides much better performance than the default internal clock. This initialization is done very early before the initial setup of the Ada program (a.k.a. the program elaboration) so that the rest of the program is executed at full speed. Implementation-wise, that initialization sequence closely follows the sequence proposed by the Atmel manual, as this is very specific to the clock controller.

The second step is to adjust the number of wait states for accessing the flash memory. The flash memory is the non-volatile storage which contains the application. As accessing the flash memory is slower than the speed at which the processor runs, we need to add delays (also called wait states) for each access to the flash from the processor, and as we increase the speed of the processor we need to increase these delays. If we forget to insert these delays (this is simply a programmable parameter of the flash device), the processor will read incorrect values from the flash memory and crash very early. Fortunately the processor has a cache memory to reduce the number of flash memory accesses and therefore to limit the effect of the wait states.

The third step is to setup the watchdog peripheral. The purpose of a watchdog is to reinitialize the application in case of crashes. To signal that it is running, the application should periodically access the watchdog. Note that by proving that the core application in SPARK is free from run-time errors, we already reduce the number of potential crashes. So we chose here to disable the watchdog.

So what happens when the board is powered up?

The processor starts by executing the program contained in flash memory. The first instructions are assembly code that copy initialized variables (that could be later modified) from flash to RAM. This is done using the default setup and therefore at low speed (but the amount of bytes to copy is very low). Then it increases the number of wait states for the flash. At this time, the performance is very low, the lowest in this application. Then the clock device is programmed, which is a long sequence (the processor has to wait until the clock is stable). This process switches the frequency from 12MHz to 120MHz, thus increasing speed by a factor of 10! The application then disables the watchdog, and now that the processor is running at full speed, the program can start to execute.

Developing a BSP for the Atmel SAM4S - Simple Drivers

So we can execute a program on the board. This is a very good news, but we cannot observe any effect of this program: there is no communication yet with the outside world.

On a standard PC, a program interacts using the keyboard, the screen, the filesystem or the network. On that SAM4S Xplained Pro board, the program will get inputs via buttons, and it will produce visible effects by switching LEDs on and off and displaying patterns on the small OLED screen.

But there is no standard or predefined API for these devices. So we should write simple device drivers, starting with the simplest one: GPIO. GPIO is a common acronym used in the bare board world: General Purpose Input/Output. When configured as output, a GPIO pin can be commanded by a hardware register (in fact a variable mapped at a specific address) to emit current or not. On the board, four of them are interesting because they are connected to a LED. So if a 1 is written in a specific bit of the hardware register, the LED will be on and if a 0 is written it will be off. When configured as input, a GPIO register will reflect the value of a pin. Like for LEDs, four buttons are connected to a GPIO. So the application may read the status of a button from a register. A bit will be set to 1 when the button is pressed and set to 0 if not pressed.

The configuration of a GPIO device is highly device specific and we simply closely follow the vendor manual. First we need to power-up the device and enable its clock:

```
--  Enable clock for GPIO-A and GPIO-C

PMC.PMC_PCER0 := 2 ** PIOA_ID + 2 ** PIOC_ID;
```

Without that the device would be inactive. Technically, the LEDs and the buttons are connected to two different devices, GPIO-A and GPIO-C. Then we need to configure the port. Each bit in a GPIO word correspond to a GPIO pin. We need to configure pins for LEDs as output, and pins for buttons as input. We also need to enable that GPIO lines (there are more settings like pull-up or multi-driver, curious readers shall refer to the SAM4D data sheet):

```
--  Configure LEDs

PIOC.PER := Led_Pin_C + Led1_Pin_C + Led3_Pin_C
  + But2_Pin_C + But3_Pin_C;
PIOC.OER := Led_Pin_C + Led1_Pin_C + Led3_Pin_C;
PIOC.CODR := Led_Pin_C + Led1_Pin_C + Led3_Pin_C;
PIOC.MDDR := Led_Pin_C + Led1_Pin_C + Led3_Pin_C;
PIOC.PUER := But2_Pin_C + But3_Pin_C;
```

The device driver for a GPIO is often very simple. Here is a procedure that sets the state of LED 1:

```
procedure Set_Led1 (On : Boolean) is
begin
   if On then
      PIOC.CODR := Led1_Pin_C;
   else
      PIOC.SODR := Led1_Pin_C;
   end if;
end Set_Led1;
```

CODR means clear output data register; so any bit written as 1 will clear the corresponding bit in the output data register. Respectively, SODR means set output data register; so any bit written as 1 will set the corresponding bit in the output data register and bit written as 0 have no effect. This particular interface (one word to set and one to clear) is common and makes atomic bit manipulation easier.

The function to read the state of a button is also simple:

```
function Button3_Pressed return Boolean is
begin
   return (PIOC.PDSR and But3_Pin_C) = 0;
end Button3_Pressed;
```

We simply need to test a bit of the hardware register. Note that this is a low level driver. The code that calls this function must debounce the button. What does it mean? Real life is somewhat more complex than boolean logic. A button is a simple device with two pins. One is connected to power supply (here +3.0V) and the other is connected to the GPIO pin, but also to the ground via a resistor. When the button is not pressed, the voltage at the GPIO pin is 0V and the program will read 0 in the register. When the button is pressed, the voltage at the GPIO pin is +3.0V (well almost) and the program will read 1 in the register. But when the button is being pressed (this is not instantaneous), the voltage will increase irregularly from 0V to +3V, and due to capacity effects and non-uniformity of the button itself, the bit in the register will oscillate between 0 and 1 during a certain amount of time. If you don't take care, the application may consider a single push on the button as multiple pushes. Debouncing is avoiding that issue, and the simplest way to debounce is not reading the status again before a certain amount of time (like 20ms - but this depends on the button). We take care of that in the main loop of the Tetris game.

Developing a BSP for the Atmel SAM4S - OLED driver

Despite being very simple, the GPIO driver is a first good step. You can write simple applications to blink the LEDs or to switch the LEDs if a user presses a button. But LEDs and buttons are too simple for a Tetris game.

The SAM4S Xplained Pro has an OLED1 extension board. The OLED is a small screen and just behind it, there is a very small chip that controls the screen. This chip, the OLED controller, contains the bitmap that is displayed, refreshes periodically the screen, handles contrast, etc. These tasks are too complex to be performed by the CPU, hence the need for an OLED controller.

But this controller is somewhat complex. It communicates with the SAM4S using a dedicated interface, the SPI bus. SPI stands for Serial Peripheral Interface and as you can guess it uses a few lines (3 in fact) for a serial communication.

Let's have a closer look at the gory details. As you can suppose, we need to talk to the OLED controller whose reference is SSD1306. It accepts many commands, most of them to configure it (number of lines, number of columns, refresh rate) and some to change the bitmap (set the column or modify 8 pixels). All is done via the SPI so we need to first create an SPI driver for the SAM4S. Finally note that if commands can be sent to the OLED controller, that chip is not able to return any status.

The initialization of the SPI isn't particularly complex: we need to power on the device, and to configure the pins. In order to reduce the size of the chip and therefore its price, the SPI pins are multiplexed with GPIO pins. So we need to configure the pins so that they are used for SPI instead of GPIO. Here is an excerpt of the code for that (package Oled, procedure Oled_Configure):

```
--  Enable clock for SPI

PMC.PMC_PCER0 := PMC.PMC_PCER0 + 2 ** SPI_ID;

--  Configure SPI pins

PIOC.PER := Spi_DC_Pin_C + Spi_Reset_Pin_C;
PIOC.OER := Spi_DC_Pin_C + Spi_Reset_Pin_C;
PIOC.CODR := Spi_DC_Pin_C + Spi_Reset_Pin_C;
PIOC.MDDR := Spi_DC_Pin_C + Spi_Reset_Pin_C;
PIOC.PUER := Spi_DC_Pin_C + Spi_Reset_Pin_C;
```

Then there is a sequence to setup the SPI features: baud rate, number of bits per byte... We just have to read the Atmel documentation and set the right bits! (package Oled, procedure Spi_Init):

```
procedure Spi_Init is
   Baudrate : constant := 200; -- 120_000_000 / 5_000_000;
begin
   --  Reset SPI
   SPI.SPI_CR := SPI_CR.SWRST;

   --  Set mode register.
   --  Set master mode, disable mode fault, disable loopback, set chip
   --  select value, set fixed peripheral select, disable select decode.
   SPI.SPI_MR := (SPI.SPI_MR and not (SPI_MR.LLB or SPI_MR.PCS_Mask
                                      or SPI_MR.PS or SPI_MR.PCSDEC
                                      or SPI_MR.DLYBCS_Mask))
     or SPI_MR.MODFDIS or SPI_MR.MSTR or 0 * SPI_MR.DLYBCS;

   --  Set chip select register.
   SPI.SPI_CSR2 := 0 * SPI_CSR.DLYBCT or 0 * SPI_CSR.DLYBS
     or Baudrate * SPI_CSR.SCBR or (8 - 8) * SPI_CSR.BITS
     or SPI_CSR.CSAAT or 0 * SPI_CSR.CPOL or SPI_CSR.NCPHA;

   --  enable
   SPI.SPI_CR := SPI_CR.SPIEN;
end Spi_Init;
```

Now that the SPI interface is on, we need to configure the OLED controller. First action is to reset it so that it is in a known state. This is done via extra GPIO lines, and needs to follow a timing:

```
procedure Oled_Reset
is
   use Ada.Real_Time;
   Now : Time := Clock;
   Period_3us : constant Time_Span := Microseconds (3);
begin
   --  Lower reset
   PIOC.CODR := Spi_Reset_Pin_C;
   Now := Now + Period_3us;
   delay until Now;

   --  Raise reset
   PIOC.SODR := Spi_Reset_Pin_C;
   Now := Now + Period_3us;
   delay until Now;
end Oled_Reset;
```

Then we send the commands to configure the screen. That cannot be guessed and we'd better follow (again !) OLED controller manual:

```
procedure Ssd1306_Init is
begin
   --  1/32 duty
   Ssd1306_Cmd (SSD1306.CMD_SET_MULTIPLEX_RATIO);
   Ssd1306_Cmd (31);

   --  Set ram counter.
```

```
   Ssd1306_Cmd (SSD1306.CMD_SET_DISPLAY_OFFSET);
   Ssd1306_Cmd (0);

...
```

The OLED is now ready to use. The interface for it is rather low-level: we need to select the 'page' (which correspond to the line of the screen when oriented horizontally):

```
--  Set the current page (and clear column)
procedure Oled_Set_Page (Page : Unsigned_8);
```

And set pixels by groups of 8:

```
--  Draw 8 pixels at current position and increase position to the next
--  line.
procedure Oled_Draw (B : Unsigned_8);
```

Putting It All Together With Ravenscar

Most of the low-level work could be done (and was initially done) using the ZFP runtime of GNAT compiler. This runtime is very lightweight and doesn't offer tasking. But for bare boards, we have a better runtime called Ravenscar which provides the well-known Ravenscar subset of tasking features. We have already used it above for implementing delays, and we will also use it for the SPI driver. In the SPI driver, we need to send packet of bytes. We can send a new byte only when the previous one has been sent. With ZFP, we needed to poll until the byte was sent. But with Ravenscar, we can use interrupts and do something else (run another task) until the hardware triggers an interrupt to signal that the byte was sent. A protected type is declared to use interrupts:

```
protected Spi_Prot is
   pragma Interrupt_Priority (System.Interrupt_Priority'First);

   procedure Write_Byte (Cmd : Unsigned_8);

   procedure Interrupt;
   pragma Attach_Handler (Interrupt, Ada.Interrupts.Names.SPI_Interrupt);

   entry Wait_Tx;
private
   Tx_Done : Boolean := False;
end Spi_Prot;
```

The protected procedure 'Interrupt' is attached to the hardware interrupt via the pragma Attach_Handler, and the protected object is

assigned to the highest priority via the pragma Interrupt_Priority (so that all interrupts are masked within the protected object to achieve exclusive access). The procedure Ssd1306_Write sends a command using the interrupt. It first programs the SPI device to send a byte and then waits for the interrupt:

```
procedure Ssd1306_Write (Cmd : Unsigned_8) is
begin
   Spi_Prot.Write_Byte (Cmd);

   Spi_Prot.Wait_Tx;
end Ssd1306_Write;
```

Writing a byte mostly consists of writing a hardware register (don't look at the PCS issue, that is an SPI low-level detail) and enabling SPI interrupts:

```
procedure Write_Byte (Cmd : Unsigned_8) is
begin
   --  Set PCS #2
   SPI.SPI_MR := (SPI.SPI_MR and not SPI_MR.PCS_Mask)
     or (SPI_MR.PCS_Mask and not ((2 ** 2) * SPI_MR.PCS));

   --  Write cmd
   SPI.SPI_TDR := Word (Cmd) * SPI_TDR.TD;

   --  Enable TXEMPTY interrupt.
   SPI.SPI_IER := SPI_SR.TXEMPTY;
end Write_Byte;
```

When the interrupt is triggered by the SPI device, the Interrupt procedure is called. It acknowledges the interrupt (by disabling it) and opens the entry (the Ssd1306_Write procedure is waiting on that entry):

```
procedure Interrupt is
begin
   if (SPI.SPI_SR and SPI_SR.TXEMPTY) /= 0 then
      --  Disable TXEMPTY interrupt
      SPI.SPI_IDR := SPI_SR.TXEMPTY;

      --  Set the barrier to True to open the entry
      Tx_Done := True;
   end if;
end Interrupt;
```

In the entry, we simply close the entry guard and clean the SPI state:

```
entry Wait_Tx when Tx_Done is
begin
   --  Set the barrier to False.
   Tx_Done := False;
```

```
   --  Deselect device
   SPI.SPI_MR := SPI.SPI_MR or SPI_MR.PCS_Mask;

   --  Last transfer
   SPI.SPI_CR := SPI_CR.LASTXFER;
end Wait_Tx;
```

Conclusion

Overall, it was one week effort to produce this demo of SPARK on ARM. The code is available in the tetris.tgz archive attached if you're interested in experimenting with formal verification using SPARK or development in Ada for ARM. The source code is bundled with the complete development and verification environment for Linux and Windows in the archive at http://adaco.re/85.

To know more about SPARK:

- The SPARK User's Guide
  http://docs.adacore.com/spark2014-docs/html/ug/index.html
- SPARK Pro page
  www.adacore.com/sparkpro

To know more about GNAT for ARM:

- GNAT Pro for ARM page
  http://www.adacore.com/gnatpro-safety-critical/
- GNAT GPL for ARM page
  http://libre.adacore.com/tools/gnat-gpl-for-bare-board-arm

Some other people blogs show projects using GNAT on ARM.

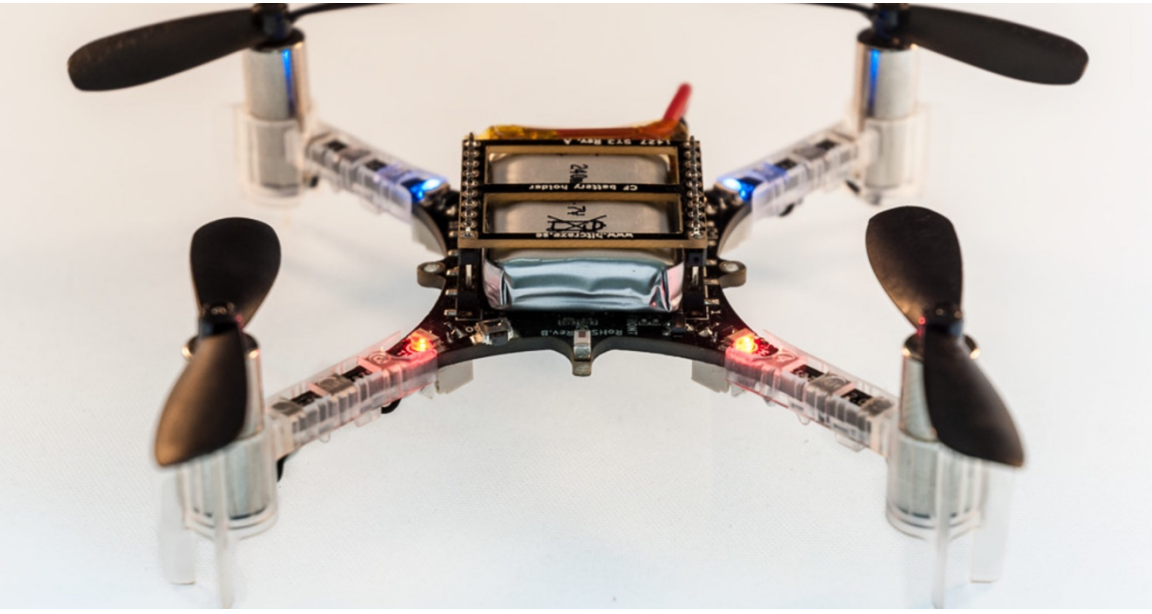
- Mike Silva has great tutorials:
  - http://www.embeddedrelated.com/showarticle/585.php
  - http://www.embeddedrelated.com/showarticle/617.php
  - http://www.embeddedrelated.com/showarticle/625.php

- Other very good tutorials by Jack Ganssle, Bill Wong and Jerry Petrey:
    - http://www.ganssle.com/video/episode9-ada-on-a-microcontroller.html
    - http://electronicdesign.com/blog/arming-ada
    - http://demo.electronicdesign.com/blog/running-ada-2012-cortex-m4
    - http://electronicdesign.com/dev-tools/armed-and-ready

This chapter was originally published at
https://blog.adacore.com/tetris-in-spark-on-arm-cortex-m4

How to Prevent Drone Crashes Using SPARK

by Anthony Leonardo Gracio
May 28, 2015



Introduction

I recently joined AdaCore as a Software Engineer intern. The subject of this internship is to rewrite a drone firmware written in C into SPARK.

Some of you may be drone addicts and already know the Crazyflie, a tiny drone whose first version has been released by Bitcraze company in 2013. But for all of those who don't know anything about this project, or about drones in general, let's do a brief presentation.

The Crazyflie is a very small quadcopter sold as an open source development platform: both electronic schematics and source code are directly available on their GitHub and its architecture is very flexible. These two particularities allow the owner to add new features in an easy way. Moreover, a wiki and a forum have been made for this

purpose, making emerge a little but very enthusiastic Crazyflie community!

Now that we know a little more about the Crazyflie, let me do a brief presentation of SPARK and show you the advantages of using it for drone-related software.

Even if the Crazyflie flies out of the box, it has not been developed with safety in mind: in case of crash, its size, its weight and its plastic propellers won't hurt anyone!

But what if the propellers were made of carbon fiber, and shaped like razor blades to increase the drone's performance? In theses circumstances, a bug in the flight control system could lead to dramatic events.

SPARK is an Ada subset used for high reliability software. SPARK allows proving absence of runtime errors (overflows, reading of uninitialized variables...) and specification compliance of your program by using functional contracts.

The advantages of SPARK for drone-related software are obvious: by using SPARK, you can ensure that no runtime errors can occur when the drone is flying. Then, if the drone crashes, you can only blame the pilot!

After this little overview, let's see how we can use SPARK on this kind of code.

Interfacing SPARK with C

Being an Ada subset, SPARK comes with the same facilities as Ada when it comes to interfacing it with other languages. This allowed me to focus on the most error-sensitive code (ex: stabilization system code), prove it in SPARK, let the less sensitive or proven-by-use code in C (ex: drivers), and mix SPARK code with the C one to produce the final executable.

Let's see how it works. The Crazyflie needs to receive the commands given by the pilot. These commands are retrieved from a controller (Xbox, PS3, or via the Android/iOS app) and are sent via Wi-Fi to the Crazyflie. This code is not related with the stabilization system and works great: for now, we just want to call this C code from our stabilization system code written in SPARK.

Here is the C procedure that interests us. It retrieves the desired angles, given by the pilot via his controller, for each axis (Roll, Pitch and Yaw).

```
void commanderGetRPY
   (float* eulerRollDesired,
    float* eulerPitchDesired,
    float* eulerYawDesired);
```

And here is the Ada procedure declaration that imports this C function.

```
procedure Commander_Get_RPY_Wrapper
   (Euler_Roll_Desired  : in out Float;
    Euler_Pitch_Desired : in out Float;
    Euler_Yaw_Desired   : in out Float);
pragma Import (C, Commander_Get_RPY_Wrapper, "commanderGetRPY");
```

Now we can use this function to get the commands and give them as inputs to our SPARK stabilization system!

Helping SPARK: constrain your types and subtypes!

Ada is well known for its features concerning types, which allow the programmer to define ranges over discrete or floating-point types. This specificity of Ada is very useful when it comes to prove absence of overflow and constraint errors using SPARK: indeed, when all the values used in the program's computations are known to be in a certain range, it becomes easy to determine if these computations will cause a runtime error!

Let's see how it works in practice. The Crazyflie comes with a PC client used to control and track the drone's movements. A lot of physical values can be seen in real-time via the PC client, including the drone's acceleration magnitude.

To calculate this magnitude, we use the accelerometer measurements given by the IMU (Inertial Measurement Unit) soldered on the Crazyflie.

Let's see how the magnitude was calculated in the original C code. The accelerometer measurements are hold in a structure containing 3 float fields, one for each axis:

```
typedef struct {
```

```
   float x;
   float y;
   float z;
} Axis3f;

static Axis3f acc;
```

In the stabilization loop, we get the fresh accelerometer, gyro and magnetometer measurements by calling a function from the IMU driver. Basically, this function simply reads the values from the IMU chip and filters the possible hardware errors:

```
imu9Read(gyro, acc, mag);
```

Now that we have the current accelerometer measurements, let's calculate its magnitude:

```
accMAG = (acc.x*acc.x) + (acc.y*acc.y) + (acc.z*acc.z);
```

The type of each 'acc' field is a simple C 'float'. This means that, for instance, 'acc.x' can possibly be equal to FLT_MAX, causing an obvious overflow at runtime... Without knowing anything about the return values of imu9Read, we can't prove that no overflow can occur here. In SPARK, you can easily prove this type of computations by constraining your ADA types/subtypes. For this case, we can create a float subtype 'T_Acc' for the IMU acceleration measurements. Looking in the Crazyflie IMU documentation, we discover that the IMU accelerometer measurements are included in [-16, 16], in G:

```
--  Type for acceleration output from accelerometer, in G
subtype T_Acc  is Float range -16.0 .. 16.0;

type Accelerometer_Data is record
   X : T_Acc;
   Y : T_Acc;
   Z : T_Acc;
end record;
```

Now that we have constrained ranges for our accelerometer measurements, the acceleration magnitude computation code is easily provable by SPARK!

Constrained subtypes can also be useful for cascaded calculations (i.e: when the result of a calculation is an operand for the next calculation). Indeed, SPARK checks each operand type's range in priority in order to

prove that there is no overflow or constraint error over a calculation. Thus, giving a constrained subtype (even if the subtype has no particular meaning!) for each variable storing an intermediate result facilitates the proof.

Ensuring absence of constraint errors using saturation

We have seen that defining constrained types and subtypes helps a lot when it comes to prove that no overflow can occur over calculations. But this technique can lead to difficulties for proving the absence of constraint errors. By using saturation, we can ensure that the result of some calculation will not be outside of the variable type range.

In my code, saturation is used for two distinct cases:

- Independently from SPARK, when we want to ensure that a particular value stays in a semantically correct range
- Directly related to SPARK, due to its current limitations regarding floating-point types

Let's see a code example for each case and explain how does it help SPARK. For instance, motor commands can't exceed a certain value: beyond this limit, the motors can be damaged. The problem is that the motor commands are deduced from the cascaded PID system and other calculations, making it difficult to ensure that these commands stay in a reasonable range. Using saturation here ensures that motors won't be damaged and helps SPARK to prove that the motor commands will fit in their destination variable range.

First, we need the function that saturates the motor power. Here is its defintion: it takes a signed 32-bit integer as input and retrieves an unsigned 16-bit integer to fit with the motors drivers.

```
--  Limit the given thrust to the maximum thrust supported by the motors.
   function Limit_Thrust (Value : T_Int32) return T_Uint16 is
      Res : T_Uint16;
   begin
      if Value > T_Int32 (T_Uint16'Last) then
         Res := T_Uint16'Last;
      elsif Value < 0 then
         Res := 0;
      else
         pragma Assert (Value <= T_Int32 (T_Uint16'Last));
```

```
         Res := T_Uint16 (Value);
      end if;

      return Res;
   end Limit_Thrust;
```

Then, we use it in the calculations to get the power for each motor, which is deduced from the PID outputs for each angle (Pitch, Roll and Yaw) and the thrust given by the pilot:

```
procedure Stabilizer_Distribute_Power
     (Thrust : T_Uint16;
      Roll   : T_Int16;
      Pitch  : T_Int16;
      Yaw    : T_Int16)
   is
      T : T_Int32 := T_Int32 (Thrust);
      R : T_Int32 := T_Int32 (Roll);
      P : T_Int32 := T_Int32 (Pitch);
      Y : T_Int32 := T_Int32 (Yaw);
   begin
       R := R / 2;
       P := P / 2;

      Motor_Power_M1 := Limit_Thrust (T - R + P + Y);
      Motor_Power_M2 := Limit_Thrust (T - R - P - Y);
      Motor_Power_M3 := Limit_Thrust (T + R - P + Y);
      Motor_Power_M4 := Limit_Thrust (T + R + P - Y);

      Motor_Set_Ratio (MOTOR_M1, Motor_Power_M1);
      Motor_Set_Ratio (MOTOR_M2, Motor_Power_M2);
      Motor_Set_Ratio (MOTOR_M3, Motor_Power_M3);
      Motor_Set_Ratio (MOTOR_M4, Motor_Power_M4);
  end Stabilizer_Distribute_Power;
```

That's all! We can see that saturation here, in addition to ensure that the motor power is not too high for the motors, ensures also that the result of these calculations will fit in the 'Motor_Power_MX' variables.

Let's switch to the other case now. We want to log the drone's altitude so that the the pilot can have a better feedback. To get the altitude, we use a barometer which isn't very precise. To avoid big differences between two samplings, we make a centroid calculation between the previous calculated altitude and the raw one given by the barometer. Here is the code:

```
      subtype T_Altitude is Float range -8000.0 .. 8000.0;  -- Deduced from the
barometer documentation

   --  Saturate a Float value within a given range
     function Saturate
        (Value     : Float;
         Min_Value : Float;
         Max_Value : Float) return Float
     is
     (if Value < Min_Value then
```

```
         Min_Value
      elsif Value > Max_Value then
         Max_Value
      else
         Value);
     pragma Inline (Saturate);

      --  Other stuff...

      Asl_Alpha            : T_Alpha       := 0.92;  --  Short term smoothing
      Asl                  : T_Altitude    := 0.0;
      Asl_Raw              : T_Altitude    := 0.0;

      --  Other stuff...

      --  Get barometer altitude estimations
      LPS25h_Get_Data (Pressure, Temperature, Asl_Raw, LPS25H_Data_Valid);

      if LPS25H_Data_Valid then
         Asl := Saturate
                     (Value          => Asl * Asl_Alpha + Asl_Raw * (1.0 -
Asl_Alpha),
                      Min_Value  => T_Altitude'First,
                      Max_Value =>T_Altitude'Last);
      end if;
```

Theoretically, we don't need this saturation here: the two involved variables are in T_Altitude'Range by definition, and Asl_Alpha is strictly inferior to one. Mathematically, the calculation is sure to fit in T_Altitude'Range. But SPARK has difficulties to prove this type of calculations over floating-point types. That's why we can help SPARK using saturation: by using the 'Saturate' expression function, SPARK knows exactly what will be the 'Saturate' result range, making it able to prove that this result will fit in the 'Asl' destination variable.

Using State Abstraction to improve the code readability

The stabilization system code of the Crazyflie contains a lot of global variables: IMU outputs, desired angles given by the pilot for each axis, altitude, vertical speed... These variables are declared as global so that the log subsystem can easily access them and give a proper feedback to the pilot.

The high number of global variables used for stabilization lead to never-ending and unreadable contracts when specifying data dependencies for SPARK. As a reminder, data dependencies are used to specify what global variables a subprogram can be read and/or written. A simple solution for this problem is to use state abstraction: state abstraction allows the developer to map a group of global variables to an 'abstract state', a symbol that can be accessed from the package where it was created but also by other packages.

Here is a procedure declaration specifying data dependencies without the use of state abstraction:

```
--  Update the Attitude PIDs
   procedure Stabilizer_Update_Attitude
     with
       Global => (Input  => (Euler_Roll_Desired,
                             Euler_Pitch_Desired,
                             Euler_Yaw_Desired,
                             Gyro,
                             Acc,
                             V_Acc_Deadband,
                             V_Speed_Limit),
                  Output => (Euler_Roll_Actual,
                             Euler_Pitch_Actual,
                             Euler_Yaw_Actual,
                             Roll_Rate_Desired,
                             Pitch_Rate_Desired,
                             Yaw_Rate_Desired,
                             Acc_WZ,
                             Acc_MAG),
                  In_Out => (V_Speed,
                             Attitude_PIDs));
```

We can see that these variables can be grouped. For instance, the 'Euler_Roll_Desired', 'Euler_Pitch_Desired' and 'Euler_Yaw_Desired' refer to the desired angles given by the pilot: we can create an abstract state Desired_Angles to refer them in our contracts in a more readable way. Applying the same reasoning for the other variables referenced in the contract, we can now have a much more concise specification for our data dependencies:

```
   procedure Stabilizer_Update_Attitude
     with
       Global => (Input  => (Desired_Angles,
                             IMU_Outputs,
                             V_Speed_Parameters),
                  Output => (Actual_Angles,
                             Desired_Rates),
                  In_Out => (SensFusion6_State,
                             V_Speed_Variables,
                             Attitude_PIDs));
```

Combining generics and constrained types/subtypes

SPARK deals very well with Ada generics. Combining it with constrained types and subtypes can be very useful to prove absence of runtime errors on general algorithms that can have any kind of inputs.

Like many control systems, the stabilization system of the Crazyflie uses a cascaded PID control, involving two kinds of PID:

- Attitude PIDs, using desired angles as inputs and outputting a desired rate
- Rate PIDs, using attitude PIDs outputs as inputs

In the original C code, the C base float type was used to represent both angles and rates and the PID related code was also implemented using floats, allowing the developer to give angles or rates as inputs to the PID algorithm.

In SPARK, things are more complicated: PID functions do some calculations over the inputs, like calculating the error between the measured angle and the desired one. We've seen that, without any information about input ranges, it's very difficult to prove the absence of runtime errors on calculations, even over a basic one like an error calculation (Error = Desired – Measured). In other words, we can't implement a general PID controller using the Ada Float type if we intend to prove it with SPARK.

The good practice here is to use Ada generics: by creating a generic PID package using input, output and PID coefficient ranges as parameters, SPARK will analyze each instance of the package with all information needed to prove the calculations done inside the PID functions.

Conclusion

Thanks to SPARK 2014 and these tips and tricks, the Crazyflie stabilization system is proved to be free of runtime errors! SPARK also helped me to discover little bugs in the original firmware, one of which directly related with overflows. It has been corrected by the Bitcraze team with this commit since then.

All the source code can be found on my GitHub.

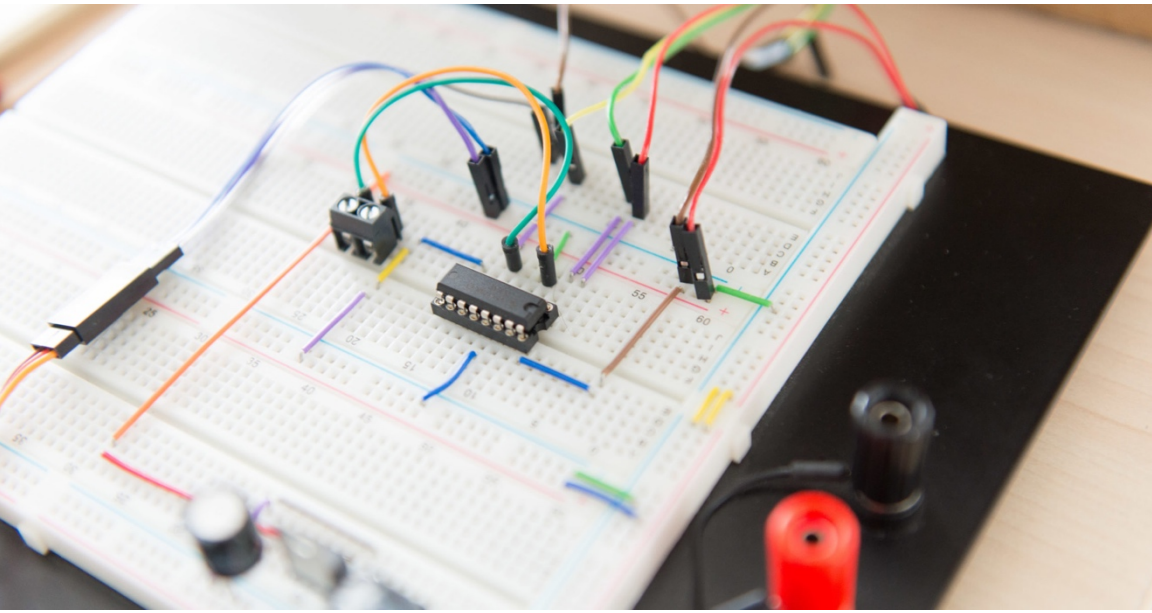
Since Ada allows an easy integration with C, the Crazyflie currently flies with its rewritten stabilization system in SPARK on top of FreeRTOS! The next step for my internship is to rewrite the whole firmware in Ada, by removing the FreeRTOS dependencies and use Ravenscar instead. All the drivers will be rewritten in Ada too.

I will certainly write a blog post about it at blog.adacore.com

This chapter was originally published at
https://blog.adacore.com/how-to-prevent-drone-crashes-using-spark

Make with Ada: All that is Useless is Essential

by Fabien Chouteau
June 19, 2015



Solenoid Engine - Part 1

A few weeks ago I discovered the wonderful world of solenoid engines. The idea is simple: take a piston engine and replace explosion with electromagnetic field. The best example being the incredible V12 of David Robert.

Efficient? Probably not. Fun? Definitely!

All the engines I found on YouTube use a mechanical switch to energize the coil at the right moment. On Robert's V12, it is the blue cylinder. This is very similar to the intake camshaft of a piston engine.

While watching the video, I thought that if we replace the mechanical switch with an electronic detection of the rotor's position, combined with a software-controlled solenoid ignition, we could:

1. remove a mechanism that is not very reliable
2. fine tune the ignition of the solenoids (when and how long it is energized) to control the motor's speed and optimize energy consumption

I will try to experiment that solution in this article.

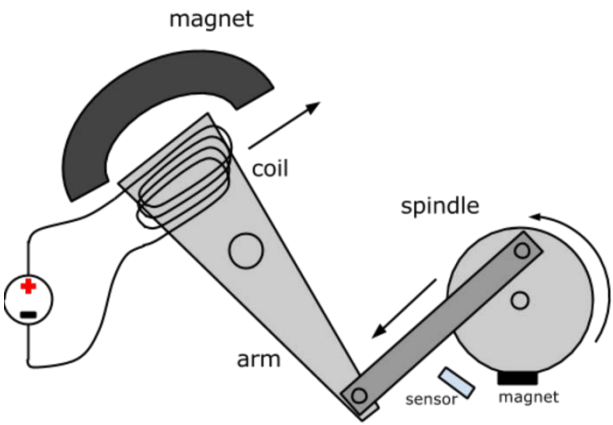
Hardware

Unfortunately, I do not have the tools nor the knowhow of David Robert. So, inspired by a lot of projects on YouTube, I took the hacker path and used an old hard drive to build my motor.

The hard drive has two mechanical elements:

1. A brushless motor spinning the disk(s) (the spindle)
2. An oscillating arm holding the read/write heads on one side and a coil surrounded by strong magnets on the other side. The arm is mounted on a pivot and moves when the coil is energized.

If I add a connecting rod from the oscillating arm to an off-center point on the spindle, I can mimic the mechanism of a solenoid engine.

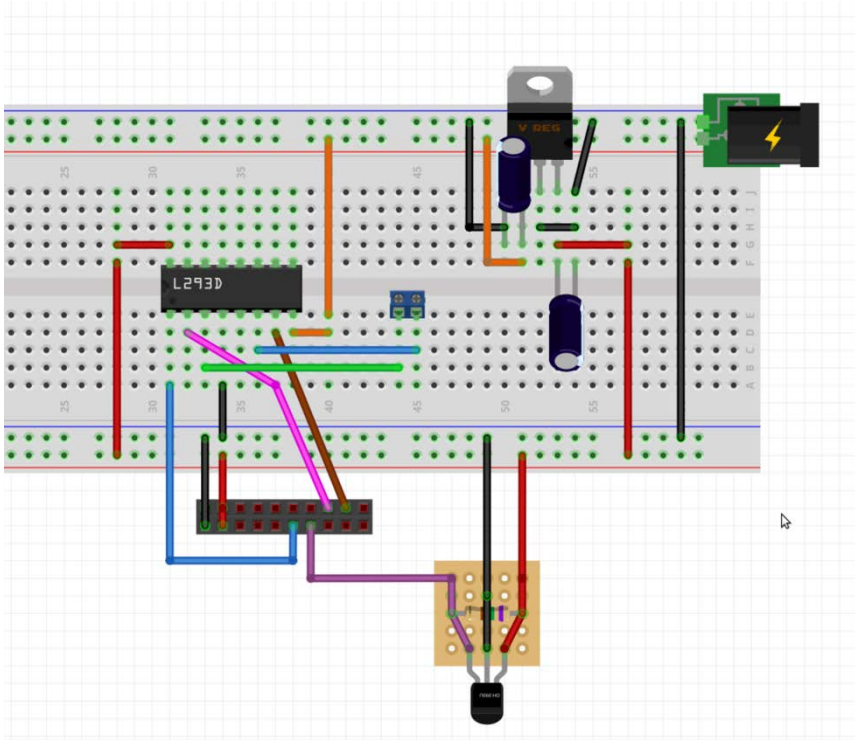


To detect the spindle's position, I use a hall effect sensor combined with a small magnet glued to the spindle. The sensor and magnet are set to detect when the spindle is at the top dead center (TDC http://en.wikipedia.org/wiki/Dead_centre).

To control the solenoid, I use one of the integrated H-bridges of a L293D. The advantage of an H-bridge is to be able to energize the coil in two directions, which means I can have two power strokes (push and pull on the spindle).

The sensor and the L293D are connected to an STM32F4-discovery board that will run the software.

Here is the schematic on Fritzing : https://github.com/Fabien-Chouteau/solenoid-engine-controller/blob/master/schematics/HDD_solenoid_engine.fzz



Software

The first version of the control software will be somewhat naive. Foremost, I want to check that I can control the engine with the STM32F4. I will explore advanced features later.

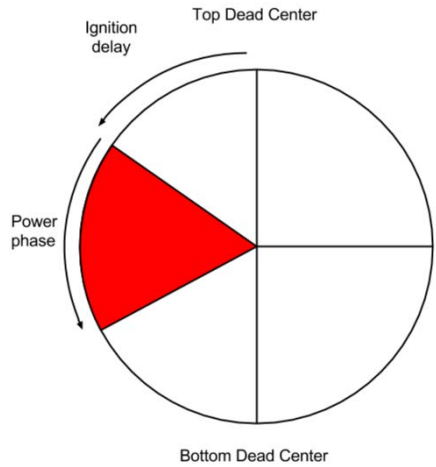
The hall effect sensor will be connected to an interrupt. The interrupt will be triggered when the spindle is at TDC. By measuring the time between two interrupts, I can compute the speed of the spindle.

```
--  What time is it?
Now := Clock;

--  Time since the last interrupt
Elapsed := To_Duration (Now - Last_Trigger);

--  Store trigger time for the next interrupt
Last_Trigger := Now;

--  Compute number of revolutions per minute
RPM := 60.0 / Float (Elapsed);
```

Then I have to compute the best moment to energize the coil (ignition) and how long it should be energized (power phase).

Intuitively, the coil is most efficient 90 degrees after top dead center (ATDC) which means the power phase should be fairly distributed around that point.

For the moment, we arbitrarily decide that the power phase should be 50% of the TDC to BDC time.




```
--  How much time will the engine take to go from Top Dead Center
--  to Bottom Dead Center (half of a turn) based on how much time
--  it took to make the last complete rotation.
TDC_To_BDC := Elapsed / 2.0;

--  We start energizing at 25% of the TDC to BDC time
Ignition    :=  TDC_To_BDC * 0.25;

--  We energize the coil during 50% of the TDC to BDC time
Power_Phase := TDC_To_BDC * 0.5;

--  Convert to start and stop time
Start := Now + Milliseconds (Natural (1000.0 * Ignition));
Stop := Start + Milliseconds (Natural (1000.0 * Power_Phase));
```

To deliver the power command, we will use timing events (the implementation is inspired by the Gem #4 — https://www.adacore.com/gems/gem-4/). Of course, a more advanced version should use the hardware timers available in the STM32F4 to reduce CPU usage. However, the engine will not exceed

3000 RPM, that's 50 events per second and therefore not a big deal for the 168MHz micro-controller.

You can find the sources on GitHub: https://github.com/Fabien-Chouteau/solenoid-engine-controller

That's it for the software,let's compile, program the board, plug in everything, give it a small push and see what happens...

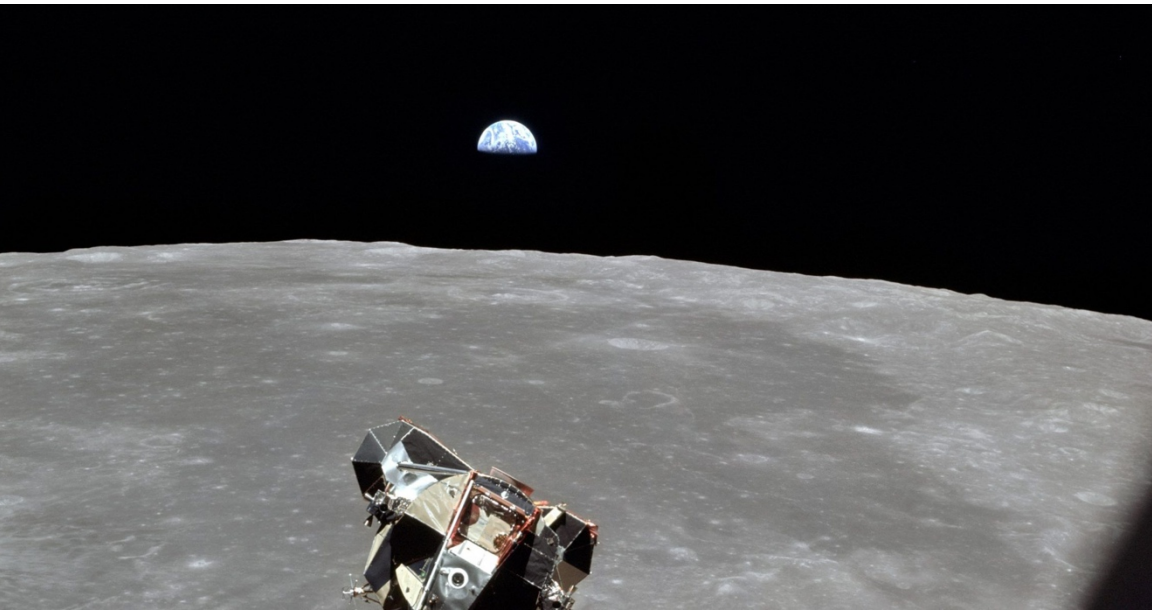
https://youtu.be/V4Et5AvYgXc

In the next part, I will use the screen on the STM32F429 Discovery board to control the ignition and power_phase parameters, we will see how this changes the behavior of the engine.

This chapter was originally published at https://blog.adacore.com/make-with-ada-all-that-is-useless-is-essential

Make with Ada: "The Eagle has landed"

by Fabien Chouteau
July 20, 2015



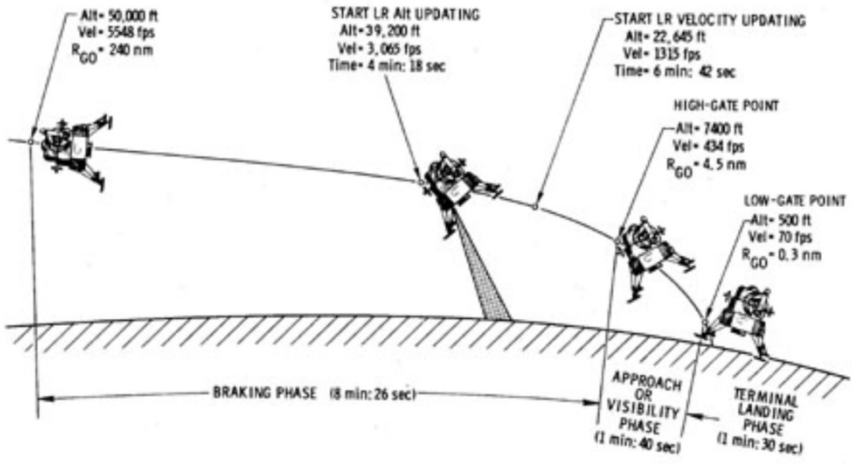
July 20, 1969, 8:18 p.m. UTC, while a bunch of guys were about to turn blue on Earth, commander Neil A. Armstrong confirms the landing of his Lunar Module (LM), code name Eagle, on the moon.

Even though the first footstep on the moon will certainly remain the most memorable moment of the Apollo 11, landing a manned spacecraft on the moon was probably the most challenging part of the mission.

To celebrate the 46th anniversary of this extraordinary adventure, I decided to challenge you. Will you be able to manually land Eagle on the Sea of Tranquillity?

In this article I will present my lunar lander simulator. It is a 2D simulation of the basic physics of the landing using GTKAda and Cairo for the graphic front-end.

The simulation starts at High-Gate point, altitude 7,500 ft (2.2 km). It is the beginning of the visibility phase, i.e. the astronauts start to see the ground and the target landing site.



You have two controls to maneuver the Lunar Module:

- Descent Propulsion System (DPS): It is the main engine of the LM. The DPS is on full throttle since the beginning of the landing (about 8 min. before High-Gate). It can only be throttled between 10% and 60%, otherwise it is either off or full throttle.
- Reaction Control System (RCS): Composed of four pods of four small thrusters that provide attitude control. In the simulation you can control the throttle of opposite RCS to rotate the LM.

In the simulator, you have the raw control of throttle for DPS and RCS, the real Apollo Lunar Module was heavily relying on software, the commander would only take manual control (still software assisted) for the very last approach where the Apollo Guidance Computer (AGC) could not evaluate the features of the landing site. For instance, Neil Armstrong had to fly the LM over a boulder field to find a suitable landing area.

Physics engine

In the physics engine I use GNAT’s dimensionality checking system. It is a static verification of dimensional correctness of the formulas. For instance if I multiply a speed by a time, GNAT will ensure that I can only put this data in a distance variable. Same goes for magnetic flux, electrical current, pressure, etc.

So in the simulator, all the calculi from net forces, acceleration, speed to position are statically checked for physical unit error. In fact, thanks to this system, I realized that I forgot the moment of inertia in my simulation.

You can learn more about GNAT dimensionality checking in Gem #136 — https://www.adacore.com/gems/gem-136-how-tall-is-a-kilogram/

Panels

On the screen you will see the representation of some of the panels of the LM's cockpit. All the panels can be moved with a left mouse click, and resized with right mouse click.

- Gauges: Percentage of remaining fuel for DPS and RCS
- Attitude: Pitch angle and pitch rate
- T/W: Thrust to weight ratio
- Alt / Alt Rate: Altitude from lunar surface and altitude rate
- X-Pointer: In the real LM the X-pointer displays forward and lateral speed. Since this is a 2D simulator, I chose to display the forward speed on the X axis and vertical speed on Y axis.

Optional help

Manually landing Eagle can be tricky, to help you I added three features:

- Speed vector direction: The speed vector direction is shown by a green line starting at the center of the LM, while the red line represents the LM's orientation. If you manage to keep those two lines close to each other, you are not far from a good landing.
- Forecast: It is a stream of blue dots showing the future positions of the LM if you do not touch the controls.
- Overall situation panel: It is the big white panel. It shows the current position in blue, the target landing site in red and the trajectory since beginning of simulation in green. (Of course, this panel was not part of the real LM cockpit...)

You can disable those two help features by clicking on the "Help" button on the top-right of the screen.

Source code

The code is available on GitHub (https://github.com/Fabien-Chouteau/eagle-lander). You can compile it with GNAT Community on Windows and Linux (https://www.adacore.com/download)

Video

To give you a preview or if you don’t bother compiling the simulator, here is a video of a landing: https://youtu.be/9JksZqToFn4

Links

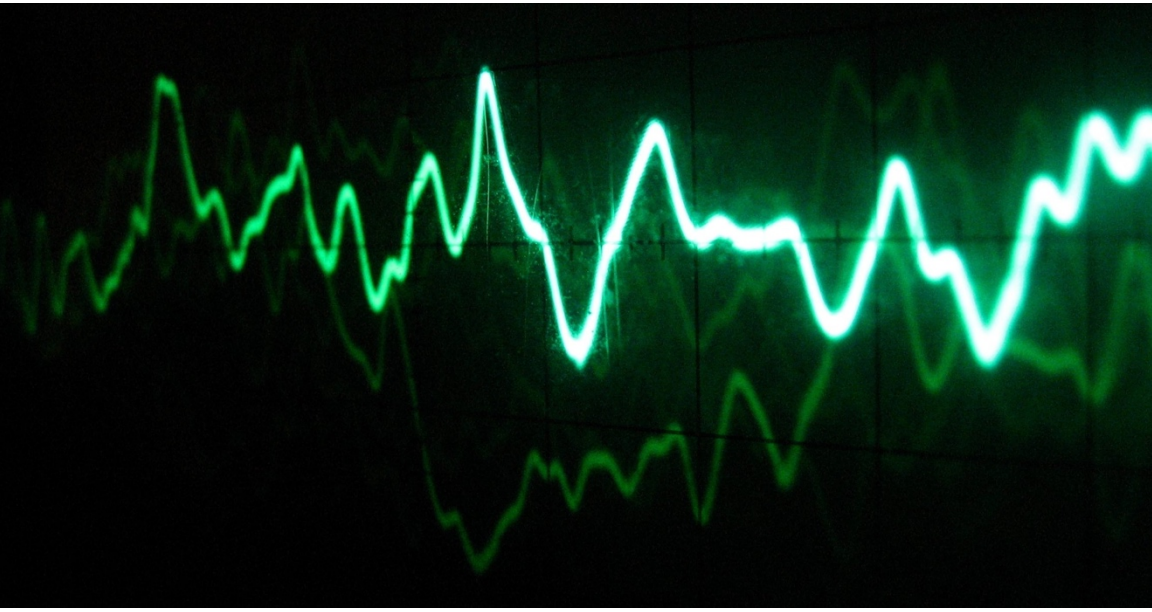
Here are a few links to great documents about the Apollo space program that I want to share with you. Many of the informations required to develop this simulator were grabbed from those sources.

1. firstmenonthemoon.com: This website is difficult to describe, I would just say that this is the closest you will ever get to actually landing on the moon…
2. TALES FROM THE LUNAR MODULE GUIDANCE COMPUTER: A paper written by Don Eyles, one of the engineer that worked on the AGC software. Eyles talks about the operating system and lunar landing guidance software. http://dodlithr.blogspot.fr/2014/08/lm-descent-to-moon-part-1-theory-and.html
3. Exo Cruiser Blog: A great blog with a lot of details about the LM and Moon landing theory. http://dodlithr.blogspot.fr/2014/08/lm-descent-to-moon-part-1-theory-and.html
4. Computer for Apollo: A video from 1965 showing the development of AGC at the MIT instrumentation lab. http://techtv.mit.edu/videos/12260-computer-for-apollo-1965-science-reporter-tv-series
5. Apollo 14: Mission to Fra Mauro: A NASA documentary from 1971, lot’s of great images. At 6:07, the documentary shortly explains how Don Eyles reprogrammed the AGC just a couple hours before landing. https://youtu.be/xY6YOISaYAI
6. Apollo lunar descent and ascent trajectories: NASA document analyzing Apollo 11 and 12 lunar landings. https://www.hq.nasa.gov/alsj/nasa58040.pdf

This chapter was originally published at https://blog.adacore.com/make-with-ada-the-eagle-has-landed

Make with Ada : From Bits to Music

by Raphaël Amiard
Aug 4, 2015



I started out as an electronic musician, so one of my original motivations when I learnt programming was so that I could eventually *program* the sounds I wanted rather than just use already existing software to do it.

https://youtu.be/L9KLnNOGczI

If you know sound synthesis a bit, you know that it is an incredibly deep field. Producing a simple square wave is a very simple task, but doing so in a musical way is actually much more complex. I approached this as a total math and DSP newbie. My goal was not to push the boundaries of digital sound synthesis, but rather to make simple stuff, that would help me understand the underlying algorithms, and produce sound that could be labelled as musical.

Also, even if I'm bad at math, I picture myself as reasonably good at software architecture (don't we all!), and I figured that producing a simple sound synthesis library, that would be easy to use and to understand, and actually small enough not to be intimidating, would be a reasonable milestone for me.

One of the other objectives was to make something that you could run on a small bareboard computer such as the stm32 or the raspberry pi, so it needs to be efficient, both in terms of memory and CPU consumption. Being able to run without an operating system would be a nice plus too!

And this is how ada-synth-lib (https://github.com/raph-amiard/ada-synth-lib) was born, for lack of a better name. The aim of the library is to present you with a toolkit that will allow you to produce sound waves, massage them into something more complex via effects and envelopes, regroup them as instruments, but also sequence them to produce a real musical sequence.

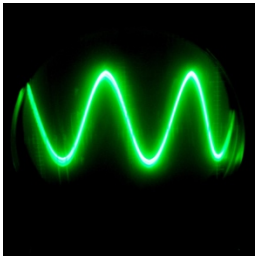
But let's walk through the basics! We'll see how to build such a sequence with ada-synth-lib, from a very simple sine generator, to a full musical sequence.

Preamble: How to compile and run the library

As its name indicates, ada-synth-lib is developped in Ada, using the AdaCore Libre suite of tools. To build and run the examples, you'll need the GPL 2015 edition of the AdaCore libre release, that you can get from here: https://www.adacore.com/download

Starting simple: The sine generator

Starting simple, we're gonna just generate a sound wave, and to make the sound not too aggressive to your ears, we'll use a sine wave, that has a smooth and soothing profile.



Holy sine wave, witnessed by an old-school oscilloscope

If you know something about sound theory, you may know that you can recreate any (periodical) sound from a carefully arranged superposition of sine waves, so the choice of the sine wave is also a way of paying respect to the theory of sound generation in general, and to fourier in particular.

Here is the code to generate a simple sine wave with ada-synth-lib. We just have a very simple sine generator, and we use the `Write_To_Stdout` helper to write the sound stream on the standard output.

```
with Waves; use Waves;
with Write_To_Stdout;

procedure Simple_Sine is
   --  Create a simple sine wave Generator.
   Sine_Gen : constant access Sine_Generator := Create_Sine (Fixed (300.0));
begin
   Write_To_Stdout (Sine_Gen);
end Simple_Sine;
```

Compiling this example and running it on the command line is simple, but we are just creating a sound stream and printing it directly to stdout! To hear it on our speakers, we need to give it to a program that will forward it to your audio hardware. There are several options to do that, the most known being the old /dev/dsp file on Unix like systems, but you have great cross platform tools such as sox that you can use for such a purpose.

```
# you should hear a sine !
$ obj/simple_sine | play -t s16 -r 44100 -
```

From bit to music – basic example (https://youtu.be/DwSyl8O1bnU)

The interesting thing is that the input to `Create_Sine` is another generator. Here we use a fixed value generator, that will provide the value for the frequency, but we could use a more complex generator, which would modulate the input frequency!

```
with Waves; use Waves;
with Write_To_Stdout;

procedure Simple_Sine is
   Sine_Gen : constant access Sine_Generator :=
     Create_Sine (
       Fixed
         (1000.0,
          -- The second parameter to the Fixed constructor is a generator
          -- that will be added to the fixed frequency generated.

          -- LFO is also a sine oscillator underneath, but you can set it to
          -- have amplitudes much larger than +- 1.0
          LFO (6.0, 200.0)));
Begin.
   Write_To_Stdout (Sine_Gen);
end Simple_Sine;
```

Going deeper

This is just the beginning of what you can do. ada-synth-lib is just a lego toolkit that you can assemble to generate the sequences you want to generate.

With only slightly more complex sequences, you can get into real musical sequences, such as the one you can hear below:

From Bits to Music – Advanced Example (https://youtu.be/2eiWnN1xWcs)

The sequencing part is done via the simple sequencer data type which you can use to create looping note sequences. Here is how it is done for the snare instrument:

```
   o : constant Sequencer_Note := No_Seq_Note;
   K : constant Sequencer_Note := (Note => (G, 3), Duration => 3000);
   Z : constant Sequencer_Note := (Note => (G, 3), Duration => 5000);
   B : constant Sequencer_Note := (Note => (G, 3), Duration => 8000);

   Snare_Seq : constant access Simple_Sequencer :=
     Create_Sequencer
       (Nb_Steps => 16, BPM => BPM, Measures => 4,
        Notes =>
          (o, o, o, o, Z, o, o, o, o, o, o, o, K, o, o, o,
           o, o, o, o, K, o, o, o, o, o, o, o, B, o, K, K,
           o, o, o, o, Z, o, o, o, o, o, o, o, K, o, o, o,
           o, o, o, o, K, o, o, K, o, o, Z, o, B, o, Z, o));
```

You can also see how we used Ada's named aggregates to make the code more readable and self documenting. Also interesting is how we can create complex synth sounds from basic bricks, as in the below example. The bricks are very simple to understand individually, but the

result is a full substractive synthetizer that can be programmed to make music!

```
Synth : constant access Disto :=
  --  We distort the output signal of the synthetizer with a soft clipper
  Create_Dist
    (Clip_Level => 1.00001,
     Coeff      => 1.5,

     --  The oscillators of the synth are fed to an LP filter
     Source     => Create_LP
       (

        --  We use an ADSR enveloppe to modulate the Cut frequency of the
        --  filter. Using it as the modulator of a Fixed generator allows us
        --  to have a cut frequency that varies between 1700 hz and 200 hz.
        Cut_Freq_Provider =>
          Fixed
            (Freq      => 200.0,
             Modulator => new Attenuator'
               (Level  => 1500.0,
                Source => Create_ADSR (10, 150, 200, 0.005, Synth_Source),
                others => <>)),

        --  Q is the resonance of the filter, very high values will give a
        --  resonant sound.
        Q => 0.2,

        --  This is the mixer, receiving the sound of 4 differently tuned
        --  oscillators, 1 sine and 3 saws
        Source =>
          Create_Mixer
            (Sources =>
                 (4 => (Create_Sine
                        (Create_Pitch_Gen
                           (Rel_Pitch => -30, Source => Synth_Source)),
                        Level => 0.6),
                  3 => (BLIT.Create_Saw
                        (Create_Pitch_Gen
                           (Rel_Pitch => -24, Source => Synth_Source)),
                        Level => 0.3),
                  2 => (BLIT.Create_Saw
                        (Create_Pitch_Gen
                           (Rel_Pitch => -12, Source => Synth_Source)),
                        Level => 0.3),
                  1 => (BLIT.Create_Saw
                        (Create_Pitch_Gen
                           (Rel_Pitch => -17, Source => Synth_Source)),
                        Level => 0.5)))));
```

The ADSR envelope is what gives the sound a dynamic nature, shaping it in the time domain. The Low Pass filter shapes the sound by removing some high frequency components from it.

That's it for today! In the next instalment of this series we'll see how to compile and run the code on a bare board system using the STM32F4 board and AdaCore GPL tools.

Links and Credits

- You can find the ada-synth-lib library on github- https://github.com/raph-amiard/ada-synth-lib

- The needed toolchain to play with it is on the libre site.

- A good, high-level guide to music synthesis here- http://beausievers.com/synth/synthbasics/

- A lot of the algorithms in ada-synth-lib were inspired by stuff I found on http://musicdsp.org/, so big thanks to every people putting algorithms in there.

- The alias free oscillators in the BLIT module are done using the Bandlimited Impulse Train method, for which the seminal paper is here- https://ccrma.stanford.edu/~stilti/papers/blit.pdf

- Thanks and credits to Mikael Altermark for the beautiful sound waves pictures, and to Bisqwit for the introduction video!

This chapter was originally published at
https://blog.adacore.com/make-with-ada-from-bits-to-music

Make with Ada: Formal Proof on My Wrist

by Fabien Chouteau
Nov 10, 2015

When the Pebble Time kickstarter went through the roof, I looked at the specification and noticed the watch was running on an STM32F4, an ARM cortex-M4 CPU which is supported by GNAT. So I backed the campaign, first to be part of the cool kids and also to try some Ada hacking on the device.

At first I wasn't sure if I was going to replace the Pebble OS entirely or only write an application for it. The first option requires to re-write all the drivers (screen, bluetooth, etc) and I would also need a way to program the chip, which to my knowledge requires opening the watch and soldering a debug interface to the chip. Since I'm not ready to crack open a $200 watch just for the sake of hacking I decided to go with the second option, write an app.

Binding the C API to Ada

The Pebble SDK is very well thought out and provides an emulator based on QEMU which is great for testing. In fact I developed and tested the binding with this SDK before I even got the real watch.

The entire API is contained in a single C header (pebble.h). I used GNAT's switch -fdump-ada-spec to generate a first version of the binding, then I reformatted it to split the features in packages and rename the subprograms. For example, the function layer_create became:

```
package Pebble.Ui.Layers is
   function Create (Frame : Grect) return Layer; -- pebble.h:3790
   pragma Import (C, Create, "layer_create");
end Pebble.Ui.Layers;
```

The API uses almost exclusively pointers to hidden structures:

```
typedef struct Window Window;

Window * window_create(void);
```

which we can conveniently map in Ada like so:

```
   type Window is private;

   function Create return Window;
   pragma Import (C, Create, "window_create");

private

   type Window is new System.Address;
```

Overall the binding was relatively easy to create.

Smartwatch app and formal proof

To use this binding I decided to port the formally proven Tetris written in SPARK by Yannick and Tristan (http://blog.adacore.com/tetris-in-spark-on-arm-cortex-m4),

The game system being the same, this port consists of a new graphical front-end, menus and button handling. The app was quite straightforward to develop, the Pebble API is well designed and quite easy to use.

The formal proof is focused on things that are impossible to test, in our case, the game system. Can you think of a way to test that the code will reject invalid moves on any piece, in any orientation, at any position and for every board state possible (that's trillions or quadrillions of combinations)?

The first thing we get from SPARK analysis is the absence of run-time error. For us it means, for instance, that we are sure not to do invalid access to the game board matrix.

Then we prove high level game properties like:

- Never move the falling piece into an invalid position
- The falling piece will never overlap the pieces already in the board
- All the complete lines are removed
- Whatever the player does, the game will always be in a valid state

This application was released on the Pebble app store under the name of PATRIS (http://apps.getpebble.com/en_US/application/559af16358bc81d930000067), and as of today more than 660 users downloaded it.

I also made a watchface (http://apps.getpebble.com/en_US/application/561a73451a1d8994a700007e) using the blocks to display time digits.

Persistent storage

The last thing I did was to create a higher interface to the persistent storage API, a mechanism to store data and retrieve it after the app is closed.

The C interface is very simple with only a couple of functions. Each data is associated with a uint32_t key, so the app can read, write and test the existence of data for a given key.

```
int persist_read_data(const uint32_t key, void * buffer, const size_t buffer_size);
int persist_write_data(const uint32_t key, const void * data, const size_t size);
bool persist_exists(const uint32_t key);
```

But of course, the Ada type system doesn't like void pointers and to be able to use the persistent storage without having to deal with nasty unchecked conversions (same as C casts) I wrote a generic package that automatically takes care of everything:

```
generic
   Data_Key : Uint32_T;
   type Data_Type is private;

package Pebble.Generic_Persistent_Storage is
    function Write (Data : Data_Type) return Boolean;
   --  Write data associated with the key, return True on success

   function Read (Data : out Data_Type) return Boolean;
   --  Read data associated with the key, return True on success

   function Exists return Boolean;
   --  Return True if there is data associated with the key

   procedure Erase;
   --  Erase data associated with the key
end Pebble.Generic_Persistent_Storage;
```

Using the persistent storage is now as simple as:

```
   type My_Data is record
      Level : Natural;
      XP    : Natural;
   end record;

   package My_Data_Storage is new Pebble.Generic_Persistent_Storage (1, My_Data);

   Player : My_Data;
begin

   if not My_Data_Storage.Read (Player) then
      Player.XP    := 0;
      Player.Level := 1;
   end if;

    --   Insert complex gameplay here…

   if not My_Data_Storage.Write (Player) then
      Put_Line ("The game could not be saved");
   end if;
```

Sources

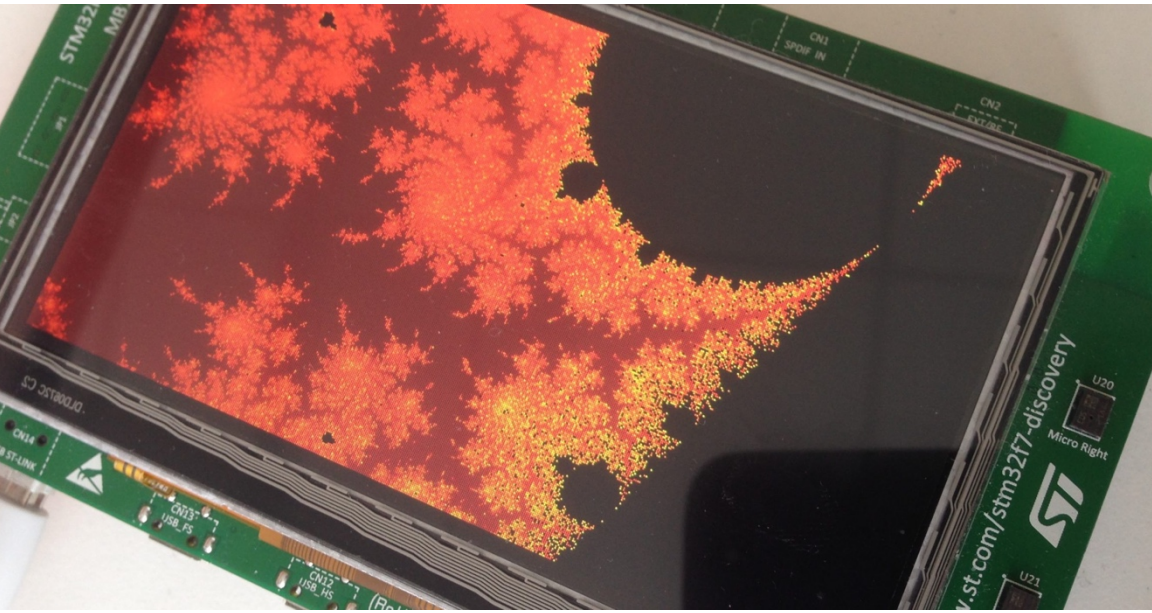
The binding and app source code are available on GitHub- https://github.com/Fabien-Chouteau/Ada_Time.

Youtube video of the watch in action: https://youtu.be/5Ng5fmdXmwk

This chapter was originally published at https://blog.adacore.com/make-with-ada-formal-proof-on-my-wrist

Porting the Ada Runtime to a New ARM Board

by Jérôme Lambourg
Jan 12, 2016



As a first article (for me) on this blog, I wanted to show you how to adapt and configure a ravenscar-compliant run-time (full or sfp) to a MCU/board when the specific MCU or board does not come predefined with the GNAT run-time.

To do so, I will use GNAT GPL for ARM ELF and 3 boards of the same family: the STM32F429I-Discovery, the STM32F469I-Discovery, and the STM32F746G-Discovery.

These boards are interesting because:

- They're fun to use, with lots of components to play with (exact features depends on the board): LCD, touch panel, audio in/out, SD-Card support, Networking, etc.
- They are pretty cheap.
- They are from the same manufacturer, so we can expect some reuse in terms of drivers.
- The first one (STM32F429I-Disco) is already supported by default by the GNAT run-time. We can start from there to add support for the other boards.
- They differ enough to deserve specific run-time adjustments, while sharing the same architecture (ARMv7) and DSP/FPU (Cortex-M4 & M7)

STM32F429I-Disco STM32F469I-Disco STM32F746G-Disco

So where to start ? First, we need to understand what is MCU-specific, and what is board-specific:

- Instructions, architecture are MCU specific. GCC is configured to produce code that is compatible with a specific architecture. This also takes into account specific floating point instructions when they are supported by the hardware.
- Initialization of an MCU is specific to a family (All STM32F4 share the same code, the F7 will need adjustments).

- The interrupts are MCU-specific, but their number and assignments vary from one minor version to another depending on the features provided by the MCU.

- Memory mapping is also MCU-specific. However there are differences in the amount of available memory depending on the exact version of the MCU (e.g. this is not a property of the MCU family). This concerns the in-MCU memory (the SRAM), not the potential external SDRAM memory that depends on the board.

- Most clock configuration can be made board-independant, using the MCU's HSI clock (High Speed Internal clock), however this is in general not desirable, as external clocks are much more reliable. Configuring the board and MCU to use the HSE (High Speed External clock) is thus recommended, but board-specific.

From this list, we can deduce that - if we consider the CPU architecture stable, which is the case here - adapting the run-time to a new board mainly consists in:

- Adapting the startup code in case of a major MCU version (STM32F7, that is Cortex-M7 based).

- Checking and defining the memory mapping for the new MCU.

- Checking and defining the clock configuration for the specific board.

- Make sure that the hardware interrupts are properly defined and handled.

Preparing the sources

To follow this tutorial, you will need at least one of the boards, the stlink tools to flash the board or load examples in memory, and GNAT GPL for ARM (hosted on Linux or Windows) that can be downloaded from www.adacore.com/download.

Install it (in the explanations below, I installed it in $HOME/gnat).

The GNAT run-times for bareboard targets are all user-customizable. In this case, they are located in <install prefix>/arm-eabi/lib/gnat.

The board-specific files are located in the arch and gnarl-arch subfolders of the run-times.

So let's create our new run-time there, and test it. Create a new folder named ravenscar-sfp-stm32f469disco, in there, you will need to copy from the original ravenscar-sfp-stm32f4 folder:

- arch/
- gnarl-arch/
- ada-object-path
- runtime.xml
- runtime_build.gpr and ravenscar_build.gpr and apply the following modifications:

```
$ diff -ub ../ravenscar-sfp-stm32f4/runtime_build.gpr runtime_build.gpr
--- ../ravenscar-sfp-stm32f4/runtime_build.gpr   2016-01-09 14:09:26.936000000 +0100
+++ runtime_build.gpr    2016-01-09 14:10:43.528000000 +0100
@@ -1,5 +1,6 @@
 project Runtime_Build is
    for Languages use ("Ada", "C", "Asm_Cpp");
+   for Target use "arm-eabi";

    for Library_Auto_Init use "False";
    for Library_Name use "gnat";
@@ -8,7 +9,8 @@
    for Library_Dir use "adalib";
    for Object_Dir use "obj";

-   for Source_Dirs use ("arch", "common", "math");
+   for Source_Dirs use
+     ("arch", "../ravenscar-sfp-stm32f4/common", "../ravenscar-sfp-stm32f4/math");

    type Build_Type is ("Production", "Debug");

$ diff -ub ../ravenscar-sfp-stm32f4/ravenscar_build.gpr ravenscar_build.gpr
--- ../ravenscar-sfp-stm32f4/ravenscar_build.gpr 2015-04-30 12:36:37.000000000 +0200
+++ ravenscar_build.gpr 2016-01-09 14:11:37.952000000 +0100
@@ -1,7 +1,9 @@
 with "runtime_build.gpr";
```

```
 project Ravenscar_Build is
   for Languages use ("Ada", "C", "Asm_Cpp");
+   for Target use "arm-eabi";

   for Library_Auto_Init use "False";
   for Library_Name use "gnarl";
@@ -10,7 +12,8 @@
   for Library_Dir use "adalib";
   for Object_Dir use "obj";

-  for Source_Dirs use ("gnarl-arch", "gnarl-common");
+   for Source_Dirs use
+     ("gnarl-arch", "../ravenscar-sfp-stm32f4/gnarl-common");

   type Build_Type is ("Production", "Debug");
```

- ada_source_path with the following content:

```
1. arch
2. ../ravenscar-sfp-stm32f4/common
3. ../ravenscar-sfp-stm32f4/math
4. ../ravenscar-sfp-stm32f4/gnarl-common
5. gnarl-arch
```

You are now ready to build your own run-time. To try it out, just do:

```
$ cd ~/gnat/arm-eabi/lib/gnat/ravenscar-sfp-stm32f469disco
$ export PATH=$HOME/gnat/bin:$PATH
$ gprbuild -p  -f -P ravenscar_build.gpr
```

If everything goes fine, then a new ravenscar-sfp run-time should have been created.

As it has been created directly within the GNAT default search path, you can use it via its short name (e.g. the directory name) just as a regular run-time: by specifying --RTS=ravenscar-sfp-stm32f469disco in gprbuild's command line for example, or by specifying 'for Runtime ("Ada") use "ravenscar-sfp-stm32f469disco"' in your project file.

```
$ ls
ada_object_path  adalib  gnarl-arch  ravenscar_build.gpr  runtime_build.gpr
ada_source_path  arch    obj         runtime.xml
```

Handling the STM32F469I-Discovery:

Let's start with the support of the STM32F469I-Discovery. Being the same MCU major version than the STM32F429, modifications to the run-time are less intrusive than the modifications for the STM32F7,

First, we need to make sure the board is properly handled by gprbuild. For that, we edit runtime.xml and change

type Boards is ("STM32F4-DISCO", "STM32F429-DISCO", "STM32F7-EVAL");
Board : Boards := external ("BOARD", "STM32F4-DISCO");

with:

type Boards is ("STM32F469-DISCO");
Board : Boards := external ("BOARD", "STM32F469-DISCO");

Now we're ready to start the real thing.

Memory mapping and linker scripts

In this step, we're going to tell the linker at what addresses we need to put stuff. This is done by creating a linker script from the base STM32F429-DISCO script:

```
$ cd arch
$ mv STM32F429-DISCO.ld STM32F469-DISCO.ld
# Additionally, you can cleanup the other STM32*.ld scripts, they are unused by this
customized run-time
```

Next, we need to find the technical documents that describe the MCU. Go to http://st.com and search for "stm32f469NI" (that is the MCU used by the discovery board), and once in the product page, click on "design resources" and check the RM0386 Reference Manual.

From the chapter 2.3.1, we learn that we have a total of 384kB of SRAM, including 64kB of CCM (Core Coupled Memory) at 0x1000 0000 and the remaining at 0x2000 0000.

Additionally, we need to check the flash size. This is MCU micro version specific, and the specific MCU of the STM32F469-Disco board has 2

MB of flash. The STM32 reference manual tells us that this flash is addressed at 0x0800 0000.

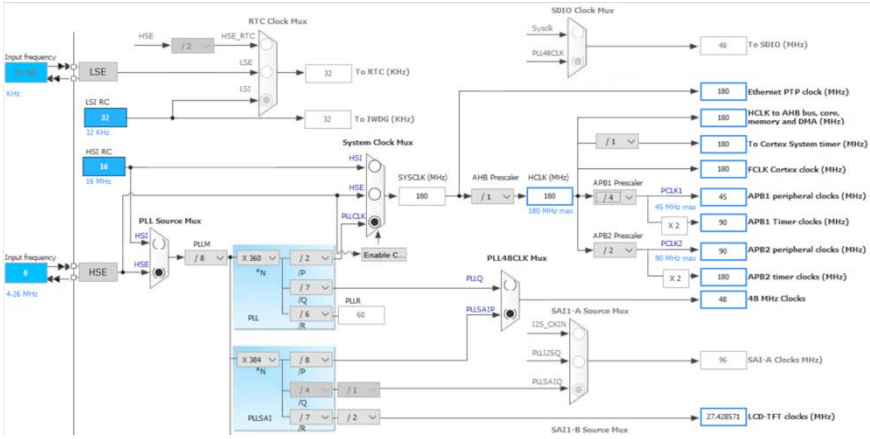
So with this information, you can now edit the STM32F469-DISCO-memory-map.ld file:

```
MEMORY
{
  flash (rx)  : ORIGIN = 0x08000000, LENGTH = 2048K
  sram  (rwx) : ORIGIN = 0x20000000, LENGTH = 320K
  ccm   (rw)  : ORIGIN = 0x10000000, LENGTH = 64K
}
```

System clocks

The run-time is responsible for initializing the system clock. We need the following information to do this - the various clock settings that are available, and the main clock source.

STMicroelectonics provides a Windows tool to help set up their MCU: STM32CubeMX. Using the tool we can verify the clock settings:



To properly setup the values, we now need to check the speed of the HSE clock on the board. So back to st.com, search for STM32F469-Disco, and from the product page, download the board's user manual UM1932: Discovery kit with STM32F469NI MCU (http://www.st.com/st-web-ui/static/active/en/resource/technical/document/user_manual/DMO

0218846.pdf). From chapter "6.3.1 HSE clock source" check that the HSE clock is running at 8MHz.

Now let's check that the run-time is doing the right thing:

arch/setup_pll.adb is responsible for the clock setup

gnarl-arch/s-bbpara.ads defines the clock constants

arch/s-stm32f.ads define some of the MCU's registers, as well as Device ID constants.

Start by adding the STM32F46x device id in s-stm32f.ads. You can search google for the device id, or use st-util to connect to the board and report the id.

```
DEV_ID_STM32F40xxx : constant := 16#413#;
DEV_ID_STM32F42xxx : constant := 16#419#;
DEV_ID_STM32F46xxx : constant := 16#434#;
DEV_ID_STM32F7xxxx : constant := 16#449#;
```

Now let's check the clock constants in s-bbpara.ads:

```
function HSE_Clock
  (Device_ID : STM32F4.Bits_12) return STM32F4.RCC.HSECLK_Range
is (case Device_ID is
       when STM32F4.DEV_ID_STM32F42xxx => 8_000_000,
       --  STM32F429 Disco board
       when STM32F4.DEV_ID_STM32F7xxxx => 25_000_000,
       --  STM32F7 Evaluation board
       when others => 8_000_000)
       --  STM32F407 Disco board and Unknown device
with Inline_Always;
```

We see in s-bbpara.ads that the HSE is OK (we fall in the 'others' case). However the Clock_Frequency constant can be bumped to 180_000_000.

```
Clock_Frequency : constant := 180_000_000;
pragma Assert (Clock_Frequency in STM32F4.RCC.SYSCLK_Range);
```

Looking now at setup_pll.adb, we can verify that this file does not require specific changes. PLLM is set to 8 to achieve a 1 MHz input clock. PLLP is a constant to 2, so PLLN is evaluated to 360 to achieve the expected clock speed : HSE / PLLM * PLLN / PLLP = 180 MHz.

However, the PWR initialization should be amended to handle the STM32F46 case, and can be simplified as we're creating a run-time specific to the MCU:

```
$ diff -u ../ravenscar-sfp-stm32f4/arch/setup_pll.adb arch/setup_pll.adb
--- ../ravenscar-sfp-stm32f4/arch/setup_pll.adb  2015-04-30 12:36:37.000000000 +0200
+++ arch/setup_pll.adb  2016-01-09 14:11:11.216000000 +0100
@@ -90,7 +90,6 @@
    procedure Initialize_Clocks is

       HSECLK    : constant Integer := Integer (HSE_Clock (MCU_ID.DEV_ID));
-      MCU_ID_Cp : constant MCU_ID_Register := MCU_ID;

       ------------------------------
       -- Compute Clock Frequencies --
@@ -194,11 +193,7 @@
       --  and table 15 p79). On the stm32f4 discovery board, VDD is 3V.
       --  Voltage supply scaling only

-      if MCU_ID_Cp.DEV_ID = DEV_ID_STM32F40xxx then
-         PWR.CR := PWR_CR_VOS_HIGH_407;
-      elsif MCU_ID_Cp.DEV_ID = DEV_ID_STM32F42xxx then
-         PWR.CR := PWR_CR_VOS_HIGH_429;
-      end if;
+      PWR.CR := PWR_CR_VOS_HIGH_429;

       --  Setup internal clock and wait for HSI stabilisation.
       --  The internal high speed clock is always enabled, because it is the
```

Interrupts

The available interrupts on the MCU can be found in the Reference Manual.

However, an easier and better way to get the list of interrupts is by generating the Ada bindings from the CMSIS-SVD file for this board using the svd2ada tool that can be found on GitHub, and by downloading the SVD file that corresponds to the current MCU (STM32F46_79x.svd) directly from ARM. This binding generates the

interrupts list and we can then check the ones that are not mapped by the current run-time.

```
$ svd2ada ~/SVD_FILES/STM32F46_79x.svd -p STM32_SVD -o temp
$ cat temp/stm32_svd-interrupts.ads
...
...
   UART7_Interrupt: constant Interrupt_ID := 84;

   UART8_Interrupt: constant Interrupt_ID := 85;

   SPI4_Interrupt: constant Interrupt_ID := 86;

   SPI5_Interrupt: constant Interrupt_ID := 87;

   SPI6_Interrupt: constant Interrupt_ID := 88;

   SAI1_Interrupt: constant Interrupt_ID := 89;

   LCD_TFT_Interrupt: constant Interrupt_ID := 90;

   LCD_TFT_1_Interrupt: constant Interrupt_ID := 91;

   DMA2D_Interrupt: constant Interrupt_ID := 92;

   QUADSPI_Interrupt: constant Interrupt_ID := 93;
```

A total of 91 interrupts are defined by the MCU, with an additional 2 required by GNAT (Interrupt Id 0 is reserved, and GNAT maps the SysTick interrupt to Id 1).

So let's amend the gnarl-arch/a-intnam.ads file:

```
   HASH_RNG_Interrupt              : constant Interrupt_ID := 82;
   FPU_Interrupt                   : constant Interrupt_ID := 83; --  This line and
below are new
   UART7_Interrupt                 : constant Interrupt_ID := 84;
   UART8_Interrupt                 : constant Interrupt_ID := 85;
   SPI4_Interrupt                  : constant Interrupt_ID := 86;
   SPI5_Interrupt                  : constant Interrupt_ID := 87;
   SPI6_Interrupt                  : constant Interrupt_ID := 88;
   SAI1_Interrupt                  : constant Interrupt_ID := 89;
```

```
   LCD_TFT_Interrupt               : constant Interrupt_ID := 90;
   LCD_TFT_1_Interrupt             : constant Interrupt_ID := 91;
   DMA2D_Interrupt                 : constant Interrupt_ID := 92;
   QUADSPI_Interrupt               : constant Interrupt_ID := 93;

end Ada.Interrupts.Names;
```

We also need to edit arch/handler.S to properly initialize the interrupt vector:

```
$ diff -bu ../ravenscar-sfp-stm32f4/arch/handler.S arch/handler.S
--- ../ravenscar-sfp-stm32f4/arch/handler.S     2014-09-15 11:28:25.000000000 +0200
+++ arch/handler.S      2016-01-09 11:58:32.456000000 +0100
@@ -145,6 +145,16 @@
            .word   __gnat_irq_trap      /* 95 IRQ79.  */
            .word   __gnat_irq_trap      /* 96 IRQ80.  */
            .word   __gnat_irq_trap      /* 97 IRQ81.  */
+           .word   __gnat_irq_trap      /* 98 IRQ82.  */
+           .word   __gnat_irq_trap      /* 99 IRQ83.  */
+           .word   __gnat_irq_trap      /* 100 IRQ84.  */
+           .word   __gnat_irq_trap      /* 101 IRQ85.  */
+           .word   __gnat_irq_trap      /* 102 IRQ86.  */
+           .word   __gnat_irq_trap      /* 103 IRQ87.  */
+           .word   __gnat_irq_trap      /* 104 IRQ88.  */
+           .word   __gnat_irq_trap      /* 105 IRQ89.  */
+           .word   __gnat_irq_trap      /* 106 IRQ90.  */
+           .word   __gnat_irq_trap      /* 107 IRQ91.  */

            .text
```

And we also need to bump the number of interrupt IDs in gnarl-arch/s-bbpara.ads:

```
   Number_Of_Interrupt_ID : constant := 93;
```

And that's it

The necessary job has now been done to support the STM32F469I-Disco. You can now install the run-time, and use it with the examples from our bareboard drivers repository on GitHub

(https://github.com/AdaCore/bareboard/tree/svd). Note that, as of the time when this article is written, only the 'svd' branch includes some drivers support for this board.

```
$ gprbuild -P ravenscar_build.gpr
$ cd ~/bareboard/ARM/STMicro/STM32/examples/balls
$ git checkout svd
$ gprbuild -p -P balls_demo.gpr -XBOARD=STM32F469-DISCO -XRTS=ravenscar-sfp -XLCH=lcd
-XLOADER=ROM --RTS=ravenscar-sfp-stm32f469disco
$ arm-eabi-objcopy -O binary obj/demo obj/demo.bin
$ st-flash write obj/demo.bin 0x8000000
```

Porting the run-time to the STM32F7-DISCOVERY

Now on to the STM32F7. This is going to be a bit more difficult for one reason: the STM32F7, being based on the Cortex-M7, can now benefit from Data and Instruction caches. These caches need explicit initialization. A minimal support for the STM32F7 already exists in the run-time, but it is incomplete as these caches are not properly initialized.

Prepare the run-time

First of all, let's create the new run-time for this board. We'll start this time from the work previously performed for the STM32F469-Discovery board to speed up the process.

```
$ cd ~/gnat/arm-eabi/lib/gnat
$ cp -r ravenscar-sfp-stm32f469disco ravenscar-sfp-stm32f7disco
```

Enable Data and Instruction caches

Initialization of the cache is described in details by ARM in the Cortex-M7 processor technical reference manual.

So let's try to update the startup code. For that, we're going to add a new file 'arch/start-common.S':

```
            .syntax unified
            .cpu cortex-m4
            .thumb
            .text
            .thumb_func
            .globl      _stm32_start_common
        .type _stm32_start_common, #function
_stm32_start_common:
        /**************/
        /* Enable FPU */
        /**************/
        movw     r0,#0xED88
        movt     r0,#0xE000
        ldr      r1,[r0]
        orr      r1,r1,#(0xF << 20)
        str      r1,[r0]
        /* Wait for store to complete and reset pipeline with FPU enabled  */
        dsb
        isb
        /********************
         * Enable I/D cache *
        ********************/
        /* Register definition for cache handling */
        .set    CCSIDR,  0xE000ED80
        .set    CSSELR,  0xE000ED84
        .set    DCISW,   0xE000EF60
        .set    ICIALLU, 0xE000EF50
        .set    CCR,     0xE000ED14
        /* First invalidate the data cache */
dcache_invalidate:
        mov     r1, #0x0
        ldr     r0, =CSSELR
        str     r1, [r0]        /* Select the data cache size */
        dsb
        ldr     r0, =CCSIDR
        ldr     r2, [r0]        /* Cache size identification */
        and     r1, r2, #0x7    /* Number of words in a cache line */
        add     r7, r1, #0x4
        ubfx    r4, r2, #3, #10  /* r4 = number of ways - 1 of data cache */
        ubfx    r2, r2, #13, #15 /* r2 = number of sets - 1 of data cache */
        clz     r6, r4           /* Calculate bit offset for "way" in DCISW */
        ldr     r0, =DCISW
inv_loop1:
        mov     r1, r4
        lsls    r8, r2, r7
```

```
inv_loop2:
        lsls    r3, r1, r6
        orrs    r3, r3, r8
        str     r3, [r0]        /* Invalidate the D-Cache line */
        subs    r1, r1, #1
        bge     inv_loop2
        subs    r2, r2, #1
        bge     inv_loop1
        dsb
        isb
        /* Now invalidate the instruction cache */
icache_invalidate:
        mov     r1, #0x0
        ldr     r0, =ICIALLU
        str     r1, [r0]
        dsb
        isb
        /* Finally enable Instruction and Data cache */
        ldr     r0, =CCR
        ldr     r1, [r0]
        orr     r1, r1, #(0x1 << 16) /* Sets the data cache enabled field */
        orr     r1, r1, #(0x1 << 17) /* Sets the i-cache enabled field */
        str     r1, [r0]
        dsb
        isb
        /*****************************
         * TCM Memory initialisation *
        *****************************/
        .set    CM7_ITCMCR, 0xE000EF90
        .set    CM7_DTCMCR, 0xE000EF94
        ldr     r0, =CM7_ITCMCR
        ldr     r1, [r0]
        orr     r1, r1, #0x1 /* set the EN field */
        str     r1, [r0]
        ldr     r0, =CM7_DTCMCR
        ldr     r1, [r0]
        orr     r1, r1, #0x1 /* set the EN field */
        str     r1, [r0]
        dsb
        isb
end:
        bx lr
        .size _stm32_start_common, . - _stm32_start_common
```

This file initializes the FPU, the data cache, the instruction cache (according to the ARM documentation), as well as the TCM memory.

We now need to call it from the startup files, start-ram.S and start-rom.S.

```
start-ram.S:

  /* Init stack */
            ldr         sp,.LinitSp

-        /* Enable FPU */
-        movw     r0,#0xED88
-        movt     r0,#0xE000
-        ldr      r1,[r0]
-        orr      r1,r1,#(0xF << 20)
-        str      r1,[r0]
-
-        /* Wait for store to complete and reset pipeline with FPU enabled  */
-        dsb
-        isb
+        bl _stm32_start_common

            /* Clear .bss */
            movw        r0,#:lower16:__bss_start

start-rom.S:

 _start_rom:
-        /* Enable FPU */
-        movw     r0,#0xED88
-        movt     r0,#0xE000
-        ldr      r1,[r0]
-        orr      r1,r1,#(0xF << 20)
-        str      r1,[r0]
+        bl _stm32_start_common
```

Clocks, interrupts, linker scripts, etc.

We will also create a linker script for the STM32F7, and add the new board to runtime.xml. We perform the same run-time modifications we did for the STM32F469-Disco board:

create arch/STM32F7-DISCO-memory-map.ld:

```
MEMORY
{
itcm (x) : ORIGIN = 0x00000000, LENGTH = 16K
flash (rx) : ORIGIN = 0x08000000, LENGTH = 1024K
dtcm (rx) : ORIGIN = 0x20000000, LENGTH = 64K
sram (rwx) : ORIGIN = 0x20010000, LENGTH = 240K
}
```

In s-stm32f.ads, DEV_ID_STM32F7xxxx is already defined.

In s-bbpara.ads, the HSE clock is also properly set to 25MHz, the MCU can run at 216 MHz, but STM32CubeMX shows some issues with such value, so we simplify by using a 200MHz value.

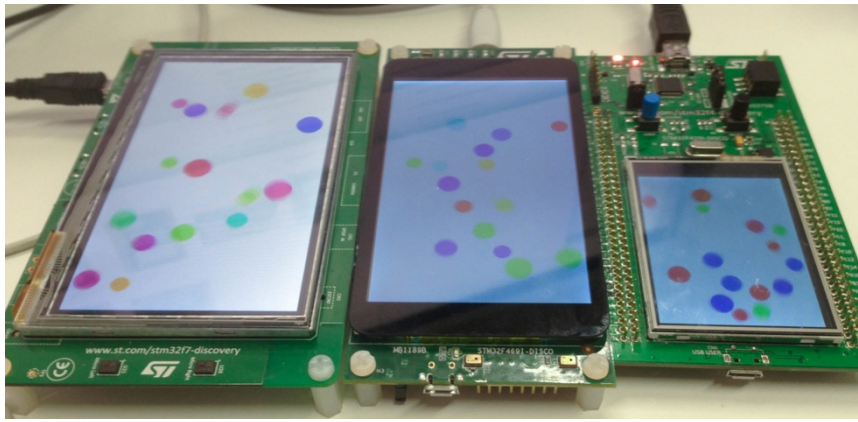
Now edit runtime.xml:

```
type Boards is ("STM32F7-DISCO");
Board : Boards := external ("BOARD", "STM32F7-DISCO");
```

The interrupts are very similar between the STM32F746 and the STM32F469, so you can benefit from the changes already performed.

Et voilà. Now you can rebuild the run-time, and test it similarly to the stm32f469-disco.

```
$ gprbuild -P ravenscar_build.gpr
$ cd ~/bareboard/ARM/STMicro/STM32/examples/balls
$ gprbuild -p -P balls_demo.gpr -XBOARD=STM32F7-DISCO -XRTS=ravenscar-sfp -XLCH=lcd -
XLOADER=ROM --RTS=ravenscar-sfp-stm32f7disco
$ arm-eabi-objcopy -O binary obj/demo obj/demo.bin
$ st-flash write obj/demo.bin 0x8000000
```



GNAT on the three boards

Final words and refinements

You will find below the source files for the runtimes.

Although part of the initial run-time for the STM32F429-Disco is delivered with GNAT, it is not necessarily well optimized (some missing interrupts and a non-optimal clock speed in particular). So I included the sfp and full ravenscar run-times for it as well in the final source packages.

Also, in the attached source package, I made use of extending projects to adjust the runtimes. The setup is a bit complex so I haven't explained it above as this is not really part of the subject, but you can have a look if you want. By using extending projects, the advantage is that I only needed to add the files that I'm actually modifying, and thus can more easily benefit from a futur upgrade of GNAT.

Finally, in the downloadable sources, I got rid of the 'BOARD' scenario variable, as the runtimes are now board specific: such scenario variable is only useful when supporting a complete board family.

To go further in customized run-time, you can refer to the following documentation: Customized run-time (https://docs.adacore.com/gnat_ugx-docs/html/gnat_ugx/gnat_ugx/customized_run-time_topics.html).

This chapter was originally published at https://blog.adacore.com/porting-the-ada-runtime-to-a-new-arm-board

Make with Ada: Candy Dispenser, with a Twist...

by Fabien Chouteau
Mar 03, 2016

A few months ago, my colleague Rebecca installed a candy dispenser in our kitchen here at AdaCore. I don't remember how exactly, but I was challenged to make it more... fun.

So my idea is to add a touch screen on which people have to answer questions about Ada or AdaCore's technology, and to modify the dispenser so that people only get candy when they give the right answer. (Evil, isn't it?)

For this, I will use the great STM32F469 discovery board with a 800x600 LCD and capacitive touch screen. But before that, I have to hack the candy dispenser....

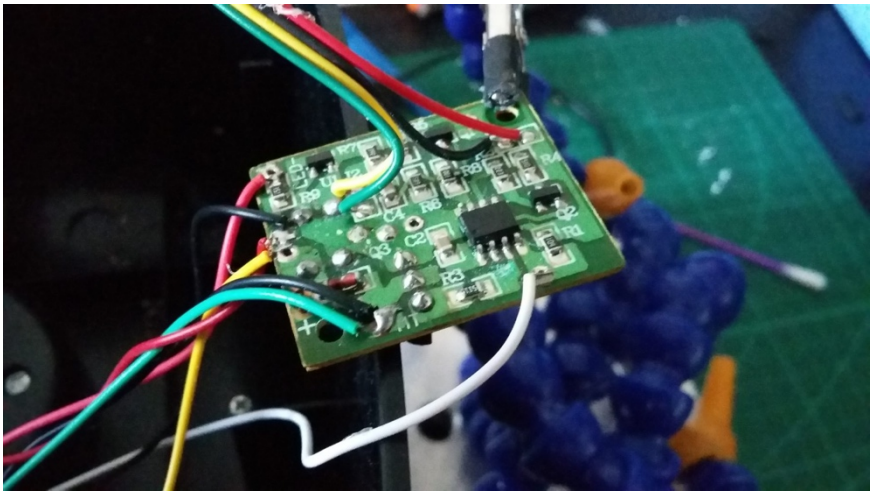
Hacking the dispenser

The first thing to do is to hack the candy dispenser to be able to control the candy delivery system.

The candy dispenser is made of :

- A container for the candies
- A motor that turns a worm gear pushing the candies out of the machine
- An infrared proximity sensor which detects the user's hand

My goal is to find the signal that commands the motor and insert my system in the middle of it. When the dispenser will try to turn on the motor, it means a hand is detected and I can decide whether or not I actually want to turn on the motor.



To find the signal controlling the motor, I started by looking at the wire going to the motor and where it lands on the board. It is connected to the center leg of a "big" transistor. Another leg of the transistor is tied to the ground, so the third one must be the signal I am looking for. By following the trace, I see that this signal is connected to an 8 pin IC: it must be the microcontroller driving the machine.

At this point, I have enough info to hack the dispenser, but I want to understand how the detection works and see what the signal is going to

look like. So I hook up my logic analyser to each pin of the IC and start recording.

Here are the 3 interesting signals: infrared LED, infrared sensor, and motor control.

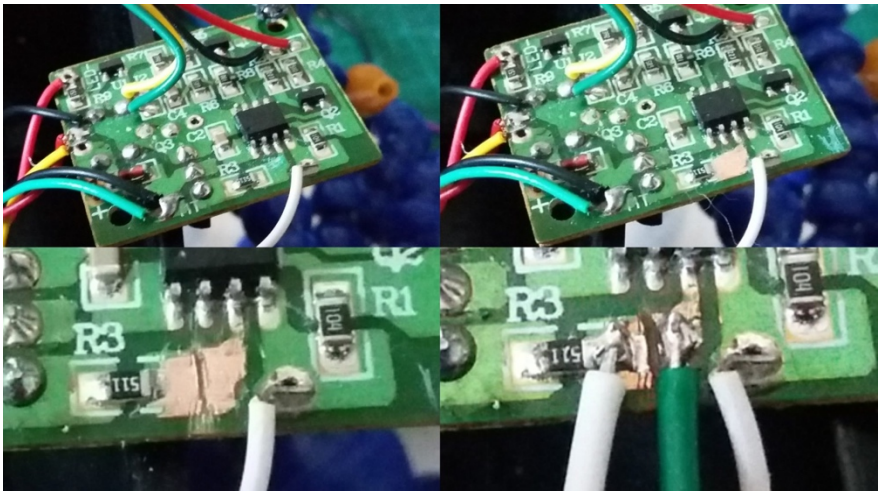
The detection works in 3 phases:

- **Wait**: The microcontroller is turning on the infra-red LED (signal 01) only 0.2 milliseconds every 100ms.This is to save power as the machine is designed to be battery powered.

- **Detection**: When something is detected, the MCU will then turn on the infra-red LED 10 times to confirm that there's actually something in front of the sensor.

- **Delivery**: The motor is turned on for a maximum of 300ms. During that period the MCU checks every 20ms to see if the hand is still in front of the sensor (signal 03).

This confirms that the signal controlling the motor is where I want to insert my project. This way I let the MCU of the dispenser handle the detection and the false positive.

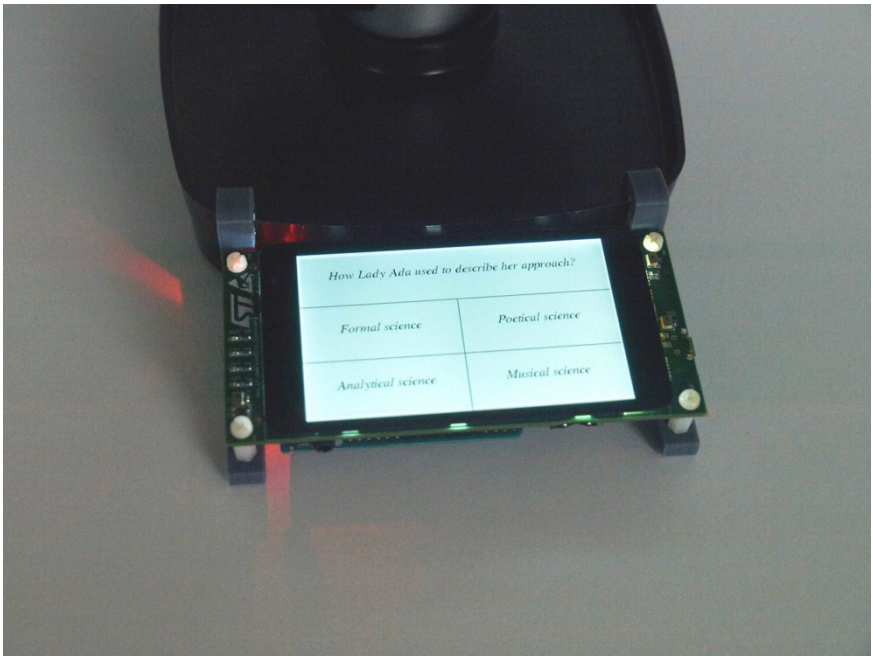


I scratched the solder mask, cut the trace and solder a wire to each side. The green wire is now my detection signal (high when a hand is detected) and the white wire is my motor command.

I ran those two wires with an additional ground wire down to the base of the dispenser and then outside to connect them to the STM32F469 discovery board.

Software

The software side is not that complicated: I use the GPL release of GNAT and the Ravenscar run-time on an ARM Cortex-M on the STM32F469. The green wire coming from the dispenser is connected to a GPIO that will trigger an interrupt, so whenever the software gets an interrupt it means that there's a hand in front of the sensor. And using another GPIO, I can turn on or off the dispenser motor.



For the GUI, I used one of my toy projects called Giza: it's a simple graphical tool kit for basic touch screen user interface. The interfaces has two windows. The first window shows the question and there is one button for each of the possible answers. When the player clicks on the wrong answer, another question is displayed after a few seconds; when

it's the right answer, the delivery window is shown. On the delivery window the player can abort by using the "No, thanks" button or put his hand under the sensor and get his candies!

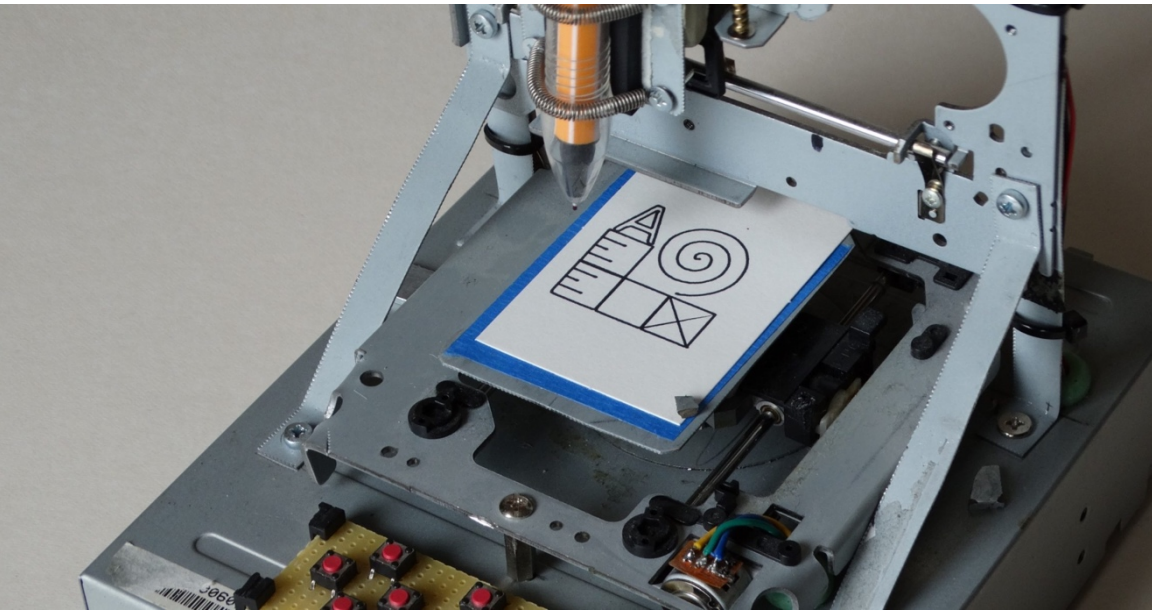
The code is available on GitHub here: https://github.com/Fabien-Chouteau/AMCQ

Now let's see what happens when I put the candy dispenser back in the kitchen: Watch the video at https://youtu.be/nO77RFpqTaE

This chapter was originally published at https://blog.adacore.com/make-with-ada-candy-dispenser-with-twist

Make with Ada: ARM Cortex-M CNC Controller

by Fabien Chouteau
Jun 01, 2016



I started this project more than a year ago. It was supposed to be the first Make with Ada project but it became the most challenging from both, the hardware and software side. https://youtu.be/uXfkWCUyjM8

CNC and Gcode

CNC stands for Computerized Numerical Control. It's the automatic control of multi-axes machines, such as lathes, mills, laser cutters or 3D printers.

Most of the CNC machines are operated via the Gcode language. This language provides commands to move the machine tool along the different axes.

Here are some examples of commands:

```
M17 ; Enable the motors
G28 ; Homing. Go to a known position, usually at the beginning of each axis
G00 X10.0 Y20.0 ; Fast positioning motion. Move to (X, Y) position as fast as
possible
G01 X20.0 Y30.0 F50.0 ; Linear motion. Move to (X, Y) position at F speed (mm/sec)
G02/G03 X20.0 Y10.0 J-10.0 I0.0 ; Circular motion. Starting from the current
position, move to (X, Y) along the circle of center (Current_Position + (I, J))
M18 ; Disable the motors
```

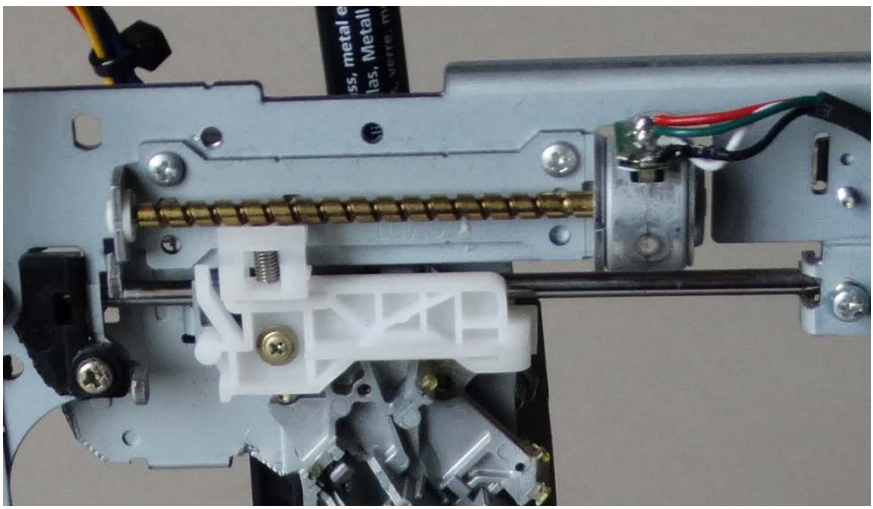
The Gcode above will produce this tool motion:

Most of the time, machinists will use a software to generate Gcode from a CAD design file. In my case, I used Inkscape (the vector graphics editor — https://inkscape.org/) and a plugin (http://wiki.inkscape.org/wiki/index.php/Extension_repository#Gcode_tools) that generates Gcode from the graphics. Here is an example of Gcode which I used with my machine: make_with_ada.gcode (https://github.com/Fabien-Chouteau/ACNC/blob/master/examples/make_with_ada.gcode).

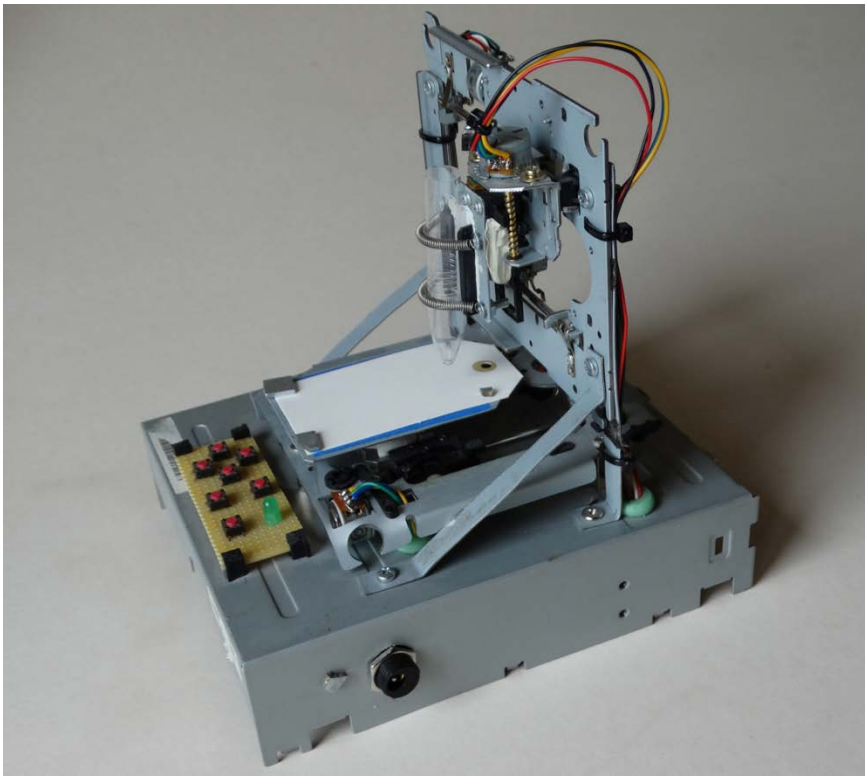
Hardware

To achieve precise movements, CNC machines use a special kind of electric motors: stepper motors. With the appropriate electronic driver, stepper motors rotate by a small fixed angle every time a step signal is received by the driver.

The rotation motion of the motor is translated to linear motion with a leadscrew. With the characteristic of both, the motor and leadscrew, we can determine the number of steps required to move one millimeter. This information is then used by the CNC controller to convert motion commands into a precise number of steps to be executed on each motor.



To create my own small CNC machine, I used tiny stepper motors from old DVD drives and floppy disk drives. I mounted them on the enclosure of one of the DVD drives. The motors are driven by low voltage stepper drivers from Pololu and the software is running on an STM32F4 Discovery.



Software

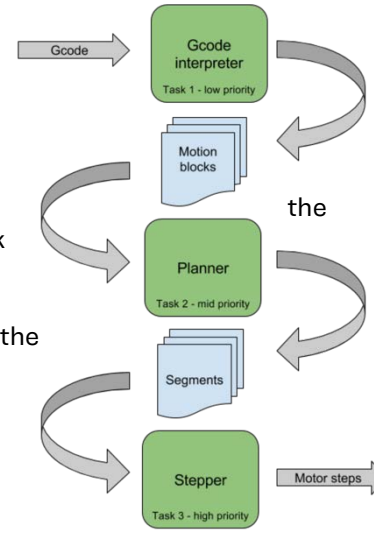
My CNC controller is greatly inspired by the Grlb project. Besides the fact that my controller is running on an STM32F4 (ARM Cortex-M4F) and Grbl is running on an Arduino (AVR 8-bit), the main difference is the use of tasking provided by the Ada language and the Ravenscar run-time.

The embedded application is running in 3 tasks:

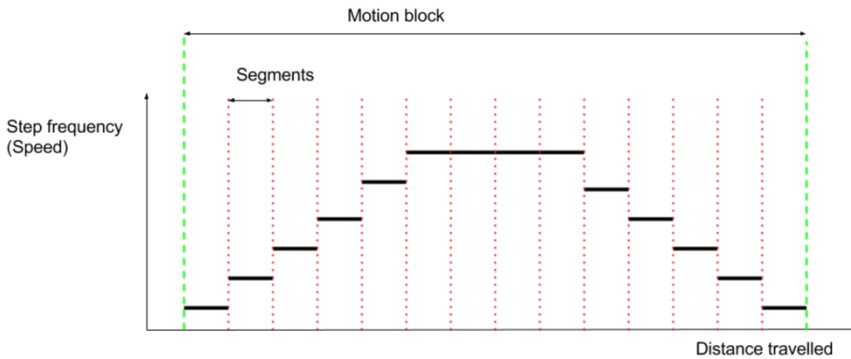
1. Gcode interpreter: This task waits for Gcode commands from the UART port, parses them and translates all the motion commands into absolute linear movements (Motion blocks). The circle interpolations are transformed into a list of linear motions that will approximate the circle.

2. The planner: This task takes motion blocks as an input and splits each one into multiple segments. A segment is a portion of a motion block with a constant speed. By setting the speed of

each segment within a block, the planner can create acceleration and deceleration profiles (as seen in the video).

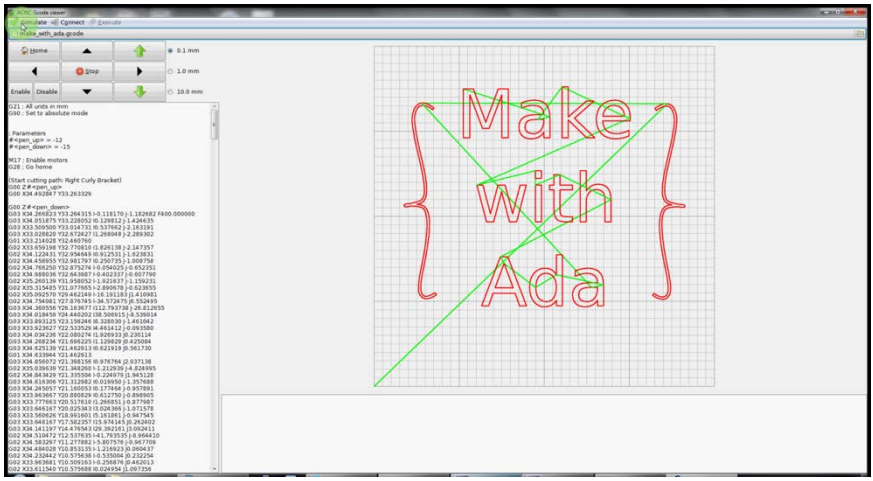


3. Stepper: This is a periodic task that will generate step signals to motors. The frequency of the task will depend on the feed rate required for the motion and the acceleration profile computed by the planner. Higher frequency means more steps per second and therefore higher motion speed.



Gcode simulator and machine interface

To be able to quickly evaluate my Gcode/Stepper algorithm and to be able to control my CNC machine, I developped a native (Linux/Windows) application.



The center part of the application shows the simulated motions from the Gcode.The top left button matrix provides manual control of the CNC machine, for example ‘move left’ by 10 mm. The left text view shows the Gcode to be simulated/executed. The bottom text view (empty on this screenshot) shows the message sent to us by the machine.

The Ada code running in the microcontroller and the code running in the simulation tool are 100% the same. This allows for very easy development, test and validation of the Gcode interpreter, motion planner and stepper algorithm.

Give me the code!!!

As with all the Make with Ada projects, the code is available on GitHub: https://github.com/Fabien-Chouteau/ACNC Fork it, build it, use it, improve it.

This chapter was originally published at https://blog.adacore.com/make-with-ada-arm-cortex-m-cnc-controller

Make with Ada: DIY Instant Camera

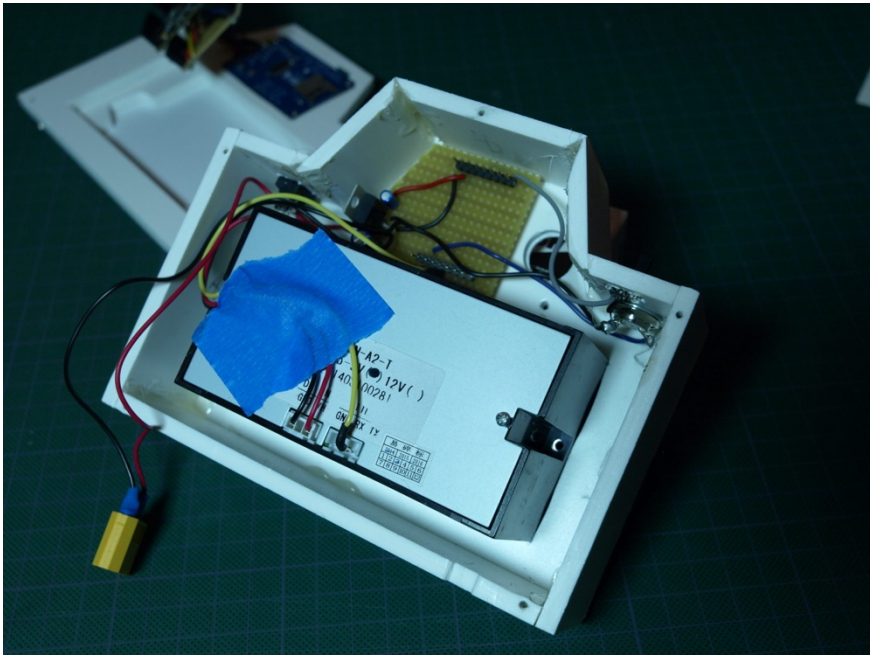
By Fabien Chouteau
Dec 12, 2016

There are moments in life where you find yourself with an AdaFruit thermal printer in one hand, and an OpenMV camerain the other. You bought the former years ago, knowing that you would do something cool with it, and you are playing with the latter in the context of a Hackaday Prize project. When that moment comes — and you know it will come — it's time to make a DIY instant camera. For me it was at the end of a warm Parisian summer day. The idea kept me awake until 5am, putting the pieces together in my head, designing an enclosure that would look like a camera. Here's the result:
https://youtu.be/5dyqzRtlMxI

The Hardware

On the hardware side, there's nothing too fancy. I use a 2 cell LiPo battery from my drone. It powers the thermal printer and a 5V

regulator. The regulator powers the OpenMV module and the LCD screen. There's a push button for the trigger and a slide switch for the mode selection, both are directly plugged in the OpenMV IOs. The thermal printer is connected via UART, while the LCD screen uses SPI. Simple.

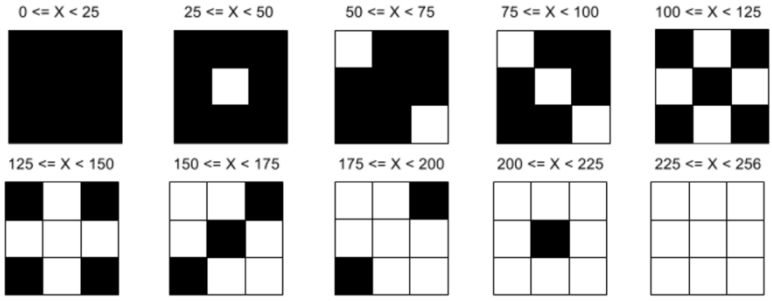


The Software

For this project I added support for the OpenMV in the Ada_Drivers_Library (https://github.com/AdaCore/Ada_Drivers_Library). It was the opportunity to do the digital camera interface (DCMI) driver as well as two Omnivision camera sensors, ST7735 LCD driver and the thermal printer.

The thermal printer is only capable of printing black or white pixel bitmap (not even gray scale), this is not great for a picture. Fortunately, the printing head has 3 times more pixels than the height of a QQVGA image, which is the format I get from the OpenMV camera. If I also multiply the width by 3, for each RGB565 pixel from the camera I can have 9 black or white pixels on the paper (from 160x120 to 480x360). This means I can use a dithering algorithm (a very naive one) to produce

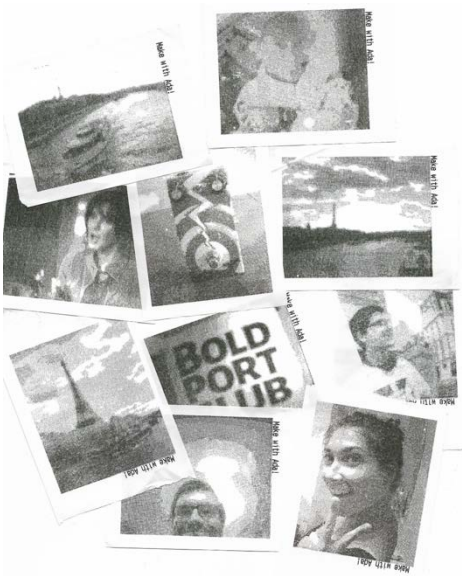
a better quality image. A pixel of grayscale value X from 0 to 255 will be transformed in a 3x3 black and white matrix, like this:



This is a quick and dirty dithering, one could greatly improve image quality by using the Floyd–Steinberg algorithm or similar (it would require more processing and more memory).

As always, the code is on GitHub (https://github.com/Fabien-Chouteau/un_pola).

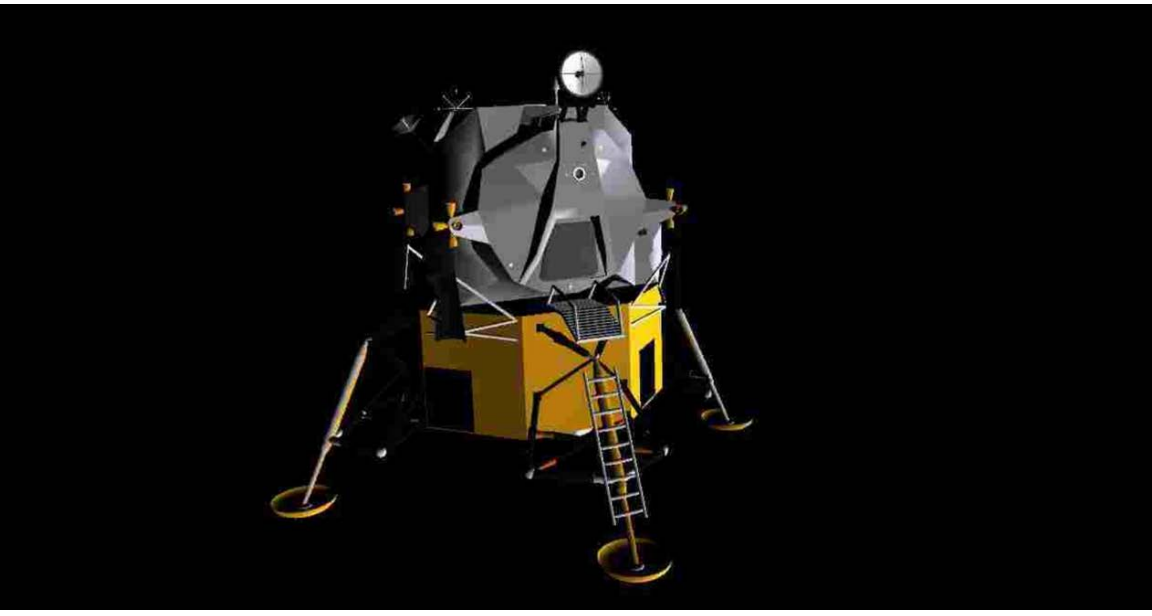
Have fun!



This chapter was originally published at https://blog.adacore.com/make-with-ada-diy-instant-camera

Driving a 3D Lunar Lander Model with ARM and Ada

by Pat Rogers
Nov 10, 2016

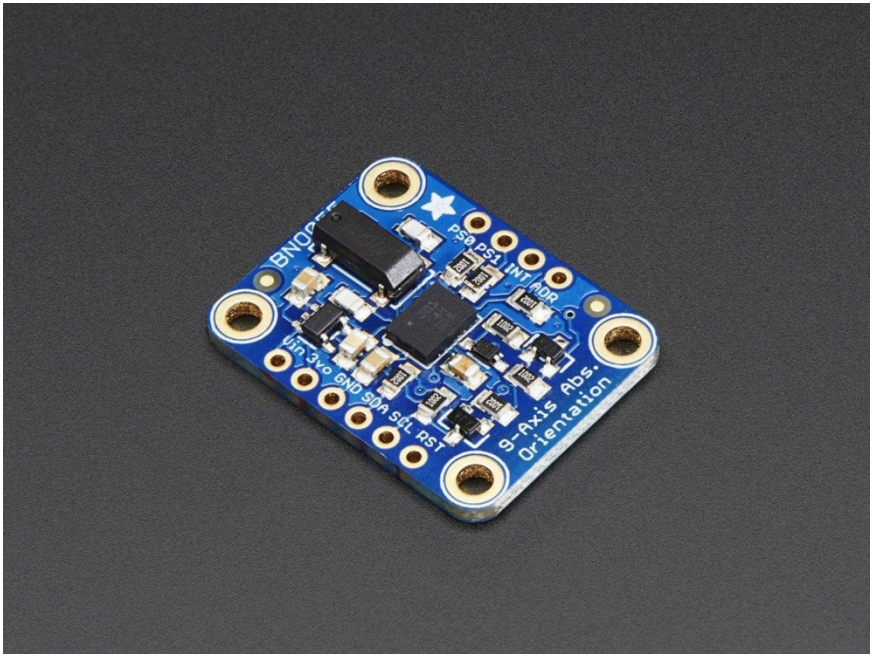


One of the interesting aspects of developing software for a bare-board target is that displaying complex application-created information typically requires more than the target board can handle. Although some boards do have amazing graphics capabilities, in some cases you need to have the application on the target interact with applications on the host. This can be due to the existence of special applications that run only (or already) on the host, in particular.

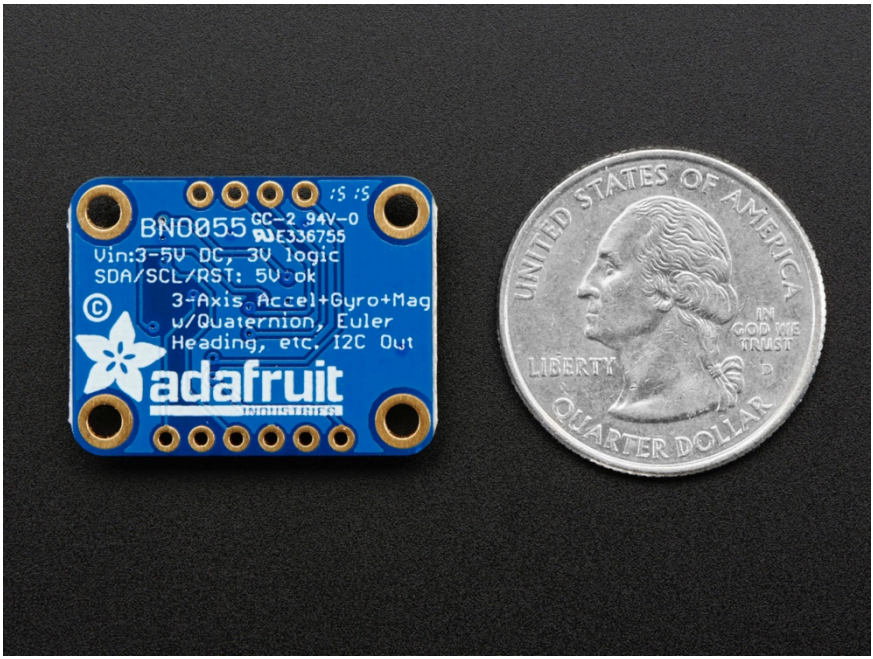
For example, I recently created an Ada driver for a 9-DOF inertial measurement unit (IMU) purchased from AdaFruit. This IMU device takes accelerometer, magnetometer, and gyroscope data inputs and produces fused values indicating the absolute orientation of the sensor

in 3D space. It is sensor-fusion on a chip (strictly, a System in Package, or SiP), obviating the need to write your own sensor fusion code. You simply configure the sensor to provide the data in the format desired and then read the Euler angles, quaternions, or vectors at the rate you require. I plan to use this orientation data in a small robot I am building that uses an STM32 Discovery board for the control system.

The device is the "AdaFruit 9-DOF Absolute Orientation IMU Fusion Breakout - BNO055" that puts the Bosch BO055 sensor chip on its own breakout board, with 3.3V regulator, logic level shifting for the I2C pins, and a Cortex-M0 to do the actual sensor data fusion.



The breakout board containing the BNO055 sensor and Cortex-M0 processor, courtesy AdaFruit



The bottom of the BNO055 breakout board and a quarter coin, for comparison, courtesy AdaFruit

See https://www.adafruit.com/products/2472 for further product details.

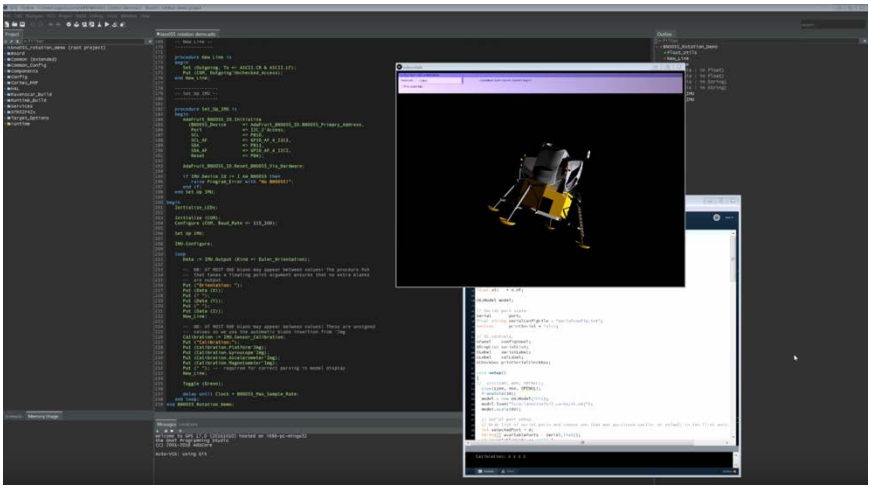
So I have easy access to fused 3D orientation data but I don't know whether those data are correct -- i.e., whether my driver is really working. I could just display the values of the three axes on the target's on-board LCD screen but that is difficult to visualize in general.

Again, AdaFruit provides a much more satisfying approach. They have a demonstration for their IMU breakout board that uses data from the sensor to drive a 3D object modeled on the host computer. As the breakout board is rotated, the modeled object rotates as well, providing instant (and much more fun) indication of driver correctness.

The current version of the demonstration is described in detail here, using a web-based app to display the model. I got an earlier version that uses the "Processing" application to display a model, and instead of using their 3D model of a cat I use a model of the Apollo Lunar Excursion Module (LEM).

The LEM model is available from NASA, but must be converted to the "obj" model data format supported by the Processing app.

The Processing app is available here for free – https://learn.adafruit.com/bno055-absolute-orientation-sensor-with-raspberry-pi-and-beaglebone-black/overview. Once downloaded and installed on the host, Processing can execute programs -- "sketches" -- that can do interesting things, including displaying 3D models. The AdaFruit demo provided a sketch for displaying their cat model. I changed the hard-coded file name to specify the LEM model and changed the relative size of the model, but that was about all that was changed.



The 3D LEM model displayed by the Processing app

The sketch gets the orientation data from a serial port, and since the sensor breakout board is connected to an STM32 Discovery board, we need that board to communicate over one of the on-board USART ports. The Ada Drivers Library includes all that is necessary for doing that, so the only issue is how to connect the board's USART port to a serial port on the host.

For that purpose I use a USB cable specifically designed to appear as a serial port on the host (e.g., a COM port on Windows). These cables are available from many vendors, including Mouser:

- Mouser Part No: 895-TTL-232R-5V

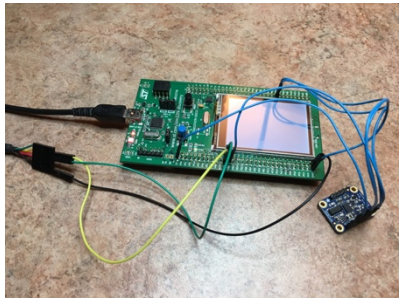
- Manufacturer Part No: TTL-232R-5V
- Manufacturer: FTDI

but note that the above is a 5-volt cable in case that is an issue. There are 3V versions available.

The end of the cable is a female header, described in the datasheet (DS_TTL-232R_CABLES-217672.pdf). Header pin 4 on the cable is TXD, the transmit data output. Header pin 5 on the cable is RXD, the receive data input. The on-board software I wrote that sends the sensor data over the port uses specific GPIO pins for the serial connection, thus I connected the cable header pins to the STMicro board's GPIO pins as follows:

- header pin 1, the black wire's header slot, to a ground pin on the board
- header pin 4, the orange wire's header slot, to PB7
- header pin 5, the yellow wire's header slot, to PB6

On the host, just plug in the USB-to-serial cable. Once the cable is connected it will appear like a host serial port and can be selected within the Processing app displaying the model. Apply power to the board and the app will start sending the orientation data.



The breakout board, STM32F429 Discovery board, and USB serial cable connections are shown in the following image. (Note that I connected a green wire to the USB cable's orange header wire because I didn't have an orange connector wire available.)

When we rotate the breakout board the LEM model will rotate accordingly, as shown in the following video:
https://youtu.be/FrhAqqUyuQ8

To display the LEM model, double-click on the "lander.pde" file to invoke the Processing app on that sketch file. Then press the Run

button in the Processing app window. That will bring up another window showing the LEM model.

In the second window showing the lander, there is a pull-down at the upper left for the serial port selection. Select the port corresponding to the USB cable attached to the STM32 board's serial port. That selection will be recorded in a file named "serialconfig.txt" located in the same directory as the model so that you don't have to select it again, unless it changes for some reason.

Note that in that second window there is a check-box labeled "Print serial data." If you enable that option you will see the IMU data coming from the breakout board via the serial port, displayed in the main Processing app window. That data includes the current IMU calibration states so that you can calibrate the IMU (by moving the IMU board slowly along three axes). When all the calibration values are "3" the IMU is fully calibrated, but you don't need to wait for that -- you can start rotating the IMU board as soon as the model window appears.

The Ada Drivers Library is available here.
https://github.com/AdaCore/Ada_Drivers_Library

The Ada source code, the 3D LEM model, and the Processing sketch file for this example are available here.
https://github.com/AdaCore/Lunar_Lander_Rotation_Demo

This chapter was originally published at https://blog.adacore.com/3d-lunar-lander-model

New Year's Resolution for 2017: Use SPARK, Say Goodbye to Bugs

By Yannick Moy
Jan 04, 2017

NIST has recently published a report called "Dramatically Reducing Software Vulnerabilities" (https://www.nist.gov/news-events/news/2016/12/safer-less-vulnerable-software-goal-new-nist-computer-publication) in which they single out five approaches which have the potential for creating software with 100 times fewer vulnerabilities than we do today. One of these approaches is formal methods. In the introduction of the document, the authors explain that they selected the five approaches that meet the following three criteria:

- Dramatic impact,
- 3 to 7-year time frame and

- Technical activities.

The dramatic impact criteria is where they aim at "reducing vulnerabilities by two orders of magnitude". The 3 to 7-year time frame was meant to select "existing techniques that have not reached their full potential for impact". The technical criteria narrowed the selection to the technical area.

Among formal methods, the report highlights strong suits of SPARK, such as "Sound Static Program Analysis" (the raison d'être of SPARK), "Assertions, Pre- and Postconditions, Invariants, Aspects and Contracts" (all of which are available in SPARK), and "Correct-by-Construction". The report also cites SPARK projects Tokeneer and iFACTS as example of mature uses of formal methods.

Another of the five approaches selected by NIST to dramatically reduce software vulnerabilities is what they call "Additive Software Analysis Techniques", where results of analysis techniques are combined. This has been on our radar since 2010 when we first planned an integration between our static analysis tool CodePeer and our formal verification toolset SPARK. We have finally achieved a first step in the integration of the two tools in SPARK 17, by using CodePeer as a first level of proof tool inside SPARK Pro.

Paul Black who lead the work on this report was interviewed a few months ago, and he talks specifically about formal methods at 7:30 in the podcast (https://www.fedscoop.com/radio/nists-paul-black/). His host Kevin Greene from US Homeland Security mentions that "There has been a lot of talk especially in the federal community about formal methods." To which Paul Black answers later that "We do have to get a little more serious about formal methods."

NIST is not the only ones to support the use of SPARK. Editor Bill Wong from Electronic Design has included SPARK in his "2016 Gifts for the Techie", saying:

> It is always nice to give something that is good for you, so here is my suggestion (and it's a cheap one): Learn SPARK. Yes, I mean that Ada programming language subset.

For those who'd like to follow NIST or Bill Wong's advice, here is where you should start:

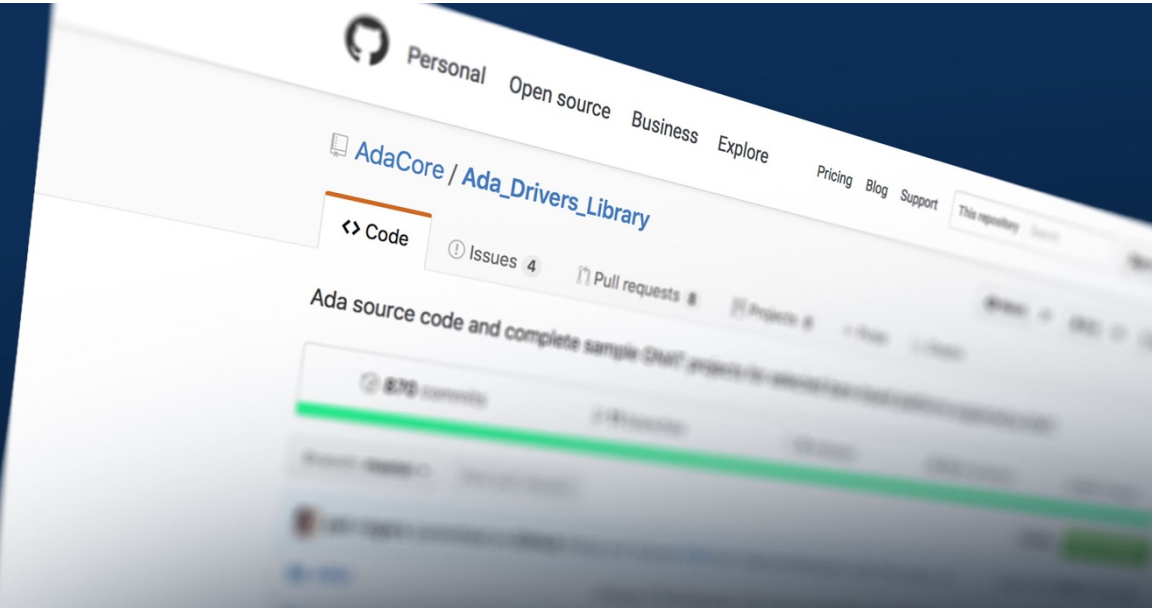
- The free SPARK Course on AdaCore U e-learning website http://university.adacore.com/courses/spark-2014/

- The online SPARK User's Guide http://docs.adacore.com/spark2014-docs/html/ug/

This chapter was originally published at https://blog.adacore.com/new-years-resolution-for-2017-no-bugs-with-spark

Getting Started with the Ada Driver's Library Device Drivers

By Pat Rogers
Feb 14, 2017



The Ada Drivers Library (ADL) is a collection of Ada device drivers and examples for ARM-based embedded targets. The library is maintained by AdaCore, with development originally (and predominantly) by AdaCore personnel but also by the Ada community at large. It is available on GitHub (https://github.com/AdaCore/Ada_Drivers_Library)and is licensed for both proprietary and non-proprietary use.

The ADL includes high-level examples in a directory at the top of the library hierarchy. These examples employ a number of independent components such as cameras, displays, and touch

screens, as well as middleware services and other device-independent interfaces. The stand-alone components are independent of any given target platform and appear in numerous products. A previous blog entry (http://blog.adacore.com/3d-lunar-lander-model) examined one such component, the Bosch BLO055 inertial measurement unit IMU). Other examples show how to create high-level abstractions from low-level devices. For instance, one shows how to create abstract data types representing serial ports.

In this entry we want to highlight another extremely useful resource: demonstrations for the low-level device drivers. Most of these drivers are for devices located within the MCU package itself, such as GPIO, UART/USART, DMA, ADC and DAC, and timers. Other demonstrations are for some of the stand-alone components that are included in the supported target boards, for example gyroscopes and accelerometers. Still other demonstrations are for vendor-defined hardware such as a random number generator.

These demonstrations show a specific utilization of a device, or in some cases, a combination of devices. As such they do not have the same purpose as the high-level examples. They may just display values on an LCD screen or blink LEDs. Their purpose is to provide working examples that can be used as starting points when incorporating devices into client applications. As working driver API references they are invaluable.

Approach

Typically there are multiple, independent demonstration projects for each device driver because each is intended to show a specific utilization. For example, there are five distinct demonstrations for the analog-to-digital conversion (ADC) driver. One shows how to set up the driver to use polling to get the converted value. Another shows how to configure the driver to use interrupts instead of polling. Yet another shows using a timer to trigger the conversions, and another builds on that to show the use of DMA to get the converted values to the user. In each case we simply display the resulting values on an LCD screen rather than using them in some larger application-oriented sense.

Some drivers, the I2C and SPI communication drivers specifically, do not have dedicated demonstrations of their own. They are used to implement drivers for devices that use those protocols, i.e., the drivers for the stand-alone components. The Bosch BLO055 IMU mentioned earlier is an example.

Some of the demonstrations illustrate vendor-specific capabilities beyond typical functionality. The STM32 timers, for example, have direct support for quadrature motor encoders. This support provides CPU-free detection of motor rotation to a resolution of a fraction of a degree. Once the timer is configured for this purpose the application merely samples a register to get the encoder count. The timer will even provide the rotation direction. See the encoder demonstration (https://github.com/AdaCore/Ada_Drivers_Library/tree/master/arch/ARM/STM32/driver_demos/demo_timer_quad_encoder) if interested.

Implementation

All of the drivers and demonstration programs are written in Ada 2012. They use preconditions and postconditions, especially when the driver is complicated. The preconditions capture API usage requirements that are otherwise expressed only within the documentation, and sometimes not expressed at all. Similarly, postconditions help clients understand the effects of calls to the API routines, effects that are, again, only expressed in the documentation. Some of the devices are highly sophisticated -- a nice way of saying blindingly complicated -- and their documentation is complicated too. Preconditions and postconditions provide an ideal means of capturing information from the documentation, along with overall driver usage experience. The postconditions also help with the driver implementation itself, acting as unit tests to ensure implementer understanding. Other Ada 2012 features are also used, e.g., conditional and quantified expressions.

The STM32.Timers package uses preconditions and postconditions extensively because the STM timers are "highly sophisticated." STM provides several kinds of timer with significantly different capabilities. Some are defined as "basic," some "advanced," and others are "general purpose." The only way to know which is which is by the timer naming scheme ("TIM" followed by a number) and the documentation. Hence TIM1 and TIM8 are advanced timers, whereas TIM6 and TIM7 are basic timers. TIM2 through TIM5 are general purpose timers but not the same as TIM9 through TIM14, which are also general purpose. We use preconditions and postconditions to help keep it all straight. For example, here is the declaration of the routine for enabling an interrupt on a given timer. There are several timer interrupts possible, represented by the enumeration type Timer_Interrupt. The issue is that basic timers can only have one of the possible interrupts specified, and only advanced timers can have two of those possible. The preconditions express those restrictions to clients.

```
procedure Enable_Interrupt
   (This   : in out Timer;
    Source : Timer_Interrupt)
with
   Pre =>
      (if Basic_Timer (This) then Source = Timer_Update_Interrupt) and
      (if Source in Timer_COM_Interrupt | Timer_Break_Interrupt then Advanced_Timer
(This)),
   Post => Interrupt_Enabled (This, Source);
```

The preconditions reference Boolean functions Basic_Timer and Advanced_Timer in order to distinguish among the categories of timers. They simply compare the timer specified to a list of timers in those categories.

The postcondition tells us that the interrupt will be enabled after the call returns. That is useful for the user but also for the implementer because it serves as an actual check that the implementation does what is expected. When working with hardware, though, we have to keep in mind that the hardware may clear the tested condition before the postcondition code is called. For example, a routine may set a bit in a register in order to make the attached device do something, but the device may clear the bit as part of its response. That would likely happen before the postcondition code could check that the bit was set. When looking throughout the drivers code you may notice some "obvious" postconditions are not specified. That may be the cause.

The drivers use compiler-dependent facilities only when essential. In particular, they use an AdaCore-defined aspect specifying that access to a given memory-mapped register is atomic even when only one part of it is read or updated. This access reflects the hardware requirements and simplifies the driver implementation code considerably.

Organization

The device driver demonstrations are vendor-specific because the corresponding devices exist either within the vendor-defined MCU packages or outside the MCU on the vendors' target boards. The first vendor supported by the library was STMicroelectroncs (STM), although other vendors are beginning to be represented too. As a result, the device driver demonstrations are currently for MCU products and boards from STM and are, therefore, located in a library subdirectory specific to STM. Look for them in the /Ada_Drivers_Library/ARM/STM32/driver_demos/ subdirectory of

your local copy from GitHub. There you will see some drivers immediately. These are for drivers that are shared across an entire MCU family. Others are located in further subdirectories containing either a unique device's driver, or devices that do exist across multiple MCUs but nonetheless differ in some significant way.

Let's look at one of the demonstration projects, the "demo_LIS3DSH_tilt" project, so that we can highlight the more important parts. This program demonstrates basic use of the LIS3DSH accelerometer chip. The four LEDs surrounding the accelerometer will come on and off as the board is moved, reflecting the directions of the accelerations measured by the device.

The first thing to notice is the "readme.md" file. As you might guess, this file explains what the project demonstrates and, if necessary, how to set it up. In this particular case the text also mentions the board that is intended for execution, albeit implicitly, because the text mentions the four LEDs and an accelerometer that are specific to one of the STM32 Discovery boards. In other words, the demo is intended for a specific target board. At the time of this writing, all the STM demonstration projects run on either the STM32F4 or the STM32F429I Discovery boards from STMicroelectronics. They are very inexpensive, amazingly powerful boards. Some demonstrations will run on either one because they do not use board-specific resources.

But even if a demonstration does not require a specific target board, it still matters which board you use because the demo's project file (the "gpr file") specifies the target. If you use a different target the executable will download but may not run correctly, perhaps not at all.

The executable may not run because the specified target's runtime library is used to build the binary executable. These libraries have configurations that reflect the differences in the target board, especially memory and clock rates, so using the runtime that matches the board is critical. This is the first thing to check when the board you are using simply won't run the demonstration at all.

The demonstration project file specifies the target by naming another project in a with-clause. This other project represents a specific target board. Here is the elided content of this demonstration's project file. Note the second with-clause that specifies a gpr file for the STM32F407 Discovery board. That is one of the two lines to change if you want to use the F429I Discovery instead.

```
with "../../../../boards/common_config.gpr";
with "../../../../boards/stm32f407_discovery.gpr";

project Demo_LIS3DSH_Tilt extends "../../../../examples/common/common.gpr" is

   ...
   for Runtime ("Ada") use STM32F407_Discovery'Runtime("Ada");
   ...

end Demo_LIS3DSH_Tilt;
```

The other line to change in the project file is the one specifying the "Runtime" attribute. Note how the the value of the attribute is specified in terms of another project's Runtime attribute. That other project is the one named in the second with-clause, so when we change the with-clause we must change the name of the referenced project too.

That's really all you need to change in the gpr file. GPS and the builder will handle everything else automatically.

There is, however, another effect of the with-clause naming a specific target. The demonstration programs must refer to the target MCU in order to use the devices in the MCU package. They may also need to refer to devices on the target board. Different MCU packages have differing numbers of devices (eg, USARTs) in the package. Similarly, different boards have different external components (accelerometers versus gyroscopes, for example). We don't want to limit the code in the ADL to work with specific boards, but that would be the case if the code referenced the targets by name, via packages representing the specific MCUs and boards. Therefore, the ADL defines two packages that represent the MCU and the board indirectly. These are the STM32.Device and STM32.Board packages, respectively. The indirection is then resolved by the gpr file named in the with-clause. In this demonstration the clause names the STM32F407_Discovery project so that is the target board represented by the STM32.Board package. That board uses an STM32F407VG MCU so that is the MCU represented by the STM32.Device package. Each package contains declarations for objects and constants representing the specific devices on that specific MCU and target board.

You'll also see a file named ".gdbinit" at the same level as the readme.md and gpr files. This is a local gdb script that automatically resets the board when debugging. It is convenient but not essential.

At that same level you'll also see a "gnat.adc" file containing configuration pragmas. These files contain a single pragma that ensures all interrupt handlers are elaborated before any interrupts can trigger them, among other things. It is not essential for the correct function of these demonstrations but is a good idea in general.

Other than those files you'll see subdirectories for the source files and compiler's products (the object and ALI files, and the executable file).

And that's it. Invoke GPS on the project file and everything will be handled by the IDE.

Application Use

We mentioned that you must change the gpr file if you want to use a different target board. That assumes you are running the demonstration programs themselves. There is no requirement that you do so. You could certainly take the bulk of the code and use it on some other target that has the same MCU family inside. That's the whole point of the demonstrations: showing how to use the device drivers! The Certyflie project (https://github.com/AdaCore/Certyflie), also on the AdaCore GitHub, is just such a project. It uses these device drivers so it uses an STM32.Device package for the on-board STM32F405 MCU, but the target board is a quad-copter instead of one of the Discovery kits.

Concluding Remarks

Finally, it must be said that not all available devices have drivers in the ADL, although the most important do. More drivers and demonstrations are needed. For example, the hash processor and the cryptographic processor on the STM Cortex-M4 MCUs do not yet have drivers. Other important drivers are missing as well. CAN and Ethernet support is either minimal or lacking entirely. And that's not even mentioning the other vendors possible. We need the active participation of the Ada community and hope you will join us!

This chapter was originally published at
https://blog.adacore.com/getting-started-with-the-ada-drivers-library-device-drivers

SPARK Tetris on the Arduboy

By Fabien Chouteau, Arnaud Charlet, Yannick Moy

One of us got hooked on the promise of a credit-card-size programmable pocket game under the name of Arduboy and participated in its kickstarter in 2015. The kickstarter was successful (but late) and delivered the expected working board in mid 2016. Of course, the idea from the start was to program it in Ada , but this is an 8-bits AVR microcontroller (the ATmega32u4 by Atmel) not supported anymore by GNAT Pro. One solution would have been to rebuild our own GNAT compiler for 8-bit AVR from the GNAT FSF repository and use the AVR-Ada project. Another solution, which we explore in this blog post, is to use the SPARK-to-C compiler that we developed at AdaCore to turn our Ada code into C and then use the Arduino toolchain to compile for the Arduboy board.

This is in fact a solution we are now proposing to those who need to compile their code for a target where we do not propose an Ada compiler, in particular small microcontrollers used in industrial automation and automotive industries. Thanks to SPARK-to-C, you can

now develop your code in SPARK, compile it to C, and finally compile the generated C code to your target. We have built the universal SPARK compiler! This product will be available to AdaCore customers in the coming months.

We started from the version of Tetris in SPARK that we already ported to the Atmel SAM4S, Pebble-Time smartwatch and Unity game engine. For the details on what is proved on Tetris, see the recording of a talk at FOSDEM 2017 conference (https://fosdem.org/2017/schedule/event/spark/). The goal was to make this program run on the Arduboy.

SPARK-to-C Compiler

What we call the SPARK-to-C compiler in fact accepts both less and more than SPARK language as input. It allows pointers (which are not allowed in SPARK) but rejects tagged types and tasking (which are allowed in SPARK). The reason this is the case is that it's easy to compile Ada pointers into C pointers but much harder to support object oriented or concurrent programming.

SPARK-to-C supports, in particular, all of Ada's scalar types (enumerations, integers, floating-point, fixed-point, and access) as well as records and arrays and subtypes of these. More importantly, it can generate all the run-time checks to detect violations of type constraints such as integer and float range checks and checks for array accesses out of bounds and access to a null pointer or invalid pointer. Therefore, you can program in Ada and get the guarantee that the executable compiled from the C code generated by SPARK-to-C preserves the integrity of the program, as if you had compiled it directly from Ada with GNAT.

Compiling Ada into C poses interesting challenges. Some of them are resolved by following the same strategy used by GNAT during compilation to binary code. For example, bounds of unconstrained arrays are bundled with the data for the array in so-called "fat pointers", so that both code that directly references Array'First and Array'Last as well as runtime checks for array accesses can access the array bounds in C. This is also how exceptions, both explicit in the code and generated for runtime checks, are handled. Raising an exception is translated into a call to the so-called "last chance handler", a function provided by the user that can perform some logging before terminating the program. This is exactly how exceptions are handled in Ada for targets that don't have runtime support. In general, SPARK-to-C provides very little runtime support, mostly for numerical computations

(sin, cosine, etc.), accessing a real time clock, and outputting characters and strings. Other features require specific source-to-source transformations of Ada programs. For example, functions that return arrays in Ada are transformed into procedures with an additional output parameter (a pointer to some preallocated space in the caller) in C.

The most complex part of SPARK-to-C deals with unnesting nested subprograms because, while GCC supports nested functions as an extension, this is not part of standard C. Hence C compilers cannot be expected to deal with nested functions. Unnesting in SPARK-to-C relies on a tight integration of a source-to-source transformation of Ada code in the GNAT frontend, with special handling of nested subprograms in the C-generation backend. Essentially, the GNAT frontend creates an 'activation record' that contains a pointer field for each uplevel variable referenced in the nested subprogram. The nested subprogram is then transformed to reference uplevel variables through the pointers in the activation record passed as additional parameters. A further difficulty is making this work for indirect references to uplevel variables and through references to uplevel types based on these variables (for example the bound of an array type). SPARK-to-C deals also with these cases: you can find all details in the comments of the compiler file exp_unst.ads

Compiling Tetris from SPARK to C

Once SPARK-to-C is installed, the code of Tetris can be compiled into C with the version of GPRbuild that ships i SPARK-to-C:

```
$ gprbuild -P<project> --target=c
```

For example, the SPARK expression function Is_Empty from Tetris code:

```
function Is_Empty (B : Board; Y : Integer; X : Integer) return Boolean is
      (X in X_Coord and then Y in Y_Coord and then B(Y)(X) = Empty);
```

is compiled into the C function tetris_functional__is_empty, with explicit checking of array bounds before accessing the board:

```
boolean tetris_functional__is_empty(tetris_functional__board b, integer y, integer x)
{
  boolean C123s = false;
  if ((x >= 1 && x <= 10) && (y >= 1 && y <= 50)) {
    if (!((integer)y >= 1 && (integer)y <= 50))
      __gnat_last_chance_handler(NULL, 0);
    if (!((integer)x >= 1 && (integer)x <= 10))
      __gnat_last_chance_handler(NULL, 0);
    if ((b)[y - 1][x - 1] == tetris_functional__empty) {
      C123s = true;
    }
  }
  return (C123s);
}
```

or into the following simpler C function when using compilation switch -gnatp to avoid runtime checking:

```
boolean tetris_functional__is_empty(tetris_functional__board b, integer y, integer x)
{
  return (((x >= 1 && x <= 10) && (y >= 1 && y <= 50)) && ((b)[y - 1][x - 1] ==
tetris_functional__empty));
}
```

Running on Arduboy

To interface the SPARK Tetris implementation with the C API of the Arduboy, we use the standard language interfacing method of SPARK/Ada:

```
procedure Arduboy_Set_Screen_Pixel (X : Integer; Y : Integer);
pragma Import (C, Arduboy_Set_Screen_Pixel, "set_screen_pixel");
```

A procedure Arduboy_Set_Screen_Pixel is declared in Ada but not implemented. The pragma Import tells the compiler that this procedure is implemented in C with the name “set_screen_pixel”.

SPARK-to-C will translate calls to the procedure “Arduboy_Set_Screen_Pixel” to calls to the C

function “set_screen_pixel”. We use the same technique for all the subprograms that are required for the game (button_right_pressed, clear_screen, game_over, etc.).

The program entry point is in the Arduino sketch file SPARK_Tetris_Arduboy.ino (link). In this file, we define and export the C functions (set_screen_pixel() for instance) and call the SPARK/Ada code with _ada_main_tetris().

It’s that simple :)

If you have an Arduboy, you can try this demo by first...

Follow the quick start guide
http://community.arduboy.com/t/quick-start-guide/2790

Download the project from GitHub
https://github.com/AdaCore/SPARK-to-C_Tetris_Demo

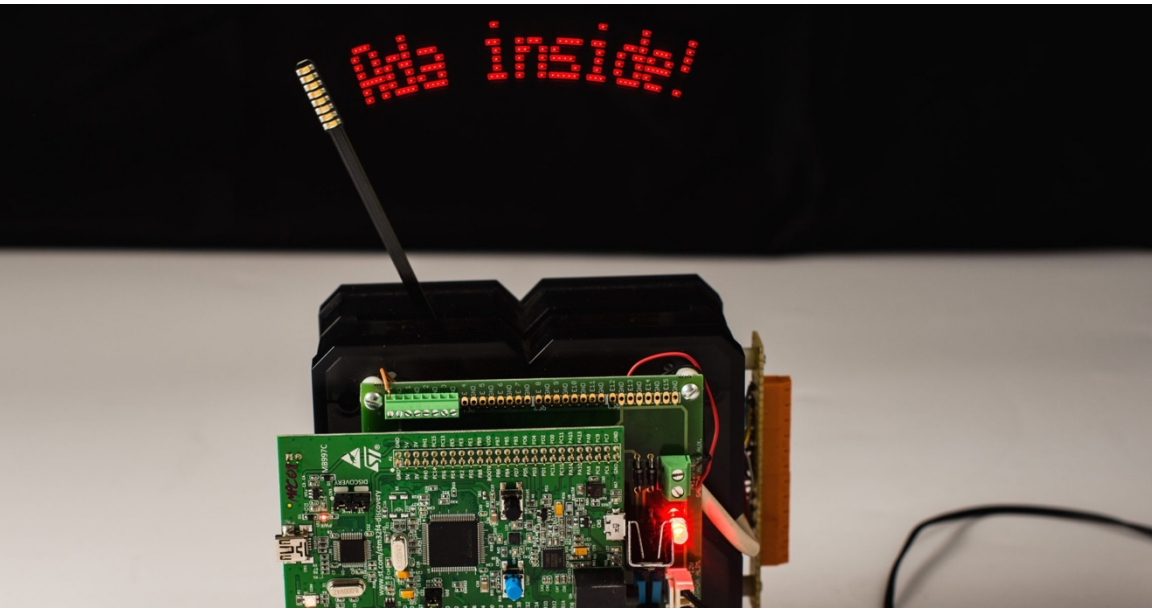
Load the Arduino sketch SPARK_Tetris_Arduboy
https://github.com/AdaCore/SPARK-to-C_Tetris_Demo/tree/master/SPARK_Tetris_Arduboy

And then click the upload button.

This chapter was originally published at
https://blog.adacore.com/spark-tetris-on-the-arduboy

Writing on Air

By Jorge Real
March 27, 2017



While searching for motivating projects for students of the Real-Time Systems course here at Universitat Politècnica de València, we found a curious device that produces a fascinating effect. It holds a 12 cm bar from its bottom and makes it swing, like an upside-down pendulum, at a frequency of nearly 9 Hz. The free end of the bar holds a row of eight LEDs. With careful and timely switching of those LEDs, and due to visual persistence, it creates the illusion of text... floating in the air!

The web shows plenty of references to different realizations of this idea. They are typically used for displaying date, time, and also rudimentary graphics. Try searching for "pendulum clock LED", for example. The picture in Figure 1 shows the one we are using.

The software behind this toy is a motivating case for the students, and it contains enough real-time and synchronisation requirements to also make it challenging.

We have equipped the lab with a set of these pendulums, from which we have disabled all the control electronics and replaced them with STM32F4 Discovery boards. We use also a self-made interface board (behind the Discovery board in Figure 1) to connect the Discovery with the LEDs and other relevant signals of the pendulum. The task we propose our students is to make it work under the control of a Ravenscar program running on the Discovery. We use GNAT GPL 2016 for ARM hosted on Linux, along with the Ada Drivers Library (https://github.com/AdaCore/Ada_Drivers_Library).

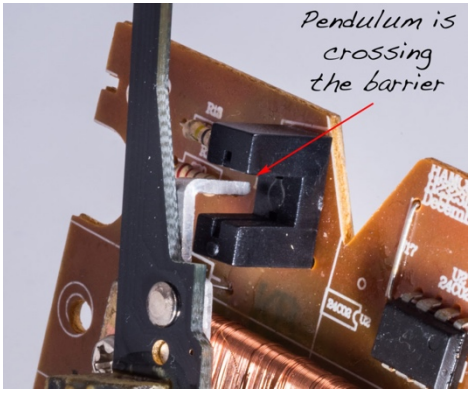
There are two different problems to solve: one is to make the pendulum bar oscillate with a regular period; the other one is to then use the LEDs to display some text.

Swing that bar!

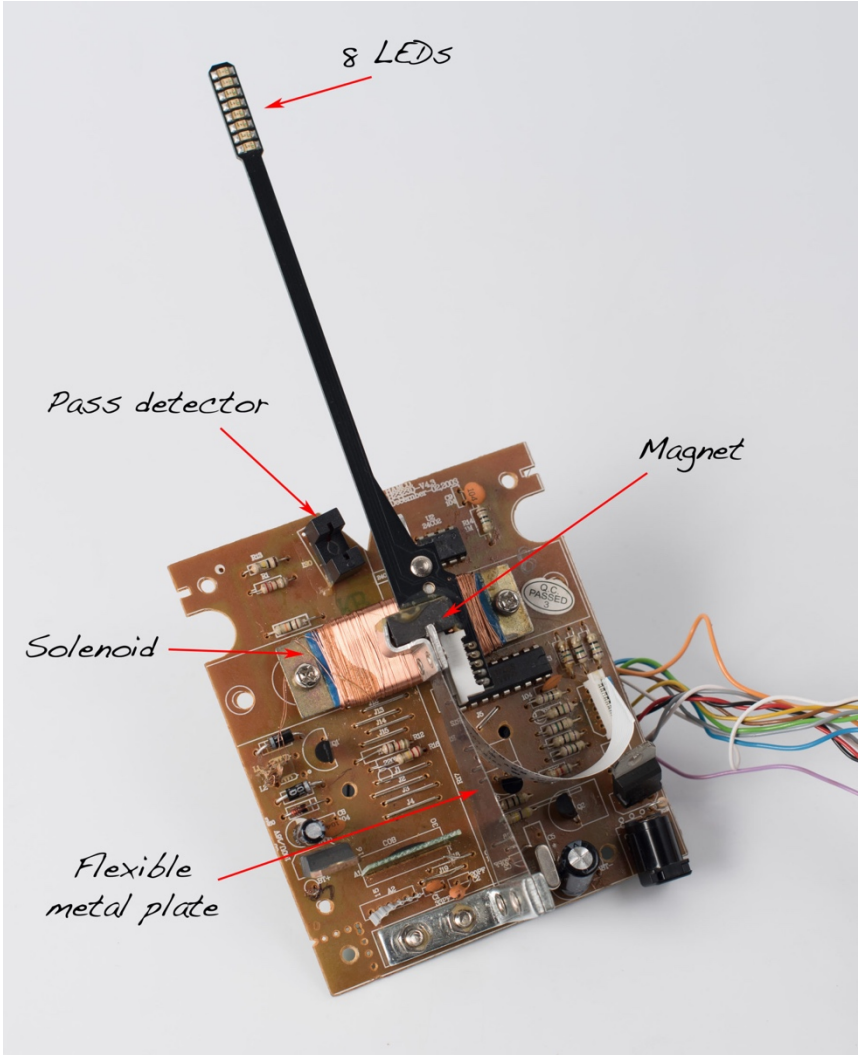
The bar is fixed from the bottom to a flexible metal plate (see Figure 2). The stable position of the pendulum is vertical and still. There is a permanent magnet attached to the pendulum, so that the solenoid behind it can be energised to cause a repulsion force that makes the bar start to swing.

At startup, the solenoid control is completely blind to the whereabouts of the pendulum. An initial sequence must be programmed with the purpose of moving the bar enough to make it cross the barrier (see detail in Figure 3), a pass detector that uses an opto-coupler sensor located slightly off-center the pendulum run. This asymmetry is crucial, as we'll soon see.

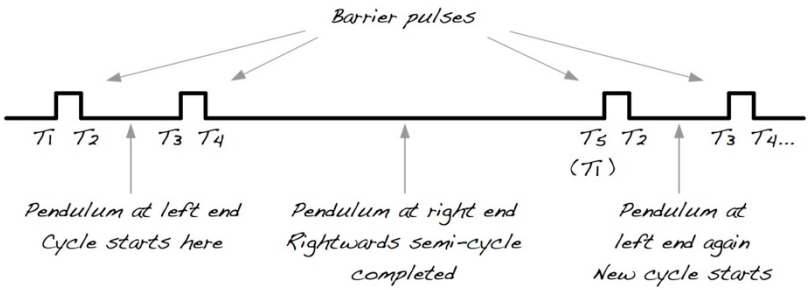
Once the bar crosses the barrier at least three times, we have an idea about the pendulum position along time and we can then apply a more precise control sequence to keep the pendulum swinging regularly. The situation is pretty much like swinging a kid swing: you need to give it a small, regular push, at the right time. In our case, that time is when the pendulum enters the solenoid area on its way to the right side, since the solenoid repels the pendulum rightwards. That happens at about one sixth of the pendulum cycle, so we first need to know when the cycle starts and what duration



it has. And for that, we need to pay close attention to the only input of the pendulum: the barrier signal.



The figure below sketches a chronogram of the barrier signal. Due to its asymmetric placement, the signal captured from the opto-coupler is also asymmetric.



Chronogram of the barrier signal and correspondence with extreme pendulum positions

To determine the start time and period of the next cycle, we take note of the times when rising and falling edges of the barrier signal occur. This is easy work for a small Finite State Machine (FSM), triggered by barrier interrupts to the Discovery board. Once we have collected the five edge times T1 to T5 (normally would correspond to 2 full barrier crossings plus the start of a third one) we can calculate the period by subtracting T5 - T1. Regarding the start time of the next cycle, we know the pendulum initiated a new cycle (reached its left-most position) just in between the two closest pulses (times T1 and T4). So, based on the information gathered, we estimate that the next cycle will start at time T5 + (T4 - T1) / 2.

But... all we know when we detect a barrier edge is whether it is rising or falling. So, when we detect the first rising edge of Barrier, we can't be sure whether it corresponds to T1 (the second barrier crossing) or T3 (the first). We have arbitrarily guessed it is T1, so we must verify this guess and fix things if it was incorrect. This check is possible precisely due to the asymmetric placement of the pass detector: if our guess was correct, then T3 - T1 should be less than T5 - T3. Otherwise we need to re-assign our measurements (T3, T4 and T5 become T1, T2 and T3) and then move on to the adequate FSM state (waiting for T4).

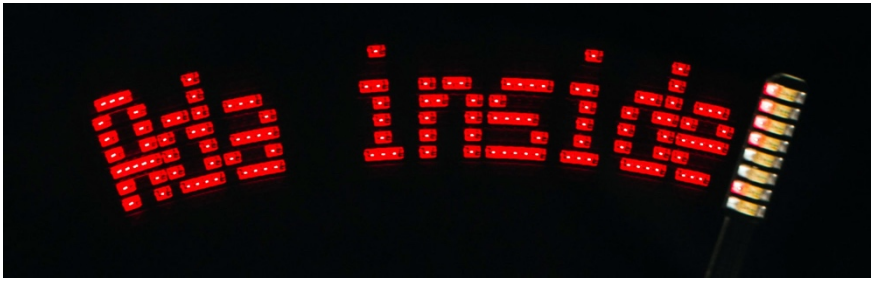
Once we know when the pendulum will be in the left-most position (the cycle start time) and the estimated duration of the next cycle, we can give a solenoid pulse at the cycle start time plus one sixth of the period. The pulse duration, within reasonable range, affects mostly the amplitude of the pendulum run, but not so much its period. Experiments with pulse durations between 15 and 38 milliseconds showed visible changes in amplitude, but period variations of only

about 100 microseconds, for a period of 115 milliseconds (less than 0.1%). We found 18-20 ms to work well.

So, are we done with the pendulum control? Well... almost, but no, we're not: we are also concerned by robustness. The software must be prepared for unexpected situations, such as someone or something suddenly stopping the bar. If our program ultimately relies on barrier interrupts and they do not occur, then it is bound to hang. A timeout timing event is an ideal mechanism to revive a dying pendulum. If the timeout expires, then the barrier-based control is abandoned and the initialisation phase engaged again, and again if needed, until the pendulum makes sufficient barrier crossings to let the program retake normal operation. After adding this recovery mechanism, we can say we are done with the pendulum control: the bar will keep on swinging while powered.

Adding lyrics to that swing

Once the pendulum is moving at a stable rate, we are ready to tackle the second part of the project: using the eight LEDs to display some text. Knowing the cycle start time and estimated period duration, one can devise a plan to display each line of a character at the proper period times. We have already calculated the next cycle start time and duration for the pendulum control. All we need to do now is to timely provide that information to a displaying task.



Time to display an exclamation mark!

The pendulum control functions described above are implemented by a package with the following (abbreviated) specification:

```
with STM32F4;       use STM32F4;
with Ada.Real_Time; use Ada.Real_Time;

package Pendulum_IO is

   --  Set LEDs using byte pattern (1 => On, 0 => Off)
   procedure Set_LEDs (Pattern : in Byte);

   --  Synchronization point with start of new cycle
   procedure Wait_For_Next_Cycle (Init_Time      : out Time;
                                  Cycle_Duration : out Time_Span);

private
      task P_Controller with Storage_Size => 4 * 1024;
end Pendulum_IO;
```

The specification includes subprograms for setting the LEDs (only one variant shown here) and procedure Wait_For_Next_Cycle, which in turn calls a protected entry whose barrier (in the Ada sense, this time) is opened by the barrier signal interrupt handler, when the next cycle timing is known. This happens at time T5 (see Figure 4), when the current cycle is about to end but with sufficient time before the calling task must start switching LEDs. The P_Controller task in the private part is the one in charge of keeping the pendulum oscillating.

Upon completion of a call to Wait_For_Next_Cycle, the caller knows the start time and period of the next pendulum cycle (parameters Init_Time and Cycle_Period). By division of the period, we can also determine at what precise times we need to switch the LEDs. Each character is encoded using an 8 tall x 5 wide dot matrix, and we want to fit 14 characters in the display. Adding some left and right margins to avoid the slowest segments, and a blank space to the right of each character, we subdivide the period in 208 lines. These lines represent time windows to display each particular character chunk. Since the pendulum period is around 115 milliseconds, it takes just some 550 microseconds for the pendulum to traverse one line.

If that seems tight, there is an even tighter requirement than this inter-line delay. The LEDs must be switched on only during an interval between 100 and 200 microseconds. Otherwise we would see segments, rather than dots, due to the pendulum speed. This must also be taken into account when designing the plan for the period, because the strategy changes slightly depending on the current pendulum direction. When it moves from left to right, the first 100 microseconds

of a line correspond to it's left part, whereas the opposite is true for the opposite direction.

Dancing with the runtime

Apart from careful planning of the sequence to switch the LEDs, this part is possibly less complex, due to the help of Wait_For_Next_Cycle. However, the short delays imposed by the pendulum have put us in front of a limitation of the runtime support. The first try to display some text was far from satisfactory. Often times, dots became segments. Visual glitches happened all the time as well. Following the track to this issue, we ended up digging into the Ravenscar runtime (the full version included in GNAT GPL 2016) to eventually find that the programmed precision for timing events and delay statements was set to one millisecond. This setting may be fine for less demanding applications, and it causes a relatively low runtime overhead; but it was making it impossible for us to operate within the pendulum's tight delays. Things started to go well after we modified and recompiled the runtime sources to make delays and timing events accurate to 10 microseconds. It was just a constant declaration, but it was not trivial to find it! Definitely, this is not a problem we ask our students to solve: they use the modified runtime.

If you come across the same issue and the default accuracy of 1 millisecond is insufficient for your application, look for the declaration of constant Tick_Period in the body of package System.BB.Board_Support (file s-bbbosu.adb in the gnarl-common part of either the full or the small footprint versions of the runtime). For an accuracy of 10 microseconds, we set the constant to Clock_Frequency / 100_000.

More fun

There are many other things that can be done with the pendulum, such as scrolling a text longer than the display width, or varying the scrolling speed by pressing the user button in the Discovery board (both features are illustrated in the video below, best viewed in HD); or varying the intensity of the text by changing the LEDs flashing time; or displaying graphics rather than just text...

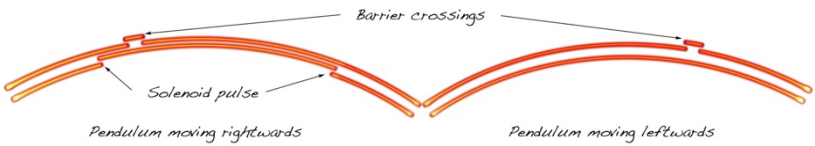
See the video at https://youtu.be/h5K41rovKmI

One of the uses we have given the pendulum is as a chronometer display for times such as the pendulum period, the solenoid pulse width, or other internal program delays. This use has proved very

helpful to better understand the process at hand and also to diagnose the runtime delay accuracy issue.

The pendulum can also be used as a rudimentary oscilloscope. Figure 6 shows the pendulum drawing the chronograms of the barrier signal and the solenoid pulse. The top two lines represent these signals, respectively, as the pendulum moves rightwards. The two bottom lines are for the leftwards semi-period and must be read leftwards. In Figure 7, the two semi-periods are chronologically re-arranged. The result cannot be read as in a real oscilloscope, because of the varying pendulum speed; but knowing that, it is indicative.

Pendulum used as an oscilloscope (original image)



Oscilloscope image, chronologically re-arranged

Credit, where it's due

My colleague Ismael Ripoll was the one who called my attention to the pendulum, back in 2005. We implemented the text part only (we did not disconnect the solenoid from the original pendulum's microcontroller). Until porting (and extending) this project to the Discovery board, we've been using an industrial PC with a digital I/O card to display text in the pendulum. The current setup is about two orders of magnitude cheaper. And it also fits much better the new focus of the subject on real-time and also embedded systems.

I'm thankful to Vicente Lliso, technician at the DISCA department of UPV, for the design and implementation of the adapter card connecting the Discovery board with the pendulum, for his valuable comments and for patiently attending my requests for small changes here and there.

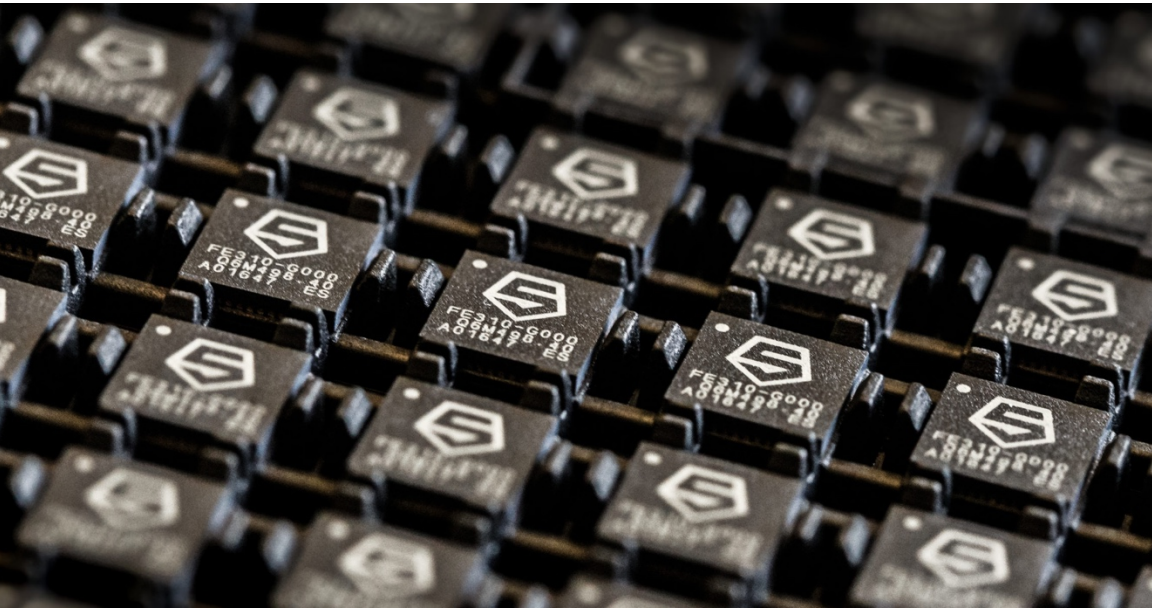
My friend and amateur photographer Álvaro Doménech produced excellent photographical material to decorate this entry, as well as the pendulum video. Álvaro is however not to be blamed for the oscilloscope pictures, which I took with just a mobile phone camera.

And many thanks to Pat Rogers, from AdaCore, who helped me with the delay accuracy issue and invited me to write this entry. It was one of Pat's excellent tutorials at the Ada-Europe conference that pushed me into giving a new angle to this pendulum project... and to others in the near future!

This chapter was originally published at
https://blog.adacore.com/writing-on-air

Ada on the First RISC-V Microcontroller

By Fabien Chouteau
June 13, 2017



The RISC-V open instruction set is getting more and more news coverage these days. In particular since the release of the first RISC-V microcontroller from SiFive and the announcement of an Arduino board at the Maker Faire Bay Area 2017.

As an Open Source software company we are very interested in this trendy, new, open platform. AdaCore tools already support an open IP core with the Leon processor family, a popular architecture in the space industry that is developed in VHDL and released under the GPL. RISC-V seems to be targeting a different market and opening new horizons.

GNAT - the Ada compiler developed and maintained by AdaCore - is part of the GCC toolchain. As a result, when a new back-end is added we can fairly quickly start targeting it and developing in Ada. In this blog

post I will describe the steps I followed to build the tool chain and start programing the HiFive 1 RISC-V microcontroller in Ada.

Building the tool chain

The first step is to build the compiler. SiFive - manufacturer of the MCU - provides an SDK repository with scripts to build a cross RISC-V GCC. All I had to do was to change the configure options to enable Ada support

```
--enable-languages=c,c++,ada
```

and disable libada since this is a bare-metal target (no operating system) we won't use a complete run-time

```
--disable-libada
```

If you want to build the toolchain yourself, I forked and modified the freedom-e-sdk repository.

Just clone it

```
$ git clone --recursive https://github.com/Fabien-Chouteau/freedom-e-sdk
```

install a native GNAT from your Linux distrib (I use Ubuntu)

```
$ sudo apt-get install gnat
```

and start the build

```
$ cd freedom-e-sdk
$ make tools
```

If you have a problem with this procedure don't hesitate to open an issue on GitHub, I'll see what I can do to help.

Building the run-time

Ada programs always need a run-time library, but there are different run-time profiles depending on the constraints of the platform. In GNAT we have the so called Zero FootPrint run-time (ZFP) that provides the bare minimum and therefore is quite easy to port to a new platform (no exception propagation, no tasking, no containers, no file system access, etc.).

I started from Shawn Nock's ZFP for the Nordic nRF51 and then simply changed the linker script and startup code, everything else is platform independant.

You can find the run-time in this repository: https://github.com/Fabien-Chouteau/zfp-hifive1

Writing the drivers

To control the board I need a few drivers. I started by writing the description of the hardware registers using the SVD format: https://github.com/AdaCore/svd2ada/blob/master/CMSIS-SVD/SiFive/FE310.svd.

I then generated Ada mapping from this file using the SVD2Ada tool. You can find more info about this process at the beginning of this blog post: https://community.arm.com/iot/embedded/b/embedded-blog/posts/ada-driver-library-for-arm-cortex-m-r---part-2-2.

From these register mappings it's fairly easy to implement the drivers. So far I wrote GPIO and UART: https://github.com/AdaCore/Ada_Drivers_Library/tree/master/arch/RISC-V/SiFive/drivers

First Ada projects on the HiFive1

The first project is always a blinky. The HiFive1 has RGB leds so I started by driving those. You can find this example in the Ada_Drivers_Library (https://github.com/AdaCore/Ada_Drivers_Library/tree/master/examples/HiFive1).

If you want to run this simple example on your board, get my fork of the freedom-e-sdk (as described above) and run:

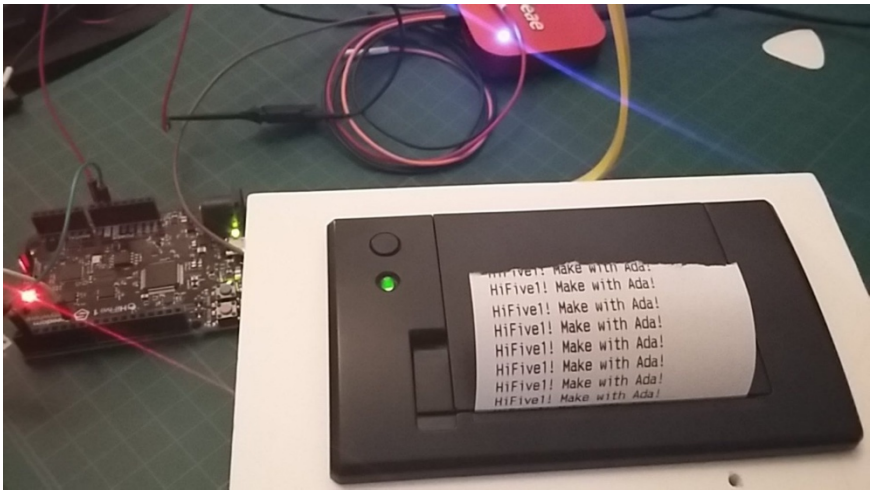
```
$ make ada_blinky_build
```

and then

```
$ make ada_blinky_upload
```

to flash the board.

For the second project, thanks to the architecture of the Ada_Drivers_Library, I was able to re-use the thermal printer driver from my DIY instant camera and it took me only 5 minutes to print something from the HiFive1.



Conclusion

All of this is only experimental for the moment, but it shows how quickly we can start programing Ada on new platforms. Proper support would require a run-time with tasking, interruptions, protected objects (Ravenscar profile) and of course complete test and validation of the compiler.

This chapter was originally published at https://blog.adacore.com/ada-on-the-first-risc-v-microcontroller

DIY Coffee Alarm Clock

By Fabien Chouteau
May 16, 2017

See it in action : https://youtu.be/w7NEzSnPe70

A few weeks ago one of my colleagues shared the kickstarter project, The Barisieur. It's an alarm clock coffee maker, promising to wake you up with a freshly brewed cup of coffee every morning. I jokingly said "just give me an espresso machine and I can do the same". Soon after, the coffee machine is in my office. Now it is time to deliver :)

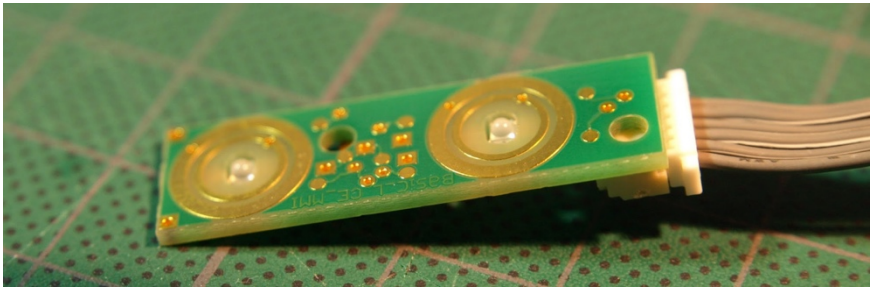
The basic idea is to control the espresso machine from an STM32F469 board and use the beautiful screen to display the clock face and configuration interface.

Hacking the espresso machine

The first step is to be able to control the machine with the 3.3V signal of the microcontroller. To do this, I open the case to get to the two push buttons on the top. Warning! Do not open this kind of appliance if you don't know what you are doing. First, it can be dangerous, second, these

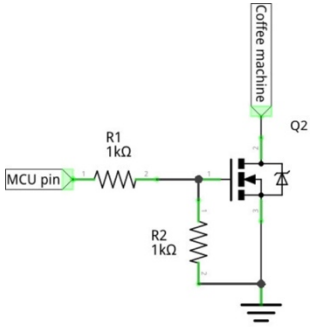
things are not made to be serviceable so there's a good chance you will never be able to put it back together.

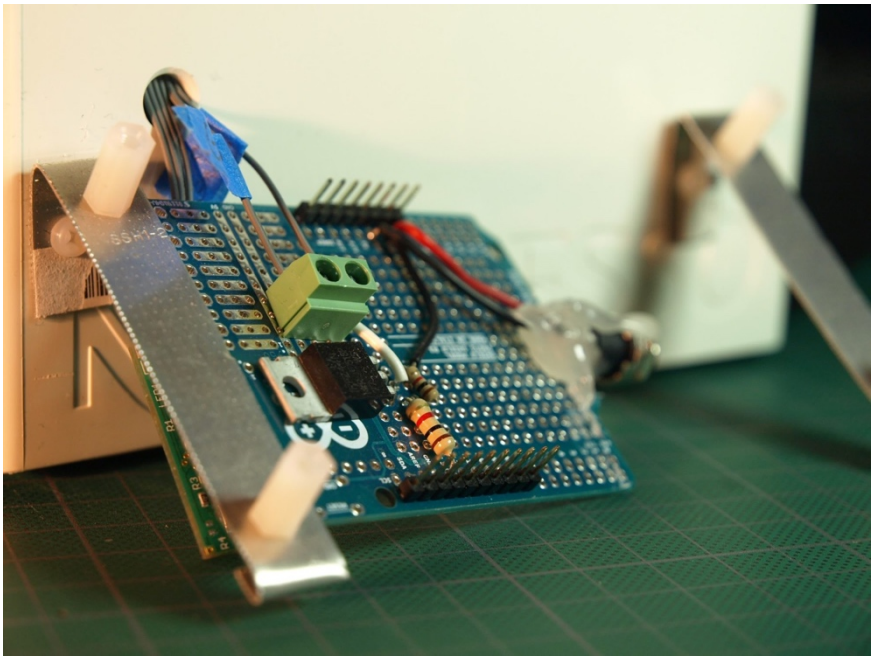
The push buttons are made with two exposed concentric copper traces on a small PCB and a conductive membrane that closes the circuit when the button is pushed.



I use a multimeter to measure the voltage between two circles of one of the buttons. To my surprise the voltage is quite high, about 16V. So I will have to use a MOSFET transistor to act as an electronic switch rather than just connecting the microcontroller to the espresso machine signals.

I put that circuit on an Arduino proto shield that is then plugged behind the STM32F469 disco board. The only things left to do are to drill a hole for the wires to go out of the the machine and to make a couple of metal brackets to attach to the board. Here's a video showing the entire hacking process: https://youtu.be/n3AEIQGGn_g





Writing the alarm clock software

For the clock face and configuration interface I will use Giza, one of my toy projects that I developed to play with the object oriented programming features of Ada. It's a simplistic/basic UI framework.

Given the resolution of the screen (800x480) and the size of the text I want to display, it will be too slow to use software font rendering. Instead, I will take advantage of the STM32F4's 2D graphic hardware acceleration (DMA2D) and have some bitmap images embedded in the executable. DMA2D can very quickly copy chunks of memory - typically bitmaps - but also convert them from one format to the other. This project is the opportunity to implement support of indexed bitmap in the Ada_Drivers_Library.

I also add support for STM32F4's real time clock (RTC) to be able to keep track of time and date and of course trigger the coffee machine at the time configured by the user.

It's time to put it all together and ask my SO to perform in the the high budget video that you can see at the beginning of this post :)

The code is available on GitHub: https://github.com/Fabien-Chouteau/coffee-clock.

This chapter was originally published at https://blog.adacore.com/diy-coffee-alarm-clock

The Adaroombot Project

by Rob Tice
June 20, 2017

See the robot in action at https://youtu.be/2KtlZcpsAcY.

Owning an iRobot Roomba® is an interesting experience. For those not familiar, the Roomba® is a robot vacuum cleaner that's about the diameter of a small pizza and stands tall enough to still fit under your bed. It has two independently driven wheels, a small-ish dust bin, a rechargeable battery, and a speaker programmed with pleasant sounding beeps and bloops telling you when it's starting or stopping a cleaning job. You can set it up to clean on a recurring schedule through buttons on the robot, or with the new models, the mobile app. It picks up an impressive amount of dust and dirt and it makes you wonder how you used to live in a home that was that dirty.

A Project to Learn Ada

I found myself new to AdaCore without any knowledge of the Ada programming language around the same time I acquired a Roomba® for my cats to use as a golf cart when I wasn't home. In order to really learn Ada I decided I needed a good project to work on. Having come from an embedded Linux C/C++ background I decided to do a project involving a Raspberry Pi and something robotic that it could control. It just so happens that iRobot has a STEM initiative robot called the Create® 2 which is aimed towards embedded control projects. That's how the AdaRoombot project was born.

The first goal of the project was to have a simple Ada program use the Create® 2's serial port to perform some control algorithm. Mainly this would require the ability to send commands to the robot and receive feedback information from the robot's numerous sensors. As part of the Create® 2 documentation package, there is a PDF detailing the serial port interface called the iRobot Create® 2 Open Interface Specification.

On the command side of things there is a simple protocol: each command starts with a one-byte opcode specifying which command is being issued and then is followed by a number of bytes carrying the

data associated with the opcode, or the payload. For example, the Reset command has an opcode of 7 and has zero payload bytes. The Set Day/Time command has an opcode of 168 and has a 3-byte payload with a byte specifying the Day, another for the Hour, and another for the Minute. The interface for the Sensor data is a little more complicated. The host has the ability to request data from individual sensors, a list of sensors, or tell the robot to just stream the list over and over again for processing. To make things simple, I choose to just receive all the sensor data on each request.

Because we are using a Raspberry Pi, it is quite easy to communicate with a serial port using the Linux tty interface. As with most userspace driver interfaces in Linux, you open a file and read and write byte data to the file. So, from a software design perspective, the lowest level of the program abstraction should take robot commands and transform them into byte streams to write to the file, and conversely read bytes from the file and transform the byte data to sensor packets. The next level of the program should perform some algorithm by interpreting sensor data and transmitting commands to make the robot perform some task and the highest level of the program should start and stop the algorithm and do some sort of system monitoring.

The high level control algorithm I used is very simple: drive straight until I hit something, then turn around and repeat. However, the lower levels of the program where I am interfacing with peripherals is much more exciting. In order to talk to the serial port, I needed access to file I/O and Linux's terminal I/O APIs.

Ada has cool features

Ada has a nifty way to interface with the Linux C libraries that can be seen near the bottom of "src/communication.ads". There I am creating Ada specs for C calls, and then telling the compiler to use the C implementations supplied by Linux using pragma Import. This is similar to using extern in C. I am also using pragma Convention which tells the compiler to treat Ada records like C structs so that they can be passed into the imported C functions. With this I have the ability to interface to any C call I want using Ada, which is pretty cool. Here is an example mapping the C select call into Ada:

```
--  #include <sys/select.h>
--  fd_set represents file descriptor sets for the select function.
--  It is actually a bit array.
type Fd_Set is mod 2 ** 32;
pragma Convention (C, Fd_Set);

--  #include <sys/time.h>
--  time_t tv_sec - number of whole seconds of elapsed time.
--  long int tv_usec - Rest of the elapsed time in  microseconds.
type Timeval is record
    Tv_Sec  : C.Int;
    Tv_Usec : C.Int;
end record;
pragma Convention (C, Timeval);

function C_Select (Nfds : C.Int;
                   Readfds   : access Fd_Set;
                   Writefds  : access Fd_Set;
                   Exceptfds : access Fd_Set;
                   Timeout   : access Timeval)
                   return C.int;
pragma Import (C, C_Select, "select");
```

The other neat low-level feature to note here can be seen in “src/types.ads”. The record Sensor_Collection is a description of the data that will be received from the robot over the serial port. I am using a feature called a representation clause to tell the compiler where to put each component of the record in memory, and then overlaying the record on top of a byte stream. By doing this, I don’t have to use any bit masks or bit shift to access individual bits or fields within the byte stream. The compiler has taken care of this for me. Here is an example of a record which consists of Boolean values, or bits in a byte:

```
type Sensor_Light_Bumper is record
    LT_Bump_Left         : Boolean;
    LT_Bump_Front_Left   : Boolean;
    LT_Bump_Center_Left  : Boolean;
    LT_Bump_Center_Right : Boolean;
    LT_Bump_Front_Right  : Boolean;
    LT_Bump_Right        : Boolean;
end record
 with Size => 8;
```

```
for Sensor_Light_Bumper use record
    LT_Bump_Left at 0 range 0 .. 0;
    LT_Bump_Front_Left at 0 range 1 .. 1;
    LT_Bump_Center_Left at 0 range 2 .. 2;
    LT_Bump_Center_Right at 0 range 3 .. 3;
    LT_Bump_Front_Right at 0 range 4 .. 4;
    LT_Bump_Right at 0 range 5 .. 5;
end record;
```

In this example, LT_Bump_Left is the first bit in the byte, LT_Bump_Front_Left is the next bit, and so on. In order to access these bits, I can simply use the dot notation to access members of the record, where with C I would have to mask and shift. Components that span multiple bytes can also include an endianness specification. This is useful because on this specific platform data is little endian, but the serial port protocol is big endian. So instead of byte swapping, I can specify certain records as having big endian mapping. The compiler then handles the swapping.

These are some of the really cool low level features available in Ada. On the high-level programming side, the algorithm development, Ada feels more like C++, but with some differences in philosophy. For instance, certain design patterns are more cumbersome to implement in Ada because of things like, Ada objects don't have explicit constructors or destructors. But, after a small change in mind-set it was fairly easy to make the robot drive around the office.

The code for AdaRoombot, which is available on Github, can be compiled using the GNAT GPL cross compiler for the Raspberry Pi 2 located at adacore.com/download. The directions to build and run the code is included in the README file in the root directory of the repo.

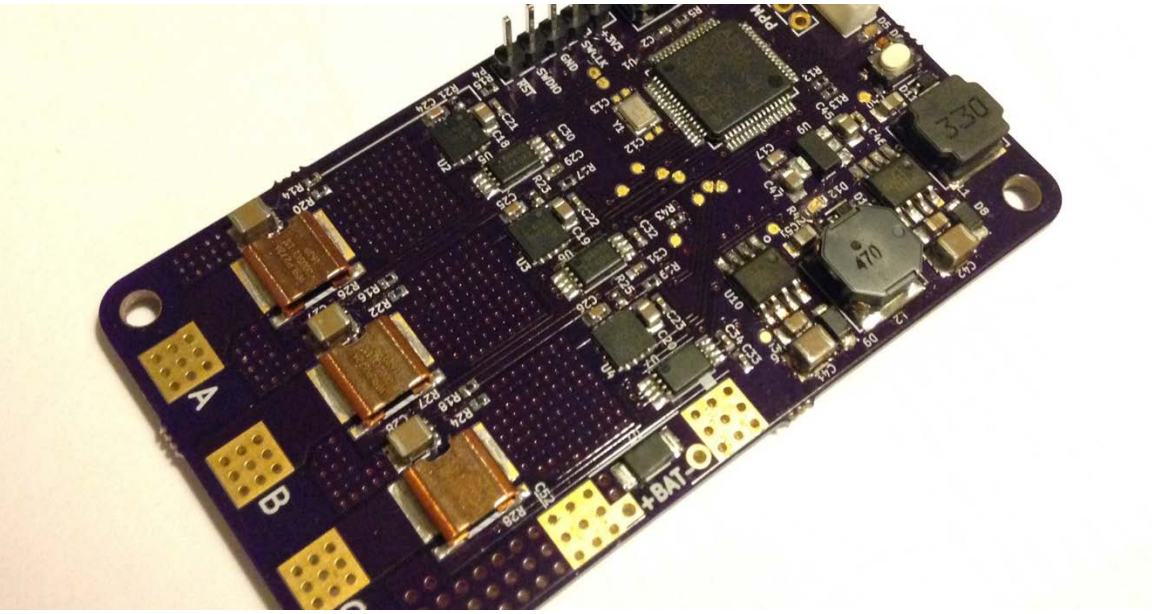
The next step is to add some vision processing and make the robot chase a ball down the hallway. Stay tuned....

The code is available on GitHub: https://github.com/Robert-Tice/AdaRoombot.

This chapter was originally published at https://blog.adacore.com/the-adaroombot-project

Make with Ada: Brushless DC Motor Controller

By Jonas Attertun
Nov 14, 2017



Jonas Attertun was a guest blogger and the winner of the Make with Ada 2017 competition. With a master's degree in Electrical Engineering from Chalmers University of Technology, he has mainly worked as an embedded software engineer within automotive applications.

Not long after my first experience with the Ada programming language I got to know about the Make With Ada 2017 contest. And, just as it seemed, it turned out to be a great way to get a little bit deeper into the language I had just started to learn

The ada-motorcontrol project involves the design of a BLDC motor controller software platform written in Ada. These types of applications

often need to be run fast and the core control software is often tightly connected to the microcontroller peripherals. Coming from an embedded systems perspective with C as the reference language, the initial concerns were if an implementation in Ada actually could meet these requirements.

It turned out, on the contrary, that Ada is very capable considering both these requirements. In particular, accessing peripherals on the STM32 with help of the Ada_Drivers_Library really made using the hardware related operations even easier than using the HAL written in C by ST.

Throughout the project I found uses for many of Ada's features. For example, the representation clause feature made it simple to extract data from received (and to put together the transmit) serial byte streams. Moreover, contract based programming and object oriented concepts such as abstracts and generics provided means to design clean and easy to use interfaces, and a well organized project.

One of the objectives of the project was to provide a software platform to help developing various motor control applications, with the core functionality not being dependent on some particular hardware. Currently however it only supports a custom inverter board, since unfortunately I found that the HAL provided in Ada_Drivers_Library was not comprehensive enough to support all the peripheral features used. But the software is organized such as to keep driver dependent code separated. To put this to test, I welcome contributions to add support for other inverter boards. A good start would be the popular VESC-board.

Motivation

The recent advances in electric drives technologies (batteries, motors and power electronics) has led to increasingly higher output power per cost, and power density. This in turn has increased the performance of existing motor control applications, but also enabled some new - many of them are popular projects amongst diyers and makers, e.g. electric bike, electric skateboard, hoverboard, segway etc.

On a hobby-level, the safety aspects related to these is mostly ignored. Professional development of similar applications, however, normally need to fulfill some domain specific standards putting requirements on for example the development process, coding conventions and verification methods. For example, the motor controller of an electric vehicle would need to be developed in accordance to ISO 26262, and if the C language is used, MISRA-C, which defines a set of programming

guidelines that aims to prevent unsafe usage of the C language features.

Since the Ada programming language has been designed specifically for safety critical applications, it could be a good alternative to C for implementing safe motor controllers used in e.g. electric vehicle applications. For a comparison of MISRA-C and Ada/SPARK, see this report (http://www.adacore.com/uploads/technical-papers/2016-10-SPARK-MisraC-FramaC.pdf). Although Ada is an alternative for achieving functional safety, during prototyping it is not uncommon that a mistake leads to destroyed hardware (burned motor or power electronics). Been there, done that! The stricter compiler of Ada could prevent such accidents.

Moreover, while Ada is not a particularly "new" language, it includes more features that would be expected by a modern language, than is provided by C. For example, types defined with a specified range, allowing value range checks already during compile time, and built-in multitasking features. Ada also supports modularization very well, allowing e.g. easy integration of new control interfaces - which is probably the most likely change needed when using the controller for a new application.

This project should consist of and fulfill:

- Core software for controlling brushless DC motors, mainly aimed at hobbyists and makers.
- Support both sensored and sensorless operation.
- Open source software (and hardware).
- Implementation in Ada on top of the Ravenscar runtime for the stm32f4xx.
- Should not be too difficult to port to another microcontroller.

And specifically, for those wanting to learn the details of motor control, or extend with extra features:

- Provide a basic, clean and readable implementation.
- Short but helpful documentation.

- Meaningful and flexible logging.
- Easy to add new control interfaces (e.g. CAN, ADC, Bluetooth, whatever).

Hardware

The board that will be used for this project is a custom board that I previously designed with the intent to get some hands-on knowledge in motor control. It is completely open source and all project files can be found on GitHub (https://github.com/osannolik/MotCtrl/tree/master/hw).

- Microcontroller STM32F446, ARM Cortex-M4, 180 MHz, FPU
- Power MOSFETs 60 V
- Inline phase current sensing
- PWM/PPM control input
- Position sensor input as *either* hall *or* quadrature encoder
- Motor and board temp sensor (NTC)
- Expansion header for UART/ADC/DAC/SPI/I2C/CAN

It can handle power ranges in the order of what is required by an electric skateboard or electric bike, depending on the used battery voltage and cooling.

There are other inverter boards with similar specifications. One very popular is the VESC (http://vedder.se/2015/01/vesc-open-source-esc/) by Benjamin Vedder. It is probably not that difficult to port this project to work on that board as well.

Rough project plan

I thought it would be appropriate to write down a few bullets of what needs to be done. The list will probably grow...

- Create a port of the Ravenscar runtime to the stm32f446 target on the custom board
- Add stm32f446 as a device in the Ada Drivers Library
- Get some sort of hello world application running to show that stuff works
- Investigate and experiment with interrupt handling with regards to overhead
- Create initialization code for all used mcu peripherals
- Sketch the overall software architecture and define interfaces
- Implementation
- Documentation...

Support for the STM32F446

The microprocessor that will be used for this project is the STM32F446. In the current version of the Ada Drivers Library and the available Ravenscar embedded runtimes, there is no explicit support for this device. Fortunately, it is very similar to other processors in the stm32f4-family, so adding support for stm32f446 was not very difficult once I understood the structure of the repositories. I forked these and added them as submodules in this project's repo (https://github.com/osannolik/ada-motorcontrol).

Compared to the Ravenscar runtimes used by the discovery-boards, there are differences in external oscillator frequency, available interrupt vectors and memory sizes. Otherwise they are basically the same.

An important tool needed to create the new driver and runtime variants is svd2ada (https://github.com/AdaCore/svd2ada). It generates device specification and body files in ada based on an svd file (basically xml) that describes what peripherals exist, how their registers look like, their address', existing interrupts, and stuff like that. It was easy to use, but a little bit confusing how flags/switches should be set when generating driver and runtime files. After some trail and error I think I got it right. I created a Makefile for generating all these file with correct switches.

I could not find an svd-file for the stm32f446 directly from ST, but found one on the internet. It was not perfect though. Some of the source code that uses the generated data types seems to make assumptions on the structure of these types. Depending on how the svd file looks, svd2ada may or may not generate them in the expected way. There were also other missing and incorrect data in the svd file, so I had to manually correct these. There are probably additional issues that I have not found yet...

It is alive!

I made a very simple application consisting of a task that is periodically delayed and toggles the two leds on the board each time the task resumes. The leds toggles with the expected period, so the oscillator seems to be initialized correctly.

Next up I need to map the different mcu pins to the corresponding hardware functionality and try to initialize the needed peripherals correctly.

The control algorithm and its use of peripherals

There are several methods of controlling brushless motors, each with a specific use case. As a first approach I will implement sensored FOC, where the user requests a current value (or torque value).

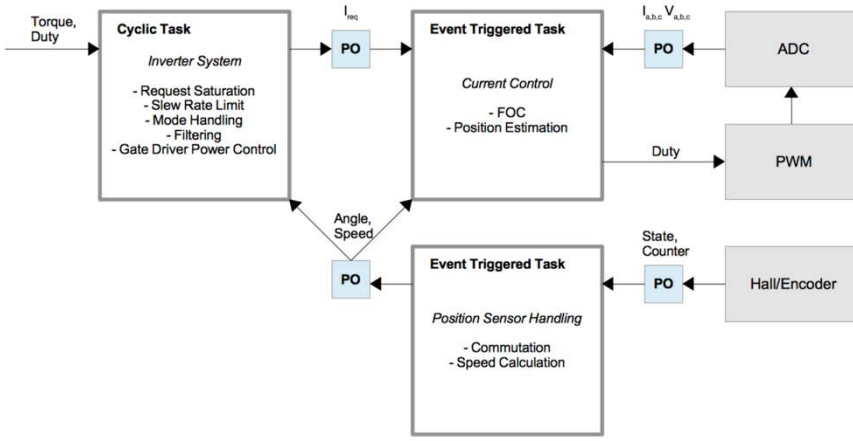
To simplify, this method can be divided into the following steps, repeated each PWM period (typically around 20 kHz):

1. Sample the phase currents
2. Transform the values into a rotor fixed reference frame
3. Based on the requested current, calculate a new set of phase voltages
4. Transform back to the stator's reference frame
5. Calculate PWM duty cycles as to create the calculated phase voltages

Fortunately, the peripherals of the stm32f446 has a lot of features that makes this easier to implement. For example, it is possible to trigger the

ADC directly from the timers that drives the PWM. This way the sampling will automatically be synchronized with the PWM cycle. Step 1 above can thus be started immediately as the ADC triggers the corresponding conversion-complete-interrupt. In fact, many existing implementations perform all the steps 1-to-6 completely within an ISR. The reason for this is simply to reduce any unnecessary overhead since the performed calculations is somewhat lengthy. The requested current is passed to the ISR via global variables.

I would like to do this the traditional way, i.e. to spend as little time as possible in the ISR and trigger a separate Task to perform all calculations. The sampled current values and the requested current shall be passed via Protected Objects. All this will of course create more overhead. Maybe too much? Need to be investigated.



PWM and ADC is up and running

I have spent some time configuring the PWM and ADC peripherals using the Ada Drivers Library. All in all it went well, but I had to do some smaller changes to the drivers to make it possible to configure the way I wanted.

- PWM is complementary output, center aligned with frequency of 20 kHz
- PWM channels 1 to 3 generates the phase voltages
- PWM channel 4 is used to trigger the ADC, this way it is possible to set where over the PWM period the sampling should occur

- By default the sampling occurs in the middle of the positive waveform (V7)
- The three ADC's are configured to Triple Multi Mode, meaning they are synchronized such that each sampled phase quantity is sampled at the same time.
- Phase currents and voltages a,b,c are mapped to the injected conversions, triggered by the PWM channel 4
- Board temperature and bus voltage is mapped to the regular conversions triggered by a timer at 14 kHz
- Regular conversions are moved to a volatile array using DMA automatically after the conversions complete
- ADC create an interrupt after the injected conversions are complete

The drivers always assumed that the PWM outputs are mapped to a certain GPIO, so in order to configure the trigger channel I had to add a new procedure to the drivers. Also, the Scan Mode of the ADCs where not set up correctly for my configuration, and the config of injected sequence order was simply incorrect. I will send a pull request to get these changes merged with the master branch.

Interrupt overhead/latency

As was described in previous posts the structure used for the interrupt handling is to spend minimum time in the interrupt context and to signal an awaiting task to perform the calculations, which executes at a software priority level with interrupts fully enabled. The alternative method is to place all code in the interrupt context.

This Ada Gem (https://www.adacore.com/gems/ada-gem-13/) and its following part describes two different approaches for doing this type of task synchronization. Both use a protected procedure as the interrupt handler but signals the awaiting task in different ways. The first uses an entry barrier and the second a Suspension Object. The idiom using the entry barrier has the advantage that it can pass data as an integrated part of the signaling, while the Suspension Object behaves more like a binary semaphore.

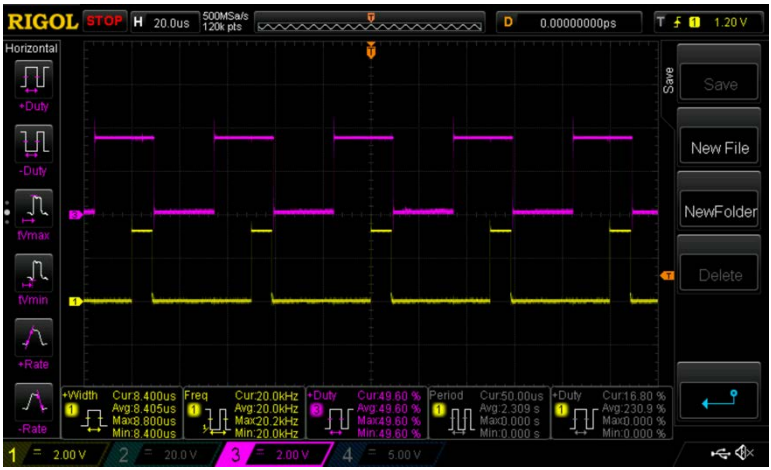
For the ADC conversion complete interrupt, I tested both methods. The protected procedure used as the ISR read the converted values consisting of six uint16. For the entry barrier method these where passed to the task using an out-parameter. When using the second method the task needed to collect the sample data using a separate function in the protected object.

Overhead in this context I define as the time from that the ADC generates the interrupt, to the time the event triggered task starts running. This includes, first, an isr-wrapper that is a part of the runtime which then calls the installed protected procedure, and second, the execution time of the protected procedure which reads the sampled data, and finally, the signaling to the awaiting task.

I measured an approximation of the overhead by setting a pin high directly in the beginning of the protected procedure and then low by the waiting task directly when waking up after the signaling. For the Suspension Object case the pin was set low after the read data function call, i.e. for both cases when the sampled data was copied to the task. The code was compiled with the -O3 flag.

The first idiom resulted in an overhead of ~8.4 us, and the second ~10 us. This should be compared to the period of the PWM which at 20 kHz is 50 us. Obviously the overhead is not negligible, so I might consider using the more common approach for motor control applications of having the current control algorithm in the interrupt context instead. However, until the execution time of the algorithm is known, the entry barrier method will be assumed...

Note: "Overhead" might be the wrong term since I don't know if during the time measured the cpu was really busy. Otherwise it should be called latency I think...



Purple: Center aligned PWM at 50 % duty where the ADC triggers in the center of the positive waveform. Yellow: Pin state as described above. High means time of overhead/latency.

Reference frames

A key benefit of the FOC algorithm is that the actual control is performed in a reference frame that is fixed to the rotor. This way the sinusoidal three phase currents, as seen in the stator's reference frame, will instead be represented as two DC values, assuming steady state operation. The transforms used (Clarke and Park) requires that the angle between the rotor and stator is known. As a first step I am using a quadrature encoder since that provides a very precise measurement and very low overhead due to the hardware support of the stm32.

Three types have been defined, each representing a particular reference frame: Abc, Alfa_Beta and Dq. Using the transforms above one can simply write:

```
declare
   Iabc  : Abc;  --  Measured current (stator ref)
   Idq   : Dq;   --  Measured current (rotor ref)
   Vdq   : Dq;   --  Calculated output voltage (rotor ref)
   Vabc  : Abc;  --  Calculated output voltage (stator ref)
   Angle : constant Float := ...;
begin
   Idq := Iabc.Clarke.Park(Angle);
```

```
   --  Do the control...

   Vabc := Vdq.Park_Inv(Angle).Clarke_Inv;
end;
```

Note that Park and Park_Inv both use the same angle. To be precise, they both use Sin(Angle) and Cos(Angle). Now, at first, I simply implemented these by letting each transform calculate Sin and Cos locally. Of course, that is a waste for this particular application. Instead, I defined an angle object that when created also computed Sin and Cos of the angle, and added versions of the transforms to use these "ahead-computed" values instead.

```
declare
   --  Same...

   Angle : constant Angle_Obj := Compose (Angle_Rad);
   --  Calculates Sin and Cos
begin
   Idq := Iabc.Clarke.Park(Angle);

   --  Do the control...

   Vabc := Vdq.Park_Inv(Angle).Clarke_Inv;
end;
```

This reduced the execution time somewhat (not as much as I thought, though), since the trigonometric functions are the heavy part. Using lookup table based versions instead of the ones provided by Ada.Numerics might be even faster...

It spins!

The main structure of the current controller is now in place. When a button on the board is pressed the sensor is aligned to the rotor by forcing the rotor to a known angle. Currently, the requested q-current is set by a potentiometer.

Watch the Ada Motorcontrol first spin https://youtu.be/SuBA_x9dE-I

As of now, it is definitely not tuned properly, but it at least it shows that the general algorithm is working as intended.

In order to make this project easier to develop on, both for myself and any other users, I need to add some logging and tuning capabilities. This should allow a user to change and/or log variables in the application (e.g. control parameters) while the controller is running. I have written a tool for doing this (over serial) before, but then in C. It would be interesting to rewrite it in Ada.

Contract Based Programming

So far, I have not used this feature much. But when writing code for the logging functionality I ran into a good fit for it.

I am using Consistent Overhead Byte Stuffing (COBS) to encode the data sent over uart. This encoding results in unambiguous packet framing regardless of packet content, thus making it easy for receiving applications to recover from malformed packets. The packets are separated by a delimiter (value 0 in this case), making it easy to synchronize the receiving parser. The encoding ensures that the encoded packet itself does not contain the delimiter value.

A good feature of COBS is that given that the raw data length is less than 254, then the overhead due to the encoding is always exactly one byte. I could of course simply write this fact as a comment to the encode/decode functions, allowing the user to make this assumption in order to simplify their code. A better way could be to write this condition as contracts.

```
Data_Length_Max : constant Buffer_Index := 253;

   function COBS_Encode (Input : access Data)
                         return Data
   with
      Pre => Input'Length <= Data_Length_Max,
      Post => (if Input'Length > 0 then
                  COBS_Encode'Result'Length = Input'Length + 1
               else
                  Input'Length = COBS_Encode'Result'Length);

   function COBS_Decode (Encoded_Data : access Data)
                         return Data
   with
```

```
      Pre => Encoded_Data'Length <= Data_Length_Max + 1,
      Post => (if Encoded_Data'Length > 0 then
                  COBS_Decode'Result'Length = Encoded_Data'Length - 1
               else
                  Encoded_Data'Length = COBS_Decode'Result'Length);
```

Logging and Tuning

I just got the logging and tuning feature working. It is an Ada-implementation using the protocol as used by a previous project of mine, Calmeas (https://github.com/osannolik/calmeas). It enables the user to log and change the value of variables in the application, in real-time. This is very helpful when developing systems where the debugger does not have a feature of reading and writing to memory while the target is running.

The data is sent and received over uart, encoded by COBS. The interfaces of the uart and cobs packages implements an abstract stream type, meaning it is very simple to change the uart to some other media, and that e.g. cobs can be skipped if wanted.

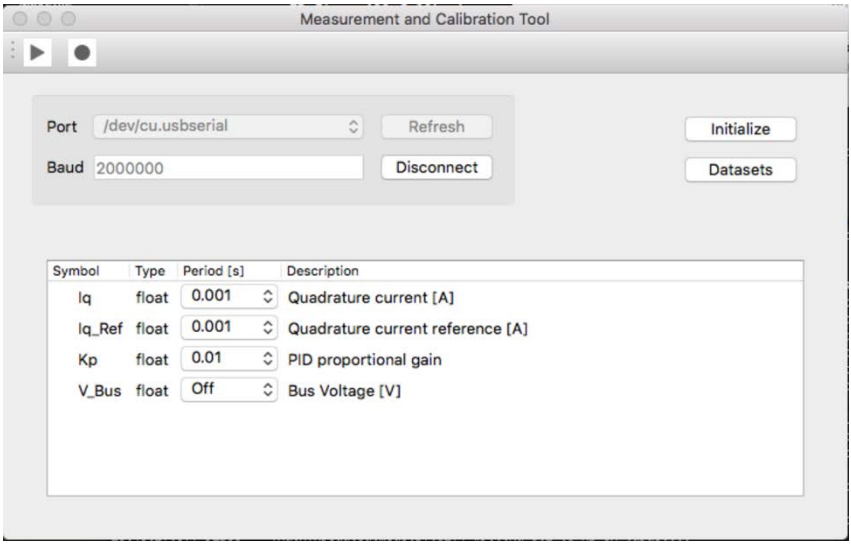
Example

The user can simply do the following in order to get the variable

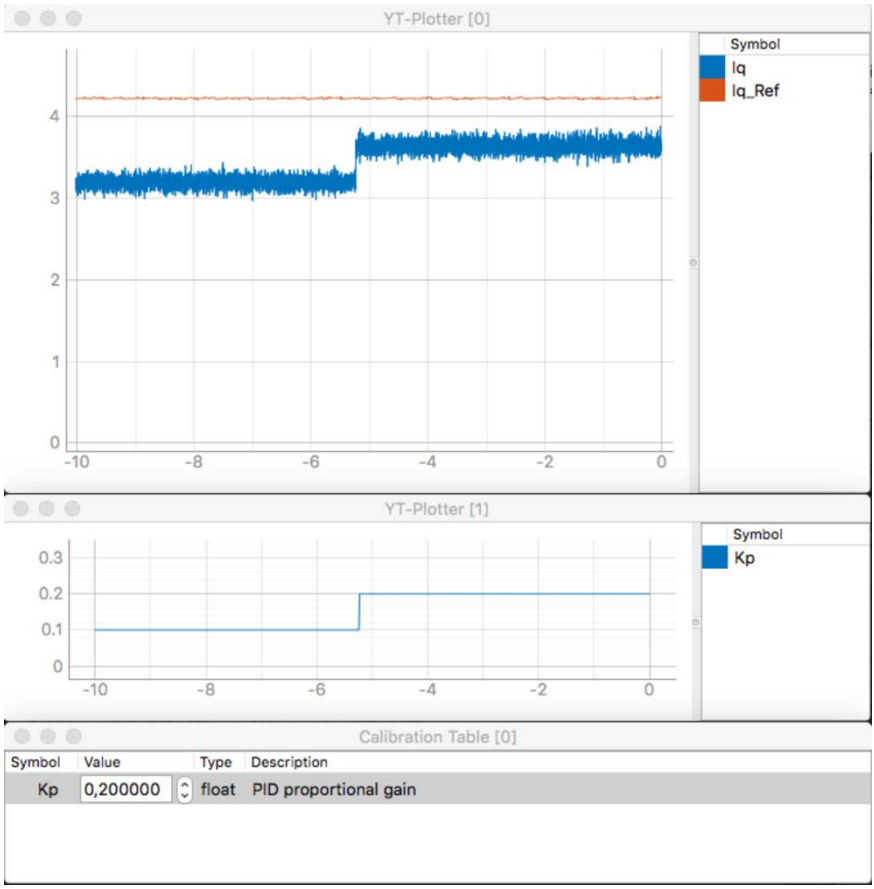
```
V_Bus_Log loggable and/or tunable:
V_Bus_Log  : aliased Voltage_V;
...
Calmeas.Add (Symbol      => V_Bus_Log'Access,
             Name        => "V_Bus",
             Description => "Bus Voltage [V]");
```

It works for (un)signed integers of size 8, 16 and 32 bits, and for floats.

After adding a few variables, and connecting the target to the gui:



As an example, this could be used to tune the current controller gains:



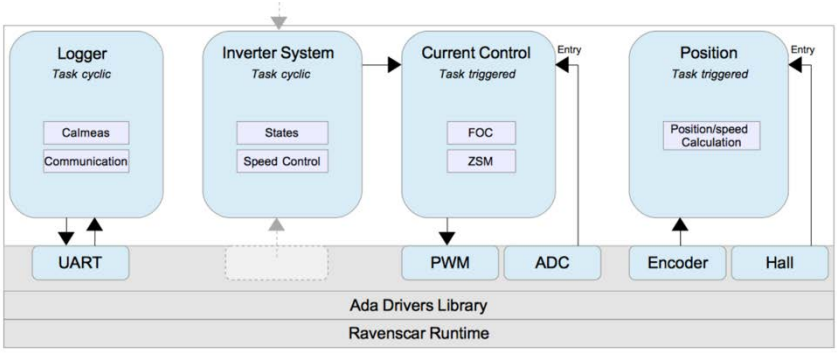
As expected, the actual current comes closer to the reference as the gain increases

As of now, the tuning is not done in a "safe" way. The writing to added symbols is done by the separate task named Logger, simply by doing unchecked writes to the address of the added symbol, one byte at a time. At the same time the application is reading the symbol's value from another task with higher prio. The optimal way would be to pass the value through a protected type, but since the tuning is mostly for debugging purposes, I will make it the proper way later on...

Note that the host GUI is not written in Ada (but Python), and is not itself a part of this project.

Architecture overview

Here is a figure showing an overview of the software:



Summary

This project involves the design of a software platform that provides a good basis when developing motor controllers for brushless motors. It consist of a basic but clean and readable implementation of a sensored field oriented control algorithm. Included is a logging feature that will simplify development and allows users to visualize what is happening. The project shows that Ada successfully can be used for a bare-metal project that requires fast execution.

The design is, thanks to Ada's many nice features, much easier to understand compared to a lot of the other C-implementations out there, where, as a worst case, everything is done in a single ISR. The combination of increased design readability and the strictness of Ada makes the resulting software safer and simplifies further collaborative development and reuse.

Some highlights of what has been done:

- Porting of the Ravenscar profiles to a custom board using the STM32F446
- Adding support for the STM32F446 to Ada_Drivers_Library project
- Adding some functionality to Ada_Drivers_Library in order to fully use all peripheral features

- Fixing a bug in Ada_Drivers_Library related to a bit more advanced ADC usage
- Written HAL-isch packages so that it is easy to port to another device than STM32
- Written a communication package and defined interfaces in order to make it easier to add control inputs.
- Written a logging package that allows the developer to debug, log and tune the application in real-time.
- Implemented a basic controller using sensored field oriented control
- Well documented specifications with a generated html version

Future plans:

- Add hall sensor support and 6-step block commutation
- Add sensorless operation
- Add CAN support (the pcb has currently no transceiver, though)
- SPARK proving
- Write some additional examples showing how to use the interfaces.
- Port the software to the popular VESC-board.

This chapter was originally published at https://blog.adacore.com/make-with-ada-2017-brushless-dc-motor-controller

Make with Ada 2017- A "Swiss Army Knife" Watch

by J. German Rivera
Nov 22, 2017

J. German Rivera is a guest blogger and the 2nd place winner for Make with Ada 2016 and 2017.

The Hexiwear is an IoT wearable development board that has two NXP Kinetis microcontrollers. One is a K64F (Cortex-M4 core) for running the main embedded application software. The other one is a KW40 (Cortex M0+ core) for running a wireless connectivity stack (e.g., Bluetooth BLE or Thread). The Hexiwear board also has a rich set of peripherals, including OLED display, accelerometer, magnetometer, gryroscope, pressure sensor, temperature sensor and heart-rate sensor. This blog article describes the development of a "Swiss Army Knife" watch on the Hexiwear platform. It is a bare-metal embedded

application developed 100% in Ada 2012, from the lowest level device drivers all the way up to the application-specific code, for the Hexiwear's K64F microcontroller.

I developed Ada drivers for Hexiwear-specific peripherals from scratch, as they were not supported by AdaCore's Ada drivers library. Also, since I wanted to use the GNAT GPL 2017 Ada compiler but the GNAT GPL distribution did not include a port of the Ada Runtime for the Hexiwear board, I also had to port the GNAT GPL 2017 Ada runtime to the Hexiwear. All this application-independent code can be leveraged by anyone interested in developing Ada applications for the Hexiwear wearable device.

Project Overview

The purpose of this project is to develop the embedded software of a "Swiss Army Knife" watch in Ada 2012 on the Hexiwear wearable device.

The Hexiwear is an IoT wearable development board that has two NXP Kinetis microcontrollers. One is a K64F (Cortex-M4 core) for running the main embedded application software. The other one is a KW40 (Cortex M0+ core) for running a wireless connectivity stack (e.g., Bluetooth BLE or Thread). The Hexiwear board also has a rich set of peripherals, including OLED display, accelerometer, magnetometer, gryroscope, pressure sensor, temperature sensor and heart-rate sensor.

The motivation of this project is two-fold. First, to demonstrate that the whole bare-metal embedded software of this kind of IoT wearable device can be developed 100% in Ada, from the lowest level device drivers all the way up to the application-specific code. Second, software development for this project will produce a series of reusable modules that can be used in the future as a basis for creating "labs" for teaching an Ada 2012 embedded software development class using the Hexiwear platform. Given the fact that the Hexiwear platform is a very attractive platform for embedded software development, its appeal can be used to attract more embedded developers to learn Ada.

The scope of the project will be to develop only the firmware that runs on the application microcontroller (K64F). Ada drivers for Hexiwear-specific peripherals need to be developed from scratch, as they are not supported by AdaCore's Ada drivers library. Also, since I will be using the GNAT GPL 2017 Ada compiler and the GNAT GPL distribution does not include a port of the Ada Runtime for the Hexiwear board,

the GNAT GPL 2017 Ada runtime needs to be ported to the Hexiwear board.

The specific functionality of the watch application for the time frame of "Make with Ada 2017" will include:

- Watch mode: show current time, date, altitude and temperature
- Heart rate monitor mode: show heart rate (when Hexiwear worn on the wrist)
- G-Forces monitor mode: show G forces in the three axis (X, Y, Z).

In addition, when the Hexiwear is plugged to a docking station, a command-line interface will be provided over the UART port of the docking station. This interface can be used to set configurable parameters of the watch and to dump debugging information.

Summary of Accomplishments

I designed and implemented the "Swiss Army Knife" watch application and all necessary peripheral drivers 100% in Ada 2012. The only third-party code used in this project, besides the GNAT GPL Ravenscar SFP Ada runtime library, is the following:

- A font generator, leveraged from AdaCore's Ada drivers library.
- Ada package specification files, generated by the the svd2ada tool, containing declarations for the I/O registers of the Kinetis K64F's peripherals.

Below are some diagrams depicting the software architecture of the "Swiss Army Knife" watch:

- **Hardware Context Diagram**
  https://github.com/jgrivera67/make-with-ada/blob/master/hexiwear_watch/doc/HW_context_diagram.pdf
- **Source Code Architecture**
  https://github.com/jgrivera67/make-with-ada/blob/master/hexiwear_watch/doc/code_architecture_diagrma.pdf

- **Ada Task Architecture**
  https://github.com/jgrivera67/make-with-ada/blob/master/hexiwear_watch/doc/Ada_task_architecture_diagram.pdf

The current implementation of the "Swiss Army Knife" watch firmware, delivered for "Make with Ada 2017" has the following functionality:

- Three operating modes:
- Watch mode: show current time, date, altitude and temperature
- Heart rate monitor mode: show heart rate monitor raw reading, when Hexiwear worn on the wrist.
- G-Forces monitor mode: show G forces in the three axis (X, Y, Z).
- To switch between modes, the user just needs to do a double-tap on the watch's display. When the watch is first powered on, it starts in "watch mode".
- To make the battery charge last longer, the microcontroller is put in deep-sleep mode (low-leakage stop mode) and the display is turned off, after 5 seconds. A quick tap on the watch's display, will wake it up. Using the deep-sleep mode, makes it possible to extend the battery life from 2 hours to 12 hours, on a single charge. Indeed, now I can use the Hexiwear as my personal watch during the day, and just need to charge it at night.
- A command-line interface over UART, available when the Hexiwear device is plugged to its docking station. This interface is used for configuring the following attributes of the watch:
    1. Current time (not persistent on power loss, but persistent across board resets)
    2. Current date (not persistent on power loss, but persistent across board resets)
    3. Background color (persistent on the K64F microcontroller's NOR flash, unless firmware is re-imaged)

4. Foreground color (persistent on the K64F microcontroller's NOR flash, unless firmware is re-imaged)

Also, the command-line interface can be used for dumping the different debugging logs: info, error and debug.

See it in action at https://youtu.be/H9N91JuWNwU

The final version of the code of the watch application and the associated peripheral drivers submitted to the "Make with Ada 2017" programming competition can be found in GitHub at: https://github.com/jgrivera67/make-with-ada/releases/tag/Make-with-Ada-2017-final

The top-level code of the watch application can be found at: https://github.com/jgrivera67/make-with-ada/tree/master/hexiwear_watch

I ported the GNAT GPL 2017 Ravenscar Small-Foot-Print Ada runtime library to the Hexiwear board and modify it to support the memory protection unit (MPU) of the K64 microcontroller, and more specifically to support MPU-aware tasks. This port of the Ravenscar Ada runtime

can be found in GitHub at https://github.com/jgrivera67/embedded-runtimes

The relevant folders are:

- https://github.com/jgrivera67/embedded-runtimes/tree/master/ravenscar-kinetis_k64f_hexiwear
- https://github.com/jgrivera67/embedded-runtimes/tree/master/bsps/kinetis_k64f_hexiwear
- https://github.com/jgrivera67/embedded-runtimes/tree/master/bsps/kinetis_k64f_common/bsp

I developed device drivers for the following peripherals of the Kinetis K64 micrcontroller as part of this project and contributed their open-source Ada code in GitHub at https://github.com/jgrivera67/make-with-ada/tree/master/drivers/mcu_specific/nxp_kinetis_k64f:

- DMA Engine
- SPI controller
- I2C controller
- Real-time Clock (RTC)
- Low power management
- NOR flash

These drivers are application-independent and can be easily reused for other Ada embedded applications that use the
Kinetis K64F microcontroller.

I developed device drivers for the following peripherals of the Hexiwear board as part of this project and contributed their open-source Ada code in GitHub at https://github.com/jgrivera67/make-with-ada/tree/master/drivers/board_specific/hexiwear:

- OLED graphics display (on top of the SPI and DMA engine drivers, and making use of the advanced DMA channel linking functionality of the Kinetis DMA engine)
- 3-axis accelerometer (on top of the I2C driver)
- Heart rate monitor (on top of the I2C driver)
- Barometric pressure sensor (on top of the I2C driver)

These drivers are application-independent and can be easily reused for other Ada embedded applications that use the Hexiwear board.

I designed the watch application and its peripheral drivers, to use the memory protection unit (MPU) of the K64F microcontroller, from the beginning, not as an afterthought. Data protection at the individual data object level is enforced using the MPU, for the private data structures from every Ada package linked into the application. For this, I leveraged the MPU framework I developed earlier and which I presented in the Ada Europe 2017 Conference. Using this MPU framework for the whole watch application demonstrates that the framework scales well for a realistic-size application. The code of this MPU framework is part of a modified Ravenscar small-foot-print Ada runtime library port for the Kinetis K64F microcontroller, whose open-source Ada code I contributed at https://github.com/jgrivera67/embedded-runtimes/tree/master/bsps/kinetis_k64f_common/bsp

I developed a CSP model (https://github.com/jgrivera67/make-with-ada/blob/master/hexiwear_watch/doc/hexiwear_watch.csp)of the Ada task architecture of the watch firmware with the goal of formally verifying that it is deadlock free, using the FDR4 tool (https://www.cs.ox.ac.uk/projects/fdr/). Although I ran out of time to successfully run the CSP model through the FDR tool, developing the model helped me gain confidence about the overall completeness of the Ada task architecture as well as the completeness of the watch_task's state machine. Also, the CSP model itself is a form a documentation that provides a high-level formal specification of the Ada task architecture of the watch code.

Open

My Project's code is under the BSD license It's hosted on Github. The project is divided in two repositories:

- Application + Drivers Repository

  The relevant folders are:

  - https://github.com/jgrivera67/make-with-ada/tree/master/hexiwear_watch
  - https://github.com/jgrivera67/make-with-ada/tree/master/drivers/portable
  - https://github.com/jgrivera67/make-with-ada/tree/master/drivers/board_specific/hexiwear

    - https://github.com/jgrivera67/make-with-ada/tree/master/drivers/mcu_specific/nxp_kinetis_k64f

- GNAT Ravenscar SFP Ada Runtime Library Port for the Hexiwear Board with Memory Protection Unit Support Repository

  The relevant Folders are:

    - https://github.com/jgrivera67/embedded-runtimes/tree/master/ravenscar-kinetis_k64f_hexiwear
    - https://github.com/jgrivera67/embedded-runtimes/tree/master/bsps/kinetis_k64f_hexiwear
    - https://github.com/jgrivera67/embedded-runtimes/tree/master/bsps/kinetis_k64f_common/bsp

To do the development, I used the GNAT GPL 2017 toolchain - ARM ELF format (hosted on Windows 10 or Linux), including the GPS IDE. I also used the svd2ada tool to generate Ada code from SVD XML files for the Kinetis microcontrollers I used.

Collaborative

I designed this Ada project to make it easy for others to leverage my code. Anyone interested in developing their own flavor of "smart" watch for the Hexiwear platform can leverage code from my project. Also, anyone interested in developing any type of embedded application in Ada/SPARK for the Hexiwear platform can leverage parts of the software stack I have developed in Ada 2012 for the Hexiwear, particularly device drivers and platform-independent/application-independent infrastructure code, without having to start from scratch as I had to. All you need is to get your own Hexiwear development kit (http://www.hexiwear.com/shop/), clone my Git repositories mentioned above and get the GNAT GPL 2017 toolchain for ARM Cortex-M (http://libre.adacore.com/download/, choose ARM ELF format).

The Hexiwear platform is an excellent platform to teach embedded software development in general, and in Ada/SPARK in particular, given its rich set of peripherals and reasonable cost. The Ada software stack that I have developed for the Hexiwear platform can be used as a base to create a series of programming labs as a companion to courses in Embedded and Real-time programming in Ada/SPARK, from basic

concepts and embedded programming techniques, to more advanced topics such as using DMA engines, power management, connectivity, memory protection and so on.

Dependable

- I used the memory protection unit to enforce data protection at the granularity of individual non-local data objects, throughout the code of the watch application and its associated device drivers.
- I developed a CSP model of the Ada task architecture of the watch firmware with the goal of formally verifying that it is deadlock free, using the FDR4 tool. Although I ran out of time to successfully run the CSP model through the FDR tool, developing the model helped me gain confidence about the completeness of the watch_task's state machine and the overall completeness of the Ada task architecture of the watch code.
- I consistently used the information hiding principles to architect the code to ensure high levels of maintainability and portability, and to avoid code duplication across projects and across platforms.
- I leveraged extensively the data encapsulation and modularity features of the Ada language in general, such as private types and child units including private child units, and in some cases subunits and nested subprograms.
- I used gnatdoc comments to document key data structures and subprograms.
- I used Ada 2012 contract-based programming features and assertions extensively
- I used range-based types extensively to leverage the Ada power to detect invalid integer values.
- I Used Ada 2012 aspect constructs wherever possible
- I Used GNAT coding style. I use the -gnaty3abcdefhiklmnoOprstux GNAT compiler option to check compliance with this standard.
- I used GNAT flags to enable rigorous compile-time checks, such as -gnato13 -gnatf -gnatwa -gnatVa -Wall.

Inventive

The most innovative aspect of my project is the use of the memory protection unit (MPU) to enforce data protection at the granularity of individual data objects throughout the entire code of the project (both application code and device drivers). Although the firmware of a watch is not a safety-critical application, it serves as a concrete example of a realistic-size piece of embedded software that uses the memory protection unit to enforce data protection at this level of granularity. Indeed, this project demonstrates the feasibility and scalability of using the MPU-based data protection approach that I presented at the Ada Europe 2017 conference earlier this year, for true safety-critical applications.

CSP model of the Ada task architecture of the watch code, with the goal of formally verifying that the task architecture was deadlock free, using the FDR4 model checking tool. Although I ran out of time to successfully run the CSP model through the FDR tool, the model itself provides a high-level formal specification of the Ada task architecture of the watch firmware, which is useful as a concise form of documentation of the task architecture, that is more precise than the task architecture diagram alone.

Short-Term Future Plans

Implement software enhancements to existing features of the "Swiss Army Knife" watch:

- Calibrate accuracy of altitude and temperature readings
- Calibrate accuracy of heart rate monitor reading
- Display heart rate in BPM units instead of just showing the raw sensor reading
- Calibrate accuracy of G-force readings

Develop new features of the "Swiss Army Knife" watch:

- Display compass information (in watch mode). This entails extending the accelerometer driver to support the built-in magnetometer
- Display battery charge remaining. This entails writing a driver for the K64F's A/D converter to interface with the battery sensor

- Display gyroscope reading. This entails writing a driver for the Hexiwear's Gyroscope peripheral
- Develop Bluetooth connectivity support, to enable the Hexiwear to talk to a cell phone over blue tooth as a slave and to talk to other Bluetooth slaves as a master. As part of this, a Bluetooth "glue" protocol will need to be developed for the K64Fto communicate with the Bluetooth BLE stack running on the KW40, over a UART. For the BLE stack itself, the one provided by the chip manufacturer will be used. A future challenge could entail to write an entire Bluetooth BLE stack in Ada to replace the manufacturer's KW40 firmware.

Finish the formal analysis of the Ada task architecture of the watch code, by successfully running its CSP model through the FDR tool to verify that the architecture is deadlock free, divergence free and that it satisfies other applicable safety and liveness properties.

Long-Term Future Plans

Develop more advanced features of the "Swiss Army Knife" watch:

- Use sensor fusion algorithms combining readings from accelerometer and gyroscope
- Add Thread connectivity support, to enable the watch to be an edge device in an IoT network (Thread mesh). This entails developing a UART-based interface to the Hexiwear's KW40 (which would need to be running a Thread (804.15) stack).
- Use the Hexiwear device as a dash-mounted G-force recorder in a car. Sense variations of the 3D G-forces as the car moves, storing the G-force readings in a circular buffer in memory, to capture the last 10 seconds (or more depending on available memory) of car motion. This information can be extracted over bluetooth.
- Remote control of a Crazyflie 2.0 drone, from the watch, over Bluetooth. Wrist movements will be translated into steering commands for the drone.

Develop a lab-based Ada/SPARK embedded software development course using the Hexiwear platform, leveraging the code developed in

this project. This course could include the following topics and corresponding programming labs:

1. Accessing I/O registers in Ada.
    - Lab: Layout of I/O registers in Ada and the svd2ada tool
2. Pin-level I/O: Pin Muxer and GPIO.
    - Lab: A Traffic light using the Hexiwear's RGB LED
3. Embedded Software Architectures: Cyclic Executive.
    - Lab: A watch using the real-time clock (RTC) peripheral with polling
4. Embedded Software Architectures: Main loop with Interrupts.
    - Lab: A watch using the real-time clock (RTC) with interrupts
5. Embedded Software Architectures: Tasks.
    - Lab: A watch using the real-time clock (RTC) with tasks
6. Serial Console.
    - Lab: UART driver with polling and with interrupts)
7. Sensors: A/D converter.
    - Lab: Battery charge sensor
8. Actuators: Pulse Width Modulation (PWM).
    - Lab: Vibration motor and light dimmer
9. Writing to NOR flash.
    - Lab: Saving config data in NOR Flash
10. Inter-chip communication and complex peripherals: I2C.
    - Lab: Using I2C to interface with an accelerometer
11. Inter-chip communication and complex peripherals: SPI.
    - Lab: OLED display
12. Direct Memory Access I/O (DMA).
    - Lab: Measuring execution time and making OLED display rendering faster with DMA
13. The Memory Protection Unit (MPU).

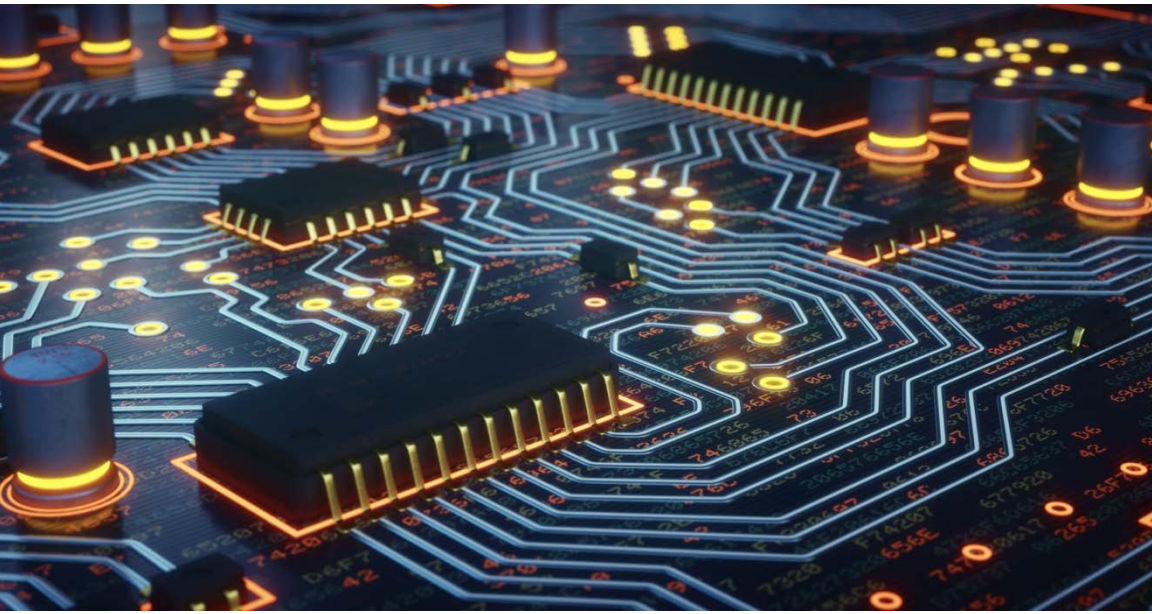
    - Lab: Using the memory protection unit
14. Power Management.
    - Lab: Using a microcontroller's deep sleep mode
15. Cortex-M Architecture and Ada Startup Code.
    - Lab: Modifying the Ada startup code in the Ada runtime library to add a reset counter)
16. Recovering from failure.
    - Lab: Watchdog timer and software-triggered resets
17. Bluetooth Connectivity
    - Lab: transmit accelerometer readings to a cell phone over Bluetooth

This chapter was originally published at
https://blog.adacore.com/make-with-ada-2017-a-swiss-army-knife-watch

‘

There's a Mini-RTOS in My Language

By Fabien Chouteau
Nov 23, 2017



The first thing that struck me when I started to learn about the Ada programing language was the tasking support. In Ada, creating tasks, synchronizing them, sharing access to resources, are part of the language

In this blog post I will focus on the embedded side of things. First because it's what I like, and also because it's much more simple :)

For real-time and embedded applications, Ada defines a profile called `Ravenscar`. It's a subset of the language designed to help schedulability analysis, it is also more compatible with platforms such as micro-controllers that have limited resources.

So this will not be a complete lecture on Ada tasking. I might do a follow-up with some more tasking features, if you ask for it in the comments ;)

Tasks

So the first thing is to create tasks, right?

There are two ways to create tasks in Ada, first you can declare and implement a single task:

```
--  Task declaration
task My_Task;
--  Task implementation
task body My_Task is
begin
   --  Do something cool here...
end My_Task;
```

If you have multiple tasks doing the same job or if you are writing a library, you can define a task type:

```
--  Task type declaration
task type My_Task_Type;

--  Task type implementation
task body My_Task_Type is
begin
   --  Do something really cool here...
end My_Task_Type;
```

And then create as many tasks of this type as you want:

```
T1 : My_Task_Type;
T2 : My_Task_Type;
```

One limitation of Ravenscar compared to full Ada, is that the number of tasks has to be known at compile time.

Time

The timing features of Ravenscar are provided by the package (you guessed it) **Ada.Real_Time**.

In this package you will find:

- a definition of the **Time** type which represents the time elapsed since the start of the system
- a definition of the **Time_Span** type which represents a period between two **Time** values
- a function **Clock** that returns the current time (monotonic count since the start of the system)
- Various sub-programs to manipulate **Time** and **Time_Span** values

The Ada language also provides an instruction to suspend a task until a given point in time: **delay until**.

Here's an example of how to create a cyclic task using the timing features of Ada.

```
task body My_Task is
    Period       : constant Time_Span := Milliseconds (100);
    Next_Release : Time;
 begin
    --  Set Initial release time
    Next_Release := Clock + Period;

    loop
       --  Suspend My_Task until the Clock is greater than Next_Release
       delay until Next_Release;

       --  Compute the next release time
       Next_Release := Next_Release + Period;

       --  Do something really cool at 10Hz...
    end loop;

 end My_Task;
```

Scheduling

Ravenscar has priority-based preemptive scheduling. A priority is assigned to each task and the scheduler will make sure that the highest priority task - among the ready tasks - is executing.

A task can be preempted if another task of higher priority is released, either by an external event (interrupt) or at the expiration of its **delay until** statement (as seen above).

If two tasks have the same priority, they will be executed in the order they were released (FIFO within priorities).

Task priorities are static, however we will see below that a task can have its priority temporary escalated.

The task priority is an integer value between 1 and 256, higher value means higher priority. It is specified with the **Priority** aspect:

```
Task My_Low_Priority_Task
  with Priority => 1;

Task My_High_Priority_Task
  with Priority => 2;
```

Mutual exclusion and shared resources

In Ada, mutual exclusion is provided by the protected objects.

At run-time, the protected objects provide the following properties:

- There can be only one task executing a protected operation at a given time (mutual exclusion)
- There can be no deadlock

In the Ravenscar profile, this is achieved with Priority Ceiling Protocol.

A priority is assigned to each protected object, any tasks calling a protected sub-program must have a priority below or equal to the priority of the protected object.

When a task calls a protected sub-program, its priority will be temporarily raised to the priority of the protected object. As a result, this task cannot be preempted by any of the other tasks that potentially use this protected object, and therefore the mutual exclusion is ensured.

The Priority Ceiling Protocol also provides a solution to the classic scheduling problem of priority inversion.

Here is an example of protected object:

```
--  Specification
protected My_Protected_Object
  with Priority => 3
is

   procedure Set_Data (Data : Integer);
   --  Protected procedues can read and/or modifiy the protected data

   function Data return Integer;
   --  Protected functions can only read the protected data

private

   --  Protected data are declared in the private part
   PO_Data : Integer := 0;
end;
```

```
--  Implementation
protected body My_Protected_Object is

   procedure Set_Data (Data : Interger) is
   begin
      PO_Data := Data;
   end Set_Data;

   function Data return Integer is
   begin
```

```
         return PO_Data;
      end Data;
   end My_Protected_Object;
```

Synchronization

Another cool feature of **protected objects** is the synchronization between tasks.

It is done with a different kind of operation called an **entry**.

An entry has the same properties as a protected procedure except it will only be executed if a given condition is true. A task calling an entry will be suspended until the condition is true.

This feature can be used to synchronize tasks. Here's an example:

```
  protected My_Protected_Object is
      procedure Send_Signal;
      entry Wait_For_Signal;
   private
      We_Have_A_Signal : Boolean := False;
   end My_Protected_Object;
```

```
protected body My_Protected_Object is

      procedure Send_Signal is
      begin
          We_Have_A_Signal := True;
      end Send_Signal;

      entry Wait_For_Signal when We_Have_A_Signal is
      begin
          We_Have_A_Signal := False;
      end Wait_For_Signal;
   end My_Protected_Object;
```

Interrupt Handling

Protected objects are also used for interrupt handling. Private procedures of a protected object can be attached to an interrupt using the **Attach_Handler** aspect.

```
protected My_Protected_Object
  with Interrupt_Priority => 255
is

private

   procedure UART_Interrupt_Handler
     with Attach_Handler => UART_Interrupt;

end My_Protected_Object;
```

Combined with an **entry** it provides and elegant way to handle incoming data on a serial port for instance:

```
protected My_Protected_Object
  with Interrupt_Priority => 255
is
   entry Get_Next_Character (C : out Character);

private
   procedure UART_Interrupt_Handler
            with Attach_Handler => UART_Interrupt;

   Received_Char  : Character := ASCII.NUL;
   We_Have_A_Char : Boolean := False;
end
```

```
protected body My_Protected_Object is

   entry Get_Next_Character (C : out Character) when We_Have_A_Char is
   begin
       C := Received_Char;
       We_Have_A_Char := False;
   end Get_Next_Character;

   procedure UART_Interrupt_Handler is
```

```
      begin
          Received_Char  := A_Character_From_UART_Device;
          We_Have_A_Char := True;
      end UART_Interrupt_Handler;
   end
```

A task calling the entry **Get_Next_Character** will be suspended until an interrupt is triggered and the handler reads a character from the UART device. In the meantime, other tasks will be able to execute on the CPU.

Multi-core support

Ada supports static and dynamic allocation of tasks to cores on multi processor architectures. The Ravenscar profile restricts this support to a fully partitioned approach were tasks are statically allocated to processors and there is no task migration among CPUs. These parallel tasks running on different CPUs can communicate and synchronize using protected objects.

The CPU aspect specifies the task affinity:

```
   task Producer with CPU => 1;
   task Consumer with CPU => 2;
   --  Parallel tasks statically allocated to different cores
```

Implementations

That's it for the quick overview of the basic Ada Ravenscar tasking features.

One of the advantages of having tasking as part of the language standard is the portability, you can run the same Ravenscar application on Windows, Linux, MacOs or an RTOS like VxWorks. GNAT also provides a small stand alone run-time that implements the Ravenscar tasking on bare metal. This run-time is available, for instance, on ARM Cortex-M micro-controllers.

It's like having an RTOS in your language.

This chapter was originally published at https://blog.adacore.com/theres-a-mini-rtos-in-my-language

Bitcoin Blockchain in Ada: Lady Ada Meet Satoshi Nakamoto

By Johannes Kanig
Feb 15, 2018

Bitcoin is getting a lot of press recently, but let's be honest, that's mostly because a single bitcoin worth 800 USD in January 2017 was worth almost 20,000 USD in December 2017. However, bitcoin and its underlying blockchain are beautiful technologies that are worth a closer look. Let's take that look with our Ada hat on!

So what's the blockchain?

"Blockchain" is a general term for a database that's maintained in a distributed way and is protected against manipulation of the entries;

Bitcoin is the first application of the blockchain technology, using it to track transactions of "coins", which are also called Bitcoins.

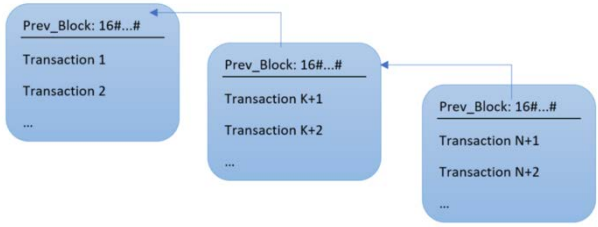
Conceptually, the Bitcoin blockchain is just a list of transactions. Bitcoin transactions in full generality are quite complex, but as a first approximation, one can think of a transaction as a triple (sender, recipient, amount), so that an initial mental model of the blockchain could look like this:

Sender	Recipient	Amount
<Bitcoin address>	<Bitcoin address>	0.003 BTC
<Bitcoin address>	<Bitcoin address>	0.032 BTC
...	...	...

Other data, such as how many Bitcoins you have, are derived from this simple transaction log and not explicitly stored in the blockchain.

Modifying or corrupting this transaction log would allow attackers to appear to have more Bitcoins than they really have, or, allow them to spend money then erase the transaction and spend the same money again. This is why it's important to protect against manipulation of that database.

The list of transactions is not a flat list. Instead, transactions are grouped into *blocks*. The blockchain is a list of blocks, where each block has a link to the previous block, so that a block represents the full blockchain up to that point in time:



Thinking as a programmer, this could be implemented using a linked list where each block header contains a *prev* pointer. The blockchain is grown by adding new blocks to the end, with each new block pointing to

the former previous block, so it makes more sense to use a *prev* pointer instead of a *next* pointer. In a regular linked list, *prev* pointer points directly to the memory used for the previous block. But the uniqueness of the blockchain is that it's a distributed data structure; it's maintained by a network of computers or *nodes*. Every bitcoin *full node* has a full copy of the blockchain, but what happens if members of the network don't agree on the contents of some transaction or block? A simple memory corruption or malicious act could result in a client having incorrect data. This is why the blockchain has various checks built-in that guarantee that corruption or manipulation can be detected.

How does Bitcoin check data integrity?

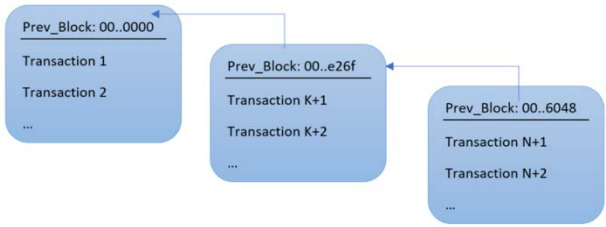
Bitcoin's internal checks are based on a cryptographic hash function. This is just a fancy name for a function that takes anything as input and spits out a large number as output, with the following properties:

- The output of the function varies greatly and unpredictably even with tiny variations of the input;

- It is extremely hard to deduce an input that produces some specific output number, other than by using *brute force*; that is, by computing the function again and again for a large number of inputs until one finds the input that produces the desired output.

The hash function used in Bitcoin is called SHA256. It produces a 256-bit number as output, usually represented as 64 hexadecimal digits. Collisions (different input data that produces the same output hash value) are theoretically possible, but the output space is so big that collisions on actual data are considered extremely unlikely, in fact practically impossible.

The idea behind the first check of Bitcoin's data integrity is to replace a raw pointer to a memory region with a "safe pointer" that can, by construction, only point to data that hasn't been tampered with. The trick is to use the hash value of the data in the block as the "pointer" to the data. So instead of a raw pointer, one stores the hash of the previous block as *prev* pointer:



Here, I've abbreviated the 256-bit hash values by their first two and last four hex digits – by design, Bitcoin block hashes always start with a certain number of leading zeroes. The first block contains a "null pointer" in the form of an all zero hash.

Given a hash value, it is infeasible to compute the data associated with it, so one can't really "follow" a hash like one can follow a pointer to get to the real data. Therefore, some sort of table is needed to store the data associated with the hash value.

Now what have we gained? The structure can no longer easily be modified. If someone modifies any block, its hash value changes, and all existing pointers to it are invalidated (because they contain the wrong hash value). If, for example, the following block is updated to contain the new *prev* pointer (i.e., hash), its own hash value changes as well. The end result is that the whole data structure needs to be completely rewritten even for small changes (following *prev* pointers in reverse order starting from the change). In fact such a rewrite never occurs in Bitcoin, so one ends up with an immutable chain of blocks. However, one needs to check (for example when receiving blocks from another node in the network) that the block pointed to really has the expected hash.

Block data structure in Ada

To make the above explanations more concrete, let's look at some Ada code (you may also want to have bitcoin documentation available : https://en.bitcoin.it/wiki/Block_hashing_algorithm).

A bitcoin block is composed of the actual block contents (the list of transactions of the block) and a block header. The entire type definition of the block looks like this (you can find all code in this post plus some supporting code in this github repository : https://github.com/kanigsson/bitcoin-ada/tree/blogpost_1)

```
type Block_Header is record
      Version : Uint_32;
      Prev_Block : Uint_256;
      Merkle_Root : Uint_256;
      Timestamp : Uint_32;
      Bits : Uint_32;
      Nonce : Uint_32;
   end record;

   type Transaction_Array is array (Integer range <>) of Uint_256;

   type Block_Type (Num_Transactions : Integer) is record
      Header : Block_Header;
      Transactions : Transaction_Array (1 .. Num_Transactions);
   end record;
```

As discussed, a block is simply the list of transactions plus the block header which contains additional information. With respect to the fields for the block header, for this blog post you only need to understand two fields:

- **Prev_Block** a 256-bit hash value for the previous block (this is the prev pointer I mentioned before)
- **Merkle_Root** a 256-bit hash value which summarizes the contents of the block and guarantees that when the contents change, the block header changes as well. I will explain how it is computed later in this post.

The only piece of information that's missing is that Bitcoin usually uses the SHA256 hash function twice to compute a hash. So instead of just computing SHA256(data), usually SHA256(SHA256(data)) is computed. One can write such a double hash function in Ada as follows, using the GNAT.SHA256 library and String as a type for a data buffer (we assume a little-endian architecture throughout the document, but you can use the GNAT compiler's Scalar_Storage_Order feature (https://www.adacore.com/gems/gem-140-bridging-the-endianness-gap) to make this code portable)

```
with GNAT.SHA256; use GNAT.SHA256;

   function Double_Hash (S : String) return Uint_256 is
      D : Binary_Message_Digest := Digest (S);
      T : String (1 .. 32);
      for T'Address use D'Address;
      D2 : constant Binary_Message_Digest := Digest (T);

      function To_Uint_256 is new Ada.Unchecked_Conversion
        (Source => Binary_Message_Digest,
         Target => Uint_256);
   begin
      return To_Uint_256 (D2);
   end Double_Hash;
```

The hash of a block is simply the hash of its block header. This can be expressed in Ada as follows (assuming that the size in bits of the block header, Block_Header'Size in Ada, is a multiple of 8):

```
   function Block_Hash (B : Block_Type) return Uint_256 is
      S : String (1 .. Block_Header'Size / 8);
      for S'Address use B.Header'Address;
   begin
      return Double_Hash (S);
   end Block_Hash;
```

Now we have everything we need to check the integrity of the outermost layer of the blockchain. We simply iterate over all blocks and check that the previous block indeed has the hash used to point to it:

```
declare
   Cur : String :=
     "000000000000000000044e859a307b60d66ae586528fcc6d4df8a7c3eff132456";
   S : String (1 ..64);
begin
   loop
      declare
         B : constant Block_Type := Get_Block (Cur);
      begin
         S := Uint_256_Hex (Block_Hash (B));
         Put_Line ("checking block hash = " & S);
```

```
         if not (Same_Hash (S,Cur)) then
            Ada.Text_IO.Put_Line ("found block hash mismatch");
         end if;
         Cur := Uint_256_Hex (B.Prev_Block);
      end;
   end loop;
end;
```

A few explanations: the *Cur* string contains the hash of the current block as a hexadecimal string. At each iteration, we fetch the block with this hash (details in the next paragraph) and compute the actual hash of the block using the *Block_Hash*function. If everything matches, we set *Cur* to the contents of the *Prev_Block* field. *Uint_256_Hex* is the function to convert a hash value in memory to its hexadecimal representation for display.

One last step is to get the actual blockchain data. The size of the blockchain is now 150GB and counting, so this is actually not so straightforward! For this blog post, I added 12 blocks in JSON format to the github repository, making it self-contained. The *Get_Block* function reads a file with the same name as the block hash to obtain the data, starting at a hardcoded block with the hash mentioned in the code. If you want to verify the whole blockchain using the above code, you have to either query the data using some website such as blockchain.info, or download the blockchain on your computer, for example using the Bitcoin Core client (https://bitcoin.org/en/bitcoin-core/), and update *Get_Block* accordingly.

How to compute the Merkle Root Hash

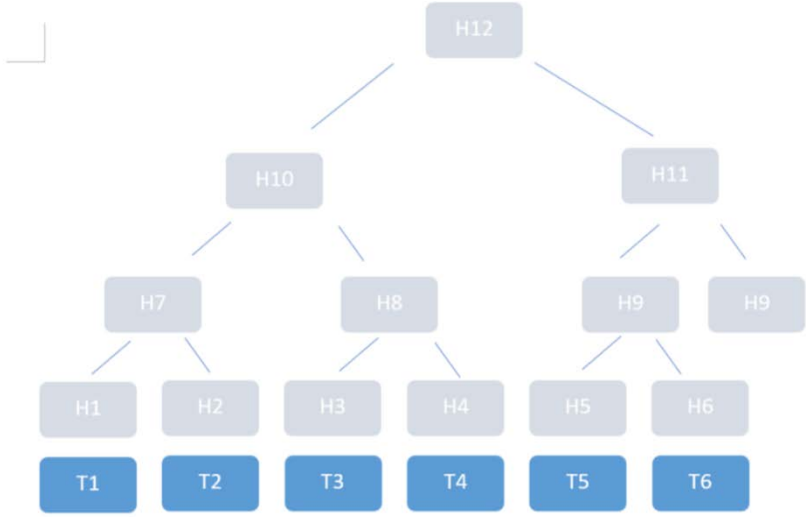
So far, we were able to verify the proper chaining of the blockchain, but what about the contents of the block? The objective is now to come up with the Merkle root hash mentioned earlier, which is supposed to "summarize" the block contents: that is, it should change for any slight change of the input.

First, each transaction is again identified by its hash, similar to how blocks are identified. So now we need to compute a single hash value from the list of hashes for the transactions of the block. Bitcoin uses a hash function which combines *two* hashes into a single hash:

```
function SHA256Pair (U1, U2 : Uint_256) return Uint_256 is
   type A is array (1 .. 2) of Uint_256;
   X : A := (U1, U2);
   S : String (1 .. X'Size / 8);
   for S'Address use X'Address;
begin
   return Double_Hash (S);
end SHA256Pair;
```

Basically, the two numbers are put side-by-side in memory and the result is hashed using the double hash function.

Now we *could* just iterate over the list of transaction hashes, using this combining function to come up with a single value. But it turns out Bitcoin does it a bit differently; hashes are combined using a scheme that's called a Merkle tree:



One can imagine the transactions (T1 to T6 in the example) be stored at the leaves of a binary tree, where each inner node carries a hash which is the combination of the two child hashes. For example, H7 is computed from H1 and H2. The root node carries the "Merkle root hash", which in this way summarizes all transactions. However, this

image of a tree is just that - an image to show the order of hash computations that need to be done to compute the Merkle root hash. There is no actual tree stored in memory.

There is one peculiarity in the way Bitcoin computes the Merkle hash: when a row has an odd number of elements, the last element is combined with itself to compute the parent hash. You can see this in the picture, where H9 is used twice to compute H11.

The Ada code for this is quite straightforward:

```
function Merkle_Computation (Tx : Transaction_Array) return Uint_256 is
   Max : Integer :=
       (if Tx'Length rem 2 = 0 then Tx'Length else Tx'Length + 1);
   Copy : Transaction_Array (1 .. Max);
begin
   if Tx'Length = 1 then
      return Tx (Tx'First);
   end if;
   if Tx'Length = 0 then
      raise Program_Error;
   end if;
   Copy (1 .. Tx'Length) := Tx;
   if (Max /= Tx'Length) then
      Copy (Max) := Tx (Tx'Last);
   end if;
   loop
      for I in 1 .. Max / 2 loop
         Copy (I) := SHA256Pair (Copy (2 * I - 1), Copy (2 *I ));
      end loop;
      if Max = 2 then
         return Copy (1);
      end if;
      Max := Max / 2;
      if Max rem 2 /= 0 then
         Copy (Max + 1) := Copy (Max);
         Max := Max + 1;
      end if;
   end loop;
end Merkle_Computation;
```

Note that despite the name, the input array only contains transaction *hashes* and not actual transactions. A copy of the input

array is created at the beginning; after each iteration of the loop in the code, it contains one level of the Merkle tree. Both before and inside the loop, *if* statements check for the edge case of combining an odd number of hashes at a given level.

We can now update our checking code to also check for the correctness of the Merkle root hash for each checked block. You can check out the whole code from this repository; the branch “blogpost_1” will stay there to point to the code as shown here.

Why does Bitcoin compute the hash of the transactions in this way? Because it allows for a more efficient way to prove to someone that a certain transaction is in the blockchain.

Suppose you want to show someone that you sent her the required amount of Bitcoin to buy some product. The person could, of course, download the entire block you indicate and check for themselves, but that’s inefficient. Instead, you could present them with the chain of hashes that leads to the root hash of the block.

If the transaction hashes were combined linearly, you would still have to show them the entire list of transactions that come after yours in the block. But with the Merkle hash, you can present them with a “Merkle proof”: that is, just the hashes required to compute the *path* from your transaction to the Merkle root. In your example, if your transaction is T3, it’s enough to also provide H4, H7 and H11: the other person can compute the Merkle root hash from that and compare it with the “official” Merkle root hash of that block.

When I first saw this explanation, I was puzzled why an attacker couldn’t modify transaction T3 to T3b and then “invent” the hashes H4b, H7b and H11b so that the Merkle root hash H12 is unchanged. But the cryptographic nature of the hash function prevents this: today, there is no known attack against the hash function SHA256 used in Bitcoin that would allow inventing such input values (but for the weaker hash function SHA1 such collisions have been found : https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html).

Wrap-Up

In this blog post I have shown Ada code that can be used to verify the data integrity of blocks from the Bitcoin blockchain. I was able to check the block and Merkle root hashes for all the blocks in the blockchain in a

few hours on my computer, though most of the time was spent in Input/Output to read the data in.

There are many more rules that make a block valid, most of them related to transactions. I hope to cover some of them in later blog posts.

This chapter was originally published at
https://blog.adacore.com/bitcoin-in-ada

Ada on the micro:bit

By Fabien Chouteau

The micro:bit is a very small ARM Cortex-M0 board designed by the BBC for computer education. It's fitted with a Nordic nRF51 Bluetooth enabled 32bit ARM microcontroller. At $15 it is one of the cheapest yet most fun piece of kit to start embedded programming.

In this blog post I will explain how to start programming your micro:bit in Ada.

How to set up the Ada development environment for the Micro:Bit

pyOCD programmer

The micro:bit comes with an embedded programming/debugging probe implementing the CMSIS-DAP protocol defined by ARM. In order to use it, you have to install a Python library called pyOCD. Here is the procedure:

On Windows:

Download the binary version of pyOCD from this link:

https://launchpad.net/gcc-arm-embedded-misc/pyocd-binary/pyocd-20150430/+download/pyocd_win.exe

Plug your micro:bit using an USB cable and run pyOCD in a terminal:

```
C:\Users\UserName\Downloads>pyocd_win.exe -p 1234 -t nrf51822
Welcome to the PyOCD GDB Server Beta Version
INFO:root:Unsupported board found: 9900
INFO:root:new board id detected: 9900000037024e450073201100000021000000009796990
1
INFO:root:board allows 5 concurrent packets
INFO:root:DAP SWD MODE initialised
INFO:root:IDCODE: 0xBB11477
INFO:root:4 hardware breakpoints, 0 literal comparators
INFO:root:CPU core is Cortex-M0
INFO:root:2 hardware watchpoints
INFO:root:GDB server started at port:1234
```

On Linux (Ubuntu):

Install pyOCD from pip:

```
$ sudo apt-get install python-pip
$ pip install --pre -U pyocd
```

pyOCD will need permissions to talk with the micro:bit. Instead of running the pyOCD as privileged user (root), it's better to add a UDEV rules saying that the device is accessible for non-privileged users:

```
$ sudo sh -c 'echo SUBSYSTEM==\"usb\", ATTR{idVendor}==\"0d28\",
ATTR{idProduct}==\"0204\", MODE:=\"666\" > /etc/udev/rules.d/mbed.rules'
$ sudo udevadm control --reload
```

Now that there's a new UDEV rule and if you already plugged your micro:bit before, you have to unplug it and plug it back again.

To run pyOCD, use the following command:

```
$ pyocd-gdbserver -S -p 1234
INFO:root:DAP SWD MODE initialised
INFO:root:ROM table #0 @ 0xf0000000 cidr=b105100d pidr=2007c4001
INFO:root:[0]<e00ff000: cidr=b105100d, pidr=4000bb471, class=1>
INFO:root:ROM table #1 @ 0xe00ff000 cidr=b105100d pidr=4000bb471
INFO:root:[0]<e000e000:SCS-M0+ cidr=b105e00d, pidr=4000bb008, class=14>
INFO:root:[1]<e0001000:DWT-M0+ cidr=b105e00d, pidr=4000bb00a, class=14>
INFO:root:[2]<e0002000:BPU cidr=b105e00d, pidr=4000bb00b, class=14>
INFO:root:[1]<f0002000: cidr=b105900d, pidr=4000bb9a3, class=9, devtype=13, devid=0>
INFO:root:CPU core is Cortex-M0
INFO:root:4 hardware breakpoints, 0 literal comparators
INFO:root:2 hardware watchpoints
INFO:root:Telnet: server started on port 4444
INFO:root:GDB server started at port:1234
[...]
```

Download the Ada Drivers Library

Ada drivers library if a firmware library written in Ada. We currently have support for some ARM Cortex-M microcontrollers like the STM32F4/7 or the nRF51, but also the HiFive1 RISC-V board.

You can download or clone the repository from GitHub: https://github.com/AdaCore/Ada_Drivers_Library

```
$ git clone https://github.com/AdaCore/Ada_Drivers_Library
```

Install the Ada ZFP run-time

In Ada_Drivers_Library, go to the microb:bit example directory and download or clone the run-time from this GitHub repository: https://github.com/Fabien-Chouteau/zfp-nrf51

```
$ cd Ada_Drivers_Library/examples/MicroBit/
$ git clone https://github.com/Fabien-Chouteau/zfp-nrf51
```

Install the GNAT ARM ELF toolchain

If you have a GNAT Pro ARM ELF subscription, you can download the toolchain from your GNATtracker account. Otherwise you can use the Community release of GNAT from this address: https://www.adacore.com/community

Open the example project and build it

Start GNAT Programming studio (GPS) and open the Micro:Bit example project:
"Ada_Drivers_Library/examples/MicroBit/microbit_example.gpr".

Press F4 and then press Enter to build the project.

Program and debug the board

Make sure your pyocd session is still running and then in GPS, start a debug session with the top menu "Debug -> Initialize -> main". GPS will start Gdb and connect it to pyOCD.

In the gdb console, use the "load" command to program the board:

```
(gdb) load
Loading section .text, size 0xbd04 lma 0x0
Loading section .ARM.exidx, size 0x8 lma 0xbd04
[...]
```

Reset the board with this command:

```
(gdb) monitor reset
```

And finally use the "continue" command to run the program:

```
(gdb) continue
```

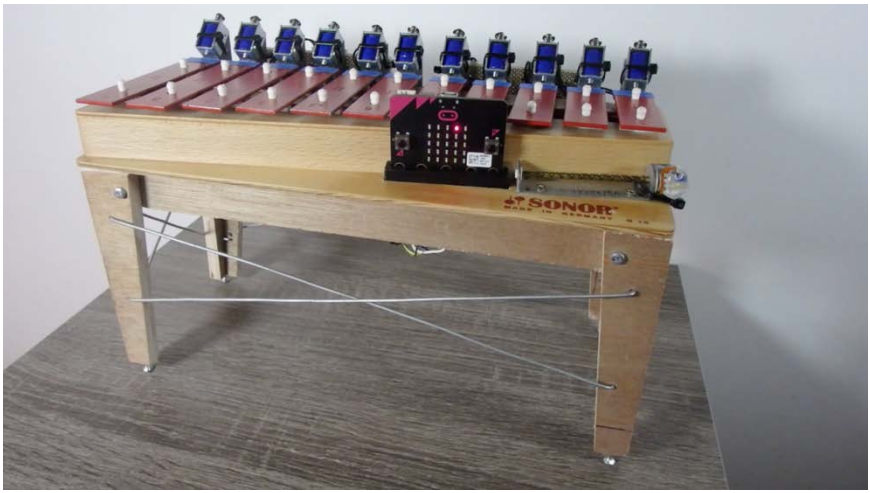
You can interrupt the execution with the "CTRL+backslash" shortcut and then insert breakpoints, step through the application, inspect memory, etc.

Conclusion

That's it, your first Ada program on the Micro:Bit! If you have an issue with this procedure, please tell us in the comments section below.

Note that the current support is limited but we working on adding tasking support (Ravenscar), improving the library as well as the integration into GNAT Programing Studio, so stay tuned.

In the meantime, here is an example of the kind of project that you can do with Ada on the Micro:Bit

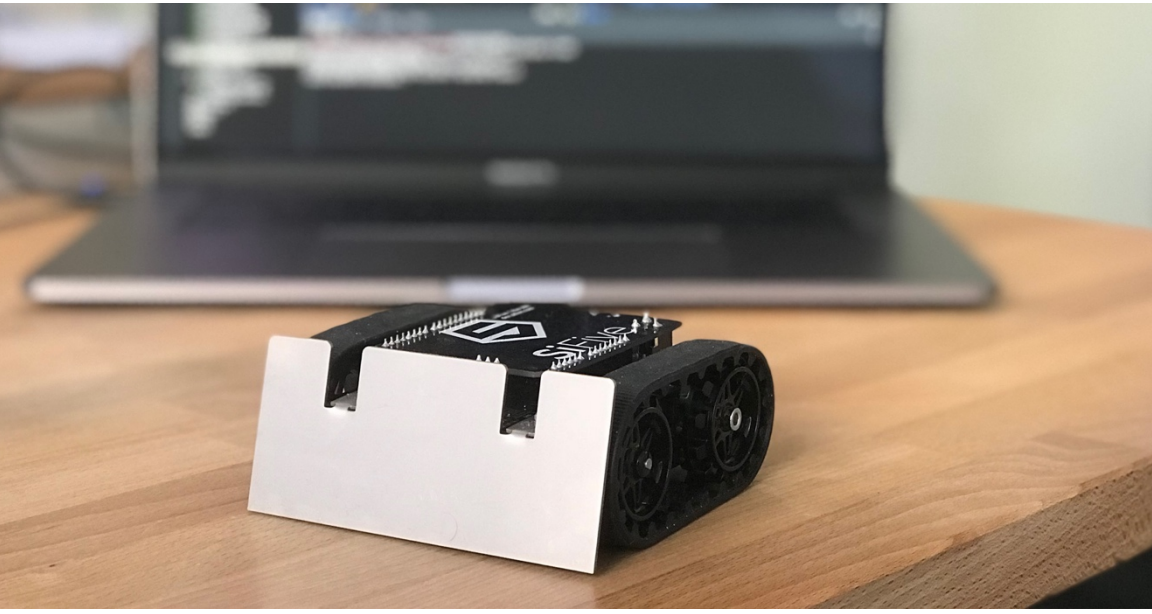


Marble Machine Cover Machine : https://youtu.be/26x4Tfyd_pQ

This chapter was originally published at https://blog.adacore.com/ada-on-the-microbit

SPARKZumo: Ada and SPARK on Any Platform

By Rob Tice



So you want to use SPARK for your next microcontroller project? Great choice! All you need is an Ada 2012 ready compiler and the SPARK tools. But what happens when an Ada 2012 compiler isn't available for your architecture?

This was the case when I started working on a mini sumo robot based on the Pololu Zumo v1.2.

The chassis is complete with independent left and right motors with silicone tracks, and a suite of sensors including an array of infrared reflectance sensors, a buzzer, a 3-axis accelerometer, magnetometer, and gyroscope. The robot's control interface uses a pin-out and footprint compatible with Arduino Uno-like microcontrollers. This is super convenient, because I can use any Arduino Uno compatible board, plug it into the robot, and be ready to go. But the Arduino Uno is an

AVR, and there isn't a readily available Ada 2012 compiler for AVR... back to the drawing board...

Or...

What if we could still write SPARK code and be able to compile it into some C code. Then use the Arduino compiler to compile and link this code in with the Arduino BSPs and runtimes? This would be ideal because I wouldn't need to worry about writing a BSP for the board I am using, and I would only have to focus on the application layer. And I can use SPARK! Luckily, AdaCore has a solution for exactly this!

CCG to the rescue!

The Common Code Generator, or CCG, was developed to solve the issue where an Ada compiler is not available for a specific architecture, but a C compiler is readily available. This is the case for architectures like AVR, PIC, Renesas, and specialized DSPs from companies like TI and Analog Devices. CCG can take your Ada or SPARK code, and "compile" it to a format that the manufacturer's supplied C compiler can understand. With this technology, we now have all of the benefits of Ada or SPARK on any architecture.

Note that this is not fundamentally different from what's already happening in a compiler today. Compilation is essentially a series of translations from one language to the other, each one being used for specific optimization or analysis phase. In the case of GNAT for example the process is as follows:

The Ada code is first translated into a simplified version of Ada (called the expanded tree).

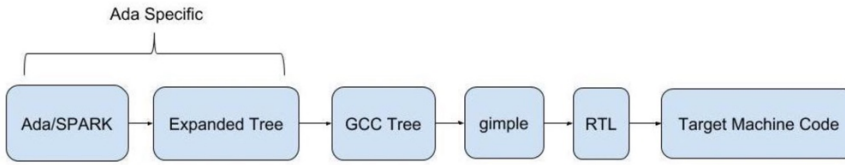
Then into the gcc tree format which is common to all gcc-supported languages.

Then into a format ideal for computing optimizations called gimple.

Then into a generic assembly language called RTL.

And finally to the actual target assembler.

With CCG, C becomes one of these intermediate languages, with GNAT taking care of the initial compilation steps and a target compiler taking care of the final ones. One important consequence of this is that the C code is not intended to be maintainable or modified. CCG is not a translator from Ada or SPARK to C, it's a compiler, or maybe half a compiler.

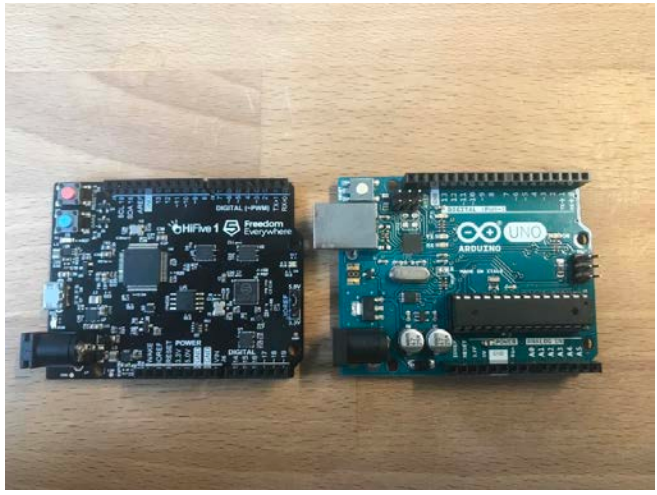


Ada Compilation Steps

There are some limitations to this though, that are important to know, which are today mostly due to the fact that the technology is very young and targets a subset of Ada. Looking at the limitations more closely, they resemble the limitations imposed by the SPARK language subset on a zero-footprint runtime. I would generally use the zero-footprint runtime in an environment where the BSP and runtime were supplied by a vendor or an RTOS, so this looks like a perfect time to use CCG to develop SPARK code for an Arduino supported board using the Arduino BSP and runtime support. For a complete list of supported and unsupported constructs you can visit the CCG User's Guide.

Another benefit I get out of this setup is that I am using the Arduino framework as a hardware abstraction layer. Because I am generating C code and pulling in Arduino library calls, theoretically, I can build my application for many processors without changing my application code. As long as the board is supported by Arduino and is pin compatible with my hardware, my application will run on it!

Abstracting the Hardware



Left to Right: SiFive HiFive1 RISC V board, Arduino Uno Rev 3

For this application I looked at targeting two different architectures, the Arduino Uno Rev 3 which has an ATmega328p on board, and a SiFive HiFive1 which has a Freedom E310 on board. These were chosen because they are pin compatible but are massively different from the software perspective. The ATmega328p is a 16 bit AVR and the Freedom E310 is a 32 bit RISC-V. The system word size isn't even the same! The source code for the project is located here.

In order to abstract the hardware differences away, two steps had to be taken:

Step 1: I used a target configuration file to tell the CCG tool how to represent data sizes during the code generation. By default, CCG assumes word sizes based on the default for the host OS. To compile for the 16 bit AVR, I used the target.atp file located in the base directory to inform the tool about the layout of the hardware. The configuration file looks like this:

```
Bits_BE                       0
Bits_Per_Unit                 8
Bits_Per_Word                16
Bytes_BE                      0
Char_Size                     8
Double_Float_Alignment        0
Double_Scalar_Alignment       0
Double_Size                  32
Float_Size                   32
Float_Words_BE                0
Int_Size                     16
Long_Double_Size             32
Long_Long_Size               64
Long_Size                    32
Maximum_Alignment            16
Max_Unaligned_Field          64
Pointer_Size                 32
Short_Enums                   0
Short_Size                   16
Strict_Alignment              0
System_Allocator_Alignment   16
Wchar_T_Size                 16
Words_BE                      0
float          15   I   32   32
double         15   I   32   32
```

Step 2: The bsp folder contains all of the differences between the two boards that were necessary to separate out. This is also where the

Arduino runtime calls were pulled into the Ada code. For example, in bsp/wire.ads you can see many pragma Import calls used to bring in the Arduino I2C calls located in wire.h.

In order to tell the project which version of these files to use during the compilation, I created a scenario variable in the main project, zumo.gpr

```
type Board_Type is ("uno", "hifive");
Board : Board_Type := external ("board", "hifive");

Common_Sources := ("src/**", "bsp/");
Target_Sources := "";
case Board is
   when "uno" =>
      Target_Sources := "bsp/atmega328p";
   when "hifive" =>
      Target_Sources := "bsp/freedom_e310-G000";
end case;

for Source_Dirs use Common_Sources & Target_Sources;
```

Software Design

Interaction with Arduino Sketch

A typical Arduino application exposes two functions to the developer through the sketch file: setup and loop. The developer would fill in the setup function with all of the code that should be run once at start-up, and then populates the loop function with the actual application programming. During the Arduino compilation, these two functions get pre-processed and wrapped into a main generated by the Arduino runtime. More information about the Arduino build process can be found here: https://github.com/arduino/Arduino/wiki/Build-Process

Because we are using the Arduino runtime we cannot have the actual main entry point for the application in the Ada code (the Arduino pre-processor generates this for us). Instead, we have an Arduino sketch file called SPARKZumo.ino which has the typical Arduino setup() and loop() functions. From setup() we need to initialize the Ada environment by calling the function generated by the Ada binder, sparkzumoinit(). Then, we can call whatever setup sequence we want.

CCG maps Ada package and subprogram namespacing into C-like namespacing, so package.subprogram in Ada would become package__subprogram() in C. The setup function we are calling in the sketch is sparkzumo.setup in Ada, which becomes sparkzumo__setup() after CCG generates the files. The loop function we are calling in the sketch is sparkzumo.workloop in Ada, which becomes sparkzumo__workloop().

Handling Exceptions

Even though we are generating C code from Ada, the CCG tool can still expand the Ada code to include many of the compiler generated checks associated with Ada code before generating the C code. This is very cool because we still have much of the power of the Ada language even though we are compiling to C.

If any of these checks fail at runtime, the __gnat_last_chance_handler is called. The CCG system supplies a definition for what this function should look like, but leaves the implementation up to the developer. For this application, I put the handler implementation in the sketch file, but am calling back into the Ada code from the sketch to perform more actions (like blink LEDs and shut down the motors). If there is a range check failure, or a buffer overflow, or something similar, my __gnat_last_chance_handler will dump some information to the serial port then call back into the Ada code to shut down the motors, and flash an LED on an infinite loop. We should never need this mechanism because since we are using SPARK in this application, we should be able to prove that none of these will ever occur!

Standard.h file

The minimal runtime that does come with the CCG tool can be found in the installation directory under the adalib folder. Here you will find the C versions of the Ada runtimes files that you would typically find in the adainclude directory.

The important file to know about here is the standard.h file. This is the main C header file that will allow you to map Ada to C constructs. For instance, this header file defines the fatptr construct used under Ada arrays and strings, and other integral types like Natural, Positive, and Boolean.

You can and should modify this file to fit within your build environment. For my application, I have included the Arduino.h at the top to bring in the Arduino type system and constructs. Because the Arduino framework defines things like Booleans, I have commented out the versions defined in the standard.h file so that I am consistent with the

rest of the Arduino runtime. You can find the edited version of the standard.h file for this project in the src directory.

Drivers

For the application to interact with all of the sensors available on the robot, we need a layer between the runtime and BSP, and the algorithms. The src/drivers directory contains all of the code necessary to communicate with the sensors and motors. Most of the initial source code for this section was a direct port from the zumo-shield library that was originally written in C++. After porting to Ada, the code was modified to be more robust by refactoring and adding SPARK contracts.

Algorithms

Even though this is a sumo robot, I decided to start with a line follower algorithm for the proof of concept. The source code for the line follower algorithm can be found in src/algos/line_finder. The algorithm was originally a direct port of the Line Follow example in the zumo-shield examples repo.

SPARKZumo Simple Line Follower :
https://www.youtube.com/watch?v=dFrLtvJ7JcE

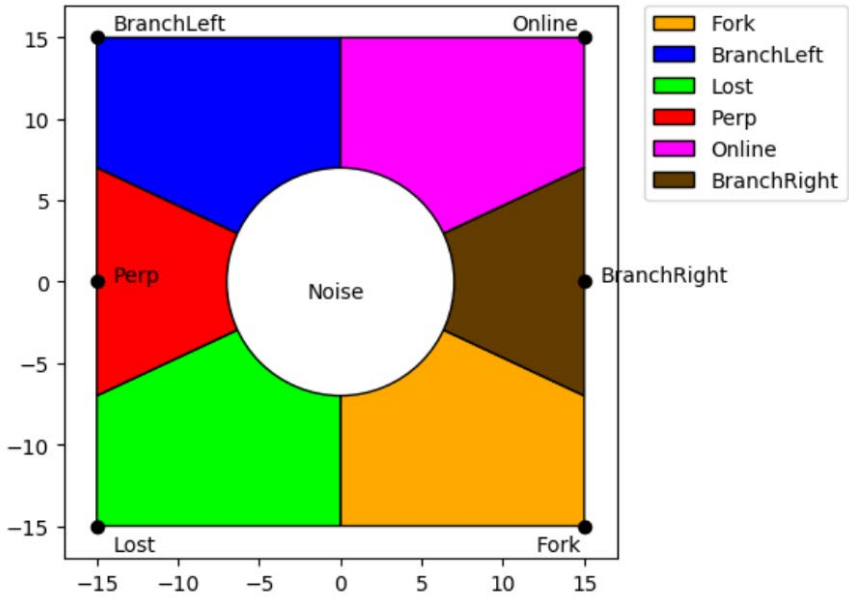
The C++ version of this algorithm worked ok but wasn't really able to handle occasions where the line was lost, or the robot came to a fork, or an intersection. After refactoring and adding SPARK features, I added a detection lookup so that the robot could determine what type of environment the sensors were looking at. The choices are: Lost (meaning no line is found), Online (meaning there's a single line), Fork (two lines diverge), BranchLeft (left turn), BranchRight (right turn), Perpendicular intersection (make a decision to go left or right), or Unknown (no clue what to do, let's keep doing what we were doing and see what happens next). After detecting a change in state, the robot would make a decision like turn left, or turn right to follow a new line. If the robot was in a Lost state, it would go into a "re-finding" algorithm where it would start to do progressively larger circles.

SPARKZumo Line Finding Algorithm:
https://www.youtube.com/watch?v=SzpKmpr4VlQ

This algorithm worked ok as well, but was a little strange. Occasionally, the robot would decide to change direction in the middle of a line, or start to take a branch and turn back the other way. The reason for this was that the robot was detecting spurious changes in state and reacting to them instantaneously. We can call this state noise. In order

to minimize this state noise, I added a state low-pass filter using a geometric graph filter.

The Geometric Graph Filter



Example plot of geometric graph filter

If you ask a mathematician they will probably tell you there's a better way to filter discrete states than this, but this method worked for me! Lets picture mapping 6 points corresponding to the 6 detection states onto a 2d graph, spacing them out evenly along the perimeter of a square. Now, let's say we have a moving window average with X positions. Each time we get a state reading from the sensors we look up the corresponding coordinate for that state in the graph and add the coordinate to the window. For instance, if we detect a Online state our corresponding coordinate is (15, 15). If we detect a Perpendicular state our coordinate is (-15, 0). And so on. If we average over the window we will end up with a coordinate somewhere in the inside of the square. If we then section off the area of the square into sections, and assign each section to map to the corresponding state, we will then find that our average is sitting in one of those sections that maps to one of our states.

For an example, let's assume our window is 5 states wide and we have detected the following list of states (BranchLeft, BranchLeft, Online, BranchLeft, Lost). If we map these to coordinates we get the following

window: ((-15, 15), (-15, 15), (15, 15), (-15, 15), (-15, -15)). When we average these coordinates in the window we get a point with the coordinates (-9, 9). If we look at our lookup table we can see that this coordinate is in the BranchLeft polygon.

One issue that comes up here is that when the average point moves closer to the center of the graph, there's high state entropy, meaning our state can change more rapidly and noise has a higher effect. To solve this, we can hold on to the previous calculated state, and if the new calculated state is somewhere in the center of the graph, we throw away the new calculation and pass along the previous calculation. We don't purge the average window though so that if we get enough of one state, the average point can eventually migrate out to that section of the graph.

To avoid having to calculate this geometry every time we get a new state, I generated a lookup table which maps every point in the polygon to a state. All we have to do is calculate the average in the window and do the lookup at runtime. There are some python scripts that are used to generate most of the src/algos/line_finder/geo_filter.ads file. This script also generates a visual of the graph. For more information on these scripts, see part #2 [COMING SOON!!] of this blog post! One issue that I ran into was that I had to use a very small graph which decreased my ability to filter. This is because the amount of RAM I had available on the Arduino Uno was very small. The larger the graph, the larger the lookup table, the more RAM I needed.

There are a few modifications to this technique that could be done to make it more accurate and more fair. Using a square and only 2 dimensions to map all the states means that the distance between any two states is different than the distance between any other 2 states. For example, it's easier to switch between BranchLeft and Online than it is to switch between BranchLeft and Fork. For the proof of concept this technique worked well though.

SPARKZumo Advanced Line Follower:
https://www.youtube.com/watch?v=5hsxAckSXgk

Future Activity

The code still needs a bit of work to get the IMU sensors up and going. We have another project called the Certyflie which has all of the gimbal calculations to synthesize roll, pitch, and yaw data from an IMU. The Arduino Uno is a bit too weak to perform these calculations properly. One issue is that there is no floating point unit on the AVR. The RISC-V has an FPU and is much more powerful. One option is to add a

bluetooth transceiver to the robot and send the IMU data back to a terminal on a laptop for synthesization.

Another issue that came up during this development is that the HiFive board uses level shifters on all of the GPIO lines. The level shifters use internal pull-ups which means that the processor cannot read the reflectance sensors. The reflectance sensor is actually just a capacitor that is discharged when light hits the substrate. So to read the sensor we need to pull the GPIO line high to charge the capacitor then pull it low and read the amount of time it takes to discharge. This will tell us how much light is hitting the sensor. Since the HiFive has the pull ups on the GPIO lines, we can't pull the line low to read the sensor. Instead we are always charging the sensor. More information about this process can be found on the IR sensor manufacturer's website under How It Works: https://www.pololu.com/product/1419

As always, the code for the entire project is available here: https://github.com/Robert-Tice/SPARKZumo

Integrating the Arduino Build Environment Into GPS

Next we go through how to actually integrate a CCG application in with other source code and how to create GPS plugins to customize features like automating builds and flashing hardware.

The Build Process

At the beginning of our build process we have a few different types of source files that we need to bring together into one binary, Ada/SPARK, C++, C, and an Arduino sketch. During a typical Arduino build, the build system converts the Arduino sketch into valid C++ code, brings in any libraries (user and system) that are included in the sketch, synthesizes a main, compiles and links that all together with the Arduino runtime and selected BSP, and generates the resulting executable binary. The only step we are adding to this process is that we need to run CCG on our SPARK code to generate a C library that we can pass to the Arduino build as a valid Arduino library. The Arduino sketch then pulls the resulting library into the build via an include.

Build Steps

From the user's perspective, the steps necessary to build this application are as follows:

1. Run CCG on the SPARK/Ada Code to produce C files and Ada Library Information files, or ali files. For more information on these files, see the GNAT Compilation Model documentation.
2. Copy the resulting C files into a directory structure valid for an Arduino library

We will use the lib directory in the main repo to house the generated Arduino library.

3. Run c-gnatls on the ali files to determine which runtime files our application depends on.
4. Copy those runtime files into the Arduino library structure.
5. Make sure our Arduino sketch has included the header files generated by the CCG tool.
6. Run the arduino-builder tool with the appropriate options to tell the tool where our library lives and which board we are compiling for.

The arduino-builder tool will use the .build directory in the repo to stage the build

7. Then we can flash the result of the compilation to our target board.

That seems like a lot of work to do every time we need to make a change to our software!

Since these steps are the same every time, we can automate this. Since we should try to make this as host agnostic as possible, meaning we would like for this to be used on Windows and Linux, we should use a scripting language which is fairly host agnostic. It would also be nice if we could integrate this workflow into GPS so that we can develop our code, prove our code, and build and flash our code without leaving our IDE. It is an Integrated Development Environment after all.

Configuration Files

The arduino-builder program is the command line version of the Arduino IDE. When you build an application with the Arduino IDE it creates a build.options.json file with the options you select from the IDE. These options include the location of any user libraries, the

hardware to build for, where the toolchain lives, and where the sketch lives. We can pass the same options to the arduino-builder program or we can pass it the location of a build.options.json file.

For this application I put a build.options.json file in the conf directory of the repository. This file should be configured properly for your build system. The best way, I have found, to get this file configured properly is to install the Arduino IDE and build one of the example applications. Then find the generated build.options.json file generated by the IDE and copy that into the conf directory of the repository. You then only need to modify:

1. The “otherLibrariesFolders” to point to the absolute path of the lib folder in the repo.
2. The”sketchLocation” to point at the SPARKZumo.ino file in the repo.

The other conf files in the conf directory are there to configure the flash utilities. When flashing the AVR on the Arduino Uno, the avrdude flash utility is used. This application takes the information from the flash.yaml file and the path of the avrdude.conf file to configure the flash command. Avrdude uses this to inform the flashing utility about the target hardware. The HiFive board uses openocd as its flashing utility. The openocd.cfg file has all the necessary configuration information that is passed to the openocd tool for flashing.

The GPS Plugin

[DISCLAIMER: This guide assumes you are using version 18.1 or newer of GPS]

Under the hood, GPS, or the GNAT Programming Studio, has a combination of Ada, graphical frameworks, and Python scripting utilities. Using the Python plugin interface, it is very easy to add functionality to our GPS environment. For this application we will add some buttons and menu items to automate the process mentioned above. We will only be using a small subset of the power of the Python interface. For a complete guide to what is possible you can visit the Customizing and Extending GPS (http://docs.adacore.com/live/wave/gps/html/gps_ug/extending.html) and Scripting API Reference for GPS (http://docs.adacore.com/live/wave/gps/html/gps_ug/GPS.html) sections of the GPS User’s Guide.

Plugin Installation Locations

Depending on your use case you can add Python plugins in a few locations to bring them into your GPS environment. There are already a handful of plugins that come with the GPS installation. You can find the list of these plugins by going to Edit->Preferences and navigating to the Plugin tab (near the bottom of the preferences window on the left sidebar). Because these plugins are included with the installation, they live under the installation directory in <installation directory>/share/gps/plug-ins. If you would like to modify you installation, you can add your plugins here and reload GPS. They will then show up in the plugin list. However, if you reinstall GPS, it will overwrite your plugin!

There is a better place to put your plugins such that they won't disappear when you update your GPS installation. GPS adds a folder to your Home directory which includes all your user defined settings for GPS, such as your color theme, font settings, pretty printer settings, etc. This folder, by default, lives in <user's home directory>/.gps. If you navigate to this folder you will see a plug-ins folder where you can add your custom plugins. When you update your GPS installation, this folder persists.

Depending on your application, there may be an even better place to put your plugin. For this specific application we really only want this added functionality when we have the SPARKzumo project loaded. So ideally, we want the plugin to live in the same folder as the project, and to load only when we load the project. To get this functionality, we can name our plugin <project file name>.ide.py and put it in the same directory as our project. When GPS loads the project, it will also load the plugin. For example, our project file is named zumo.gpr, so our plugin should be called zumo.ide.py. The source for the zumo.ide.py file is located here: https://github.com/Robert-Tice/SPARKZumo/blob/master/zumo.ide.py

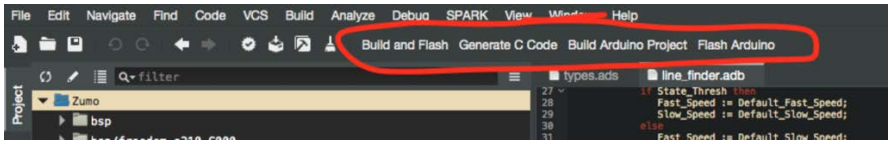
The Plugin Skeleton

When GPS loads our plugin it will call the initialize_project_plugin function. We should implement something like this to create our first button:

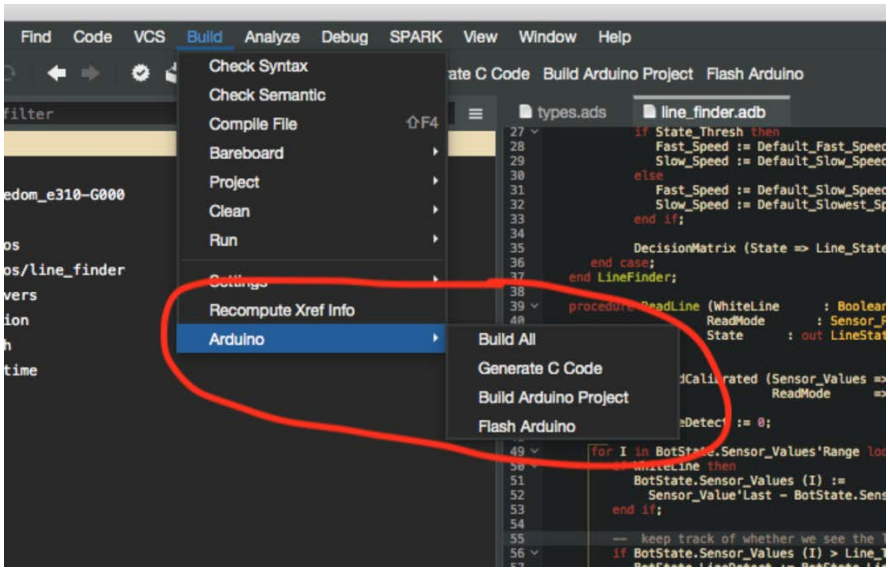
```
import GPS
import gps_utils
class ArduinoWorkflow:
   def __somefunction(self):
       # do stuff here
```

```
    def __init__(self):
        gps_utils.make_interactive(
                callback=self.__somefunction,
                category="Build",
                name="Example",
                toolbar='main',
                menu='/Build/Arduino/' + "Example",
                description="Example")
def initialize_project_plugin():
    ArduinoWorkflow()
```

This simple class will create a button and a menu item with the text Example. When we click this button or menu item it will callback to our somefunction function. Our actual plugin creates a few buttons and menu items that look like this:



Buttons in GPS created by user plug-in



Menus in GPS created by user plug-in

Task Workflows

Now that we have the ability to run some scripts by clicking buttons we are all set! But there's a problem; when we execute a script from a button, and the script takes some time to perform some actions, GPS hangs waiting for the script to complete. We really should be executing our script asynchronously so that we can still use GPS while we are waiting for the tasks to complete. Python has a nice feature called coroutines which can allow us to run some tasks asynchronously. We can be super fancy and implement these coroutines using generators!

Or...

ProcessWrapper

GPS has already done this for us with the task_workflow interface. The task_workflow call wraps our function in a generator and will asynchronously execute parts of our script. We can modify our somefunction function now to look like this:

```
def __somefunction(self, task):
       task.set_progress(0, 1)
       try:
           proc = promises.ProcessWrapper(["script", "arg1", "arg2"],
spawn_console="")
       except:
           self.__error_exit("Could not launch script.")
           return
       ret, output = yield proc.wait_until_terminate()
       if ret is not 0:
           self.__error_exit("Script returned an error.")
           return
       task.set_progress(1, 1)
```

In this function we are going to execute a script called script and pass 2 arguments to it. We wrap the call to the script in a ProcessWrapper which returns a promise. We then yield on the result. The process will run asynchronously, and the main thread will transfer control back to the main process. When the script is complete, the yield returns the stdout and exit code of the process. We can even feed some information back to the user about the progress of the background processes using the task.set_progress call. This registers the task in the task window in GPS. If we have many tasks to run, we can update the task window after each task to tell the user if we are done yet.

TargetWrapper

The ProcessWrapper interface is nice if we need to run an external script but what if we want to trigger the build or one of the gnat tools?

Triggering CCG

Just for that, there's another interface: TargetWrapper. To trigger the build tools, we can run something like this:

```
builder = promises.TargetWrapper("Build All")
retval = yield builder.wait_on_execute()
if retval is not 0:
     self.__error_exit("Failed to build all.")
     return
```

With this code, we are triggering the same action as the Build All button or menu item.

Triggering GNATdoc

We can also trigger the other tools within the GNAT suite using the same technique. For example, we can run the GNATdoc tool against our project to generate the project documentation:

```
gnatdoc = promises.TargetWrapper("gnatdoc")
retval = yield gnatdoc.wait_on_execute(extra_args=["-P",
GPS.Project.root().file().path, "-l"])
   if retval is not 0:
           self.__error_exit("Failed to generate project
documentation.")
           return
```

Here we are calling gnatdoc with the arguments listed in extra_args. This command will generate the project documentation and put it in the directory specified by the Documentation_Dir attribute of the Documentation package in the project file. In this case, I am putting the docs in the docs folder of the repo so that my GitHub repo can serve those via a GitHub Pages website: https://robert-tice.github.io/SPARKZumo/

Accessing Project Configuration

The file that drives the GNAT tools is the GNAT Project file, or the gpr file. This file has all the information necessary for GPS and CCG to process the source files and build the application. We can access all of this information from the plugin as well to inform where to find the

source files, where to find the object files, and what build configuration we are using. For example, to access the list of source files for the project we can use the following Python command: GPS.Project.root().sources().

Another important piece of information that we would like to get from the project file is the current value assigned to the “board” scenario variable. This will tell us if we are building for the Arduino target or the HiFive target. This variable will change the build configuration that we pass to arduino-builder and which flash utility we call. We can access this information by using the following command: GPS.Project.root().scenario_variables(). This will return a dictionary of all scenario variables used in the project. We can then access the “board” scenario variable using the typical Python dictionary syntax GPS.Project.root().scenario_variables()[‘board’].

Determining Runtime Dependencies

Because we are using the Arduino build system to build the output of our CCG tool, we will need to include the runtime dependency files used by our CCG application in the Arduino library directory. To detect which runtime files we are using we can run the c-gnatls command against the ali files generated by the CCG tool. This will output a set of information that we can parse. The output of c-gnatls on one file looks something like this

```
$ c-gnatls -d -a -s obj/geo_filter.ali
geo_filter.ads
geo_filter.adb
<CCG install direction>/libexec/gnat_ccg/lib/gcc/x86_64-pc-linux-
gnu/7.3.1/adainclude/interfac.ads
<CCG install directory>/libexec/gnat_ccg/lib/gcc/x86_64-pc-linux-
gnu/7.3.1/adainclude/i-c.ads
line_finder_types.ads
<CCG install directory>/libexec/gnat_ccg/lib/gcc/x86_64-pc-linux-
gnu/7.3.1/adainclude/system.ads
types.ads
```

When we parse this output we will have to make sure we run c-gnatls against all ali files generated by CCG, we will need to strip out any files listed that are actually part of our sources already, and we will need to remove any duplicate dependencies. The c-gnatls tool also lists the Ada versions of the runtime files and not the C versions. So we need to determine the C equivalents and then copy them into our Arduino

library folder. The __get_runtime_deps function is responsible for all of this work.

Generating Lookup Tables

If you had a chance to look at the first blog post in this series, I talked about a bit about code in this application that was used to do some filtering of discrete states using a graph filter. This involved mapping some states onto some physical geometry and sectioning off areas that belonged to different states. The outcome of this was to map each point in a 2D graph to some state using a lookup table.

To generate this lookup table I used a python library called shapely to compute the necessary geometry and map points to states. Originally, I had this as a separate utility sitting in the utils folder in the repo and would copy the output of this program into the geo_filter.ads file by hand. Eventually, I was able to bring this utility into the plugin workflow using a few interesting features of GPS.

GPS includes pip

Even though GPS has the Python env embedded in it, you can still bring in outside packages using the pip interface. The syntax for installing an external dependency looks something like:

```
import pip
ret = pip.main(["install"] + dependency)
```

Where dependency is the thing you are looking to install. In the case of this plugin, I only need the shapely library and am installing that when the GPS plugin is initialized.

Accessing Ada Entities via Libadalang

The Libadalang library is now included with GPS and can be used inside your plugin. Using the libadalang interface I was able to access the value of user defined named numbers in the Ada files. This was then passed to the shapely application to compute the necessary geometry.

```
ctx = lal.AnalysisContext()
unit = ctx.get_from_file(file_to_edit)
myVarNode = unit.root.findall(lambda n: n.is_a(lal.NumberDecl) and
n.f_ids.text=='my_var')
value = int(myVarNode[0].f_expr.text)
```

This snippet creates a new Libadalang analysis context, loads the information from a file and searches for a named number declaration

called ‘my_var’. The value assigned to ‘my_var’ is then stored in our variable value.

I was then able to access the location where I wanted to put the output of the shapely application using Libadalang:

```
array_node = unit.root.findall(lambda n: n.is_a(lal.ObjectDecl) and
n.f_ids.text=='my_array')
agg_start_line =
int(array_node[0].f_default_expr.sloc_range.start.line)
agg_start_col =
int(array_node[0].f_default_expr.sloc_range.start.column)
agg_end_line = int(array_node[0].f_default_expr.sloc_range.end.line)
agg_end_col = int(array_node[0].f_default_expr.sloc_range.end.column)
```

This gave me the line and column number of the start of the array aggregate initializer for the lookup table ‘my_array’.

Editing Files in GPS from the Plugin

Now that we have the computed lookup table, we could use the typical python file open mechanism to edit the file at the location obtained from Libadalang. But since we are already in GPS, we could just use the GPS.EditorBuffer interface to edit the file. Using the information from our shapely application and the line and column information obtained from Libadalang we can do this:

```
buf = GPS.EditorBuffer.get(GPS.File(file_to_edit))
agg_start_cursor = buf.at(agg_start_line, agg_start_col)
agg_end_cursor = buf.at(agg_end_line, agg_end_col)
buf.delete(agg_start_cursor, agg_end_cursor)
array_str = "(%s));" % ("(%s" % "),\n(".join([', '.join([item for
item in row]) for row in array]))
buf.insert(agg_start_cursor, array_str[agg_start_col - 1:])
```

First we open a buffer to the file that we want to edit. Then we create a GPS.Location for the beginning and end of the current array aggregate positions that we obtained from Libadalang. Then we remove the old information in the buffer. We then turn the array we received from our shapely application into a string and insert that into the buffer.

We have just successfully generated some Ada code from our GPS plugin!

Writing Your Own Python Plugin

Most probably, there is already a plugin that exists in the GPS distribution that does something similar to what you want to do. For this plugin, I used the source for the plugin that enables flashing and debugging of bare-metal STM32 ARM boards. This file can be found in your GPS installation at <install directory>/share/gps/support/ui/board_support.py. You can also see this file on the GPS GitHub repository here: https://github.com/AdaCore/gps/blob/master/share/support/ui/board_support.py

In most cases, it makes sense to search through the plugins that already exist to get a starting point for your specific application, then you can fill in the blanks from there. You can view the entire source of GPS on AdaCore's Github repository: https://github.com/AdaCore/gps

That wraps up the overview of the build system for this application. The source for the project can be found here: https://github.com/Robert-Tice/SPARKZumo. Feel free to fork this project and create new and interesting things.

Happy Hacking!

Embedded SPARK & Ada Use Cases
Updated March 2018