Master of Science HES-SO in Engineering
Av. de Provence 6
CH-1007 Lausanne

Master of Science HES-SO in Engineering

Orientation : Technologies industrielles (TIN)

ESPecIaL :
an Embedded Systems Programming Language

Fait par

Christopher Métrailler

Sous la direction de
Dr Pierre-André Mudry
HES-SO // Valais, Systems Engineering

Expert
Prof. Pascal Felber
Université de Neuchâtel, Institut d'informatique

Lausanne, HES-SO//Master, 6 février 2015

Accepté par la HES SO//Master (Suisse, Lausanne) sur proposition de

Prof. Pierre-André Mudry, conseiller de travail de Master

Prof. Pascal Felber, expert principal

Lausanne, 6 février 2015

Prof. Pierre-André Mudry	Prof. Pierre Pompili
Conseiller	Responsable de la filière Systèmes industriels

Layout based on the EPFL EDOC LaTeX 2$_\varepsilon$ template and BibTeX.
Report version 1.1 - April 2015.

Abstract

Nowadays embedded systems, available at very low cost, are becoming more and more present in many fields such as industry, automotive and education. This master thesis presents a prototype implementation of an embedded systems programming language.

This report focuses on a high-level language, specially developed to build embedded applications, based on the dataflow paradigm. Using ready-to-use blocks, the user describes the block diagram of his application, and its corresponding C++ code is generated automatically, for a specific target embedded system.

With the help of this prototype Domain Specific Language (DSL), implemented using the Scala programming language, embedded applications can be built with ease. Low-level C/C++ codes are no more necessary. Real-world applications based on the developed Embedded Systems Programming Language are presented at the end of this document.

Contents

Chapter 1

Introduction

Embedded systems are becoming more and more present. They are very popular, widely used and cheap. A multitude of these systems are available on the market. As an example, Arduino boards have a great success. These development boards are available at very low cost and allow to build applications using a specific language, similar to C or C++. Other ARM based computers, like the Raspberry Pi, are more powerful systems. They are used a lot in the educational field, especially to introduce programming concepts at school (for instance using the Python programming language).

Several Visual Programming Languages (VPL) have been developped for educational purposes. They help to build applications using a visual editor and avoiding to use a specific programming language. Some of these visual languages will be presented in the chapter 2. The majority of them are used to build games or animations on computer. The other part have been developed specially for embedded systems.

Developing applications for embedded systems is hard because low-level programming languages are necessary. Some proprietary tools helps to convert an UML model (for instance a class diagram) to a generated C/C++ or Java code. Rational Rhapsody[1] is one of this tool. Unfortunately, the generated code produced by this tool cannot be executed directly : some methods must be implemented manually using low-level C/C++ code. Depending on the target, other adaptations are necessary.

Based on theses observations, the goal of this project is to build a new prototype programming language for embedded systems. Using a specific syntax based on the dataflow paradigm, this high-level language allows to describe the behavior and the block diagram of an application (not its concrete implementation). Using ready-to-use components available in the framework, the user can connect blocks together to build the application. Each block can be seen as a black-box, with inputs and outputs, used to compute predefined functions. These blocks are used to build the logic of the application and also to control specific peripherals of the embedded system. Then, the developed framework generates automatically the C++ code corresponding to the application. The code can be used out of the box to program the target embedded system.

1 IBM - Software - Rational Rhapsody family - http://www-03.ibm.com/software/products/en/ratirhapfami

As a proof of concept, the embedded systems programming language is developed using an internal Domain Specific Language (DSL) in Scala. With the help of this high-level language, embedded applications can be built with ease, avoiding the use of low level programming languages like C or C++. This prototype language helps non programmer users to build portable applications for embedded systems.

Outline

Before presenting the developed framework specification, some existing visual programming languages and tools will be presented in chapter 2. Instead of writing textual programs, two visual approaches are introduced : block-based and dataflow programming.

Then, the project architecture overview and the specification of the prototype language are detailed in chapter 3. Some specific features of the Scala programming languages, especially Domain Specific Language (DSL) are introduced.

Chapters 4 and 5 focus on the programming language implementation. The features of the language are presented using simple example applications. The whole transformation process from the application definition to the code generation, simulation and tests is described in details.

Then, the language is shown "in action" in the chapter 6. Two real-world applications developed using the embedded system language are presented. To check the behavior of these applications, two testing approaches have been used.

Finally, the chapter 7 concludes the report, by presenting achieved results and by giving some recommendations and further directions for the project.

Chapter 2

State of the art

Visual programming languages (VPL) are now common, for instance in the educational field to learn basic programming concepts to children. Several programs also includes a visual programming editor to develop simulations, games, automated processes or to create sounds and movies. Instead of writing textual programs using codes, these programs are built graphically in an easy-to-use editor. The model of the application is described visually. It is an abstraction of the concrete implementation, and no code is required. A complete snapshot of available visual programming languages can be found online [Hos14]. Today, VPL are available to describe all kind of programs, in many areas.

Programs can be represented visually in different manners. It can be using blocks, flowcharts or dataflows for instance. A selection of some existing programming languages will be presented in this chapter. Some of them have been developed for education purpose, others especially for embedded systems. Two visual paradigms will be presented in the next two sections.

2.1 Block-based programming

The first type of visual programming language presented here consists of writing applications like puzzle. An application is composed by blocks, which can be combined together to create a program. No code is required, the application is built graphically using a drag&drop interface. Some block-based programming language are presented in the next sections.

2.1.1 The Scratch programming language

Scratch [MIT14] is a free visual programming language developed by the MIT Media Lab. Scratch 1.0 was released in 2007, but it is based on a quite old language called Squeak (created in 1996), which is a dialect of the Smalltalk language, created in 1972.

Scratch is a programming language and online community, designed specially for young people, to create interactive stories, games, and animations with ease, using a visual editor. Ready-to-use blocks are connected to each other, like a puzzle, to create a program (a script). 145 blocks[1] can be used to control active objects, called "sprites" (inspired by EToys programming language). An offline desktop editor is available, and also a web-based editor, presented in figure 2.1. They are both implemented using Flash.

1 List of blocks in Scratch 2.0 - http://wiki.scratch.mit.edu/wiki/Blocks#List_of_Blocks

The program is described visually using blocks on the right of figure 2.1. A color corresponds to block category, like motion, events, control, or sensing. All these blocks, combined together, are used to control and animate one or more sprites, drawn on the top left on figure 2.1. Each block has a special slot, to prevent syntax errors. Basic programming concepts, like loops, conditional tests and variables can be used. Images and sounds can be also added and the user can control a program using mouse and keyboard events. A nice feature of Scratch is that programs can be updated dynamically, when running.

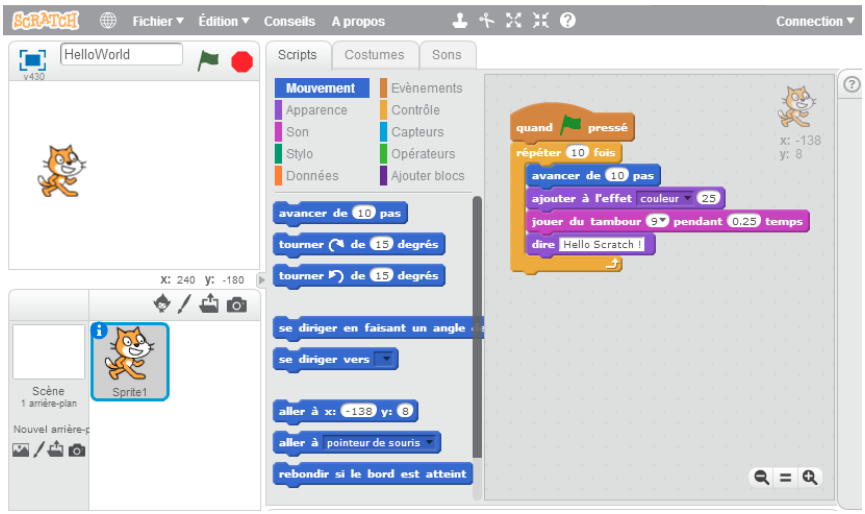


Figure 2.1 – Web based development environment of Scratch 2.0

Limitations

Scratch is multimedia and educational oriented. Programs are described using colorful blocks only. It is not possible to add code (in text format) to write more complex or specific blocks. Scratch supports only basic types: from Scratch 1.4, floating point numbers and some limited functions on strings are supported. The last Scratch version, released on May 2013, now supports custom blocks. They can be used ot create reusable functions with input parameters if needed, but no code can be inserted. An extended implementation of Scratch, called Snap! [MH14] is another programming language which allows to build custom blocks.

The scope of the Scratch programming language is limited. The version 2.0 of the language adds an extension protocol[2] available to connect Scratch desktop applications with the physical world. Javascript or HTTP extensions add specific blocks, available to control external hardware. Some projects are listed bellow :

- The first project is a simple extension board used to connect desktop Scratch programs to the physical world. The *PicoBoard* contains a slider, a button, analogs inputs and a few other sensors. Connected through USB, custom blocks are available to control the extension board.
- The LEGO WeDo Javascript extension add blocks to control the LEGO WeDo Robotic Kit, composed by motors and different sensors (light distance, tilt).

2 http://wiki.scratch.mit.edu/wiki/Scratch_Extension_Protocol_(2.0)

- Finally, some other experimental projects allow Scratch programs (running on desktop) to communicate with Arduino boards. *Arduino For Scratch* (A4S)[3] use the Firmata protocol to communicate with the Arduino board. A second project, *S4A* (Scratch For Arduino)[4] interact with Arduino by sending actuators states and receiving sensor states each 75 ms. Both of these projects are experimental and use the Scratch extension protocol to interact with the editor. All values are transmitted from/to Arduino using a serial connection over USB, and then sent over HTTP to the Scratch IDE.

Scratch has been developed for children, for educational and multimedia purpose, but not for embedded systems. It is widely used by students, schools or teachers to easily create simple games or simulations and learn how to program using a graphical editor. There is also a big community and a lot of Scratch program example are shared. Programs are build like puzzle using blocks, which are great to prevent syntax error.

Some experimental projects presented above can be used to communicate with embedded systems, like Arduino boards. Scratch programs are always running on the desktop and inputs/outputs values are transmitted from the embedded device to the PC over USB.

Finally, Stencyl[5] is a project which extends the Scratch's simple block-snapping interface with new functionality and blocks, to create games for mobile, web and desktop, always without writing code.

2.1.2 Turtle graphics

Turtle graphics is inspired by the Logo programming language [Tho83] in the late 1960s. The TurtleArt [Sug14] has been developed to draw images, using a relative cursor in a two-dimensional plan. The cursor used to draw the image, represents the turtle. Here is an example of a turtle graphics environment :

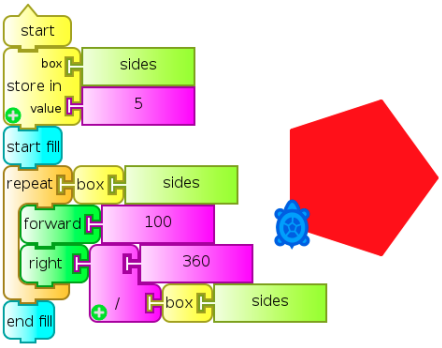


Figure 2.2 – Drawing a filled shape using TurtleArt (source [Sug14])

The turtle, used to draw images, has 3 main attributes: a position, an orientation and a pen (with a color, a width, etc.). A program is composed by a sequence of instructions used to control or move the turtle. In figure 2.2, the program is defined using blocks, in a similar way to the Scratch language. Using these blocks, it is very easy to draw simple images.

3 https://github.com/damellis/A4S
4 http://s4a.cat/
5 Create games for mobile, web and desktop without code - http://www.stencyl.com/

Extended TurtleArt versions are also available. They offer advanced functions to build complex images (using geometrical operations for instance) and more than one turtle can be used. Some advanced applications and results are presented on this website [Sug14].

Many TurtleArt implementations are available. They usually offer a graphical environment to create programs using blocks, like Scratch (see previous section 2.1.1), but TurtleArt programming languages are also available. Programs can be built using specific sequential commands. As an example, a Java implementation is available here[6]. Another interesting open source application, named Kojo[7], can be used to program turtle graphics. Kojo is a complete learning environment developed in Scala, which includes many different features, not only turtle graphics.

2.1.3 Modkit

Modkit [Mod15] is a more recent project, born in 2010 from a Kickstarter project. It is again a visual programming environment, specially developed to program embedded systems. It is heavily inspired by Scratch. Based on the same blocks concept, this drag&drop programming environment makes embedded systems programming easy and accessible to everyone. The Modkit desktop application is presented in figure 2.3.

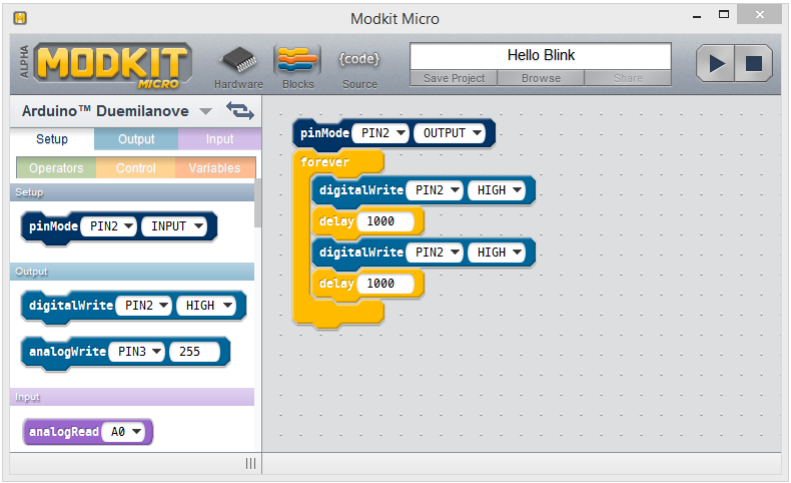


Figure 2.3 – Modkit micro desktop application

Modkit is compatible with many hardware. All Arduino[8] and Arduino like boards are compatible targets. Specific blocks are available to configure and control input and output pins of the board. The program is composed by these blocks, like in Scratch or TurtleArt.

With Modkit, in addition to the block interface of figure 2.3, two other views are available. The first view can be used to configure the pins of the selected target as input or output. Depending on the target, specific pins can be configured as analog or digital. The second view displays the generated code, but this code cannot be modified. It is generated automatically from the visual interface (the opposite way is not available).

6 Turtle Graphics with Java - http://www.java-online.ch/lego/legoEnglish/turtleGrafik.php
7 The Kojo Learning Environment - http://www.kogics.net/kojo
8 Arduino open-source electronics platform - http://www.arduino.cc/

The generated code 2.1 corresponds to the LED blinking program of figure 2.3 :

```
void setup() {
  pinMode(PIN2, OUTPUT);
}

void loop() {
  digitalWrite(PIN2, HIGH);
  delay(1000);
  digitalWrite(PIN2, LOW);
  delay(1000);
}
```

Listing 2.1 – The generated code for an Arduino board

This code is automatically generated from the blocks and cannot be edited. Each block are translated to a C code to create a sequential program (imperative paradigm). The generated code can be used to program the Arduino board, without modification. The board can be linked to the Modkit environment using an USB cable. This part has not been tested, but once the environment is configured and the board detected, the play button should program the target and run the application automatically. The program can only be ran, not debugged, and unfortunately no simulator are available.

Modkit makes embedded programming easy. It is compatible with many targets and a single application is used to develop, compile and run the program with ease. It is great to learn programming in the educational field and to build simple applications.
Unfortunately, only a few control and operator blocks are available. Furthermore, it is only possible to control analog and digital I/O. Embedded systems have more and more power and advanced peripherals. For now, all target features cannot be used. Modkit is still in an alpha version, more functionalities will be probably added soon.

2.1.4 Related work

Several visual block-based programming languages have been presented in the previous sections. Open source libraries, like Google Blocky [Goo14], can be used to build such visual editors with ease. These type of block-based languages are very common today. Other related projects like *Bitbloq* or *Ardublock*, based on the same programming paradigm as Scratch, have been developed specifically to generate programs for embedded systems (like Arduino).

One advantages of these languages is that the sequential generated code looks like the visual program description, and using blocks makes syntax errors completely impossible [Goo14] (no unbalanced parentheses, no missing semicolons, etc.). Visual block-based languages are excellent educative tools and make the programming accessible to kids (no code is required). Unfortunately, complex programs are verbose and not easy to develop, and blocks functionalities can be limited (see the Modkit project in section 2.1.3 for instance).

2.2 Dataflow and flow-based programming

Dataflow and flow-based programming are two other approaches to describe programs visually. These visual programing paradigms will be covered and explained in this section. Some projects that use each of theses approaches will be presented later.

First, terms of dataflow and flow-based can be confusing because they both define visual programming paradigms and execution models. In both cases, applications are defined using "black-boxes" components, connected together to exchange data and information. Applications are represented using graphs. The nodes of the graph are the components and connections are the arcs between them. A simple Flow-Based Programming (FBP) diagram is presented in figure 2.4 :

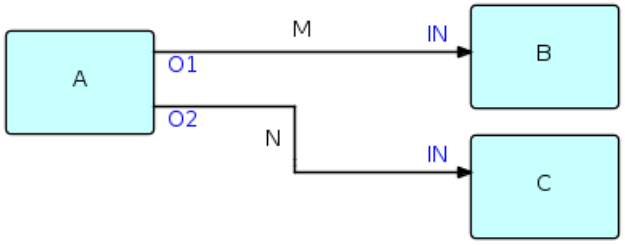


Figure 2.4 – A Flow-Based Programming diagram (source [Mor13])

Before describing how such applications are defined, it is important to distinguish the differences between these two paradigms. According to this article [Mai13], the two models are explained bellow :

1. **Flow-based programming**
   Flow-Based Programming (FBP) is a particular form of dataflow programming, invented by J. Paul Morrison in the early 1970s [Mor13]. This paradigm usually mean asynchronous dataflow programming [Mai13]. In this model, nodes are constantly waiting for messages. Data between nodes are event-based and asynchronous channels are used. Each node of the graph are connected with ports, implemented for instance with queues to continually receive messages from other nodes (like streams). Applications are not defined by a single sequential process, but by a complex network of asynchronous processes [Mai13].

2. **Dataflow programming**
   Dataflow is the synchronous approach. It corresponds more to conventional synchronous programming. This time, the graph is executed in a sequential order. A node is executed once when the data of all its inputs are ready. The node output is computed and the result "flows" to the input of the next node. The same process is executed again and again, until all nodes have been executed [Mai13]. In this model. the scheduler is much more simple because nodes are executed once, in a sequential order.

The "Advances in dataflow programming languages" [JHM04] article reviews many developments that have taken place within dataflow programming languages in the past decade. Other dataflow execution models are also presented in details. This other article [WP94] presents the history and evolution of dataflow languages. Not all of them will be covered in this document, but some projects and software based on the synchronous and asynchronous dataflow programming model will be presented in th next two sections.

2.2.1 NoFlo

Different FBP implementations are available. The NoFlo[9] project is one of them. It was created in 2013 from a KickStarter campaign and is presented here.

NoFlo is a JavaScript (more precisely CoffeeScript) implementation of Flow-Based Programming. It is based on NodeJs and runs in a browser. NoFlo components can be used to build software, defined as graphs. Components are connected together with ports and messages are used to communicate between the components. For instance, components can react to HTTP requests, to writing data to a database. NoFlo applications are built using a web-based editor. NoFlo UI is the name of their IDE for flow-based programming.

NoFlo became popular and similar project have been created or are based on it. One example is the MicroFlo[10] experimental project. It is a flow-based programming runtime for microcontrollers (like Arduino). It is inspired by and designed for integration with NoFlo. Simple applications can be developed visually using the NoFlo UI, but according to the documentation project, it is too early to use it for general tasks.

An other interesting point of NoFlo is that applications can also be developed using a specific Flow-Based Programming language[11]. This is a Domain-Specific Language (DSL) for easy graph definition. The following example presents how to use this DSL to blink a LED. Using MicroFlo, the code 2.2 can be executed on Arduino for instance.

```
# LED blinking on Arduino
# https://github.com/microflo/microflo/blob/master/examples/blink.fbp
timer(Timer) OUT -> IN toggle(ToggleBoolean)
toggle() OUT -> IN led(DigitalWrite)
'300' -> INTERVAL timer()
'13' -> PIN led()
```

Listing 2.2 – Led blinking example in Microflo

The graph application can be created using a visual tool or using the DSL (by writing code). Then, the program 2.2 is converted to a command stream and embedded into the firmware image, which can be loaded on the microcontroller. Finally, the network graph is executed on the device (standalone).

At this stage of the project, the execution model and the scheduler implementation are quite simple and limited (see[10], network execution). Components are naively scheduled. Messages are stored in queues and are delivered to the corresponding components in the main loop of the program. It is expected that scheduling will grow more complicated over time, to support for instance asynchronous input events/interrupts, fair division of the processing time, etc.

Systems based on FBP are growing. The flow-based programming introduction article of J. Paul Morrison [Mor13] presents a lot more projects based on FBP, which implement some or all FBP concepts.

One more interesting example is the Node-RED project. It is a visual tool for wiring Internet of Things application, developed by IBM. Node-RED also provides a browser-based flow editor. It works on the same principle as NoFlo and can be extended with custom blocks (Javascript code).

9 Flow-Based Programming for JavaScript - http://noflojs.org/
10 Flow-based programming runtime for microcontrollers - https://github.com/microflo/microflo
11 FBP flow definition language parser - https://github.com/noflo/fbp

Finally, Node-RED can be executed on the BeagleBone of the Raspberry Pi, using a NodeJs server. Custom components are available to use GPIO within the Node-RED Javascript environment. For this, native libraries must be installed on the hardware.

2.2.2 LabVIEW

The synchronous dataflow approach is presented using the LabVIEW software as an example.

LabVIEW is a scientific software system for laboratory automation and simulation [KMR91]. This professional software, developed by National Instruments and available since 1986, is widely used in the industry. It is a very complete software used to describe visually many types of applications with advanced structures, hierarchical diagrams, etc. It can be also used to develop graphical interfaces.

LabVIEW programs (block diagrams) are described using a dataflow representation. An example is shown in figure 2.5 :

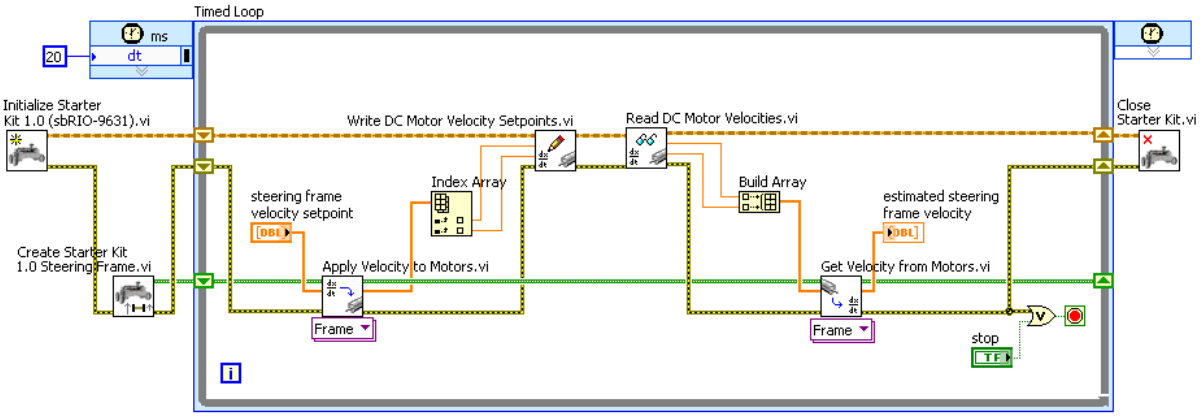


Figure 2.5 – Dataflow programming example in LabVIEW (source [Ins15])

LabVIEW is a well-known DataFlow Visual Programming Language (DFVPL) [JHM04] developed for professional users but not programmers. Ready-to-use components are available in the software and applications can be built using a drag & drop interface. This helps non-programmer users to build complex applications to control an equipment or an automated process in LabVIEW for instance.

According to the official LabVIEW documentation [Ins15], LabVIEW follows a dataflow model. A block diagram node is executed when it receives all required inputs. When a node executes, it produces output data and passes the data to the next node in the dataflow path.

The visual language embedded in LabVIEW is called "G". It is based on the dataflow model, but it is extended with graphical control flow structures. This allows to extend the pure dataflow model, because it is too restrictive for the typical applications of LabVIEW [VM99]. Control structures are available in the "G" language. For instance, a `for` loop structure allows to run a subgraph for a predefined number of times. The summary of the "G" language is available in this article from National Instrument [KMR91].

Finally, many additional toolkits are available to extend the software, for instance to build real time or embedded (DSP) applications.

Chapter 3

An embedded programming language

3.1 Specification

Several existing visual programming languages (VPL) have been presented in the previous chapter. They use different approaches to represent a program visually, instead of describing it with codes, using blocks (flowcharts) or dataflow diagrams. In this chapter, the overview and the specification of the prototype embedded system programming language will be presented.

Modkit, Bitbloq or Ardublock are some projects, presented in the last chapter, developed specifically for embedded systems. All these environments are block-based. The program is described visually, and a sequential C code is generated. The generated code is very similar to its visualization because blocks (like loops, if/case statements and math operators) are very close to the language.

In contrast to block-based VPL, the language developed during this project allows to describe the model and the behavior of a program, and not its concrete implementation. It is an high level language for embedded systems, used to express in a natural way the block diagram / the behavior of the application. According to these articles [HCRP91, Qua04, JHM04], dataflow programming languages are particularly well suited for this purpose.
The embedded system can be seen as a black box (shown in figure 3.1), connected to several external components like buttons, LEDs, actuators, sensors, etc.

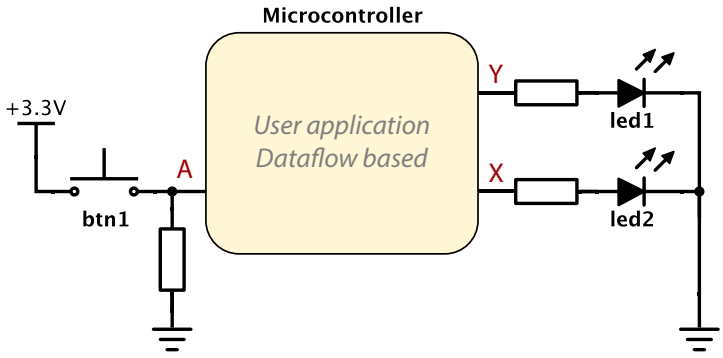


Figure 3.1 – Embedded dataflow programming language

Figure 3.2 shows a typical (simple) application which can be expressed using the developed dataflow language (it will be presented later). The pseudo-code on the left defines the application specification. Its dataflow representation is available on the right : instead of using a textual representation (a code), the application specification is described visually.

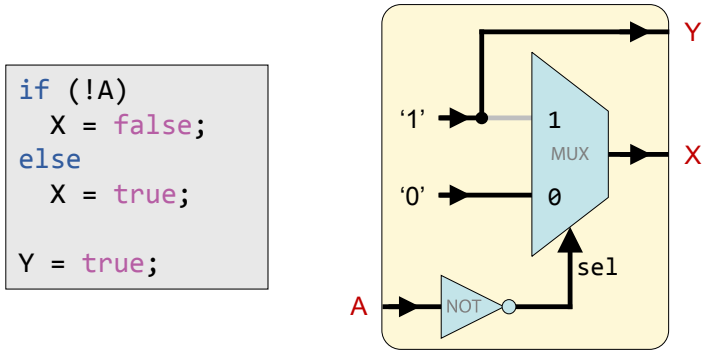


Figure 3.2 – Pseudo-code translated to a dataflow representation (adapted from [VBCG04])

In this simple example, The output `Y` is always ON. `X` is ON only when the input `A` is low.
The dataflow representation (see section 2.2 on page 8) is composed of "black-box" components. Each box is a single function : the application above is composed by a multiplexer bloc, constants values and an inverter gate. Inputs and outputs of these blocs are connected together to form a DataFlow Graph (DFG). The DFG of the application 3.2 is the following :

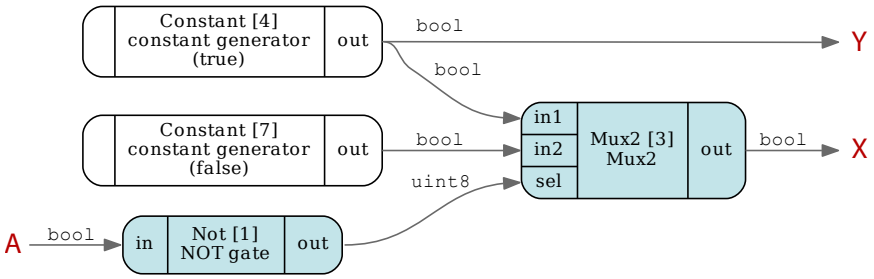


Figure 3.3 – DataFlow Graph (DFG) of the application 3.2

The graph contains the four blocks that compose the application. Inputs are on the left, outputs on the right. Components output ports are connected to input ports in a directed graph. Dependencies between the components (nodes), and the type of the data that "flows" between them are also represented in such a graph. Primitive types of data are exchanged between the components, like bool, integer or floating point values. Each component has a generic number of input and output ports. By convention, they are named `out` or `out1`, `out2`, etc. if many are available (the same for inputs). Ports with a specific function are named with a custom name, like the selection (`sel`) input port of the multiplexer.

The developed language is a high-level language used to describe any DAG (the block diagram of the application) in a natural and concise way, using a specific "dataflow" syntax. The code 3.1 is the textual representation of the application graph 3.3.

```
val mux = Mux2()                // Components decalartion
val not = Not()

mux.out --> X.in                // Connecting ports
A.out --> not.in
not.out --> mux.sel
Constant(true).out --> mux.in1
Constant(false).out --> mux.in2
Constant(true).out --> Y.in
```

Listing 3.1 – Dataflow Domain Specific Language of the application 3.2

This program is a Scala Domain Specific Language (DSL), similar to the flow-based programming language available in NoFlo (see section 2.2.1 on page 9). Severals components are available in the developed framework. Using a specific Scala syntax, components ports are connected together using the "`-->`" operator (an output port to an input port).

Unlike imperative programming, a dataflow program is not represented by a linear instruction sequence, but by its DFG [Fin95, Chapter 6]. This graph give the execution order of the code. The instruction order of the code 3.1 is not important. It must only be a valid Scala code. Components must be declared before being used, but the ports connections order is not important. Output can be connected before inputs for instance. The execution model will be presented later in this chapter.

3.2 Architecture

The code architecture overview of the project is presented in figure 3.4.

The traditional way used to program embedded systems is shown on the left. The user application is developed in C or C++, and libraries are used to control the microcontroller peripherals, I/O, etc. Depending on the target, proprietary tools and toolchain are necessary to program/debug the target.

Another approach used in this project is presented on the right of figure 3.4. This other approach is now composed by two parts. The main difference is that an automated tool converts the model of the application, described in a dataflow graph, to the native C/C++ application for the embedded system.

The frontend part, written in Scala, transform the dataflow model (its graph) to a C/C++ code. This generated code is generic and based on a Hardware Abstraction Layer (HAL) to control the microcontroller peripherals. This allow to support multiple targets, without modifying the high-level application. The backend must be developed once for each target, and then no more low-level C/C++ is required. As a proof of concept, two targets will be supported for this project : a generic ARM development kit and a simulator based on *QEMU* which emulates the real target. They will be presented in the next chapters.

The application model is specified using a custom dataflow DSL, like the program 3.1. The Scala programming language and more specifically Domain Specific Language (DSL) are presented in the next section.

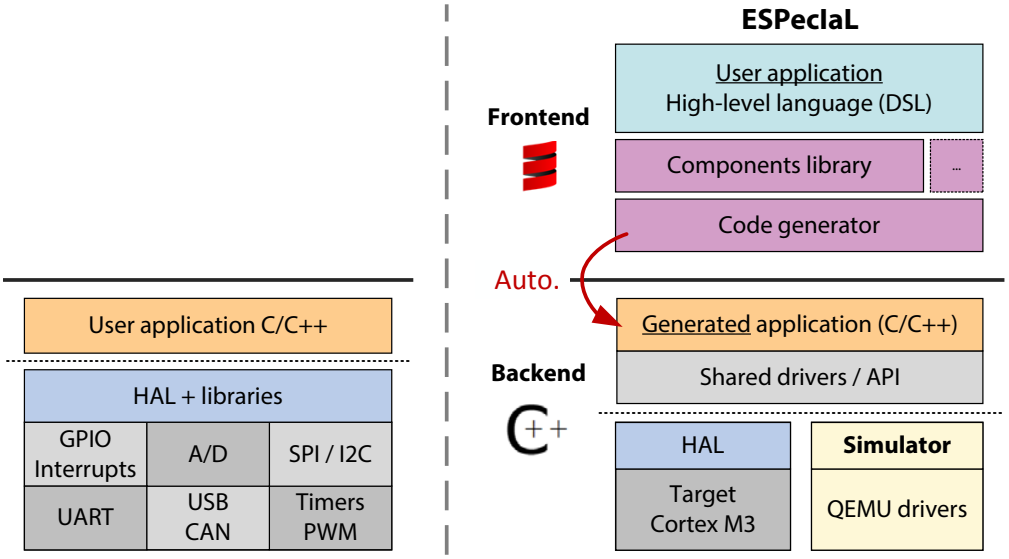


Figure 3.4 – Project architecture overview

3.3 The Scala programming language

The SCALA programming language[1], has been developed since 2001 in the programming methods laboratory at EPFL (Switzerland).
According to [Typ14a], Scala is a general purpose programming language designed to express common patterns in a concise, elegant, and type-safe way. The Scala syntax is lightweight and its primitives expressive [OSV08]. It integrates features of object-oriented and functional languages. Scala programs run on the Java Virtual Machine (JVM) and are compiled directly to regular Java bytecode. Therefore, Scala is interoperable with Java: a Java code can be accessed from and to Scala programs.

In addition to these advantages, Scala also supports Domain Specific Language (DSL). DSL are languages developed to express a custom problem (for a specific need). One well-known example is Matlab. It integrates a DSL to do numerical computing, using a specific syntax. Another example is the NoFlo DSL, presented in section 2.2.1 on page 9.

In this project, the user application will be described using a custom high-level language, based on Scala. The next two sections are focused on two Scala DSL types. They will be presented and also compared to other languages, like DSL build in Java or with other libraries.

3.3.1 Internal DSL

The first type of DSL are called internal DSL. Internal DSL are great to extend the Scala language itself. Features can be added to language to meet specific needs. Internal DSL are built using the Scala compiler, this implies that its syntax must be valid in Scala (the host language).

DSL are a powerful Scala feature [Bag09]. To extend the language, natural looking DSL can be written using features directly available in the Scala language itself. Possibilities are somewhat limited, but the Scala syntax [Ode14, p. 159] is flexible enough to built great internal DSL.

1 The Scala Programming Language - http://www.scala-lang.org/

Some key advantages, described in this book [Gho10] and these articles [Oa04] [Ber11] are summarized bellow :

- First, semicolons, dots and parenthesis, are optional in Scala (but still necessary in some cases). Compared to Java where these characters are mandatory, they are now considered as "noise" in Scala. For instance, Scala has a special syntax for invoking methods of arity-1 (one argument) without using dot and parenthesis. The Scala syntax [Ode14, p. 159] is flexible (more than Java for instance), so it possible to create less verbose, more user-friendly and more natural languages without much effort.

- The Scala syntax also allow to use any ASCII character for method names and Unicode symbols for operators. This can greatly improve the readability of the code.

- Implicit conversions are widely used to automatically convert built-in type to custom objects to extend the language. Combined with operator overloading, they allow to write concise codes, once again.

- Finally, Higher-order Functions (HOF), currying and curly braces (to make code blocks) can be used to makes user-friendly DSL.

DSL are widely used in Scala to extend the language. To illustrate some of the features enumerated above, the code 3.2 presents a DSL available within the Scala library. The `Duration` class and the `Future` trait (both available in the `scala.concurrent` package[2]) extend the language to write and use asynchronous computations with ease, in a more natural way.

```
1  import scala.concurrent._
2  import scala.concurrent.duration._
3  import scala.concurrent.ExecutionContext.Implicits._
4
5  object DemoFuture extends App {
6
7    val f: Future[Long] = future {
8      val t = (Math.random() * 3000).toLong // Simulate an HTTP request
9      Thread.sleep(t)
10     t // Return the computation time
11   }
12
13   f onSuccess {
14     case r => println(s"Future done in $r ms.")
15   }
16
17   // Waits 2 seconds for the Future result, or throws a TimeoutException
18   Await.ready(f, 2 seconds)
19   // Same as : "Await.ready(f, Duration(2, SECONDS))" (without DSL)
20 }
```

Listing 3.2 – A Scala internal DSL example using a `Future`

2 Scala Standard Library - http://www.scala-lang.org/api/current/index.html#scala.concurrent.package

In the example 3.2, an HTTP request is simulated with random time. On line 21, we wait on the `Future` result (a success or a failure) for 2 seconds at most. If this maximum time is reached, a `TimeoutException` is thrown and the program terminates. If not, future result is printed.

On lines 7, 13 and 18, some of Scala features enumerated above make the code more readable and concise (with a minimum of "noise"). For instance, using implicit conversions, timeouts, which are often used with futures, can be simply written as `"2 seconds"`, like a human readable text.

Internal DSL built using Java for instance are more verbose because characters must be added to be conform to the Java syntax. `Factory` or `Builder` patterns [Gho10] can be used to create fluent interfaces [FE05]. This help to build more natural Java internal DSL, but the Java syntax is more strict and more rigid. As a result, internal Java DSL are less attractive because they will always be more verbose (with noise) and less flexible than one built in Scala.

3.3.2 External DSL

Another type of DSL are called external DSL. In this case, the language is parsed using an external tool or a built-in library, and the DSL syntax is totally independent of the host language. Unlike internal DSL, all syntaxes can be supported.
Parser combinators[3] are a way to build parsers using the Scala programming language. As of Scala 2.11, parser combinators are available in an external library, but they are part of the language specification. The grammar of the parser can be described using a particular syntax, directly in Scala using an internal DSL.

Other Scala parser libraries are available. One alternative to the Scala parser combinators is *Parboiled*[4], which can also be used to build external Java DSL. Finally, *ANTLR* (ANother Tool for Language Recognition) [Par13] is another parser generator that can be used to built external DSL in Java. In this case, the grammar of the parser is described in an external file using an *EBNF* syntax, and not directly within Java.

External DSL can be built either in Scala or Java with one of these tools. Using Scala parser combinators, the grammar of the parser is written using an internal DSL, in a very consice form, directly in the Scala language. No external files are needed, so the program is less verbose.

3.4 Dataflow model

In a previous introductory project, I had the opportunity to work with internal and external DSL, using Java and Scala parser combinator and the aforementioned libraries.

For this prototype language, the dataflow program will be described using an internal Scala DSL. According to the section 3.3.1, Scala is great to build internal DSL. Its syntax is flexible enough to write a dataflow program (like the code 3.1 on page 13) in a natural and concise way. Furthermore, the Scala compiler helps a lot to check the validity of the user-application at the compilation time. Finally, the user can extend the DSL with ease, which is not possible with an external DSL.

3 Scala Standard Parser Combinator Library - https://github.com/scala/scala-parser-combinators
4 Parsing library for PEGs grammars - http://parboiled.org/

Projects like LabVIEW, NoFlo or Node-RED (presented in section 2.2) have a graphical interface to build applications. In this project, as a proof of concept, dataflow applications will be described textually and not graphically.
In a next version, a web-based editor, similar to Node-RED or NoFlo, could be implemented. Once the programming language is implemented, the development of the graphical interface is straightforward, because the program view can be translated one-to-one to the corresponding code.

3.5 Execution model

The block diagram of the application, stored in a dataflow-graph is transformed to a embedded C++ application. The goal of the project is not to develop a compiler (which could generates an assembly code or an LLVM bytecode for instance), but to generate a ready-to-use C/C++ application.

The C/C++ code is generated specially for embedded systems. A quite simple execution model has been chosen, so the application can be ran on most all embedded systems (also those with limited resources). The skeleton of the generated C++ application is presented below :

```
int main() {
  init(); // Components, target and peripherals initialization

  while(1) {
    // 1) I: Read Input
    // 2) P: Process logic (execute the graph)
    // 3) O: Update outputs
  }
}
```

Listing 3.3 – Skeleton of the generated C++ application

The target system is used as "bare machine". This means that the application runs in a single and monolithic "thread", without any operating system or complex execution frameworks. When the application starts, the `main` function is called, followed by initializations functions and finally the infinite main loop, as presented in the code 3.3. Using the developed HAL library, the application can be compiled for the target using a standard toolchain.

The dataflow graph is transformed to a sequential application and executed in the main loop, according to the Input-Process-Output (IPO) model. It consists in periodically reading the program inputs, then computing the dataflow graph, and finally writing the program output values.

The generated sequential application corresponds to the Grafcet execution in a Programmable Logic Controller (PLC) [GF13]. In this execution model, all inputs of the program are read at the same time, and then all outputs are updated, also at the same time.

The dataflow graph is executed in a periodic or non periodic cycle. A program cycle corresponds to one iteration of the while loop. The C/C++ code of each node/component of the dataflow graph is executed in the main loop, in a particular order, to take into account the dependencies between the nodes.

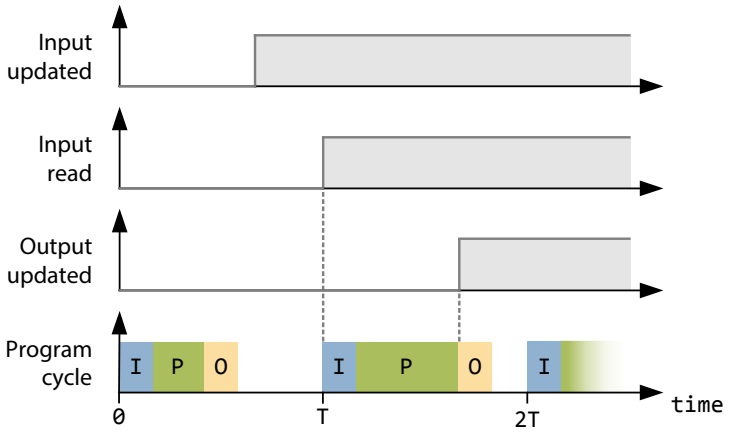


Figure 3.5 – Periodic I/O scanning, IPO model (adapted from [GF13, p. 10])

3.5.1 Synchronous dataflow

To be able to transform any dataflow graph into an efficient sequential program, the dataflow must be restricted to a synchronous dataflow [HCRP91]. A synchronous dataflow is a particular type of dataflow which allows a static scheduling (known at compilation time) [HCRP91]. From a dataflow graph perspective, this means that the graph must be restricted to a Direct Acyclic Graph (DAG). The topological sort of the DAG gives the order in which nodes must be executed.
DAG are a specific type of graphs. Nodes are connected together with directed edges (arcs), such that no *directed cycles* can be created. The following figure shows a DAG :

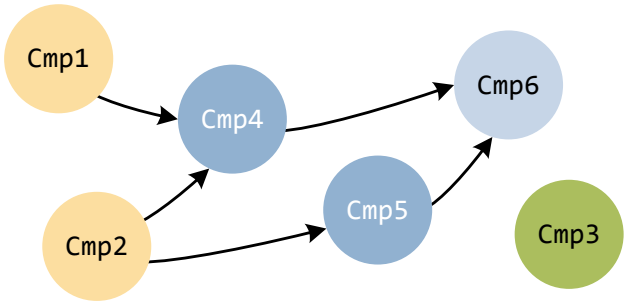


Figure 3.6 – An example of a Directed Acyclic Graph (DAG)

The graph of the figure 3.6 is a valid DAG because no directed cycles are formed by the arcs. The dataflow representation allows isolated nodes (like `Cmp3`). We can also identify source (input) nodes, like `Cmp1` and `Cmp2`, because they have 0 incoming edges (indegree zero). As opposite, the node `Cmp6` is a sink (outdegree zero) connected with 2 incoming edges.

The order of the nodes execution in the `while` loop is given by computing the topological order of the DAG. The graph sorting is done by the frontend part, before generating the application. This allows to run an efficient sequential program on the embedded system, avoiding the overhead of a run-time scheduling [TMPL95], which can be a performance issue for embedded systems.

The topological ordering of a DAG is not unique and must corresponds to the IPO model. It will be explained in details in the chapter 5, in the frontend implementation description.

3.6 Components library

We have developed several components that can be used by the user to build the dataflow applications. The figure 3.7 shows the main components (dataflow blocks), classified by types.

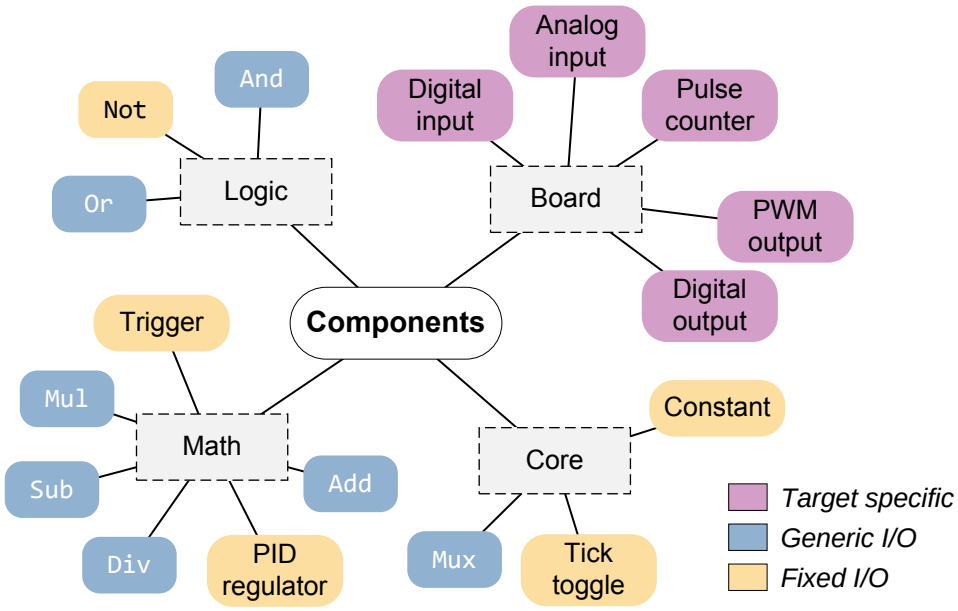


Figure 3.7 – Available components classed by type

A first category of component, called "target specific" allows to control the hardware and the peripherals of the target, to read or write digital/analog values for instance. To do this, specific libraries and low-level code is necessary, for each supported target. The model of these components is described in the Scala frontend, and their concrete C/C++ implementation in the backend. The implementation of these target specific components is based on a generic interface, an Hardware Abstraction Layer, presented in the next chapter.

The other components allows to describe the logic of the program. The majority of the developed components can be used with a generic number of input, like multiplexer, all logic gates and math blocks. These components are implemented in the frontend, using a generic C/C++ implementation.

Currently, not so many components are available, but thanks to the Scala internal DSL, new components can be added with ease, depending on the user needs. In the chapter 6, real-world applications, based on these components will be presented.

Chapter 4

Backend

The first technical part of the project consists in developing a low level C/C++ code for the chosen embedded system (the target). First of all, a development board has been selected for the project. The hardware and its toolchain are presented below.

Then, an Hardware Abstraction Layer (HAL) has been developed. Its specifications and implementation are detailed in the second part of the chapter. The role of this software layer is to provide a generic way to access to the microntroller inputs, outputs and peripherals. All (developed or generated) applications running on the target will use this abstraction layer to control the hardware. Target specific components presented in figure 3.7 on page 19, like digital I/O, PWM output, etc. are implemented in C/C++ for the chosen target.

In the last part of the chapter, a sample C++ application based on the developed HAL presents how the hardware and its peripherals can be accessed and controlled with ease. The HAL provides a high abstraction level and helps to add new compatible targets in the future.

4.1 Target selection

The embedded system programming language developed during this project can be compatible with several hardware, as long as a backend is available (has been developed) for the target.

For this project prototype and as a proof of concept, one hardware as been selected among many possible choices. The goal of the project is not to develop a specific hardware, so a development board available in the market has been chosen.

Two constraints have been considered for the selection. First, it is important to have a low-level access to the hardware and its peripherals, and the possibility to develop on the hardware without operating system (bare machine). This is why Raspberry Pi and alternatives boards have not been selected. Secondly, the target should be sufficiently powerful to not be limited during the developments (PIC, AVR or MSP processors can be too limited).

4.2 Development kit

The selected target for the project is based on a 32-bit ARM Cortex-M3 processor. These processors are very common, they offer rich connectivity, good performance and are available at low-cost. Many manufacturers propose this type of processor, like *Atmel, NXP Semiconductors, Texas Instruments* and *STMicroelectronics*.

Several development kits based on these processors are available at low-cost. As a result, I chose a development kit available on the market I had experience with. It is based on the `STM32F103` Performance Line MCU [STM14a]. This microcontroller combines a powerful ARM Cortex-M3 CPU with an extensive range of peripheral functions and enhanced I/O capabilities :

- `STM32F103RBT6` ARM 32-bit Cortex-M3 CPU Core (128KB flash, 20KB SRAM)
- 72 MHz maximum frequency
- `LQFP64` package (10x10mm), 51 fast I/O ports
- Main peripherals : USB, CAN, I2C/SPI, ADC, UART, timers

The chosen development kit, presented in figure 4.1, is manufactured by *Olimex* [Oli14] and is available for less than 50$. Its features are summarized bellow. Not all theses peripherals will be used during the project.

- One user buttons, a joystick and a status LED
- LCD NOKIA 3310 black/white, 3-axis accelerometer, SD-MMC connector, 2.4 Ghz wireless transceiver, audio in/out, USB mini connector
- Two extension connectors
- Standard JTAG connector for programming and debugging

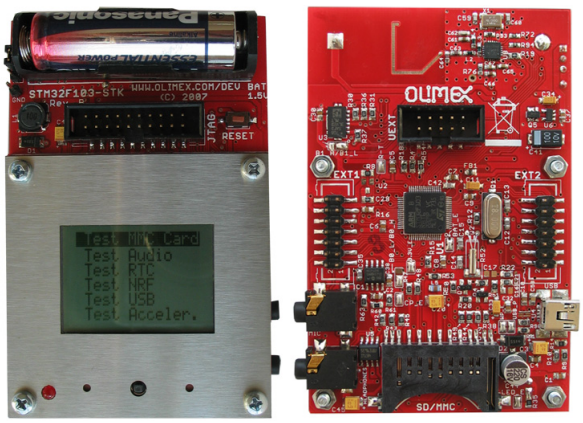


Figure 4.1 – Olimex ARM Cortex-M3 `STM32-103STK` starter-kit board (90x65mm)

One user button, a joystick and a LED are available directly on the board (an Analog to Digital Converter is necessary to get the joystick direction). The kit is flexible and has been designed for many applications purposes. External peripherals can be added using the two 14-pin connectors to

extend the board capabilities. For the needs of the project, and extension board has been developed. It will be presented later in this chapter, in section 4.5 on page 28.

All technical specifications and the development kit schematics are available on the manufacturer website [Oli14]. I/O types and pins numbers are detailed in appendix A on page 78.

4.2.1 Development environment and toolchain

The backend has been developed and tested on the `STM32-103STK` starter kit. The figure 4.2 is an overview of the tools and the hardware used to developed on the ARM Cortex M3 target.

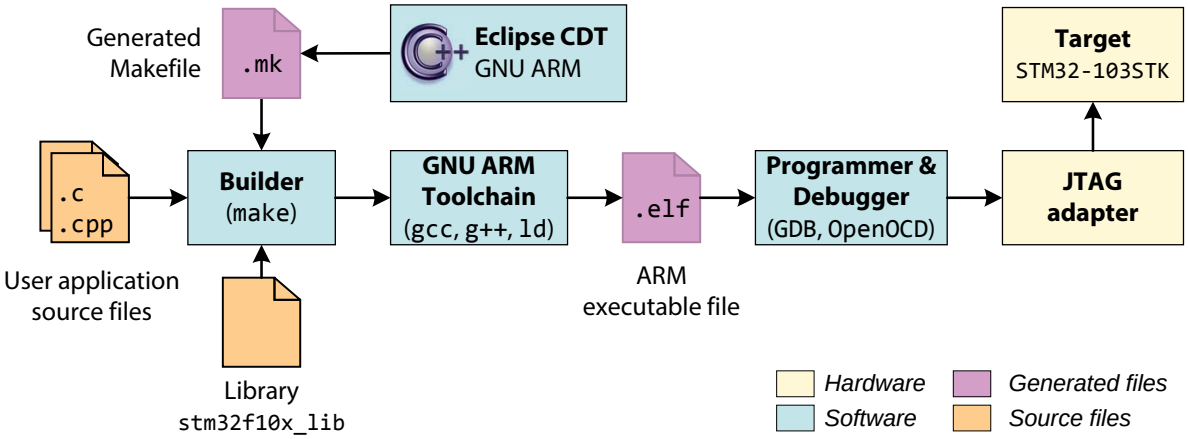


Figure 4.2 – Software tools and hardware overview (adapted from [HES14])

The development environment is free of charge. It is based on the Eclipse CDT (C/C++ support) IDE and the GNU ARM toolchain[1] (`arm-none-eabi`), which compiles all C/C++ source files into an ARM executable file. The project makefile is generated automatically by Eclipse.
When the elf file is generated, GDB and OpenOCD are used to program and debug the code on the target. A standard ARM JTAG 20-pin connector is available on the target. An home-made JTAG is used, but any generic JTAG should be fine (an OpenOCD configuration file is required). Detailed tools versions and hardware references are available in appendix B on page 79.

This environment has been tested on Linux, but it is cross-platform. Eclipse settings and tools setup are described on the Wiki of the school [HES14]. All developments, later described in this chapter, are included in the developed `stm32f10x_lib` C++ library (see figure 4.2). It is shared and used by different projects and demo applications.

The chosen development board [Oli14] is shipped with demo applications. The kit is widely used and many example codes are available. Projects like this one[2] and the `libheivs_stm32` library [HES14] have been used as a starting point. Some files have been used and adapted for the project needs. Finally, the source code documentation has been generated with *Doxygen*.

1 GNU Tools for ARM Embedded Processors - https://launchpad.net/gcc-arm-embedded
2 Demo applications for the `STM32-P103` board - https://github.com/beckus/stm32_p103_demos

4.3 Hardware Abstraction Layer

Many embedded systems are available, with different specifications and peripherals. To hide differences between hardware, an Hardware Abstraction Layer (HAL) has been developed.

The HAL presented in the center of the figure 4.3, is a software layer used by all applications to uniform the control and the access of the hardware (the MCU and its peripherals). This layer allows to run the same user application on different targets, without modifications, because all I/O and peripherals are accessed through this abstraction layer. For each target, a specific HAL must be implemented.

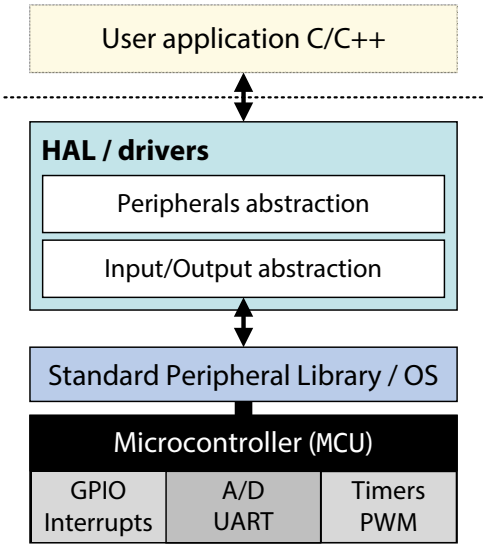


Figure 4.3 – Hardware Abstraction Layer

To support several targets, the HAL unifies the control and the access to the five following inputs, outputs and peripherals :

1. **General Purpose Input/Output and external interrupts (IRQs)**
   General Purpose Input/Output (GPIO) are used to read or write digital values from generic pins of the processor. All GPIO are identified by a unique port and a pin number. They can be configured either as input or output to read or write boolean values. Input values can be polled or inputs can be configured as external interrupts to be notified automatically (using a function callback) when its value change.

2. **Analog input**
   Some specific inputs can be configured as analog inputs. In addition to the GPIO, they are identified by a unique channel number, connected to one or several Analog-to-Digital Converters.

3. **Analog output**
   The reverse operation consist of converting a digital number to an output voltage, using a Digital-to-Analog Converter (DAC). DAC are not often available on low-cost embedded systems. An alternative consist of using Pulse-Width Modulation (PWM) signal to simulate an analog output. The frequency and duty cycle of the output signal can be configured. If necessary, an external analog circuit can be used to filter this signal.

4. **Logging and debugging**
   A serial interface is available to log events and print messages from the microcontroller to an external console (`UART` over RS232). This logging interface is useful for debugging purpose. Events can be sent to trace the code execution.

5. **Timer (delay and time measurement)**
   Finally, delays (busy waiting) are available to wait some time, expressed in milli or microseconds. An hardware timer can also be used to count the elapsed time without blocking the code execution.

4.4 Design and implementation

This section describes the HAL implementation developed for the `STM32-103STK` development kit, presented in section 4.2 on page 22.

The toolchain (see section 4.2.1) supports C++, and the target is powerful enough (no performance issue), so it would be too bad to not use object oriented features of the C++ language. The HAL implementation will be more structured, and OO features makes the code more flexible and highly portable to support other targets, by using `Builder` and `Factory` patterns, interfaces, etc.

The C++ HAL implementation for the `STM32-103STK` board is summarized in the UML class diagram of figure 4.4 (it has been simplified, the full documentation is available in Doxygen) :

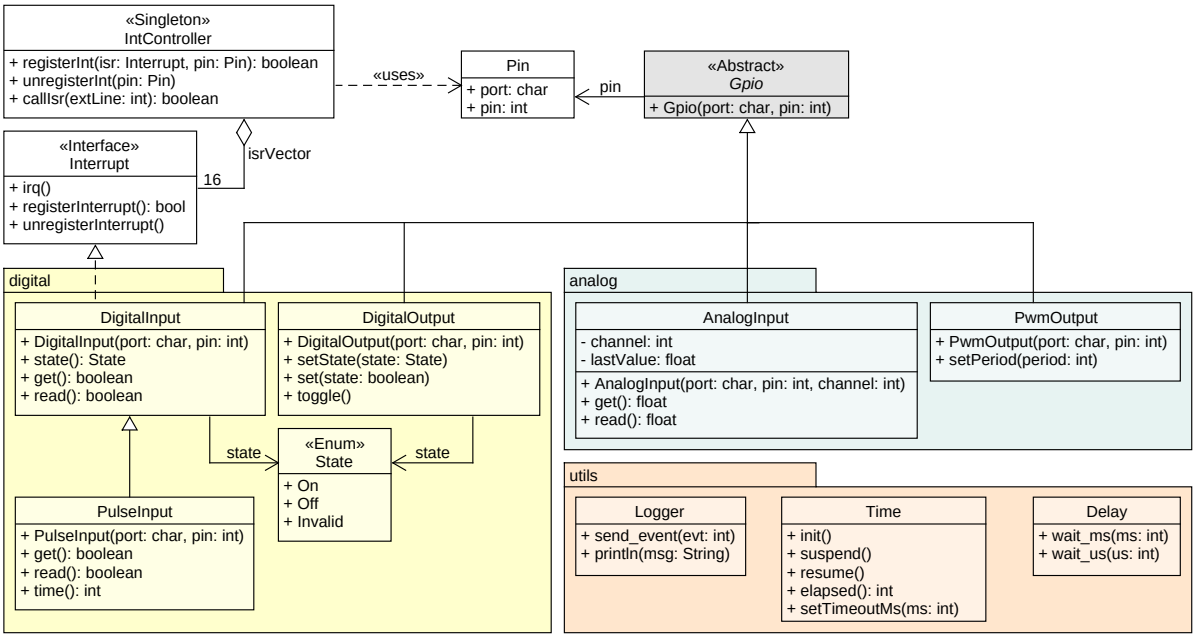


Figure 4.4 – UML class diagram of the HAL implementation

The HAL contains the low level implementation of all target specific components available in the framework (see figure 3.7 on page 19), like digital inputs, PWM outputs, A/D converter, etc. Their implementation is detailed separately later in this section.

Inputs and outputs are grouped in two main packages : `digital` and `analog`. Each I/O is identified by a port and pin number, which correspond to a physical pin of the MCU. The abstract `Gpio` class is the base class used by all I/O. The `initialize` method is implemented by each subclass to configure the pin automatically, behind the scene, as required. All pins can be used as general purpose. Some specific pins can also be configured as alternate function, to be used as analog inputs, timer outputs, external interrupts, etc.

The HAL uses the `StdPeriph` library to access to the MCU peripherals. The *STM32F10x Standard Peripherals Library* [STM14c] is a vendor specific hardware abstraction layer for Cortex-M3 processors. The library is provided by *STMicroelectronics* and contains low level C functions to configure and access all peripherals of the `STM32F10x` Cortex-M3 MCU.

All information about the processor implementation and its peripherals are available in the reference manual [STM14b]. This document contains more than 1000 pages. The main hardware specificities and implementation details used to develop the `STM32` HAL are summarized in the next sections.

4.4.1 General Purpose Input/Output

Each physical pins of the microcontroller are grouped by ports and identified by a unique number. Up to 51 GPIO are available. Each of these pins can be configured for a general purpose, as input or output.

The `DigitalOutput` class configure any pin, specified by a port letter (from `A` to `G`) and a pin number (from 1 to 15) as a digital output. The clock of the corresponding port is enabled and the pin mode configured as an output in push-pull mode. Once initialized, is it set automatically to off. Operator overloading are available to `set` a boolean value to the output (positive logic).

Any pin can also be configured as a digital input using the `DigitalInput` class. Inputs pins are configured as floating inputs (without any internal pull-up or pull-down resistor). To read the current input value, the `read` function must be used. This allows to read an input pin value by polling.

External interrupts

It is also possible to configure the pin as an external interrupt. The `IntController` class manages all external interrupt lines (up to 16). When an input is configured as an external interrupt, an interrupt is triggered on the rising and falling edges and the input value is automatically saved. The `get` function will return the input pin value, automatically saved in the ISR routine. This is faster than polling the input pin.

To use an input as external interrupt, some hardware limitations [STM14b, p. 208] must be taken into account. First, only 16 external interrupts lines are available and secondly, they are internally connected in the specific manner, presented in figure 4.5.

GPIO are multiplexed and grouped by pin number to use all the 16 available lines [STM14b]. This limitation must be taken into account when external interrupts pins are chosen. For instance, it is not possible to differentiate the `PA0` and the `PB0` input because there are connected to the same `EXTI0` interrupt line (pin numbers must be different). In the current implementation, digital inputs are initialized as external interrupts by default. This is more efficient than polling the input value. The `registerInt` method of the interrupt controller is called automatically when the input is initialized.

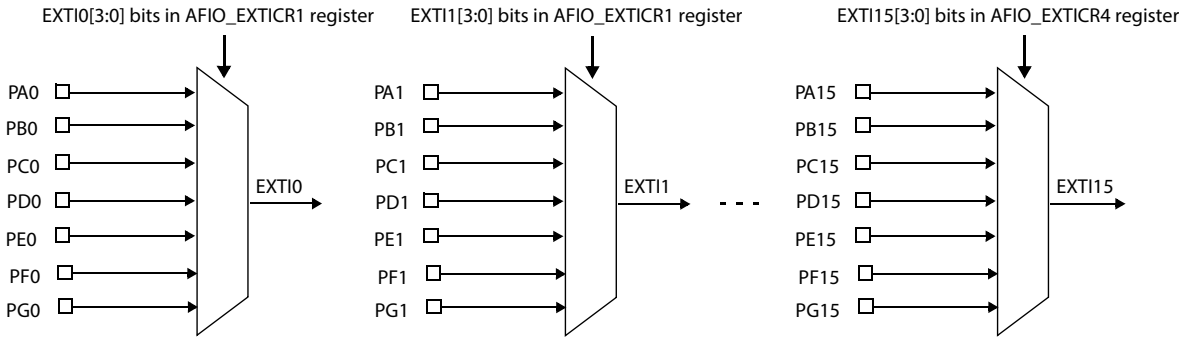


Figure 4.5 – External interrupt/event GPIO mapping (adapted from [STM14b])

4.4.2 Analog-to-Digital Converter

Two 12-bit A/D converters are available to measure 16 external analog sources (not all GPIO can be used). The `ADC1` converter is used to measure analog inputs using a single conversion mode. An `AnalogInput` is a specific pin identified by a unique port, pin and channel number (from 0 to 15). The `read` function start and A/D conversion and returns the result when it is completed.

4.4.3 Pulse-width modulation

PWM outputs are generated using 16-bit hardware timers. The frequency and the duty cycle of these outputs can be configured. Only specific pins can be used as PWM outputs. They must be configured as alternate function to be internally connected to timer outputs, using the `PwmOutput` class.
The pin must be configured as a push-pull output. Then, depending on the pin number, the corresponding timer is automatically configured in PWM mode. For now, only the 4 channels of the timer 4 can be used as PWM outputs.
The PWM duty cycle of each output can be configured using the `setPeriod` function. A 100% duty cycle corresponds to a period of 0xFFF (12-bit resolution). This implies a maximum PWM frequency of about 9 kHz. No Ditigal-to-Analog Converter (DAC) are available in the chosen MCU, but PWM outputs can "simulate" it.

4.4.4 Universal asynchronous transmitter

The `Logger` class can be used to transmit (Tx) messages from the MCU to an external serial console. This is useful for debugging and to trace the code execution. The receiver module (Rx) is currently not available and not used. The `USART2` serial output is configured as follow :

- 115200 baud, 8-bit data and 1 stop bit, no parity bit / no hardware flow control

A custom tracer is used to connect the serial output of the development kit to a PC USB port.

4.4.5 Delays and timers

The developed HAL provides utilities functions to deal with the time :

1. Delays functions are available to wait from 1 µs to a few seconds, using busy-waiting. `wait_us` and `wait_ms` are both blocking functions.

2. To count the elapsed time (without blocking the code execution), the timer 2 is used as an hardware `SysTick` timer. Each 1 ms, a shared counter variable is incremented to count the elapsed time in milliseconds (using the`time_get` function).

Time and delays implementation has been adapted from the `libheivs_stm32` library [HES14] to work with the `STM32F103` processor running at 72 MHz.

4.4.6 Available inputs and outputs

The table 4.1 summarizes all inputs, outputs and peripherals of the `STM32F103-STK` development kit that can be accessed using the developed HAL. They are classed by types.

I/O type	Available	Limitation
General-purpose input	51	-
General-purpose output	51	-
External input interrupt	16	`EXTI` lines only. Pin numbers must be different
Analog input (12-bit ADC)	16	All analog channels
PWM output	4	`Timer4` output channels only
UART (Tx, debug)	1	`USART2` Tx only

Table 4.1 – Accessible I/O using the Hardware Abstraction Layer

4.5 Extension board

An extension board has been designed to add inputs and outputs to the `STM32-103STK` kit (presented in section 4.2). With only one button and one LED available on the main kit, possibilities were too limited to test real-word and complex applications.

The prototype extension board, presented on the right in figure 4.6, has been designed on a stripboard, a rapid prototyping technique.

A 4-pin connector (`EXT`) can be used to connect external peripherals with ease. LEDs or buttons can be disconnected using jumpers. To develop this board, constraints described in the previous section have been taken into account (see table 4.1). For instance, buttons are connected to external interrupts lines on different pin numbers.

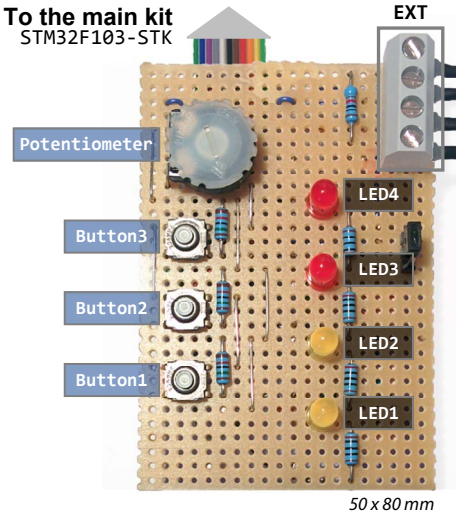


Figure 4.6 – The extension board

The extension board schematic with pin numbers and I/O types is available in appendix A on page 78. In a next version, the extension board can be miniaturized using *SMD* components to be connected directly on the bottom of the main kit.

The following I/O are available on the developed extension board to extend the main kit :

I/O type	Description
Digital input	3 push buttons, connected on interrupts lines (`EXTI`)
Analog input	1 potentiometer connected to an analog input
Digital output	4 LEDs (2 yellow, 2 red)
PWM output	The 2 red LEDs can also be controlled as PWM outputs

Table 4.2 – Available inputs and outputs on the developed extension board

4.6 Testing

The developed HAL, packed into the `stm32f10x_lib` library, allows to access with ease and a high abstraction level to the main peripherals of the MCU. The sample C++ application 4.1 demonstrates how the HAL can be used to control inputs and outputs pins of the extension board and components presented in the previous sections. For space reasons, some lines have been removed. The full application code is available in appendix C on page 82.

The demo application 4.1 uses several peripherals of the extension board. When the application starts, a debug message is printed and the `led1` is switched on. The button 2 controls the `led2` and the potentiometer value set the intensity of the `led3` using a PWM output. Finally, using the time utilities functions, the `led4` blinks every 0.5 seconds. Thanks to the C++ implementation of the HAL, methods like `set` or `get` can also be replaced by equals operators to make applications less verbose and more user friendly (see line 20 or 24 of the code 4.1 instance).

The toolchain presented in section 4.2.1 on page 23 is used to compile the application. A sample Eclipse project is available in the `stm32` Git repository to compile and program the application on the real target. The extension board must be connected to the main kit. The figure 4.7 shows an oscilloscope screen capture of two PWM signals, generated by two PWM outputs (≈ 9 kHz, duty cycle of 25% and 50%).

The developed HAL uniforms the access and the control to all inputs and outputs, so they are all created and initialized using the same approach (see lines 1 to 4 of code 4.1). Methods with an high abstraction level are available to use all target specific components presented in the figure 3.7 on page 19. If the constraints of the table 4.1 on page 28 are satisfied, any component can be automatically initialized on the desired pin. This was a quite big implementation challenge, because clocks, peripherals and all registers must be initialized correctly in a generic manner, according to the port and the pin number of the components.

All C++ applications running on the `STM32P103-STK` board will access to the peripherals and I/O through the HAL. This makes user applications highly portable and new targets, like ARM Cortex development kits, can be added without much effort. Thanks to the HAL, the component model implementation in the frontend part is much simplified. It is presented in the next chapter.

```
AnalogInput adc1('B', 0, 8); // Potentiometer
PwmOutput pwm3('B', 8);      // led3
DigitalInput btn2('C', 1);
DigitalOutput led4('B', 9), led2('C', 4), led1('C', 3);

void initIO() {
    btn2.initialize();
    // ...
}

int main() {
    time_init();
    initIO(); // Init all I/O and set to '0'

    println("HAL sample application"); // Print to USART2
    led1 = true; // Led1 is on

    timeout_t time = time_get();
    while (1) {
        led2 = btn2.get();  // Led2 is controlled by btn2 (use EXT interrupt line)
        pwm3 = adc1.read(); // Start an A/D conversion to read the potentiometer value

        if(time_diff_ms(time_get(), time) > 500) {
            time = time_get();
            led4.toggle();  // Led4 toggles every 1/2 seconds
        }
    }
    return 0;
}
```

Listing 4.1 – Part of the HAL demonstration application

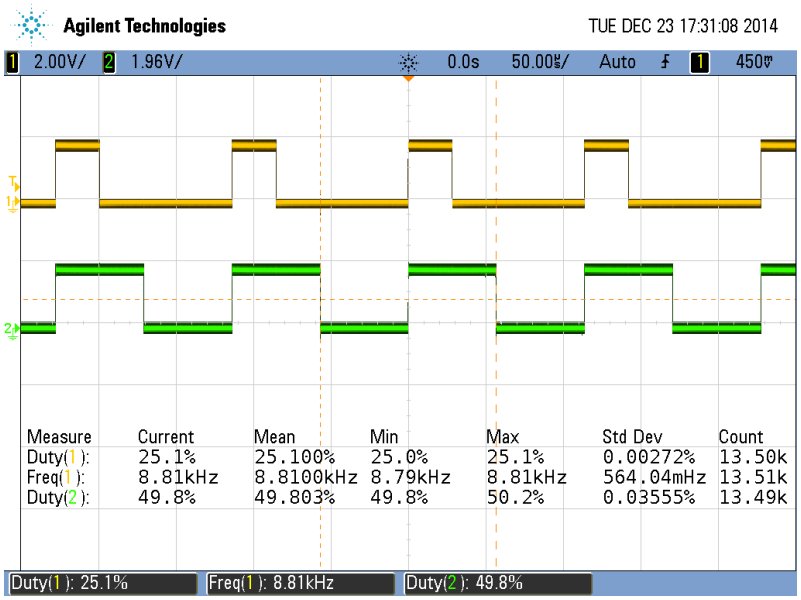


Figure 4.7 – Oscilloscope capture of two generated PWM signals

Chapter 5

Frontend

5.1 Introduction

The second technical part of the project, the frontend, is implemented in Scala. The user writes its application using severals components available in the framework (presented in section 3.6 on page 3.6). The concept of the internal dataflow DSL and the implementation of some components will be introduced.

This chapter is composed in two main parts. The first part presents the compilation pipeline. Severals steps are necessary to convert the dataflow application to a C++ code for the embedded system. These steps and transformations will be presented separately in details.

Then, the second part is focused on the code generation and simulation. Two approaches to test the language and the generated application will be introduced. The first one consist in programming the target and testing the application manually. The other is a fully automated approach, not widely used for embedded systems. It allows to simulate and test the generated application without the real hardware, using an existing ARM emulator.

Based on simple examples and basic applications, concepts of components, ports and dataflow graph are presented here. The implementation of target-specific components (see figure 3.7 on page 19) is based on the developed HAL library, presented in the last chapter.

At the end of the chapter, some advanced features to build complex applications, in a concise and natural way are introduced. These features will be used to build real-world applications.

5.1.1 Development environment

The frontend part of the project is developed in Scala. I chose the free and open-source IDE *IntelliJ* (with the Scala plugin) to develop this part of the project. Moreover, I use *sbt* [Typ14b] as build tool. The directory structure of the Scala project recommended for sbt (the same as Maven projects)[1] is used. Unit tests and Scala source files are well separated.

1 http://www.scala-sbt.org/0.13/tutorial/Directories.html

To run the project, Scala and sbt must be installed. The sbt project definition file and tools versions are listed in appendix B.2 on page 80, with useful sbt commands to compile, test and run the project. Unit tests are based on the ScalaTest[2] library. Some of these tests will be presented later in this chapter. All used Scala libraries are listed in the sbt project file available in appendix B.2. Finally, *ScalaDoc* is used to document the code. The documentation can be generated using sbt. The sbt build definition makes the project highly portable. It can be imported in severals other IDE, like Eclipse.

5.1.2 Pipeline

From the input, the user program written in Scala, to the compiled ARM elf output file, many stages, operations and transformations are required. These stages, grouped in a pipeline, will be presented later in this chapter. First, the concept of pipeline will be introduced with a simple code example.

In general, pipelines are used in compilers, to transform the input language *I* to another output language *O*. A pipeline consists of a chain of blocks, and each of them applies a specific transformation. These blocks are useful to decompose a problem in smaller and simpler parts. That is the reason why a pipeline is used and has been developed for this project.

To demonstrates how the pipeline works, a simple example is presented. Three blocks are composing the pipeline, like described in the figure 5.1 on the right. The first block (on the top) is a source. It produces a random integer number. It is the input of the second block. The block 2 applies a transformation on this number and throws an error if the output integer is even. Finally, the last block prints the final number as a string, if no error occurred (`Int` to `String` transformation).

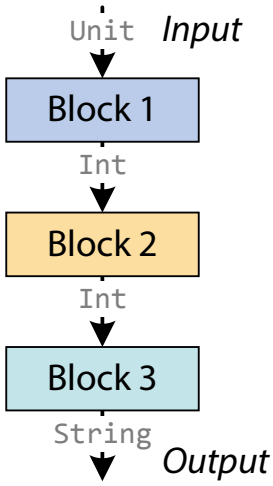


Figure 5.1 – Pipeline blocks

The Scala implementation of the pipeline application of figure 5.1 is the following :

```
1  case class Block1() extends Pipeline[Unit, Int] {                    // Pipeline block 1
2    override def run(ctx: Context)(i: Unit) = {
3      val nbr = new Random().nextInt(100)
4      ctx.log.info(s"Random number is $nbr"); nbr
5    }
6  }
7
8  case class Block2() extends Pipeline[Int, Int] {                      // Pipeline block 2
9    override def run(ctx: Context)(i: Int) = i % 2 == 0 match {
10     case true => 2 * i
11     case _ => ctx.log.fatal(s"$i is not an even number !")
12   }
13 }
14
15 case class Block3() extends Pipeline[Int, String] {                   // Pipeline block 3
16   override def run(ctx: Context)(i: Int) = s"The result is $i."
17 }
18
19
```

2 Scala and Java unit test framework - http://www.scalatest.org/

```
20 test("Chain of 3 blocks") {
21   val pipe = Block1() -> Block2() -> Block3()  // Build the pipeline
22   val ctx = new Context("Demo")
23   val res = pipe.run(ctx)(Unit)               // Run the pipeline
24   ctx.log.info(res)                           // and print the result using the logger
25 }
```

Listing 5.1 – Pipeline application of figure 5.1

Pipeline blocks are defined with an input and output type, using the `Pipeline` template class. If no input or output are required, the type will be `Unit` (see `Block1`). To create a pipeline, blocks can be chained using the custom "`->`" operator (on line 20). The `run` method of each block must be overridden to implement the block code. A `Context` must be provided to run a pipeline. It contains global settings, available for all blocks. Finally, the context also contains a logger, used to report information or errors.

As an example, on line 11, an error is reported if the number is odd. This will stop the pipeline execution. The error will be printed and the last block number 3 will not be executed at all. If no error occurs, the pipeline result is printed using the `Logger` class. It allows to trace messages, informations or errors. It is based on the `Slf4J` logging library.

The custom "`->`" operator used on line 20 is similar to the `andThen` Scala method, used to compose functions. It allows to build pipelines with ease. The pipeline implementation has been adapted from the Scala *Leon* [3] project. A more exhaustive pipeline test has been developed in the "`test/utils/PipelineTest.scala`" test file. It is available in the Git repository of the project.

5.2 Code generation pipeline

The concept of pipeline has been introduced in the previous section. The first concrete pipeline developed for the project is presented here.

The code generation pipeline consists in transforming the user dataflow application, written in Scala, to a generated C++ code for the target embedded system. To do this, several transformations are necessary. They will be presented separately in the next sections. These transformation blocks are chained together to create the code generation pipeline, shown in figure 5.2.

3 The Leon synthesis and verification system - http://lara.epfl.ch/w/leon and https://github.com/epfl-lara/leon

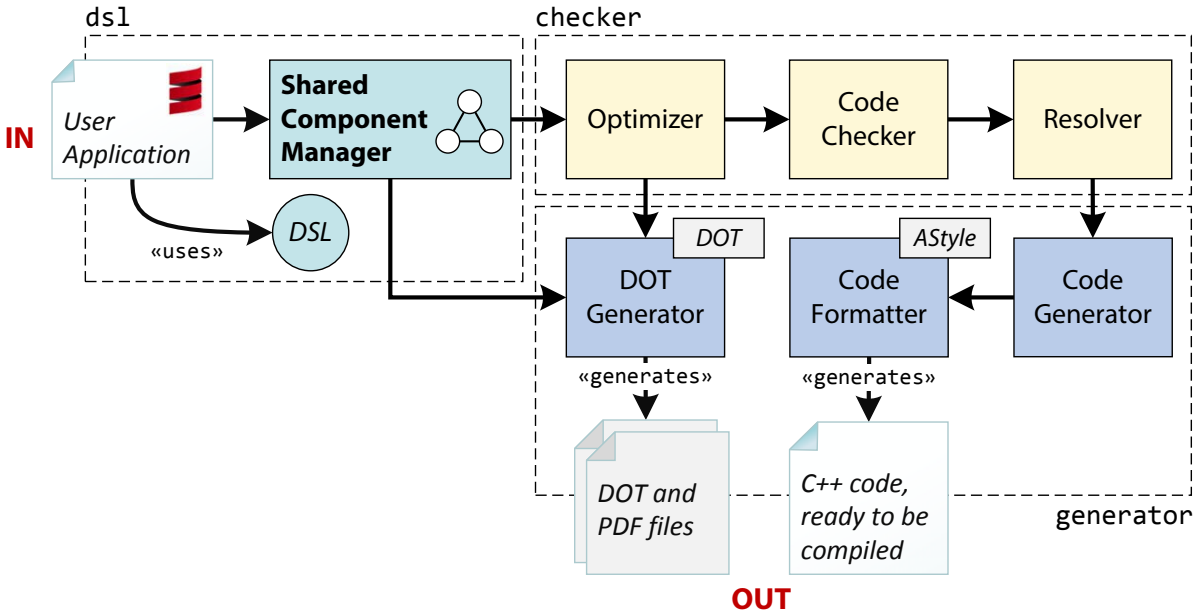


Figure 5.2 – Overview of the code generation pipeline

To generate a C++ code from the user application, the following main operations are performed :

1. The user application is written with the help of the developed dataflow internal DSL. Severals components have been implemented (see section 3.7 on page 19) and are ready-to-use. They can be used by the user to build the application.
2. The dataflow application is compiled using the Scala compiler and its dataflow graph is saved in memory. All transformations are applied on this graph.
3. Severals optimizations and checks are applied on the graph. Then, it is sorted to be transformed to a sequential application. The last step consists in generating the output application by adding the code of each component. The generated C++ application file can be compiled and ran on the target embedded system.
4. Each pipeline block generates several debug information during the code generation process. The visual representation of dataflow graph is also generated automatically by the `DotGenerator` block.

Each block of the pipeline are detailed separately in the next sections, with the help of code samples and simple application examples. Complete applications are presented in a next chapter.
For each application, the generated files are available in the `output` folder. Finally, the pipeline blocks can be configured using the shared `Setting` class.

5.2.1 Component manager

The user application is written using an internal domain-specific language, this means that it is compiled by the Scala compiler. It checks the validity and the correctnesses of the application.

Once the user application is successfully compiled, the component manager is used to create the dataflow graph of the application. Using this first pipeline block, graph nodes and edges can be

added. This block contains the Directed Acyclic Graph (DAG) representation of the application and implements all methods to query the graph, to find specific type of nodes, to connect components ports, etc.

A graph library

To store the application DAG in memory, I have choose to use a generic and open source Scala library, called *Graph for Scala* [Emp15b], because this particular data structure is not available within the Scala programming language.
The Graph for Scala library provides basic graph functionalities. It is part of the EPFL Scala incubator space since 2011. It seamlessly fits into the Scala standard collections library. Like members of `scala.collection`, graph instances are in-memory containers that expose a rich, user-friendly interface.

Using this existing library speeds up the development of the project. The main reasons [Emp15a] are discussed here. First, different type of graph can be created, manipulated, queried and customized intuitively, like other regular Scala collections, in a concise and functional style. Mutable and immutable graph collection implementations are available. The graph consistency is maintained by the library. For instance, it prevents nodes or edges duplicates. Finally, the Scala-graph library is composed of several modules, which help to add constraint or to export the graph for instance. Some of these features will be presented later in this chapter. The library is well documented and a user guide is available for each module on the official website [Emp15a].

Graph structure

The dataflow DAG is saved in memory using the Scala graph library. Graph nodes represent components of the program, presented in section 3.6 on page 19. The low-level implementation of each target-specific component is implemented in the backend. In the frontend, components are defined by a name, a unique identifier, and a list of input and output ports (inbound and outbound ports) :

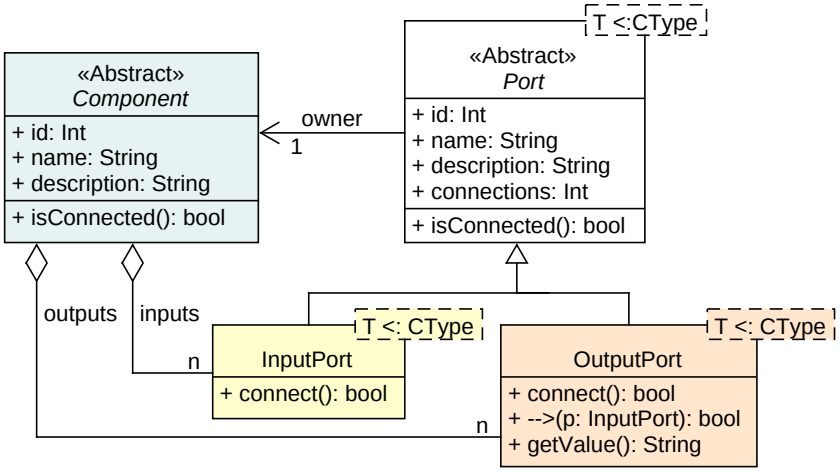


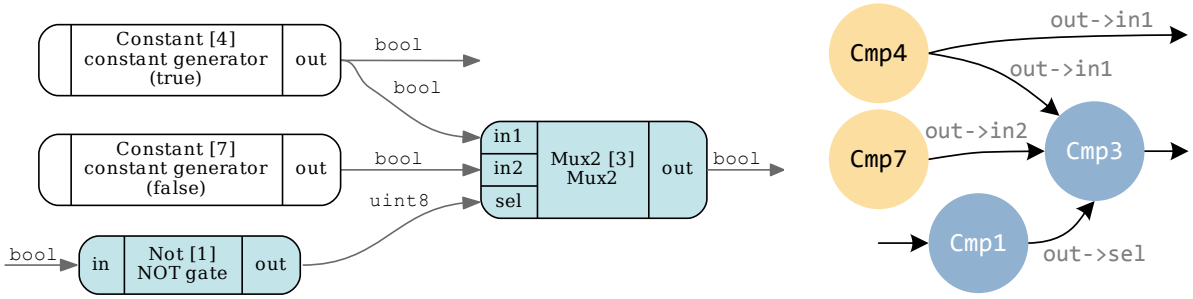
Figure 5.3 – UML class diagram of components and ports

The following graph structure allows to save the DAG in memory in an efficient way. The internal graph representation is presented below :

```
1 import scalax.collection.constrained.mutable.Graph
2 // ...
3
4 implicit val config: Config = Acyclic // DAG constraint (unconnected nodes are allowed)
5
6 // Mutable graph representation (DAG) with components as nodes and dirceted edges with a label
7 val cpGraph = Graph.empty[Component, LDiEdge]
```

Listing 5.2 – Mutable graph strucutre declaration

In the graph, components (nodes) are connected together through their ports. A directed edge connects and output port to an input port. To create this connection in the graph, a directed edge (arc) with a label is necessary (`LDiEdge`), because a component can have multiple input and output ports. The label attached to the graph arc corresponds to a `Wire` (basically a tuple), which identify the output and input ports (source and destination) of the connection. The figure 5.4b shows the internal graph representation for the application 3.2 presented on page 12.



(a) Generated DAG

(b) In-memory graph structure

Figure 5.4 – Two corresponding DAG representations

The figure 5.4a has been generated automatically using a pipeline block. It will be presented in the next section. Two main steps are necessary to populate the DAG :

1. First, all components of the user application are added as isolated nodes in the graph. They are all identified by a unique ID, required by the library to store them.
2. Then, wires / connections between component's ports are added. The "`-->`" operator in the Scala DSL corresponds to a unique arc in the graph. Each port of a component is also identified by a unique ID.

To meet the sequential execution model of the generated application (see section 3.5 on page 17), the graph must be a acyclic. This means that arcs can be added to the graph only if no cycle are created. The `constrained` module of the scala graph library has been used to constraint the graph to be `Acyclic` (see code 5.2). This is an automatic way to ensure that the graph will always be acyclic : the custom `CycleException` is thrown is cycles are created.

5.2.2 Components ports

Dataflow components exchange data through their ports. A port is characterized by a direction (input or output) and by a data type which "flows" through the components ports (see figure 5.3 on page 35). The `getValue` method returns a specific C code to read the output value of a port. The returned code can be a constant value or the name of a local variable which contains the port value.
A component can have multiple I/O ports. In the DSL, components ports are accessible by name. By convention, port names of generic components (like logic gates) are named `in` or `in1`, `in2`, etc. if more than one input exist. Output ports are named `out` or `out1`, `out2`, etc. The `IOtraits.scala` file implements predefined traits for components from 1 to 4 input or output.

The Scala implementation of components ports is shown in the following code :

```
abstract class Port[+T <: CType : TypeTag](owner: Component) {
  val id = owner.nextPortId // Unique ID of the component
  val tpe = typeOf[T]       // Data type of the port
  var connections = 0       // Number of connections (only 1 for inputs)

  def connect(): Unit       // Override by input and output ports
}

abstract class InputPort[+T <: CType : TypeTag](owner: Component) extends Port[T](owner) {

  final override def connect(): Unit = (connections > 0) match {
    case true => throw PortInputShortCircuit(this)      // Input already connected
    case _ => connections = 1
  }
}

abstract class OutputPort[+T <: CType : TypeTag](owner: Component) extends Port[T](owner) {

  final override def connect(): Unit = connections += 1 // Multiple connections allowed

  def -->[A <: CType : TypeTag](that: InputPort[A]): Boolean = {
    checkType(that) // PortTypeMismatch exception can be thrown

    that.connect()  // PortInputShortCircuit exception can be thrown
    this.connect()

    ComponentManager.addWire(this, that) // Add the directed edge in the graph
  }

  def getValue: String // C code to read the output value
}
```

Listing 5.3 – Components ports implementation

Ports are template classes. Predefined data types can be "transported" with ports, like integer (`uint8,16,32` and `int8,16,32`) and floating point values (`float` and `double`). These types are all defined as subclasses of `CType`, so it is used as a upper type bound. In this case, templates have been used to build generic components. For instance, the `Constant` component has only one implementation an can be used with any type of data. Moreover, to help developing components

with a generic number of I/O (of different types), like logic gates or multiplexers, ports are covariant[4] in its parameter `T`. This allows to store ports with different types in a `Seq[Port[CType]]` (see the `GenericCmp.scala` file as an example).

The "`-->`" method in the Scala DSL code is used to connect an output port to an input port. According to the code 5.3, severals checks must be performed before adding an arc in the dataflow graph (before connecting two ports) :

1. First of all, the "`-->`" method is only available for output ports, so outputs can only be connected to inputs (see line 19). The user cannot do it wrong.
2. To connect two ports together, they must have the same type `T`. This test is performed using reflection and TypeTags. If types are not valid, the custom `PortTypeMismatch` exception is thrown at runtime :

```
Constant(uint8(128)).out --> DigitalOutput(Pin('A', 1)).in // uint8 to bool not allowed

> [ERROR] Ports types mismatch. Connection error !
> Cannot connect the output ‘out‘ (type ‘uint8‘) of Cmp[1] 'Constant'
> to the input ‘in‘ (type ‘bool‘) of Cmp[3] 'DigitalOutput'.
```

3. Data between nodes are not event based, this implies that an input can be connected only once because only one data can be on each arc. Each time a port is connected, the `connections` counter is incremented. If an input is connected more than once, the `PortInputShortCircuit` exception is thrown at runtime (see line 10 in listing 5.3) :

```
val cst = Constant(bool(true))
val led = DigitalOutput(Pin('A', 1))
cst.out --> led.in // Connection valid
cst.out --> led.in // Short circuit here

> [ERROR] Short circuit !
> The input 'in' of Cmp[2] 'DigitalOutput' is already connected.
```

   There are no restrictions for output ports. They can be connected more than once with several inputs.

4. Finally, if these conditions are satisfied and if no cycle are created, the edge between the two components (with its corresponding label) can be added safely into the graph (line 25).

Custom exceptions presented above are implemented in the `Exceptions.scala` file. More specific exceptions will be presented later in this chapter, in section 5.4.

4 http://docs.scala-lang.org/glossary/#covariant

5.2.3 Dot generator

The user application is stored in memory in a complex graph structure. The `DotGenerator` block allows to represent the dataflow graph of the application in an elegant and graphical view. All graph diagrams available in this report have been generated automatically using the developed `DotGenerator` pipeline block. This block does not contribute directly to the code generation, but it helps to visualize the user application, to display informations or warnings and finally it is useful for documentation purposes.

The chosen Scala graph library (see section 5.2.1 on page 35) contains two additional modules, which help to export the in-memory graph to different output types :

- The `Json` module can be used to import/export graph instances from/to Json text. It is useful to save a graph to the disk for instance, but not really to produce its visual representation.
- The second module of the Scala graph library has been developed specially to visualizing graphs. The `DOT` module contains helper functions to convert each nodes and edges to a structured dot file (Graphviz). This solution has been choose and will be presented below.

Graphviz [Gra14] is an open source and cross-platform graph visualization software that can be used to draw diagrams, graphs, networks, interfaces, etc. It has been designed to draw many different diagrams types, shapes, and output drawings are highly customizable. Some examples are available in the official gallery [Gra14]. An interesting point of Graphviz is that diagram elements (nodes, edges, labels) are positioned automatically on the output figure and the orientation can be set (left to right, top-down, etc.). This is great because any graph can be converted to a nice vector image or pdf file (no worry about objects placement).

Many shapes types are available within Graphviz. It was not easy to find a good way to represent any DAG to a nice formated figure, with an automatic placement. Different layout engine, diagrams and shapes types have been tested. After several attempts, the following result has been adopted :

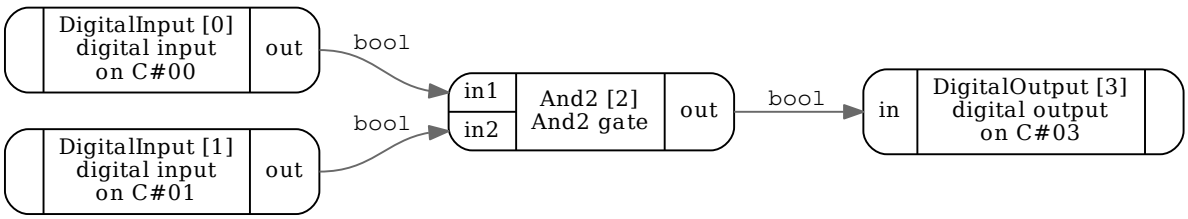


Figure 5.5 – A simple dot graph representation generated automatically

The figure 5.5 is generated using the dot layout engine and represent the dataflow graph of a simple application, composed by 4 components (2 digital inputs/buttons, an `And` logic gate with two inputs, and a digital output/LED). A `digraph` diagram is used to represents this directed graph. Nodes are placed automatically, using a left to right placement. Each components (graph nodes) are represented using `Mrecord` shapes. I chose this type of shapes to clearly display input and output ports of each

component, with connections (wires). Edges are drawn between these ports to represent an accurate view of the DAG stored in memory. The type of the value transported through a wire is also displayed on the graph. The following dot file corresponds to the figure 5.5 :

```
// File generated automatically. Visualisation of the 'DotSch' program.
digraph G {
    // Diagram settings
    graph [rankdir=LR labelloc=b, fontname=Arial, fontsize=14];
    node  [fontname=serif, fontsize=11, shape=Mrecord];
    edge  [fontname=Courier, color=dimgrey fontsize=12];

    // Exported nodes from the components graph (with I/O ports)
    cmp000 [label = "{{}|DigitalInput [0]\ndigital input\non C#00|{<p00> out}}"]
    cmp001 [label = "{{}|DigitalInput [1]\ndigital input\non C#01|{<p00> out}}"]
    cmp002 [label = "{{<p00> in1|<p01> in2}|And2 [2]\nAnd2 gate|{<p02> out}}"]
    cmp003 [label = "{{<p00> in}|DigitalOutput [3]\ndigital output\non C#03|{}}"]

    cmp000:p00 -> cmp002:p00 [label = bool]    // Wires between component's ports
    cmp001:p00 -> cmp002:p01 [label = bool]
    cmp002:p02 -> cmp003:p00 [label = bool]

    label = "\n\nVisualisation of the 'DotSch' program.\nDotSch.dot" // Figure label
}
```

Listing 5.4 – Generated dot file. It corresponds to the figure 5.5

This file contains general settings (orientation, shapes, font, etc.), the list of nodes and the list of edges (wires) between components ports. The `DotGenerator` block generates this dot text file. It can also be automatically converted to a vector pdf or image, using the dot engine. To do it, an external process is called withing the Scala code, so the graph representation (see figure 5.5) is directly available in the output folder as a pdf file (Graphviz must be installed and available in the path, see appendix B.2).

The dot module helps a lot to generate a valid dot file. Unfortunately, in a previous version of the scala-graph library, record-based shapes where not supported. I had the chance to contribute[5] to this open-source library and now the last dot module version officially supports these types of shapes.

As an alternative, the *PGF* [Tan14] graphic TEX package has also been tested to generate graph diagrams, but this library does not handle automatic object placement, that is why it is not used. Graphviz is well adapted for the project needs and is also used by other projects, for instance to visualize *NoFlo* graphs[6].

5.2.4 Code optimizer

The code optimizer block is the first operation performed on the component graph. To keep a clean output code, the `CodeOptimizer` block is able to remove unconnected/isolated nodes and also unconnected components path (several unused nodes connected together). The dot diagram shown in figure 5.6 presents an unoptimized graph :

5 Bad export format for record shapes - https://github.com/scala-graph/scala-graph/issues/23
Scala-graph 1.10 release notes - http://www.scala-graph.org/download/#1.10
6 NoFlo visualization tools for Graphviz - https://github.com/noflo/noflo-graphviz

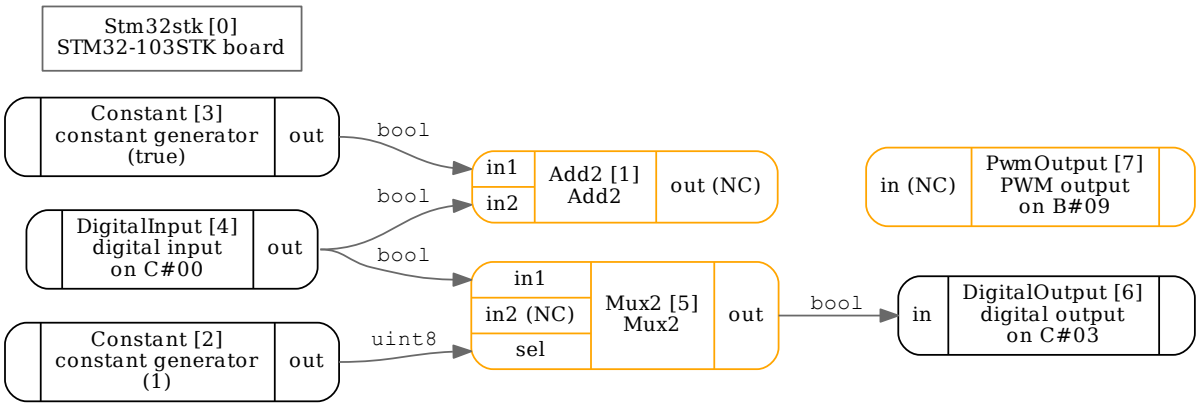


Figure 5.6 – An unoptimized graph

This diagram is generated automatically using the dot generator block (see section 5.2.3) from an unoptimized graph. In the figure 5.6, isolated and partially unconnected nodes are displayed in orange to indicates warnings (more details in the next section 5.2.5). To optimize the graph, some of them will be removed, but only if one of these two conditions is satisfied :

- If a nodes is isolated (all input and outputs are disconnected), then it can be simply removed because it is not connected at all. The only exception is for nodes without input or output. These nodes cannot be removed because they are used to modify the generated output code. With zero I/O, they are not considered as unconnected.
- Secondly, a node can be removed if all its input or output are disconnected (both conditions are satisfied for isolated nodes).

According to this second condition, several passes can be necessary to optimize a graph. Once a component is removed, the graph is updated and its state change. All components must be evaluate again to test if new ones can be removed. To optimize the graph of the figure 5.6, 2 passes are necessary. Optimization steps for the figure 5.6 are available below. Output informations of the optimizer block are printed to the console using the logger.

```
Optimizer started for 'DemoOptimizer'.
Pass [001]
  > Remove Cmp[1] 'Add2': all outputs are unconnected.
  > Remove Cmp[7] 'PwmOutput' (B.09): all inputs are unconnected.
  > 2 components removed in pass 1.
Pass [002]
  > Remove Cmp[3] 'Constant': all outputs are unconnected.
  > 1 component removed in pass 2.
Optimizer ended successfully after 2 passes. 3 components removed.
The final graph has 5 nodes and 3 edges.
```

Listing 5.5 – Code optimizer output log for the graph of the figure 5.6

As excepted, the three unused components have been removed. By removing the nodes 1 and 7, the node 3 has also been subsequently removed because its output was not connected any more. The visualization of the optimized graph is shown in figure 5.7. A node is still in orange, because it has an unconnected input. This component is used in the program and cannot be removed.

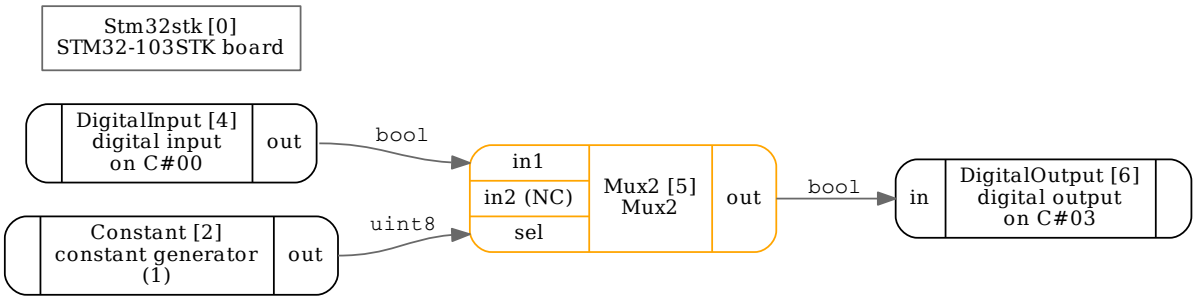


Figure 5.7 – The optimized graph, generated after the code optimizer

This code optimizer block is not absolutely necessary to generate the output code. If some components are not used, the C/C++ compiler will do some optimizations on the code, but this block is useful to keep the output code clean and not "polluted" with unnecessary code. Moreover, when components are removed, warnings are printed and the user is able to correct his program.

If necessary, this block can be disabled by setting the `PIPELINE_RUN_CODE_OPTIMIZER` flag to `false`, in the pipeline setting file. It is activated by default, and the optimized graph diagram is generated to the output folder, with the "`_opt`" suffix in the filename, like the figure 5.7.

5.2.5 Code checker

The dataflow graph stored in memory is valid : ports types mismatch and short circuits have been tested during the graph construction (see section 5.2.1 on page 5.2.1), but the graph can still contains warnings, like in the figure 5.6.

The code checker block will not modify the graph. It only analyses it and prints warning messages if unconnected components or ports are found :

- A warning is printed for each unconnected input or output ports.
- If the code optimizer block is disabled, isolated nodes can be available in the graph. A warning is printed when a node is not connected at all (with degree zero). This check is done with the following code. Thanks to the Scala graph library [Emp15b], the graph can be queried with an elegant functional programming style :

```
1 def findUnconnectedComponents: Set[Component] = {
2  val nc = cpGraph.nodes filter { c =>
3    val cp = c.value.asInstanceOf[Component]
4    val io = cp.getInputs.getOrElse(Nil) ++ cp.getOutputs.getOrElse(Nil) // All I/O ports
5    c.degree == 0 && io.length != 0 // Connected if 0 I/O, isolated if degree 0
6  }
7  nc.map(x => x.value.asInstanceOf[Component]).toSet
8 }
```

Listing 5.6 – Find unconnected components in the graph

With the help of these warnings, the user can correct his program. As excepted, isolated nodes and unconnected ports of figure 5.7 are detected in listing 5.7. Moreover, if warnings are found, the corresponding component is drawn in orange in the graph visualization.

```
[INFO] Code checker started for 'DemoOptimizer'
[WARN] Warnings found:
[WARN] 1 component declared but not connected at all:
  - Cmp[7] 'PwmOutput' (B.09)

[WARN] 3 unconnected ports found (input value set to '0'):
  - InputPort[1] 'in2' of Cmp[05] 'Mux2' (NC)
  - OutputPort[2] 'out' of Cmp[02] 'Add2' (NC)
  - InputPort[0] 'in' of Cmp[07] 'PwmOutput' (NC)
```

Listing 5.7 – Code checker result for the graph 5.6

Unconnected nodes (like the `PwmOutput[7]` in figure 5.6) are simply removed from the graph and will not appears in the generate code.

Unconnected input ports

According to the execution model presented in section 3.5 on page 17, a node can be "fired" (executed) only when all its inputs are ready. To generate a valid sequential code, a default value must be set to all unconnected input ports. For instance, if we look at the application 5.7, the multiplexer input `in2` will be automatically read as "0". A constant block is added and connected before generating the graph code. In this example, the output LED will be off. Depending on the port type, the unconnected input is set to "0" or `false`.

5.2.6 Resolver

The final stage before the code generation consist in resolving the DAG stored in memory, before converting it into a sequential C++ program. The demo application, first presented in figure 3.2 on page 12, is used again to explain how the resolver block works and why it is needed. The application can be programmed with ease in a few lines, using the developed Scala DSL and components available in the framework. The full application code can be found in appendix D.1 on page 84. The application block diagram is shown in figure 5.8.

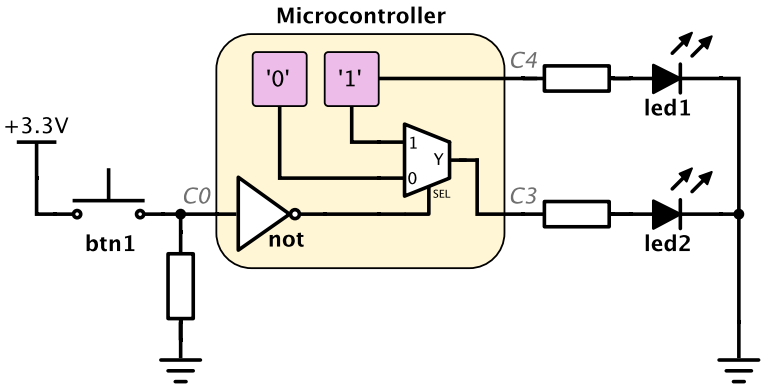


Figure 5.8 – Resolver demonstration application

Any applications which can be described in a DAG can be translated to sequential C/C++ program automatically. The generated code is build using the input-process-output (IPO) pattern. After I/O initializations, all inputs are read, then values are computed and finally output values are updated. The order of these operations is important and must be respected. If not, the program will not compile, or will produce wrong results. To avoid this problem, the resolver block will generate the code of each component is the appropriate order.

Resolving a directed acyclic graph consist of computing its topological sort. The DAG of the application presented in figure 5.8 has been generated automatically and is available in the figure 5.9. With the help of this graph, it is easy to find all inputs (nodes without predecessor) and all outputs (nodes without successor).

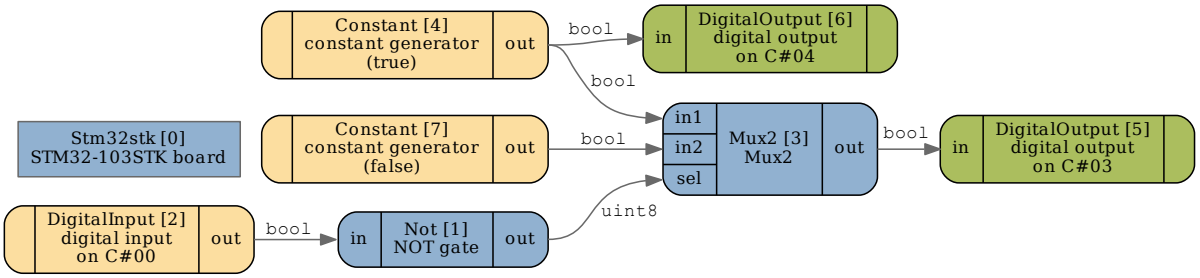


Figure 5.9 – Generated DAG for the application 5.8

Nodes of the DAG can be seen as tasks or jobs, with dependencies between them. In the figure 5.9, the component with ID 3 cannot be executed before components 1,2,4 and 7 for instance. Moreover, the code of a component can be executed (fire) as soon as all its inputs are available. The topological sort will determine the code generation oder for each components of the graph. Most of the time, a topological sort is not unique (if the graph contains more than one path). Different DAG scheduling are available, two of them are explained in this article [Qua04] and presented in the figure 5.10 to resolve the graph 5.9.

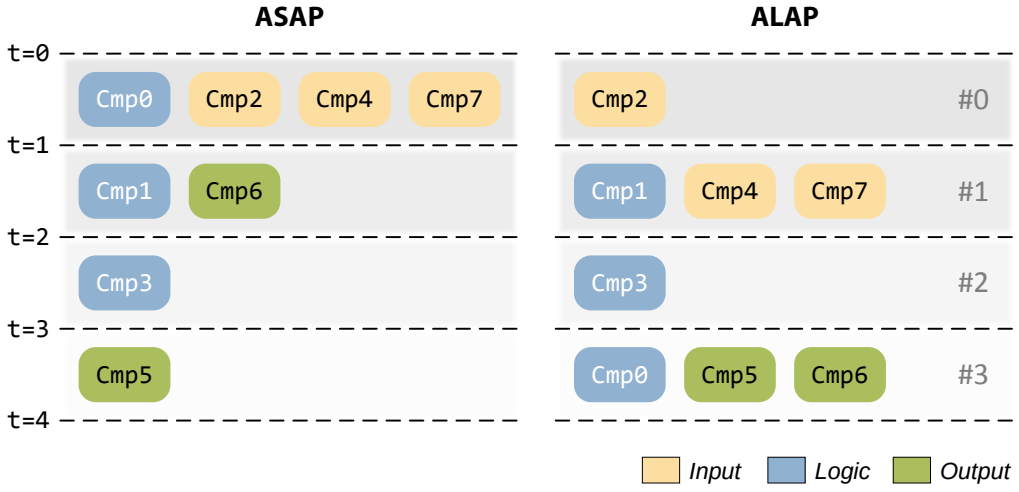


Figure 5.10 – DAG scheduling for the application 5.8

This generated graph (and application) is interesting because its contains different paths with various length, so the results of these two scheduling methods are different. They work as follow [Qua04] :

1. The as-soon-as-possible (*ASAP*) method puts every operation as early in time as possible. The graph is resolved from input to output and nodes are fired whenever the input data are available (as soon as all node inputs are ready).
2. In contrast, the as-late-as-possible (*ALAP*) schedule puts every operation as late in schedule as possible. Nodes fire when absolutely necessary, like `Cmp0` and `Cmp6` in the above diagram.

None of these two scheduling methods are satisfactory because they do not correspond to the IPO model and the chosen execution model (see section 3.5 on page 17), this is way a custom topological sort has been implemented. It forces to read inputs as-soon-as possible, then compute the program logic, and finally update outputs.

The developed resolver takes the program graph as input and returns a `Map[Int, Set[Component]]`. The map index corresponds to a pass number, and the value to the set of components that must be generated. The resolver output for the eight nodes of the application 5.8 is the following :

```
[INFO] Resolver ended successfully after 4 passes for 8 connected components. Result:
  Pass 001: Cmp[0] 'Stm32stk', Cmp[2] 'DigitalInput' (C.00), Cmp[4] 'Cst', Cmp[7] 'Cst'
  Pass 002: Cmp[1] 'Not'
  Pass 003: Cmp[3] 'Mux2'
  Pass 004: Cmp[5] 'DigitalOutput' (C.03), Cmp[6] 'DigitalOutput' (C.04)
```

Listing 5.8 – Resolver result for the application of the figure 5.8 (IPO model)

This result is a mix of the presented *ASAP* and *ALAP* scheduling methods. This time, it fully matches with the IPO specification : all inputs are read during the first pass and all outputs are updated during the last one, like expected (see I/O scanning model presented in figure 3.5 on page 18). This order must be respected to generate the sequential C++ application. Note that components grouped in the same pass do not need to be generated in a particular order (a `Set` of component is returned). Only the components groups for each pass are important.

5.2.7 Code generator

One of the final operation consist of translating the component graph into a sequential and valid C++ program. At this stage, all information are available to generate the code that corresponds to the defined application (without compilation error or wrong computation results).

All generated programs have the same structure, based on the IPO model (see section 3.5 on page 17). Each program is composed by initializations functions, custom functions if needed, and finally by the main function with an infinite `while` loop. Furthermore, generated programs are dived into seven sections, like presented below :

```
1 // Section 00: <includes>
2
3 // Section 01: <global definitions>
4
5 // Section 02: <functions definitions>
```

```
6
7  // Section 03: <I/O initialization>
8  void initOutputs() { /* ... */ }
9  void init()        { /* ... */ }
10
11 int main() {
12     initOutputs(); init();
13
14     // Section 04: <before while loop>
15     while(1) {
16         // Section 05: <while loop, IPO model>
17         //  1) Read inputs
18         //  2) Loop logic
19         //  3) Update outputs
20     }
21     // Section 06: <end of program>
22 }
```

Listing 5.9 – Skeleton of a generated program, divided into seven sections

All sections of the generated application file are "filled" individually with the code of each component, using the order given by the resolver block. Each component is mixed with the `HwImplemented` trait, which defines the seven sections below. Each component is responsible to generate its own C/C++ code using these seven available sections. The component code must be valid because it is paste "as-is" in the generate file (no checks can be performed).

Section 0 First, all necessary header files are included, by calling the `getIncludeCode` method of each components. Each component can include several files (if needed), so duplicates are automatically removed before being added using the `#include` directive.

Section 1 The section 1 is used to declare all global variables. The `getGlobalCode` methods is called for each components. It can be a global variable definition or a class declaration.

Section 2 If needed, a component can implement custom functions. They are pasted "as-is" on the top of the file, in section 2, using the `getFunctionsDefinitions` method.

Section 3 The section 3 is used to initialize all components when the program starts. First, output components (nodes without successor in the graph) are initialized in the `initOutputs` function. All other components are initialized afterward in the general `init` function.

Section 4 The section 4 can be used by any component to declare a local variable or call a specific function once, after the program initialization, using the `getBeginOfMainAfterInit` function.

Section 5 The `while` loop of the program correspond to the section 5. It is divided in three parts, according to the IPO model. The `getLoopableCode` method of each component is called, in a very specific order. This order is given by the code resolver.

Section 6 Finally, a last code can be added at the end of the loop, to indicate if an error has occurred or if the program has been stopped, using the `getExitCode` method of the `HwImplemented` trait.

To summarize the code generation process, we use again the application example defined in the figure 5.8 on page 43. The component generation order is given by the resolver block and is shown in the figure 5.11 :

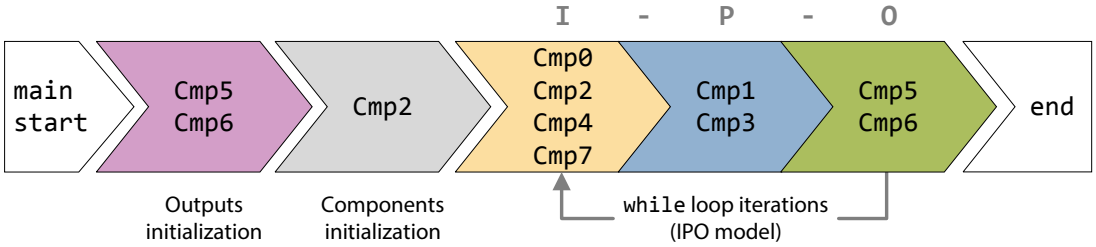


Figure 5.11 – Code generation sequence for the application 5.8

A part of the generated code (the `while` loop only) for the application 5.8 is available below. The resolver order of the figure above has been used to generate the file.

```
while(1) {
    // 1) Read inputs                          ------------
    bool in_C0 = in_cmp02.get();               // [I]: Cmp2

    // 2) Loop logic                           ------------
    uint8_t out_cmp01 = !in_C0;                // [P]: Cmp1

    uint8_t sel_cmp03 = out_cmp01;             // [P]: Cmp3
    bool out_cmp03;
    if(sel_cmp03 == 0) out_cmp03 = true;
    else out_cmp03 = false;

    // 3) Update outputs                       ------------
    out_cmp05.set(out_cmp03);                  // [O]: Cmp5
    out_cmp06.set(true);                       // [O]: Cmp6
}
```

Listing 5.10 – Part of the generated code for the application 5.8

The code of each component, defined using the `HwImplemented` trait, has been simply added in each sections to compose the final application. The full generated application code is available in appendix D on page 84. As an example, the implementation of the `Not` component (`cmp1` in the previous code) is also available in appendix D. The `Not` component can be used to invert any type of data. Depending on the input type, the generated code can change. For boolean values, the `"!"` operator is used, otherwise and if/else statement is necessary.
By default, the generated file contains comments and debug informations. The `GEN_VERBOSE_CODE` flag can be disabled in the `Settings` class to remove them automatically.

5.2.8 Code formatter

The last code transformation applied is a code formatting operation. Like explained in the previous section, each component generates its own C/C++ code. This implies that the style and the format of the generated code can looks a bit different from a component to another (space/tab indentation, spaces, curly braces, etc.).
To keep the generated code clean, readable and uniform, the *AStyle* [Pat14] tool is used to re-indent and re-format the generated code. Artistic Style is a free and fast formatter for C/C++ and other languages. It is a small, automatic and cross-platform tool, highly customizable. All its configuration parameters are given from a command line, within the Scala code by creating a `Process`.

The developed formatter block is fully automated. It uses AStyle to make the code more readable for the user. The original generated code is available in the output folder with a "`.org`" extension. AStyle executables (for Linux and Windows) are available in the "`third_party`" folder. It is not necessary to install the tool.

The generate code of the application 5.11 presented in appendix D has been formatted using Astyle.

5.3 Compilation & simulation pipeline

At this stage, different files have been automatically generated, from the user Scala program (see figure 5.2 on page 34). The previous section explained how the user application, written using a Scala DSL, is transformed to a sequential C/C++ code. Now it is time to use this generated code. These two approaches will be discussed in this section.

1. **Testing using the real target**
   The first option consists in compiling the code using a standard ARM toolchain (presented in section 4.2.1 on page 4.2.1) and then programing manually the `STM32-103STK` target board, like any other embedded system.
2. **Code simulation**
   In the second approach, an ARM emulator has been modified to simulate the generated application, without using a real target. This allows to automatize test procedures to validate the behavior of the generated code.

5.3.1 Real target

The user Scala application has been converted to a standard "`.cpp`" file. The normal approach discussed here consists in using the toolchain presented on page 23 to program the real target and test the application manually.

To do this, the `stm32` repository contains a ready-to-use Eclipse project, with all necessary dependencies, files, running and debugging Eclipse configurations. The application uses the `stm32f10x_lib` library and can be compiled using the provided Makefile directlly in the Eclipse IDE (see figure 4.2 on page 23). Once the application is compiled, the board can be programmed using a generic JTAG interface, and finally the code can be executed.
The generated code is clean and can be modified with ease if necessary. A compilation block has also been developed. This block generates the ARM executable file (ELF format) automatically, within the Scala code. It is presented below.

This standard approach is necessary to test if the generated code really works on the target, but its major drawback is that the testing process cannot be automatized (and programming the ARM board takes quite a long time). To test the generated applications with ease using automated tests, a simulated environment has been setup, in addition to this standard approach.

5.3.2 ARM emulator

A significant portion of the project consisted in setup a simulated environment to execute and test the generated application, without the real hardware. Automated tests have been developed to check if the generated application corresponds or not to its specification. This was a challenging part, because the hardware is emulated in software, and its implementation must exactly match with the real hardware.

QEMU [Bel14] is a generic and open source machine emulator and virtualizer. This existing project has been used to emulate the ARM Cortex-M3 board, presented in section 4.2 on page 22. QEMU runs programs made for one machine on a different machine, with very good performance. Many architectures are supported : X86_64, ARM (iMX, Cortex-M3, OMAP), etc.

The official QMEU version does not support the `STM32` microcontroller, so a modified version of QEMU has been used. The `qemu_stm32` project [Bec14] is a fork of the official QEMU version. It adds the STM32 microcontroller implementation with some peripherals in QEMU, and its also implements the `STM32-P103` Olimex development board. With the help of this project, the `STM32F103` Cortex-M3 processor can be emulated in software. This allows to simulate the development kit with some MCU peripherals on a PC, without the real hardware. The generated program is compiled into an ARM binary file, and this same file can be run either on the real target or in the simulated environment in QEMU.

Simulation environments (like ModelSim) are widely used to simulate hardware description languages, but this approach is not often used for embedded systems. Emulating a processor in QEMU adds a very high overhead. Thanks to the `qemu_stm32` project, the implementation is already done. With some developments, simulating the generated code in QEMU rather than on the real target allows to run automated tests, by controlling the code execution and input and output pins of the processor. Automated tests are presented in section 5.3.3.

The compilation and simulation pipeline is the second pipeline developed in this project. It is presented in the figure 5.12 :

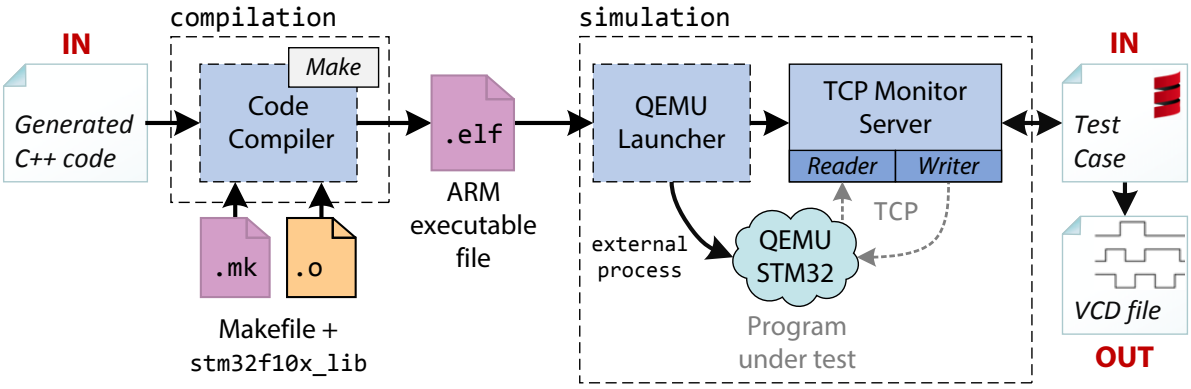


Figure 5.12 – The compilation and simulation pipeline

The code compiler block compiles the C++ application to an ARM executable file with the help of a generated Makefile and the backend library, which contains the low level components implementation (details in section 4.2.1 on page 23). The `make` command is called within the Scala code by creating a `Process`. The ARM toolchain must be correctly installed and available in the path. To run the application in QEMU, a modified linker script is used (to modify the offset address of the flash memory), but all other files are exactly the same as the files used for the real target.

The generated application has been compiled and must be now tested in the simulated environment. QEMU is launched from the Scala code in an external process, using the `stm32-p103` machine and the generated elf file as kernel. The command 5.11 is used to launch the QEMU emulator, without graphical interface and in a separated process, so the Scala test-case execution can continue in parallel.

```
info("Start QEMU in a new process.")
val cmd= s"./${Settings.PATH_QEMU_STM32}/arm-softmmu/qemu-system-arm -M stm32-p103 -kernel" +
    " csrc/target-qemu/csrc.elf -serial null -monitor null -nographic"
val qemuProcess = OSUtils.runInBackground(cmd) // Run QEMU in a new process

// Application test code...

qemuProcess.destroy() // Kill QEMU
```

Listing 5.11 – Launch QEMU in an external process from the Scala code

5.3.3 Automated tests

The main purpose of the simulated environment in QEMU is to automatically test the generated application. As for hardware description languages, a Scala test case controls and monitors the application execution in QEMU. The simulated environment allows to :

1. Control the code execution is QEMU. The microcontroller code executed in QEMU may be paused or stopped by the Scala test case.
2. Set input pin values of the microcontroller, to simulate when a button is pressed for instance.
3. Monitor the output pins values of the MCU at any time, to check if the simulated program correspond to its specification.

To do this, the `qemu_stm32` has been extended. A TCP/IP gateway allows to communicate from the QEMU simulator to the Scala frontend. Messages and events can be sent/received from/to the Scala side using the TCP monitor block (see figure 5.12).

On the Scala side (the frontend), two TCP/IP server are created by the `Monitor` block. One connection is used to send messages to QEMU, the other to read events from the simulator. Exchanged messages are formatted in *JSON*. A concrete sequence diagram is presented in the next chapter.

On the backend part, the QEMU source files have been modified and two new TCP client threads have been implemented to communicate with the Scala side. Messages and events produced by the main QEMU thread (the MCU code execution) are stored in queues to be sent to the frontend by another specific thread.

Developments in QEMU were challenging because QEMU is implemented in C. Low-level functions have been used to implement queues, to send Json messages over TCP/IP, etc. Moreover, debugging codes in QEMU is hard and time consuming because I was not very familiar with it. I had the opportunity to take an introductory course on QEMU during my master studies. Some code developed and used during this course[7] has been adapted for the project.

QEMU developments are very specific and will not be detailed here. Information about the QEMU compilation procedure are available in appendix B.3 on page 81.

7 Systèmes d'exploitation et environnements d'exécution embarqués (*SEEE*) - http://reds.heig-vd.ch/formations/master/SEEE

Value change dump export

When a program is running in QEMU, digital output pin values are monitored by the frontend. When an output pin value changes, its new state is forwarded to the Scala side through TCP/IP (in a Json message). This message identifies the port, the pin number and its new boolean value. At the end of the test, saved output values can be compared to the expected values to check if the program corresponds to its specification. Output pin values can be displayed textually, but also graphically in a digital timing diagram. This visual representation helps to check if the program, executed in the simulated environment, works correctly.

The Value Change Dump (VCD) format [IEE01, pp. 348-339] has been choose to save and export digital output values. VCD is ASCII based and is used since 1995 by EDA logic simulation tools. This file format is quite old, but still used and supported by man tools. Its format is simple and can be generated with ease. Basically, output values are simply dumped in the file, with a corresponding timestamp. The `VcdGenerator` pipeline block has been developed to generate VCD files.
The code 5.12 is a simple Scala test case used to generate a VCD file with predefined output values. The VCD file is generated automatically to the output folder.

```
1 // Output boolean values of several pins.
2 val pinValues: Map[Pin, Seq[Int]] = Map(
3   Pin('A', 1) -> Seq(0, 1, 0, 1),
4   Pin('B', 2) -> Seq(1, 0, 1, 0),
5   Pin('C', 3) -> Seq(0, 1, 1, 0),
6   Pin('D', 4) -> Seq(0, 1, 0)
7 )
8 val ctx = new Context(this.getClass.getSimpleName, true)
9 new VcdGenerator().run(ctx)(pinValues)  // Generate the VCD file to the output folder
```

Listing 5.12 – VCD generation test case in Scala (`VcdGeneratorTest.scala`)

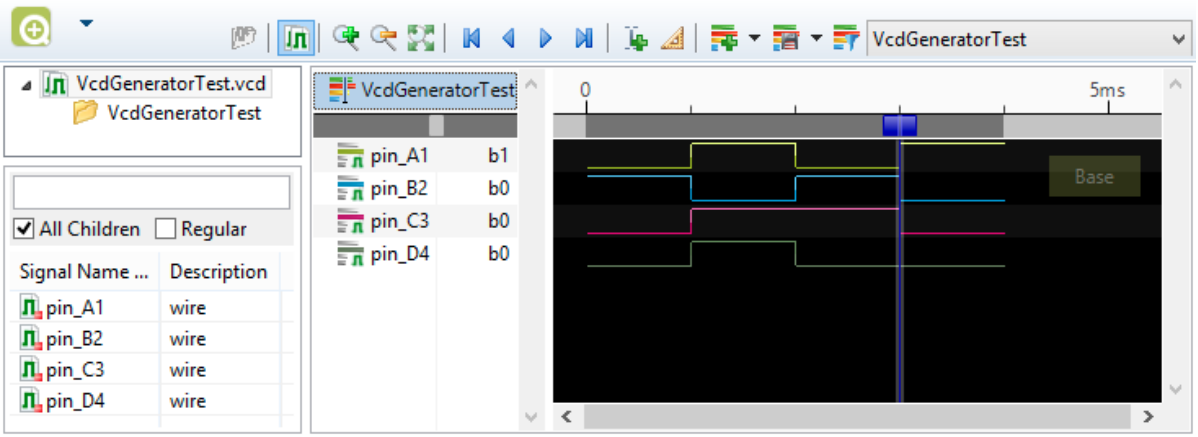


Figure 5.13 – Digital timing diagram of the test case 5.12 in Impulse (Eclipse plugin)

In this simple VCD generation example, outputs have predefined values. In reality, real output pin values, collected from the QEMU execution, are used. For space reason, the VCD file generated from the code 5.12 is only available in appendix E on page 88. Its graphical representation is shown in figure 5.13.

VCD is a generic text format. To be displayed graphically in a digital timing diagram, it must be imported in an external tool. Several EDA softwares supports this file format (like Modelsim for instance, using the `vcd2wlf` command). The digital timing diagram shown in figure 5.13 has been generated using *Impulse*, a free Eclipse plugin[8]. It is light, includes a lot of features to draw advances waveforms, and it is fully customizable. I recommend it, but many other visualization tools exist.

In the next chapter, automated tests will be used to check the behavior of a concrete application. The Scala test-case and its corresponding digital timing diagram will be presented in section 6.1.1 on page 63.

5.4 DSL improvements

A first version of the Scala internal DSL has been developed with basic functionalities. It can be improved in order to write dataflow applications with less code, in a user-friendly and more natural way. Based on Scala features described in the beginning of the document (see section 3.3.1 on page 14), some improvements are presented below.

5.4.1 Components and pins definitions

Input/outputs pins and components available on the `STM23F103-STK` extension board are widely use in the DSL to build applications. With this in mind, the `Stm32stk` and the `Stm32stkIO` objects allows to access to predefined pins and components with ease :

```
1  // Components and pins definitions of the extension board
2  object Stm32stkIO {
3      val led4_pin = Pin('B', 9)
4      lazy val led4 = DigitalOutput(led4_pin)     // Red LED on ‘PB.9‘
5
6      val adc1_pin = Pin('B', 0)
7      lazy val adc1 = AnalogInput(adc1_pin, 8)    // Potentiometer on ‘PB.0‘ (ADC channel 8)
8
9      val pwm4_pin = led4_pin
10     lazy val pwm4 = PwmOutput(pwm4_pin)         // PWM for led4 on ‘PB.9‘ (Timer4 channel 4)
11     // etc.
12 }
```

Listing 5.13 – Useful pins and components definitions (`Stm32stk.scala`)

All available components (and their pins) are defined in these objects. Components are declared as `lazy` so if they are imported but not used, they will not be instantiated and added to the graph. These components and pins definitions are used in the rest of the document. All ports and pin numbers of the development kit and its extension board are available in appendix A on page 78.

5.4.2 Anonymous components

The following code uses the previous components definitions. It simply connect a button to two LEDs. They should be both powered on when the `btn1` is pressed.

8 Impulse is a waveform viewer integrated into Eclipse - http://toem.de/index.php/projects/impulse

```
import hevs.especial.dsl.components.target.stm32stk.Stm32stkIO
Stm32stkIO.btn1.out --> Stm32stkIO.led1.in // same as DigitalInput(Pin('C', 0)).out --> ...
Stm32stkIO.btn1.out --> Stm32stkIO.led2.in
```

Listing 5.14 – Use of anonymous components (`Sch1Code.scala`)

In this example, if nothing is done, two buttons will be instantiated and added to the graph. In reality, only one button must be added. This problem does not appear if temporary variable is used, it only appears with anonymous variables.

All components are identified by a port and pin number (the `Pin` Scala class). So to solve this bug, before inserting a component in the graph, we must first check if it already exists in the graph. If the answer is yes, the existing component is returned, otherwise the new one is added. To check if a components is already in the graph a generic solution has been implemented. It consists of using the Java `equals` and `hashCode` methods. Here is a sample code which prevents to add more than one `DigitalInput` in the graph for each port and pin :

```
1  class Pin { // ...
2   override def equals(other: Any) = other match {
3      case that: Pin =>
4        // Compare two pins. Check if the port and the pin number are the same.
5        that.pinNumber == this.pinNumber && that.port == this.port
6      case _ => false
7    }
8  }
9
10 class DigitalInput private(private val pin: Pin) { // ...
11  override def equals(other: Any) = other match {
12     case that: DigitalInput => that.pin == this.pin // Pin must be unique
13     case _ => false
14   }
15 }
16
17 object DigitalInput {
18   def apply(pin: Pin): DigitalInput = {
19     val tmpCmp = new DigitalInput(pin)
20     // Check if the component already exists in the graph (return 'Some').
21     val isAdded: Option[DigitalInput] = ComponentManager.addComponent(tmpCmp)
22     isAdded.getOrElse(tmpCmp) // Return the existing component, otherwise the new one.
23   }
24 }
```

Listing 5.15 – Prevent components duplicates in the graph (simplified code)

Thanks to this code, now all digital inputs created on the same port and pin number will appear only once in the graph. To prove it, the graph of the program 5.14 has been generated in figure 5.14. It contains three nodes (and not four), as expected.
The constructor of the `DigitalInput` class is private to force to use the companion object for its instantiation (see code 5.15). The same pattern is used by all inputs and outputs components, identified by a `Pin`. Any other component that should appear only once in the graph must implement the same pattern.

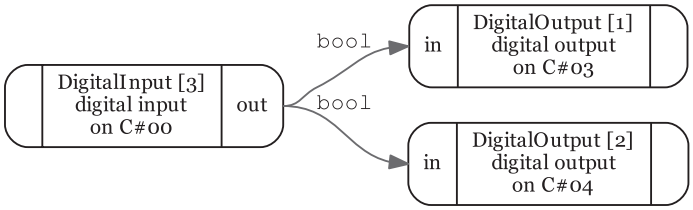


Figure 5.14 – Generated graph of the application 5.14

5.4.3 Pins functions

Another problem appears if the same input or output pin is used with two different configurations. The code 5.16 presents the problem. It will generate an exception at runtime.

```
1 val pin = Pin('A', 1)          // Sahred component pin
2
3 val out1 = DigitalOutput(pin) // Same output pin used as ‘DigitalOutput‘ AND ‘PwmOutput‘
4 val out2 = PwmOutput(pin)     // -> Error !
```

Listing 5.16 – Output pin used with two different configurations

In this particular example the output pin `A#1` is configured as a digital and a PWM output. This does not make much sense, furthermore this is not supported because all components are initialized once when the program starts : reconfiguring a pin or a controller on the fly would cause performance issues.

Before adding the component in the graph, a new case must be tested. If the component already exists in the graph and if it do not correspond to the current component type, an `IoTypeMismatch` exception must be thrown. The following message is printed for the code 5.16 :

```
[ERROR] I/O already used !
The component Cmp[3] is already used as 'DigitalOutput'. It cannot be used as 'PwmOutput'.
```

This runtime exception is thrown due to the line 4 of the code 5.16. The application is not valid and no code will be generated.

5.4.4 Variadic constructors

The DSL has been improved by using variadic constructors to instantiate and connect components inputs in one line. Up to now, the "`-->`" operator has always been used to connected components ports, but this can sometimes create verbose codes. The idea is to write less code in a more natural way. Here is an example on how variadic constructors can be used in a simple application composed by an `And` logic gate with two inputs :

```
1 object And {
2   def apply(inputs: OutputPort[bool]*) = inputs.size match {
3     case 0 | 1 | 2 => And2(inputs: _*) // 2 inputs are necessary
4     case 3 => And3(inputs: _*)         // 3 inputs are necessary
5     case 4 => And4(inputs: _*)         // 4 inputs are necessary
6   }
7 }
```

```
8
9  case class And2(inputs: OutputPort[bool]*) extends MathOps(2, "&", inputs: _*) with In2 {
10   override val description = s"And$nbrIn gate"
11   override val in1 = in(0)
12   override val in2 = in(1)
13 } // Same for And3, And4, Or2, etc.
```

```
1  val and = And(Stm32stkIO.btn1.out, Stm32stkIO.btn2.out) // -> And2 (from 0 to 4 inputs)
2  // Same as: val and = And2()
3  //           Stm32stkIO.btn1.out --> and.in1
4  //           Stm32stkIO.btn2.out --> and.in2
5  and.out --> Stm32stkIO.led1.in                          // And2 logic gate to a LED
```

Listing 5.17 – User friendly DSL code using variadic constructors

As an example, variadic constructors have been added to logic gates. This includes `And`, `Or` and `Not` gates with 1 to 4 inputs. Several case classes are available in the framework to connect manually all components inputs, but this can be a bit bothering and verbose.
In the code 5.17, the generic component `And` is instantiated with a variadic constructor. The two inputs (`btn1` and `btn2`) will be automatically connected to a `And` inputs. It is no more necessary to specify if a `And2`, `And3` or `And4` gate must be used. Moreover, all component's inputs are connected automatically. This make the DSL code shorter.

The component inputs connections are "hidden" in a generic implementation. As always, the "`-->`" is used to connect outputs to inputs (see line 7 of the code 5.18). If less parameters as the input number are given, some inputs will be not connected. In contrast, if more parameters are given, they will be ignored.

```
1  abstract class MathOps[T <: CType : TypeTag](val nbrIn: Int, operator: String) /* ... */ {
2
3    def this(nbrIn: Int, operator: String, inputs: OutputPort[T]*) = {
4      this(nbrIn, operator)
5      for (i <- 0 until nbrIn) {
6        if (inputs.indices.contains(i))   // Ignore if too much or too less parameters are given
7          inputs(i) --> in(i)             // Automatically connect component's inputs
8      }
9    }
10 }
```

Listing 5.18 – Connection of a generic number of inputs

Variadic constructors are available for all logic gates and multiplexers. They help to write a more concise and understandable code for component with a generic number of inputs. Of course, if it is necessary, inputs can still be connected manually.

5.4.5 Boolean operators

Variadic constructors help to connect logical gates with multiple inputs, but it can be even more simple to use it. The following code demonstrates how implicit conversions can improve the internal DSL. As an example, logic gates can now be instantiated and connected in a very elegant way, using only one line of code (improvement of the code 5.17) :

```
1 import hevs.especial.dsl.components.core.logic._  // Use impl. conv. from the logic package
2
3 val A = Stm32stkIO.btn1.out
4 val B = Stm32stkIO.btn2.out
5
6 (A & B) --> Stm32stkIO.led1.in  // Same as: And2(A, B).out --> Stm32stkIO.led1.in
```

Listing 5.19 – Logic gates and boolean operators

Implicit conversions (see section 3.3.1 on page 14) have been added to the `logic` package. New operators are now available for `Boolean` ports. These new operators are syntactic sugar to help making the code sorter and easier to understand. In line 6 of code 5.19, the `&` methods allow to create a logic `And` gate with two inputs in a very natural way.

To do this, the implicit conversion for output boolean ports (`OutputPort[bool]`) is defined as follow. Boolean operators are implemented using existing components (And, Or and Not gates). Some examples are presented below :

```
1  package object logic {
2    implicit def toBooleanOutputPort(out: OutputPort[bool]): BooleanOutputPort =
3        new BooleanOutputPort(out) // Converts an OutputPort to a rich BooleanOutputPort
4  }
5
6  class BooleanOutputPort(port: OutputPort[bool]) {
7    private type T = OutputPort[bool]
8
9    def &(out: T): T = And2(port, out).out    // And operator
10   def |(out: T): T = Or2(port, out).out     // Or operator
11   def unary_!(): T = Not(port).out          // Unary Not operator
12 }
```

Listing 5.20 – Implicit convertion for boolean ports

The `toBooleanOutputPort` implicit method converts an output port of boolean type to a rich `BooleanOutputPort` class. This rich class implements additional operators, only available for boolean types (and, or, not). The component output port is returned so boolean operators can be chained if more than two inputs are used.

As a proof of concept, implicit conversions have been added for logic gates. The same pattern can be applied to all other components, for instance for math blocks to do additions, subtraction, etc. with one single operator. Thanks to the Scala internal DSL, implicit conversions can be added with ease to extend the language, depending on the user needs. A more complete example using boolean implicit conversions will be presented in the chapter 6.

5.4.6 Custom components

The dataflow internal DSL can be extended with ease by the user. To help him creating new (custom) components, the `CFct` helper class has been developed. It allows to create a custom component with one input and one output of any type. The component logic is defined using a small piece of native C/C++ code directly.

A simple threshold component has been implemented using this helper class. It converts an analog value (`uint16`) to a boolean value (`bool`). Its Scala implementation is the following :

```
1  case class Threshold(threshold: Int = 512) extends CFct[uint16, bool]() {
2    private val outVal = valName("threshold")
3    override val globalVars: Map[String, CType] = Map.empty  // No global variables used
4    override def getOutputValue: String = s"$outVal"
5
6    override def loopCode: String = {  // C code implementation inserted in the while loop
7      val in = getInputValue   // Read the input value of the block (stored in a variable)
8      val outType = getTypeString[bool]
9      s"""$outType $outVal = false;
10      |if($in > $threshold)
11      |  $outVal = true;""".stripMargin // Must return a valid C/C++ code
12   }
13 }
```

Listing 5.21 – Custom threshold component (`CustomThreshold.scala`)

This component is used in the demo application 5.22. Its input and output are accessible as always with the `in` and `out` attributes. The figure 5.15 is the generate graph of the application.

```
1 val adc1 = Stm32stkIO.adc1.out
2 adc1 --> Stm32stkIO.pwm3.in
3
4 val threshold = Threshold(512)          // Custom component declaration
5 adc1 --> threshold.in                   // and connection
6 threshold.out --> Stm32stkIO.led1.in
```

Listing 5.22 – Threshold demo application (`CustomThreshold.scala`)

In the application 5.22, a potentiometer (connected on an analog input pin) controls a PWM output. It is also connected on the custom threshold component. When the potentiometer value (`uint16`) is higher than the threshold value, the `led1` switch on, otherwise it is off (a very basic trigger). The threshold value is a parameter of the block. In this example, it is fixed at the compilation time and cannot be modified during the code execution (a constant component could be added on a second input to update the threshold value on the fly).

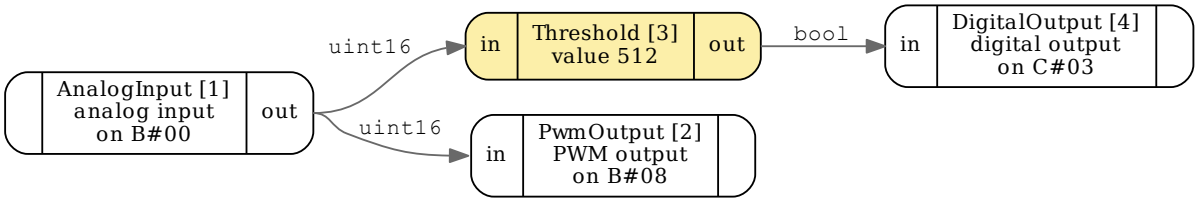


Figure 5.15 – Generated graph of the application 5.22

The logic of the component is written in C or C++. The returned code of the `loopCode` method is pasted "as-is", without any check or verification, into the generate while loop code (see lines 7 to 11 below).

A part of the generated code of the program 5.22 is available here :

```
1  while(1) {
2    // 1) Read inputs
3    uint16_t in_B0 = getlAnalogInputB0();
4
5    // 2) Loop logic
6
7    // ---- User input code of the custom component ‘Threshold (512)‘ ----
8    bool threshold_cmp03 = false;
9    if(in_B0 > 512)
10     threshold_cmp03 = true;
11   // ----
12
13   // 3) Update outputs
14   out_cmp02.setPeriod(in_B0);
15   out_cmp04.set(threshold_cmp03);
16 }
```

Listing 5.23 – Part of the generated code for the application 5.22

Custom components are great to create new components with a few line of codes, but they can be dangerous because their C/C++ implementation cannot be verified. It must be a valid C/C++ code or the program will not compile.
Moreover, the implementation code must be conform to the IPO model : it must not be blocking or taking a long execution time, all the component inputs must be read and all its outputs must be updated, or the dataflow graph will not be executed correctly and the execution model (see section 3.5 on page 17) will be broken.

If the look at the threshold component implementation (see code 5.21), the input port value is read on line 7 (from a local C variable). Then, the logic of the component, implemented in C, checks this value and store the boolean result in the `outVal` local variable. The `getOutputValue` method is used by the successor component (the digital output) to read the boolean result of the threshold block.

Custom components have been implemented to create simple blocks with ease, in a few lines of codes (see listing 5.21). They are useful and powerful but they must be used carefully : if the component implementation is not valid, the code compiler block will report a compilation error, or worse, the program results can be wrong.

Another custom block implementation will be presented in a concrete application in the next chapter.

Chapter 6

Real-world applications

In this last technical chapter, two real-word applications built using the developed dataflow DSL will be presented. These concrete applications show what type of programs can be build using the high-level dataflow representation and present some results of the project.

Two application types have been selected. The first application is a combinatory logic circuit. It presents some specific features of the dataflow internal DSL. The application is tested on the real target and in the simulation environment. Its behavior is fully checked using automated tests written in Scala and QEMU.
The second application will be tested on the real target only. It uses the extension board and several peripherals of the microcontroller (not implemented yet in the simulator) to create a full regulation application using a fan. Scala codes and generated files of each application will be presented bellow.

6.1 Majority function

As a first demonstration application, a majority circuit has been developed. The program computes the majority off three inputs, which are the three buttons available on the extension board (see section 4.5 on page 28). When two buttons or more are pressed, the `led1` of the extension board is switched on, otherwise, the LED is off (see table 6.1).

A	**B**	**C**	**Majority** $(A \wedge B) \vee (B \wedge C) \vee (A \wedge C)$
0	1	1	1
1	0	1	1
1	1	0	1
1	1	1	1

Table 6.1 – Truth table of a three-input majority circuit (all other cases are 0)

The truth table 6.1 can be implemented using the DSL with three `And` and two `Or` logic gates. Rather than connect all logic components manually without any help, some features presented in the section 5.4 are used to write less code in a more natural way : *a*) input ports are connected automatically using variadic constructors; *b*) implicit conversions are used as syntactic sugar to write less code in a more elegant way.

A first implementation of the majority circuit is available below. It presents three ways to instantiate and connect the application components :

```
1  import hevs.especial.dsl.components.core.logic._  // Necessary to use impl. conv.
2
3  val A = Stm32stkIO.btn1.out             // Input buttons
4  val B = Stm32stkIO.btn2.out
5  val C = Stm32stkIO.btn3.out
6  val O = Stm32stkIO.led1.in              // Output LED
7
8  val and1 = And2()                       // 1) Manual connection
9  A --> andA1.in1
10 C --> andA1.in2
11
12 val and2 = And2(B, C).out               // 2) Variadic constructor
13
14 val and3 = (A & C)                      // 3) Implicit convertions
15 (and1.out | and2.out | and3.out) --> O
```

Listing 6.1 – Three-input majority circuit (verbose)

Using implicit conversions, the equation of the majority circuit (available in table 6.1) can be translated in no more than one line (see line 6 below). The following code give the same result as the application 6.1.

```
1  val A = Stm32stkIO.btn1.out      // Input buttons
2  val B = Stm32stkIO.btn2.out
3  val C = Stm32stkIO.btn3.out
4  val O = Stm32stkIO.led1.in       // Output LED
5
6  (A & B | B & C | A & C) --> O    // Majority function
```

Listing 6.2 – Three-input majority circuit

Implicit conversions make the code pretty clear, and any logic operation can be written in natural way. The following diagram helps to understand how the line 8 of the code 6.2 is translated to a graph. Infix operations precedence rules [Ode14, pp. 84-85] are used :

```
1  (A & B | B & C | A & C) --> O
2  ((A & B) | (B & C) | (A & C)) --> O
3  (((A & B) | (B & C)) | (A & C)) --> O      Implicit conversions
4  ((And2(A, B).out | And2(B, C).out) | And2(A, C).out) --> O
5  (Or2(And2(A, B).out, And2(B, C).out).out | And2(A, C).out) --> O
6  Or2(Or2(And2(A, B).out, And2(B, C).out).out, And2(A, C).out).out --> O
```

Figure 6.1 – Infix operations precedence rules and implicit conversions for logic operators

The first line in figure 6.1 corresponds to the majority circuit equation available in the DSL code. According to the precedence rules in Scala, letters have the lowest precedence, followed by "`|`", then "`&`" and finally the "`-->`" operator. All lines of the figure 6.1 are equivalent, only implicit parenthesis have been removed. Note that parenthesis around the equation are necessary because the "`-->`" operator (with the highest priority) not only applies to "`C`" but to the whole equation. In the three last lines of the figure 6.1, implicit conversions are replaced with logic gates (see code 5.20 on page 57).

The 3-input `Or` gate is transformed to two `Or` gates with 2 inputs, which is an equivalent circuit (the `Or` operation is associative and commutative). The generated graph 6.2 can be inferred from the line 6 of the figure 6.1. The tree input buttons are on the left and the output is on the right.

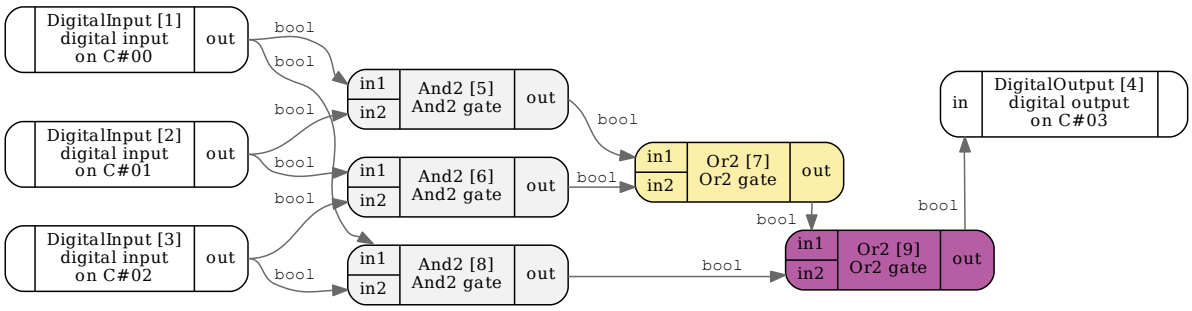


Figure 6.2 – Generated graph for the majority circuit (code 6.2)

The generated code for the 3-input majority application is available in appendix F on page 89.

6.1.1 Simulation

As a first step, the generated code has been compiled and loaded into the `STM32F103-STK` development kit, to be tested on the real hardware (details in section 5.3.1 on page 49). The application works as expected, using the three buttons and the LED of the extension board. The list of hardware and software necessary to run this demonstration application is available in appendix B.

The other approach, consisting of simulating the application, is presented in this section. The majority circuit is simulated in QEMU (details in section 5.3.2). This simulation allows to test the application without using the real hardware. An exhaustive test of the application is performed : the eight possible input values are tested. The test case, written in Scala, controls the code execution running in QEMU. On the beginning of each `while` iterations, it also set input values to simulate when a button is pressed.

A part of the generated code for the majority application (program 6.2) is shown below. The full generated code is available on page 89.

```
1  while(true) {
2    // While Loop iteration event (ACK is needed from the Scala side to continue the execution)
3    QemuLogger::send_event(SECTION_LOOP_TICK, true);
4
5    // 1) Read inputs
6    bool in_C2 = in_cmp03.get();
7    bool in_C0 = in_cmp01.get();
8    bool in_C1 = in_cmp02.get();
9
10   // 2) Loop logic
11   bool out_cmp05 = in_C0 & in_C1;
12   bool out_cmp06 = in_C1 & in_C2;
13   bool out_cmp08 = in_C0 & in_C2;
14   bool out_cmp07 = out_cmp05 | out_cmp06;
15   bool out_cmp09 = out_cmp07 | out_cmp08;
16
17   // 3) Update output
18   out_cmp04.set(out_cmp09);
19 }
```

Listing 6.3 – Generated code for the majority application (only the main loop)

To simulate an application in QEMU, specific pieces of code must be added to the generated program (see line 3 in code 6.3). These codes allow to control and monitor the code execution from the Scala side. When the code on line 2 is executed in QEMU, an event is automatically sent to the Scala side over TCP/IP. These events are identified by a name and are placed at specific places in the generated code, at the beginning or at the end of code sections (available sections are presented on page 45). The second parameter of the `send_event` method can be used to wait for a confirmation from the Scala test case. Until no acknowledge is received, the code execution in QEMU is paused. This is useful to update the microcontroller input pins at a specific time, for instance at the beginning on each loop iterations.

The majority circuit is tested using this principle. The exhaustive Scala test case of the majority circuit is summarized in the sequence diagram 6.3. It shows the TCP/IP messages exchanged between the Scala test case and QEMU.

All 2^3 (8) inputs values are sent over TCP/IP to QEMU. This means that after 8 loop iterations, the application has been fully tested. At the end of the test the QEMU output values are compared to the expected results, to check if the program works correctly or not. Output values are exported to a VCD file to be analyzed with ease. The full Scala test case of the majority application, which corresponds to the sequence diagram 6.3, is available in appendix F.

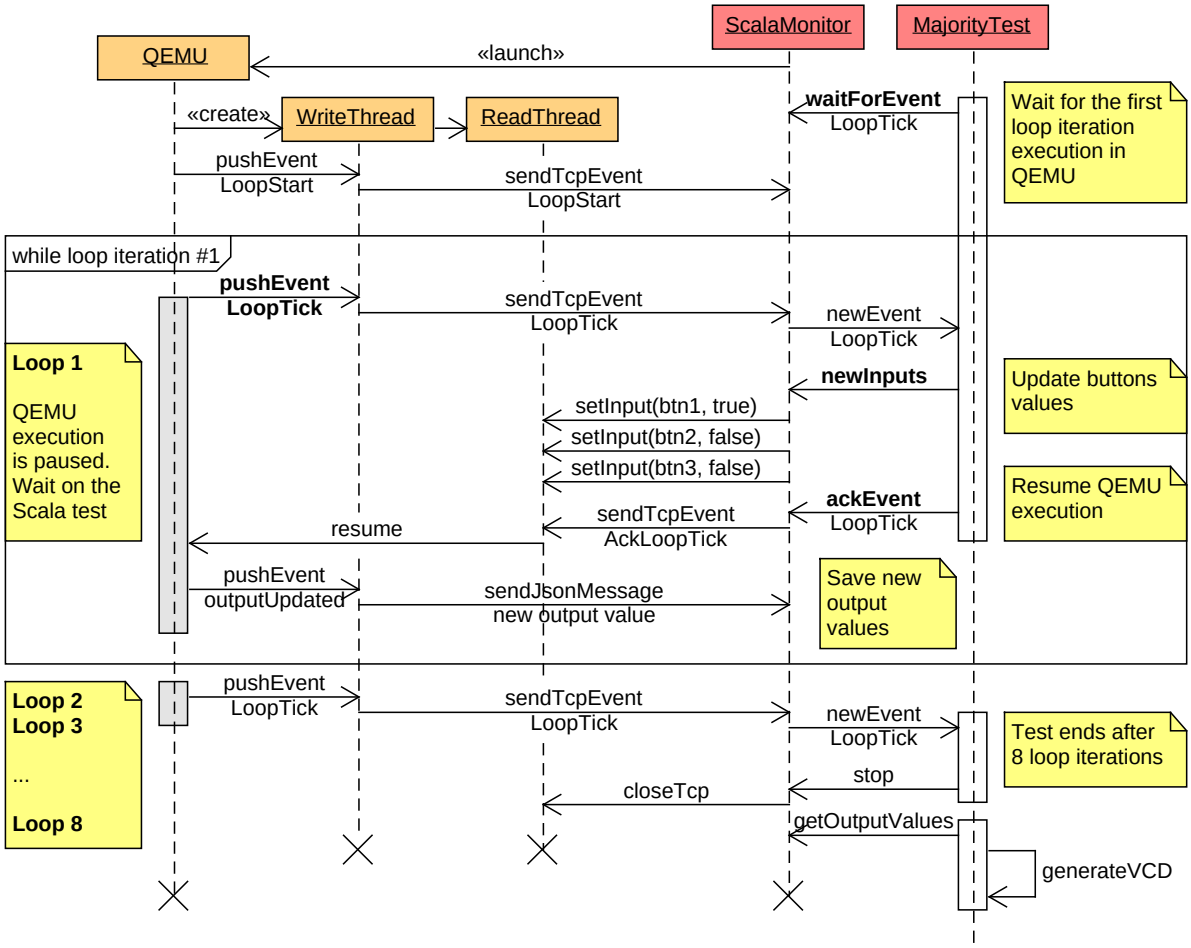


Figure 6.3 – Test case sequence diagram of the majority application

6.1.2 Digital timing diagram

On each while loop iterations, the majority output is updated. In QEMU, outputs values are monitored and automatically sent to the Scala side. They are first saved in a queue and sent later by a specific thread in QEMU. This is necessary to be sure that output values are sent in the correct order.

Output values sent from QEMU are compared to the expected results to check if the program output (the majority function) is correct or not. At the end of the test, the digital timing diagram of the input and output pins is generated :

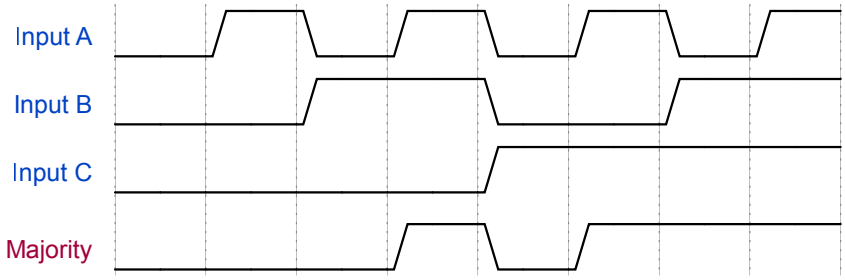


Figure 6.4 – Exhaustive test of the majority circuit represented in a digital timing diagram

One time unit in the diagram 6.4 corresponds to one while loop iteration. This generated timing diagram fully corresponds to the majority truth table 6.1. It proves that the generated application works as expected in the simulated environment, like on the real hardware.

All generated files of the majority application are available in appendix F on page 89.

6.2 Regulation application

A second developed application has been selected and will be presented in this section. This time, this concrete application allows to regulate the speed of a PC fan.
The application is running on the real hardware. The connector of the extension board (see section 4.5) is used to connect the PC fan. Using the potentiometer, the user adjusts the desired speed of the fan (from stop to full speed). In addition, the real speed of the fan is measured (feedback), and is adjusted to correspond to the desired speed, using a PID regulator.

This application is not only a control application. The potentiometer controls the speed of the fan, but a feedback is used to measure its real speed. The command of the fan is more than just proportional to the potentiometer value. The block diagram of this regulation application is presented in figure 6.5 :

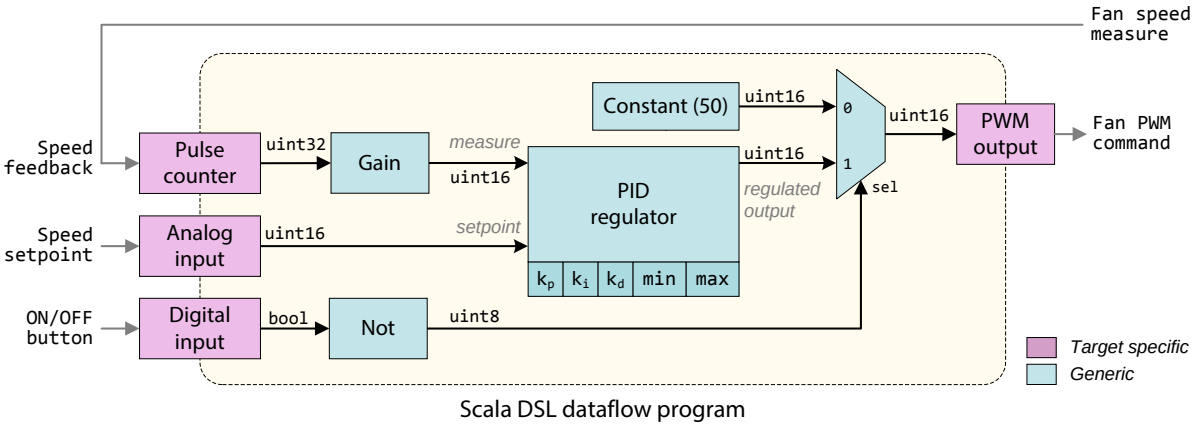


Figure 6.5 – Block diagram of the fan speed regulation application

The chosen PC fan [Swi15] (80x80mm) has a 4-pin socket. It is a 12VDC fan (nominal), but in this application it is connected to a 5V power supply (its maximum speed will be reduced). A PWM signal controls the speed of the fan. From a PWM duty cycle of 20 to 100%, its speed is approximately proportional to the duty cycle [Swi15]. The 4th pin of the fan is the speed measure of the fan. This pin is used as a feedback to control its real speed (input measure of the PID regulator).

The on/off button (`btn1` on the extension board) controls directly the PWM output signal. When it is pressed, a fixed PWM duty cycle ($\approx 2\%$) stops the fan. When released, it is controlled by the potentiometer. This logic is implemented using an inverter (not gate), a constant value and a multiplexer (see figure 6.5). The speed characteristic of the fan is available online [Swi15]. Note that the fan will run at full speed if the PWM duty cycle is 0% (to protect the hardware in a PC for instance).

The real speed of the fan is indicated by a rectangular pulse signal. Two pulses correspond to one turn. A target specific pulse counter component has been developed in the backend. The pulse signal of the fan is connected to an external interrupt line, so rising and falling edges can be automatically detected by the hardware to be counted. The speed fan is estimated using the technique presented in figure 6.6 :

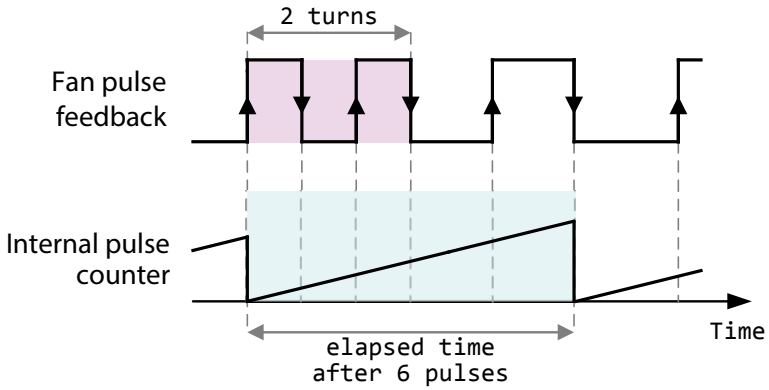


Figure 6.6 – Pulse feedback from the fan

When the fan turns at full speed, the time between two pulses is the shortest. When it is stopped, no pulse are generated. With a 5V power supply, the speed of the fan is not very high (≈ 900 RPM), this implies that the time between two pulses has a very little variation. To get a more accurate measure, the time between 6 edges is measured, like indicated in figure 6.6. This time, measured in millisecond, is directly proportional to the RPM of the fan.

The pulse counter result (an `uint32` value) is multiplied by a fixed gain and connected to the PID input measure. The gain of this block has been estimated after different measures of the fan pulse signal. The PID regulator can be tunned with different factors : only the k_p and k_i factors are used. These values were chosen empirically and tuned for the application purpose. These factors are not updated when the application runs. Finally, the minimum and maximum values of the regulator corresponds respectively to a duty cycle of 5% and 100% to control the whole speed range of the fan.

The Scala code of the application is presented below. The full code is available in appendix G.

```
val speedGain = SpeedGain(50)                       // Logic components
val mux = Mux2[uint16]()
val not = Not()

val pwm = Stm32stkIO.pwm3                           // Output

val pulse = PulseInputCounter(Pin('B', 9)).out      // Inputs
val measure = Stm32stkIO.adc1.out
Stm32stkIO.btn1.out --> not.in

val pid = PID(1.2, 0.9, 0, 205, 4095)               // PID factors
pulse --> speedGain.in                              // Pulse counter gain
speedGain.out --> pid.measure                       // PID measure
measure --> pid.setpoint                            // PID target speed (potentiometer)


Constant(uint16(50)).out --> mux.in1                // Stop the fan
pid.out --> mux.in2
not.out --> mux.sel

mux.out --> pwm.in                                  // Fan PWM command
```

Listing 6.4 – Fan speed regulation appplication

Components like the pulse counter, analog/digital inputs and the PWM output are all target specific components (see figure 6.5). The `Gain` block has been developed specially for this application. The code of this custom component (see section 5.4.6 on page 57) is presented here :

```
1  case class SpeedGain(gain: Int = 50) extends CFct[uint32, uint16]() {
2    private val outVal = outValName()          // Local variable name
3    override val description = "Custom gain"
4
5    override def loopCode = {                  // While loop code
6      val outType = getTypeString[uint16]
7      val in = getInputValue
8      s"$outType $outVal = 4096 - (($in - 110) * $gain); // Speed gain"
9    }
10   override def getOutputValue = outVal       // Compuation result stored in a variable
11 }
```

Listing 6.5 – Custom `SpeedGain` block

The speed gain is computed in the formula on line 9, using a simple C code. The result of the block is saved in a local variable to be used by the PID regulator. Using a custom component is the shortest way to compute these mathematical operations. Another solution consist of using available math blocks. More code is necessary to connect the components, but the result is the same, and no C/C++ code is necessary, which is great. The graph corresponding to the `SpeedGain` block, implemented with math components, has been generated and is shown in figure 6.7.

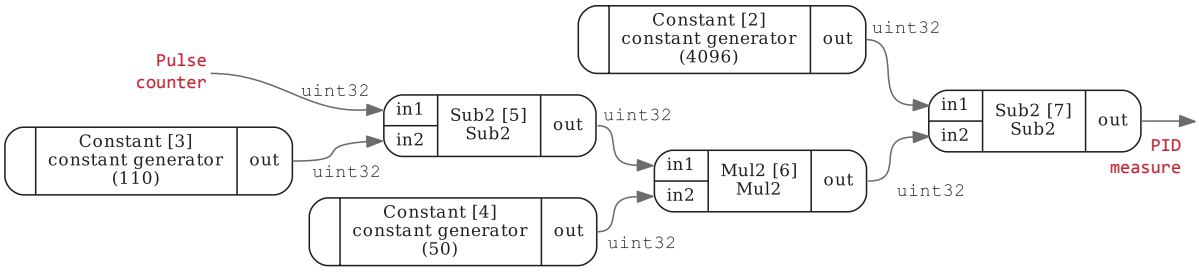


Figure 6.7 – Gain block using several math operations

To implement the `SpeedGain` block using math components, several lines of code are necessary. To write it with more ease and in a shorter way, implicit conversion for math blocks could be added. Thereby, it would be possible to automatically transform the equation on line 8 to math components.

This regulation application has been tested on the development board. The list of the used hardware is available in appendix B.4. PID constants and its gain have been fixed empirically, but actual values give satisfactory results. We can clearly see the speed regulation process, especially when the fan turns at low speed, or by making perturbations. The fan PWM command is also connected to a LED, which helps to see the signal command variations. The generated code of the application is available in appendix G on page 95.

This concludes the technical part of the report. These two real-world applications demonstrates the type and the complexity of programs that can be implemented using the developed framework.

Chapter 7

Conclusion

To conclude this report, developments and achieved results are summarized in the first part of this chapter. Then, recommendations for future work are briefly discussed.

7.1 Summary

The main objective of this project was to specify and implement a prototype language for embedded systems. This task was achieved by developing an internal dataflow DSL in Scala. The dataflow paradigm helps the user to describe the block diagram / the model of its application using an high-level description language. No specific or low-level code is required. The application is automatically converted from the Scala DSL to a C++ program (its concrete implementation). The developed back-end provides an abstraction for inputs, outputs and main peripherals of the target microcontroller.

To test the language and the generated code, two methods have been presented. The first one consists in compiling and running the generated application, without modifications, on the target embedded system. Using a standard toolchain, the set up environment and the developed HAL library, the user can run and test his application, like any other embedded software.
The second testing approach allows to simulate the generated code using an existing (modified) ARM emulator. This solution has a great advantage : the generated application can be tested automatically, using a Scala test case corresponding to the application. The modified ARM emulator allows to modify input pin values of the microcontroller and to monitor output values. At the end of the test, a digital timing diagram can be generated , to automatically check the behavior of the program, without using the real hardware. Testing the application require less time because the toolchain is simplified and only one tool is necessary. Furthermore, the test case is also written in Scala and the same compiled application can be simulated or executed on the target.

All applications presented in this document have been built using the developed dataflow DSL. They are all working as expected. All presented applications have been tested on the target development board, without modifying the generated code. Output codes are working "out of the box" on the real target or on the ARM emulator.

The last version of the source code is available online (see appendix B). Today, all opened issues have been fixed for both backend and frontend parts.

7.2 Limitations

The developed dataflow DSL is flexible and allows to build (complex) real-world applications. Some of them have been presented in the chapter 6. All applications that can be expressed in an directed acyclic dataflow graph can be converted to a sequential execution model.
This model can be too restrictive because some applications types cannot be built yet. If we look again at the regulation application (see section 6.2 on page 66), it would be great to build the regulator manually, using cycles and math components, instead of a ready-to-use component. To do this, the execution model must be upgraded to an event-driven asynchronous dataflow. A more complex scheduler must be implemented on the embedded system to execute the graph in a network of processes and no more in a single infinite loop.

The code simulation in QEMU was a challenging part and a very time consuming job because I had quite no previous experience with it. Thanks to the `qemu_stm32` project [Bec14], the chosen development kit and processor were already implemented in QEMU, but I had to extend the emulator to control it from the Scala side. Dealing with network, events and queues in a low-level C implementation was not easy (especially when debugging). Unfortunately, bugs are life. During a long period, I was not able to simulate correctly external interrupt lines (`EXTI`) in QEMU. This was due to a bug in the GPIO peripheral implementation. With the help of the project developer, it is fixed today.
Basic applications which use external interrupts and digital outputs can be simulated in the current emulator version. Scala test cases are great to automatically test the generated applications. Simulating embedded systems is not very common, but this approach give very good results. In my opinion, it would be interesting to do future work on the emulator, to add new MCU peripherals, like analog input values, PWM outputs, etc., even if the emulator is not a central part of the embedded language.

7.3 Future work

The project objectives are fulfilled. In the previous chapter, two applications have been presented. They demonstrate the type and the complexity of programs that can be implemented with this prototype language. In this section, some future work and nice to have features are presented.

As a proof of concept, only a few component are currently available in the framework. Thanks to the Scala internal DSL, the framework can be extended with ease, but severals components are missing. Components like SPI/I2C controllers (for sensors), LCD controllers or Ethernet/Bluetooth interfaces could be added. The developed extension board is minimalist, it could be miniaturized and extended.

Creating a visual editor was not an objective of the project, but it would be great to build one. It could really help non-programmer users to build applications, for instance in a web-based editor using drag & drop components, etc. Furthermore, it would be great to be able to define sub-components in a program. A component hierarchy could be used in the regulation application to build the gain block for instance. This helps to better decompose complex programs.

For this prototype project, only one target is supported, but the developed architecture has been designed to support new ones. Other ARM development boards can be added without much effort, like the ARMEBS4 board of HES-SO Valais [HES14].

Finally, some optimizations on the dataflow graph could be nice to have. Unconnected nodes and path are automatically removed before generating the application code. Optimizations like constant propagation or boolean circuits minimizations could be added. They are minor improvements because the compiler already do these optimizations.

In conclusion, this project allows to describe applications using a high-level language. Several components are ready-to-use. The model / the block diagram of the application is written using a custom dataflow DSL and the generated code can be used out of the box (without modification) to program the embedded target. This work allows non-programmer users to build portable applications without using low-level code.

Through this project, I had the opportunity to use a lot of tools, libraries and programming languages. It was interesting to deal with low-level C programming (in QEMU for instance) and also to write high-level functional programming using the Scala programming language. Developments were various and challenging. Thanks to this project, I have improved my Scala programming skills and learn a lot about the QEMU emulator. Furthermore, I had the opportunity to contribute to open-source libraries used during the project, like *Graph for Scala* and *QEMU for Stm32*.

Lausanne, 6th February 2015

Christopher Métrailler

Bibliography

[Bag09] Phil Bagwell. DSLs - A powerful Scala feature. http://www.scala-lang.org/old/node/1403, 2009.

[Bec14] Andre Beckus. QEMU with STM32 Microcontroller Implementation. http://beckus.github.io/qemu_stm32/, version 2.1.1, 2014.

[Bel14] Fabrice Bellard. QEMU - open source processor emulator. http://qemu.org/, 2014.

[Ber11] Alexander Bernauer. Internal DSLs in Scala. http://blogs.ethz.ch/copton/2011/04/08/internal-dsls-in-scala/, 2011.

[Emp15a] Peter Empen. Core User Guide of the `Graph for Scala` library. http://www.scala-graph.org/guides/core-introduction.html, 2015.

[Emp15b] Peter Empen. `Graph for Scala` library. https://github.com/scala-graph/scala-graph/, `graph-core` version 1.9.1 and `graph-dot` version 1.10.0, 2015.

[FE05] Martin Fowler and Eric Evans. Fluent Interface. http://martinfowler.com/bliki/FluentInterface.html, 2005.

[Fin95] Raphael A. Finkel. *Advanced Programming Language Design, Chapter 6 - Dataflow, pp. 169-184.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GF13] Anaïs Guignard and Jean-Marc Faure. Formal models for conformance test of programmable logic controllers. *Journal Européen des Systèmes Automatisés*, 47(4-8):423–446, 2013.

[Gho10] Debasish Ghosh. *DSLs in Action.* Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010.

[Goo14] Google. Blockly is a library for building visual programming editors. https://developers.google.com/blockly/about/faq, December 2014.

[Gra14] Graphviz. Graph Visualization Software. http://www.graphviz.org/, version 2.38.0, 2014.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. In *Proceedings of the IEEE*, pages 1305–1320, 2002 (first version published in 1991).

[HES14] HES-SO Valais//Wallis, Infotronics Unit. `ARMEBS4` board and `libheivs_stm32` library. Contains tools and setup instructions for ARM Cortex-M development. http://armebs4.hevs.ch and http://wiki.hevs.ch/uit/index.php5/EclipseArmebs4/DemoProjects#libheivs_stm32, December 2014.

[Hos14] Eric Hosick. Visual Programming Languages - Snapshots. http://blog.interfacevision.com/design/design-visual-progarmming-languages-snapshots/, 20 February 2014.

[IEE01] IEEE. IEEE standard Verilog hardware description language. *IEEE Std. 1364-2001*, pages 348–339, 2001.

[Ins15] National Instruments. Graphical Programming in LabVIEW. http://www.ni.com/getting-started/labview-basics/dataflow, 2015.

[JHM04] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, March 2004.

[KMR91] J. Kodosky, J. MacCrisken, and G. Rymar. Visual programming using structured data flow. In *Proc. of the 1991 IEEE Workshop on Visual Languages*, pages 34–39, Kobe, Japan, 1991.

[Mai13] Tomi Maila. Dataflow and Flow-Based Programming – Two Approaches To Visual Programming. http://expressionflow.com/2013/11/17/dataflow-and-flow-based-programming-two-approaches-to-visual-programming/, 17 November 2013.

[MH14] Jens Mönig and Brian Harvey. Snap! - Build Your Own Blocks (BYOB). http://byob.berkeley.edu/, 2014. University of California at Berkeley.

[MIT14] MIT Media Lab. Scratch - Imagine, Program, Share. http://scratch.mit.edu/ and http://wiki.scratch.mit.edu, 2014.

[Mod15] Modkit LLC. Program Real Things with Modkit. http://www.modkit.com/, 2015.

[Mor13] J. Paul Morrison. Flow-Based Programming, Journal of Application Developers' News. http://ersaconf.org/ersa-adn/Paul-Morrison.php, 2013.

[Oa04] Martin Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[Ode14] Martin Odersky. The Scala Language Specification. http://www.scala-lang.org/docu/files/ScalaReference.pdf, version 2.9, 11 June 2014.

[Oli14] Olimex Ltd. `STM32-103STK` starter-kit board. https://www.olimex.com/Products/ARM/ST/STM32-103STK/, 2014.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 1st edition, 2008.

[Par13] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[Pat14] Jim Pattee. Artistic Style, A Free, Fast, and Small Automatic Formatter. http://astyle.sourceforge.net/, version 2.04, 2014.

[Qua04] Gang Quan. "Data Flow Graph Intro". http://web.cecs.pdx.edu/~mperkows/temp/JULY/data-flow-graph.pdf, 2004.

[STM14a] STMicroelectronics. Mainstream Performance line, `ARM Cortex-M3 MCU` with 128 Kbytes Flash, 72 MHz CPU. http://www.st.com/web/catalog/mmc/FM141/SC1169/SS1031/LN1565/PF164487, 2014.

[STM14b] STMicroelectronics. Reference manual (RM0008): STM32F101xx, STM32F102xx, STM32F103xx, STM32F105xx and STM32F107xx advanced ARM-based 32-bit MCUs, 2014. DocID 13902 (June 2014), rev 15.

[STM14c] STMicroelectronics. STM32F10x standard peripheral library. http://www.st.com/web/en/catalog/tools/PF257890, version 3.5.0, 2014.

[Sug14] Sugarlabs. Turtle Art. http://wiki.sugarlabs.org/go/Activities/Turtle_Art, May 2014.

[Swi15] Arctic Switzerland. 4-Pin PWM fan with standard case (`F8 PWM PST`) - performance and electrical characteristics. http://www.arctic.ac/ch_en/arctic-f8-pwm-pst.html, 2015.

[Tan14] Till Tantau. PGF and TikZ - Graphic systems for TeX. http://sourceforge.net/projects/pgf/, 2014.

[Tho83] Thornburg, David D. . Friends of the Turtle: On Logo And Turtles (Compute! p. 148). http://www.atarimagazines.com/compute/issue34/068_1_FRIENDS_OF_THE_TURTLE.php, 1983. retrieved in 2013.

[TMPL95] José Luis Pino Thomas M. Parks and Edward A. Lee. A Comparison of Synchronous and Cyclo-Static Dataflow. *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, University of California, 1995.

[Typ14a] Typesafe. Build reactive applications with scala. Technical report, Typesafe, 2014.

[Typ14b] Typesafe. Simple-Build-Tool (sbt) for Scala. http://www.scala-sbt.org/index.html, version 0.13.7, 2014.

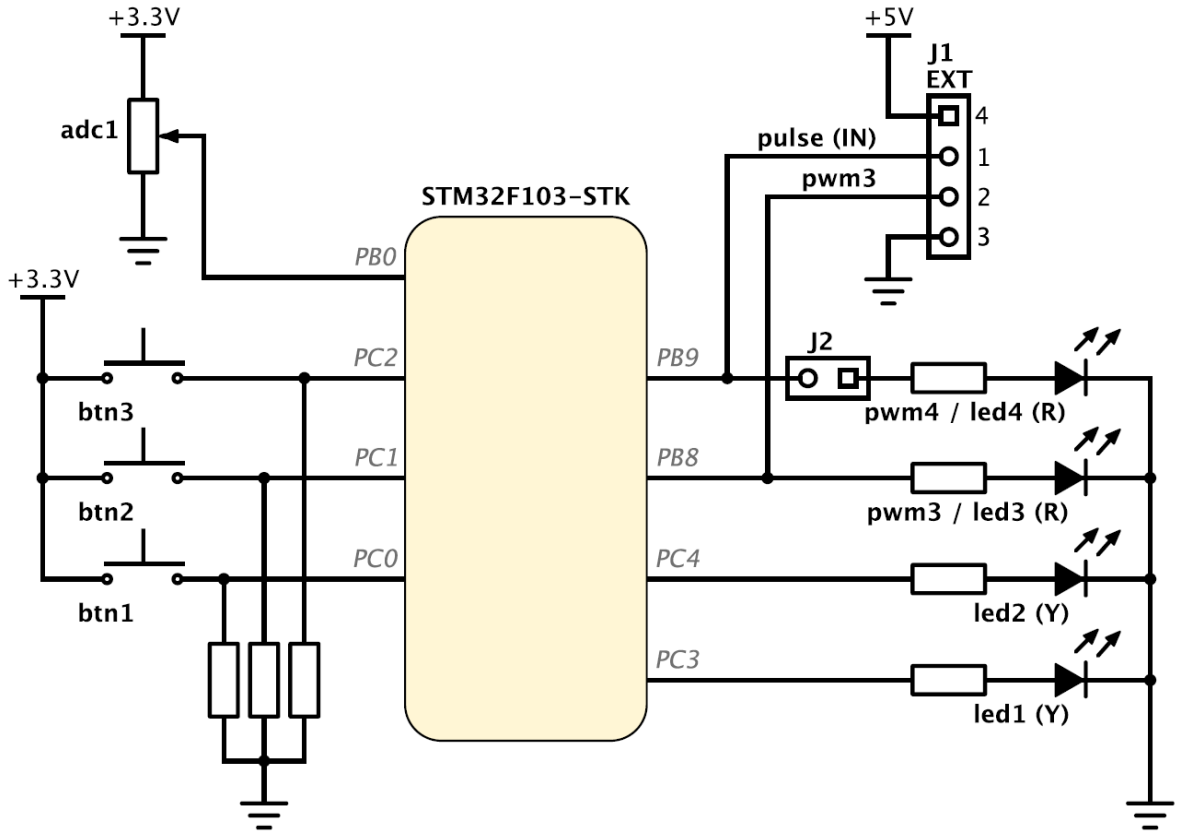
[VBCG04] Girish Venkataramani, Mihai Budiu, Tiberiu Chelcea, and Seth Copen Goldstein. C to Asynchronous Dataflow Circuits: An End-to-End Toolflow. In *IEEE 13th International Workshop on Logic Synthesis (IWLS)*, Temecula, CA, June 2004.

[VM99] Brian K. Vogel and Andrew Mihal. LabVIEW: Visual Programming Using a Dataflow Model Extended With Graphical Control Structures. EECS Berkeley, http://embedded.eecs.berkeley.edu/research/hsc/class.F99/ee249/discussions/vogel-mihal.2.pdf, 1999.

[WP94] Paul G. Whiting and Robert S.V. Pascoe. A history of data-flow languages. *IEEE Annals of the History of Computing*, 16(4):38–59, 1994.

All links were valid on February 2th, 2015.

A | Extension board



Digital input name	Description	Pin	Board
`btn3`	Button 3	`PC02`	Extension board
`btn2`	Button 2	`PC01`	Extension board
`btn1`	Button 1	`PC00`	Extension board
`btn0`	Joystick center	`PC06`	Main board

Analog input name	Description	Pin	Board
`adc1`	Potentiometer	`PB00 / CH08`	Extension board
`adc0`	Joystick directions	`PC05 / CH15`	Main board

Digital output name	Description	Pin	Board
`led4 / pwm4`	Led 4 Red or PWM 4	`PB09 / TIM4_CH4`	Extension board
`led3 / pwm3`	Led 3 Red or PWM 3	`PB08 / TIM4_CH3`	Extension board
`led2`	Led 2 Yellow	`PC04`	Extension board
`led1`	Led 1 Yellow	`PC03`	Extension board
`led0`	Led 0 Red (active low)	`PC12`	Main board

B | Development environment

A CD is attached to this report. It contains the full source code of the project, as well as a copy of the report. The project source code is also available online in the two following private *Git* repositories. The access to these repositories is available upon request.

- Frontend Scala code
  https://github.com/metc/especial-frontend
- Backend C/C++ code : QEMU for STM32 and HAL for the `STM32-103STK` board
  https://github.com/metc/especial-backend

B.1 Backend (C/C++)

The following tools are required to build the backend. The code compilation has only been tested on Linux (`Ubuntu 14.04.1 LTS`) :

Tool name	**Version**	**Website**
Eclipse IDE for C/C++	Luna 4.4.1	https://eclipse.org/cdt/downloads.php
Eclipse cross compiler plugin	1.11.1	http://sourceforge.net/projects/gnuarmeclipse/
GNU ARM toolchain `gcc-arm-none-eabi`	4_8-2014q3	https://launchpad.net/gcc-arm-embedded
GNU Make (Linux)	3.81	http://www.gnu.org/software/make/
Open On-Chip Debugger	0.7.0	http://openocd.sourceforge.net/
Impulse VCD viewer	1.0.2	http://toem.de/index.php/projects/impulse
Doxygen	1.8.9	http://www.stack.nl/~dimitri/doxygen/manual/

Table B.1 – Backend tools and versions

Once these tools are installed, the backend project can be imported in Eclipse directly. Run and debug configurations are available in the project directory, with several demo applications.

B.2 Frontend (Scala)

The Scala development environment has been tested on Linux, using the following tools and versions :

Tool name	Version	Online installation guide
Java Development Kit (JDK)	1.8.0_25	http://www.oracle.com/technetwork/java/
Scala Programming Language	2.11.4	http://www.scala-lang.org/download/install.html
Simple-Build-Tool (sbt)	0.13.7	http://www.scala-sbt.org/0.13/tutorial/Setup.html
IntelliJ IDEA	14.0.2	https://www.jetbrains.com/idea/download/
IntelliJ Scala plugin	1.2	https://plugins.jetbrains.com/plugin/?id=1347
Graph Visualization Software	2.38.0	http://www.graphviz.org/Download.php
Artistic Style	2.0.4	http://astyle.sourceforge.net/install.html

Table B.2 – Frontend tools and versions

Frontend settings can be modified using the `Settings.scala` file. More information about library dependences are available in the `third_party` folder. The sbt project definition file is available in the listing B.1. All used Scala libraries are detailed in this file :

```
name := "especial"
version := "1.0"

scalaVersion := "2.11.4"

// Graph for Scala - http://www.scala-graph.org/
libraryDependencies ++= Seq(
  "com.assembla.scala-incubator" %% "graph-core" % "1.9.1",
  "com.assembla.scala-incubator" %% "graph-dot" % "1.10.0",
  "com.assembla.scala-incubator" %% "graph-constrained" % "1.9.0"
)

// Grizzled-SLF4J, a Scala-friendly SLF4J Wrapper
libraryDependencies ++= Seq(
  "org.slf4j" % "slf4j-api" % "1.7.5",
  "org.slf4j" % "slf4j-simple" % "1.7.5",
  "org.clapper" %% "grizzled-slf4j" % "1.0.2"
)

// Lift-json - http://liftweb.net/download
libraryDependencies += "net.liftweb" %% "lift-json" % "2.6-M4"

// Scala tests - http://www.scalatest.org/
libraryDependencies += "org.scalatest" % "scalatest_2.11" % "2.2.1" % "test"

// Scala compiler options
scalacOptions ++= Seq("-unchecked", "-deprecation", "-feature")


```

```
30 // Disable parallel execution of tests (shared instance of the ComponentManager)
31 parallelExecution in ThisBuild := false
32
33 // Remove some tests which must be ran manually, one after one.
34 testOptions in Test := Seq(Tests.Filter(s => !(s.contains("generator."))))
35
36 // Generate ScalaDoc diagrams using dot
37 scalacOptions in(Compile, doc) ++= Seq("-diagrams")
38
39 // Custom clean task
40 // Delete all generated files from the output directory
41 clean ~= {x => println("Remove generated output files...")}
42 cleanFiles <++= baseDirectory { base => (base / "output/" * "*").get }
```

Listing B.1 – Project build file definition (`build.sbt`)

The sbt Scala project can be opened into IntelliJ using the sbt import wizard. The `sbteclipse` plugin can be used to convert the project to an Eclipse project. Finally, to compile and launch the project unit tests, simply run "`sbt compile`" and "`sbt test`" in a console, from the project root directory.

B.3 ARM emulator (QEMU)

The `STM32f103-STK` board can be simulated in a specific QEMU version with an STM32 microcontroller implementation [Bec14]. This project is based on the official QEMU version `2.2.1`.

The `stm32` repository contains the modified QEMU version for this project, with the TCP/IP emulation to communicates with the Scala frontend. The installation and compilation procedure is available online : https://github.com/beckus/qemu_stm32/blob/stm32/README.

B.4 Hardware

The list of the hardware used for the project and real world applications :

Hardware name	**Reference**
Starterkit board for `STM32F103RBT6` Cortex-M3 microcontroller `STM32-103STK` rev. B	https://www.olimex.com/Products/ARM/ST/STM32-103STK/
HES-SO Valais//Wallis Universal Tracer (JTAG & UART)	http://wiki.hevs.ch/uit/index.php5/Inventory/CPU/DevKits#Universal_Tracer_STM32F103_Dev_Kit
4-Pin PWM fan with standard case `F8 PWM PST`	http://www.arctic.ac/ch_en/arctic-f8-pwm-pst.html

Table B.3 – Hardware references

C | Hardware Abstraction Layer (HAL) test application

```
/**
 * Test application for the I/O extension board. Use the developped HAL.
 *  - led1 is ON when running
 *  - led2 is controlled by btn2
 *  - led3 is controlled by the potentiometer (PWM)
 *  - led4 toggles each 0.5 seconds (use Timer2)
 *
 * \author  Christopher Metrailler
 * \version 1.0
 */
#include "digitaloutput.h"
#include "digitalinput.h"
#include "analoginput.h"
#include "pwmoutput.h"
#include "utils/time.h"
#include "helper.h"

AnalogInput adc1('B', 0, 8); // Potentiometer
DigitalInput btn2('C', 1);   // btn2

PwmOutput pwm3('B', 8);      // led3
DigitalOutput led4('B', 9);  // led4
DigitalOutput led2('C', 4);  // led2
DigitalOutput led1('C', 3);  // led1

void initIO() {
    btn2.initialize();       // Inputs
    adc1.initialize();

    pwm3.initialize();       // Outputs
    led4.initialize();
    led2.initialize();
    led1.initialize();
}

int main() {
    timeout_t time;
    time_init();

    initIO();  // Init all I/O and set to '0'

    println("Sample application started.");

    led1 = true; // Led1 is on. Same as 'led1.set(true);' or 'led1.setState(On);'
    time = time_get();

```

```
47      while (1) {
48          // Led2 is controlled by btn2
49          // bool val = btn1.get();
50          // led2.set(val);
51          led2 = btn2.get(); // Use EXT interrupt line
52
53          // Potentiometer value on led3 (PWM signal)
54          // uint16_t period = adc1.read();
55          // pwm3.setPeriod(period);
56          pwm3 = adc1.read(); // Start A/D conversion
57
58          // Led4 toggles each 0.5 seconds
59          if(time_diff_ms(time_get(), time) > 500) {
60              time = time_get();
61              led4.toggle(); // Toggle the led4
62          }
63      }
64      return 0;
65  }
```

Listing C.1 – Test application of the HAL

D | Sample code generation

Application block diagram :

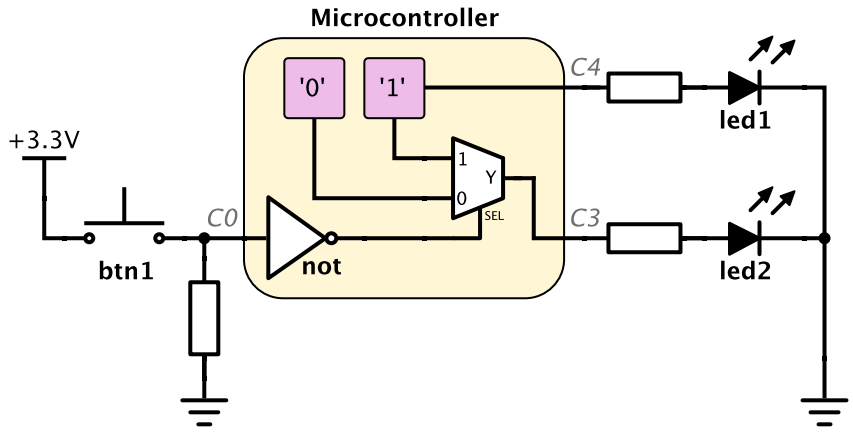


Figure D.1 – Sample application

Application implementation in Scala, using the dataflow DSL :

```
1  package hevs.especial.doc
2
3  import hevs.especial.dsl.components._
4  import hevs.especial.dsl.components.core.{CFct, Constant, Mux2}
5  import hevs.especial.dsl.components.target.stm32stk.Stm32stkIO
6  import hevs.especial.generator.STM32TestSuite
7
8  /**
9   * Application to demonstrate how the resolver and the code genertor block work.
10  *
11  * Read a button, invert its state and convert it value to an [[uint8]] to control the
12  * selection pin of a [[Mux2]]. The ‘led1‘ is always ON. When the ‘btn1‘ is pressed,
13  * the ‘led2‘ switch OFF. Application without warnings.
14  *
15  * @version 1.0
16  * @author Christopher Metrailler (mei@hevs.ch)
17  */
18 class DemoResolver extends STM32TestSuite {
19
20   def isQemuLoggerEnabled = false
21
22   import hevs.especial.dsl.components.CType.Implicits._
23
24   def runDslCode(): Unit = {
25     val not = Not()                          // Not and ‘uint8‘ conversion
26     Stm32stkIO.btn1.out --> not.in           // Read input
27
28
```

```
29
30      val mux = Mux2[bool]()
31      val cst1 = Constant[bool](true).out
32      mux.out --> Stm32stkIO.led1.in          // Update outputs LEDs
33      cst1 --> Stm32stkIO.led2.in
34
35      not.out --> mux.sel
36      Constant[bool](false).out --> mux.in2
37      cst1 --> mux.in1
38    }
39
40    case class Not() extends CFct[bool, uint8] {
41      override val description = s"NOT gate"
42      private val convValue = outValName()
43
44      /* I/O management */
45      override def getOutputValue: String = convValue
46
47      /* Code generation */
48      override def loopCode = s"${uint8().getType} $convValue = if($getInputValue) ? 0 : 1;"
49    }
50
51    runDotGeneratorTest()
52    runCodeCheckerTest()
53    runCodeOptimizer()
54    runDotGeneratorTest(optimizedVersion = true)
55
56    runCodeGenTest()
57  }
```

Listing D.1 – Scala generation test for the application 5.11

```
1  package hevs.especial.dsl.components.core.logic
2
3  import hevs.especial.dsl.components._
4  import scala.reflect.runtime.universe._
5
6  /**
7   * Invert any input type.
8   *
9   * The output value as the same type as the input. For [[bool]] values, the output is simply
10  * inverted. For all other types, in the input "0", it is set to "1" and vice-versa.
11  *
12  * @version 2.0
13  * @author Christopher Metrailler (mei@hevs.ch)
14  *
15  * @tparam T the type of the value to invert
16  */
17 case class Not[T <: CType : TypeTag]() extends Component
18 with In1 with Out1 with HwImplemented {
19
20   override val description = s"NOT gate"
21
22   /* I/O management */
23   val in = new InputPort[T](this) {
```

```
    override val name = s"in"
    override val description = "input to invert"
  }

  val out = new OutputPort[T](this) {
    override val name = s"out"
    override val description = "inverted value"
    override def getValue: String = outValName() // inverted value stored in a local variable
  }

  override def getOutputs = Some(Seq(out))
  override def getInputs = Some(Seq(in))

  /* Code generation */

  override def getLoopableCode = {
    // Read the input
    val inValue = ComponentManager.findPredecessorOutputPort(in).getValue

    // Invert the value and store the result in a variable
    val sInvert = getTypeClass[T] match {
      case _: `Class`[bool] => s"!$inValue"
      case _ => s"""($inValue == 0) ? 1 : 0"""
    }
    Some(s"${getTypeString[T]} ${outValName()} = $sInvert;")
  }
}

object Not {
  def apply[T <: CType : TypeTag](input: OutputPort[T]): Not[T] = {
    // Connect automatically the single input
    val n = Not[T]()
    input --> n.in
    n
  }
}
```

Listing D.2 – Not (inverter) component implementation

Generated C++ application code :

```
// DemoResolver application
// Code generated automatically from the Scala DSL program 'DemoResolver'.

//*// Section 00
#include "digitalinput.h"
#include "digitaloutput.h"
//*// ----------

//*// Section 01
DigitalInput in_cmp02('C', 0);  // OutputPort[0] 'out' of Cmp[02] 'DigitalInput'
DigitalOutput out_cmp05('C', 3); // InputPort[0] 'in' of Cmp[05] 'DigitalOutput'
DigitalOutput out_cmp06('C', 4); // InputPort[0] 'in' of Cmp[06] 'DigitalOutput'
//*// ----------

```

```
//*// Section 02
//*// ----------

//*// Section 03
void initOutputs() {
    out_cmp05.initialize();
    out_cmp06.initialize();
}

void init() {
    in_cmp02.initialize();
}
//*// ----------

int main() {
    initOutputs();
    init();

    //*// Section 04
    //*// ----------

    //*// Section 05
    while(1) {
        // 1) Read inputs
        bool in_C0 = in_cmp02.get();

        // 2) Loop logic
        uint8_t out_cmp01 = !in_C0;

        uint8_t sel_cmp03 = out_cmp01;
        bool out_cmp03;

        if(sel_cmp03 == 0)
            out_cmp03 = true;
        else
            out_cmp03 = false;

        // 3) Update outputs
        out_cmp05.set(out_cmp03);
        out_cmp06.set(true);
    }
    //*// ----------

    //*// Section 06
    //*// ----------
}
```

Listing D.3 – C++ code generated from the Scala DSL program D.1

E | Generated VCD file

```
$date      Tue Jan 20 18:00:32 CET 2015     $end
$version   ESPecIaL version B2.0            $end
$comment
   VCD file generated automatically for 'VcdGeneratorTest'.
$end

$timescale 1 ms $end

$scope module VcdGeneratorTest $end
$var wire 1 A1 pin_A1 $end
$var wire 1 B2 pin_B2 $end
$var wire 1 C3 pin_C3 $end
$var wire 1 D4 pin_D4 $end
$upscope $end

$enddefinitions $end

$dumpvars
xA1
xB2
xC3
xD4
$end

#0
0A1
1B2
0C3
0D4

#1
1A1
0B2
1C3
1D4

#2
0A1
1B2
1C3
0D4

#3
1A1
0B2
0C3
```

Listing E.1 – Generated VCD file for the test case 5.12 (`VcdGeneratorTest.vcd`)

F | Majority circuit

Application code of the three-input majority circuit :

```
val A = Stm32stkIO.btn1.out      // Input buttons
val B = Stm32stkIO.btn2.out
val C = Stm32stkIO.btn3.out
val O = Stm32stkIO.led1.in       // Output LED

(A & B | B & C | A & C) --> O    // Majority function
```

Listing F.1 – Three-input majority circuit

The generated dataflow graph of the application F.1 :

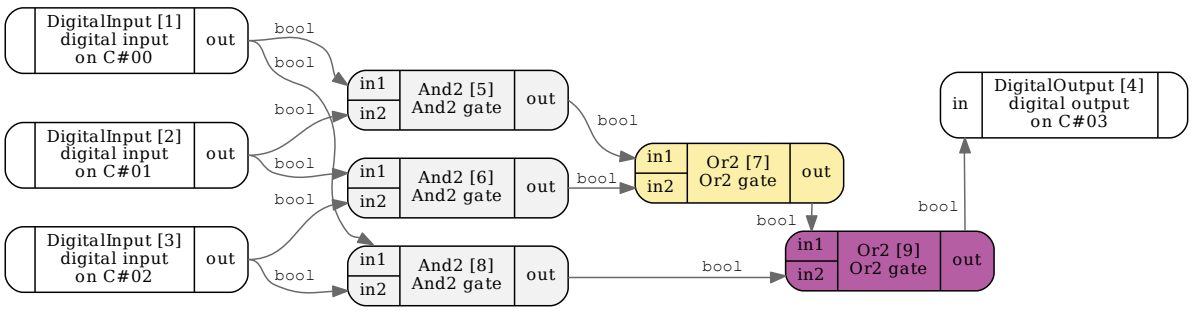


Figure F.1 – Generated graph for the majority circuit

Generated code of the application F.1. Runs in QEMU :

```
// Majority application.
// Code generated automatically from the DSL program 'Majority'.

//*// Section 00
#include "digitalinput.h"
#include "digitaloutput.h"
//*// ----------

//*// Section 01
DigitalInput in_cmp03('C', 2);  // OutputPort[0] 'out' of Cmp[03] 'DigitalInput'
DigitalInput in_cmp01('C', 0);  // OutputPort[0] 'out' of Cmp[01] 'DigitalInput'
DigitalInput in_cmp02('C', 1);  // OutputPort[0] 'out' of Cmp[02] 'DigitalInput'
DigitalOutput out_cmp04('C', 3); // InputPort[0] 'in' of Cmp[04] 'DigitalOutput'
//*// ----------

//*// Section 02
//*// ----------

//*// Section 03
void initOutputs() {
    out_cmp04.initialize();
```

```
}

void init() {
    in_cmp03.initialize();
    in_cmp01.initialize();
    in_cmp02.initialize();
}
//*// ----------

int main() {
    QemuLogger::send_event(SECTION_START);

    initOutputs();
    init();

    QemuLogger::send_event(SECTION_INIT_END);

    //*// Section 04
    //*// ----------

    QemuLogger::send_event(SECTION_LOOP_START);

    //*// Section 05
    while(1) {

        QemuLogger::send_event(SECTION_LOOP_TICK, true); // Ack event is needed

        // 1) Read inputs
        bool in_C2 = in_cmp03.get();
        bool in_C0 = in_cmp01.get();
        bool in_C1 = in_cmp02.get();

        // 2) Loop logic
        bool out_cmp05 = in_C0 & in_C1;
        bool out_cmp06 = in_C1 & in_C2;
        bool out_cmp08 = in_C0 & in_C2;
        bool out_cmp07 = out_cmp05 | out_cmp06;
        bool out_cmp09 = out_cmp07 | out_cmp08;

        // 3) Update outputs
        out_cmp04.set(out_cmp09);
    }
    //*// ----------

    //*// Section 06
    //*// ----------

    QemuLogger::send_event(SECTION_END);
}
// END of file 'Majority.cpp'
```

Listing F.2 – Generated code of the application F.1

F.1 Simulation

Scala test case of the three-input majority circuit :

```
1  package hevs.especial.apps
2
3  import hevs.especial.dsl.components.Pin
4  import hevs.especial.dsl.components.target.stm32stk.Stm32stkIO
5  import hevs.especial.generator.STM32TestSuite
6  import hevs.especial.simulation.{MonitorWriter, Events, Monitor, MonitorTest}
7  import hevs.especial.utils.OSUtils
8
9  /**
10  * Test case used to simulate the [[hevs.especial.apps.Majority]] application.
11  *
12  * QEMU events are used to monitor the MCU code execution in QEMU. This Scala test case
13  * controls the execution of the code running in QEMU. When a loop iteration is terminated,
14  * input values are modified and the execution restart for a new loop iteration with new
15  *  input values. After a number of iterations, the QEMU program is stopped and the VCD
16  * result file is generated.
17  *
18  * @version 1.0
19  * @author Christopher Metrailler (mei@hevs.ch)
20  */
21 class MajorityInputSimulation extends MonitorTest {
22
23   /** Input buttons values (all possible combinations, see truth table). */
24   private val inValues = Map(
25     3 -> "0,1,0,1,0,1,0,1", // Input C (btn3)
26     2 -> "0,0,1,1,0,0,1,1", // Input B (btn2)
27     1 -> "0,0,0,0,1,1,1,1" // Input A (btn1)
28   )
29
30   /** Expected output values. */
31   private val outValues = Array(
32     1 -> "0,0,0,1,0,1,1,1" // Output 0 (led1) = majority function
33   )
34
35   private def updateInputValues(w: MonitorWriter, tick: Int): Unit = {
36     inValues.foreach(input => {
37       val btId = input._1
38       val curVal = input._2.split(",")(tick)
39       info(s"Set input '$btId' to '$curVal'.")
40       w.setButtonInputValue(btId, curVal == "1") // Send the input boolean value
41     })
42   }
43
44   /**
45    * Simulate a test program in QEMU.
46    * @param code the test code to simulate
47    */
48   override def simulateCode(code: STM32TestSuite) = {
49     val m = new Monitor()
50
51     info("Start QEMU in a new process...\n\n")
```

```
52      val qemuProcess = OSUtils.runInBackground(runQemu)
53      Thread.sleep(1000)
54      info(" > QEMU started.")
55
56      m.waitForClient() match {
57        case false => fail("Timeout. Test aborted.")
58        case _ =>
59      }
60
61      // *********************************
62      var countTick = 0
63      while (countTick < 8) {
64        m.reader.waitForEvent(Events.LoopTick)
65
66        // Set input value
67        updateInputValues(m.writer, countTick)
68
69        m.writer.ackEvent(Events.LoopTick)
70        info(s"> Loop iteration $countTick ended.")
71        countTick += 1
72      }
73
74      // Wait for the last loop iteration
75      m.reader.waitForEvent(Events.LoopTick)
76      info(s"> Program terminated after $countTick iterations.")
77
78      // Print result values
79      pins = m.reader.getOutputValues
80      printOutputValues(pins)
81
82      // Check output values
83      val expectedValues = outValues.head._2.split(",").map(x => x.toInt)
84      val realValues = pins.get(Stm32stkIO.led4_pin).get
85      val success = expectedValues sameElements realValues
86      if (!success)
87        fail("Error in the majority function !")
88      // *********************************
89
90      // Close QEMU
91      info("Kill QEMU.")
92      disconnect(m)
93      qemuProcess.destroy()
94    }
95
96
97    /** DSL program under test. */
98    runTests(new Majority())
99  }
```

Listing F.3 – Scala test case of the three-input majority circuit (`MajorityInputSimulation.scala`)

Result of the test case F.3 in a digital timing diagram :

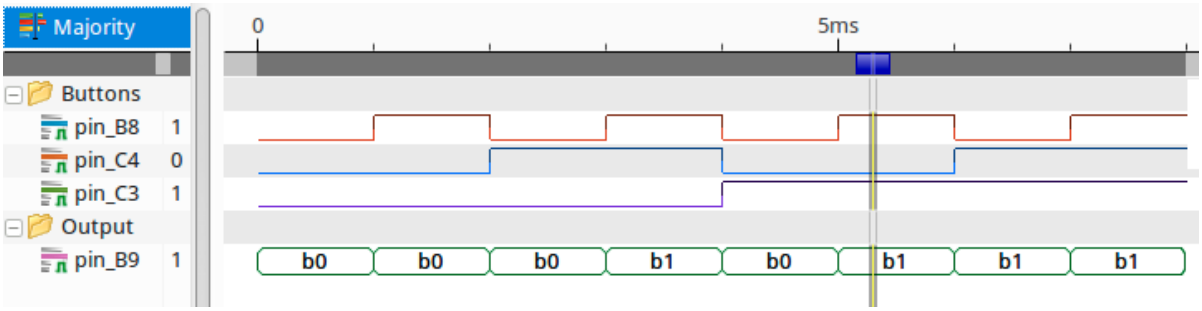


Figure F.2 – Exhaustive test of the majority circuit represented in a digital timing diagram (Impulse)

The generated VCD file of the digital timing diagram F.2 :

```
$date
   Wed Jan 21 12:48:09 CET 2015
$end
$version
   ESPecIaL version B3.0
$end
$comment
   VCD file generated automatically for 'Majority'.
$end

$timescale 1 ms $end

$scope module Majority $end
$var wire 1 C3 pin_C3 $end
$var wire 1 B9 pin_B9 $end
$var wire 1 C4 pin_C4 $end
$var wire 1 B8 pin_B8 $end
$upscope $end

$enddefinitions $end

$dumpvars
xC3
xB9
xC4
xB8
$end

#0
0C3
0B9
0C4
0B8

#1
0C3
0B9
0C4
```

```
1B8

#2
0C3
0B9
1C4
0B8

#3
0C3
1B9
1C4
1B8

#4
1C3
0B9
0C4
0B8

#5
1C3
1B9
0C4
1B8

#6
1C3
1B9
1C4
0B8

#7
1C3
1B9
1C4
1B8
```

Listing F.4 – Generated VCD file of the tree-input majority circuit

G | Regulation application

Code of the fan PID regulation application :

```
package hevs.especial.apps

import hevs.especial.dsl.components._
import hevs.especial.dsl.components.core.math.PID
import hevs.especial.dsl.components.core.{CFct, Constant, Mux2}
import hevs.especial.dsl.components.target.stm32stk.{PulseInputCounter, Stm32stkIO}
import hevs.especial.generator.STM32TestSuite

/**
 * Complete demo application to regulate a fan speed using a PPID regulator.
 *
 * The speed of the fan is measure using a pulse counter. Pulses are captured and counted
 * using external interrupts. When the button 1 is pressed, the fan is off. By default, the fan
 * speed is regulated by a PID controller. The seed setpoint can be adjusted using the
 * potentiometer, from 0 to 100% speed.
 * A custom math block is used to adapt the number of counted pulses to the desired fan speed.
 * PID kp, ki and kd constants are fixed when the program starts.
 *
 * To run this demo, the fan must be connected correctly
 * and the jumper must connect the fan output (not the led).
 *
 * @version 1.0
 * @author Christopher Metrailler (mei@hevs.ch)
 */
class FanPid extends STM32TestSuite {

  def isQemuLoggerEnabled = false

  def runDslCode(): Unit = {
    // Inputs
    val pid = PID(1, 1, 0, 255, 4095)
    val pulse = PulseInputCounter(Pin('B', 9)).out
    val measure = Stm32stkIO.adc1.out

    // Logic
    val speedGain = SpeedGain(50)
    val mux = Mux2[uint16]()
    val not = Not()

    // Output
    val pwm = Stm32stkIO.pwm3

    // PID input measure from the pulse counter
    pulse --> speedGain.in
    speedGain.out --> pid.measure

    // PID input setpoint from the potentiometer
```

```
    measure --> pid.setpoint

    // Mux logic to stop the fan using the button 1
    Constant(uint16(50)).out --> mux.in1
    pid.out --> mux.in2
    Stm32stkIO.btn1.out --> not.in
    not.out --> mux.sel

    // Fan PWM command
    mux.out --> pwm.in
  }

  /**
   * Math block used to adapt the speed of the fan.
   * @param gain speed gain from measure
   */
  case class SpeedGain(gain: Int) extends CFct[uint32, int32]() {
    override val description = "Custom gain"
    private val outVal = outValName()
    override def getOutputValue = outVal

    /* Code generation */
    override def loopCode = {
      val outType = getTypeString[int32]
      val in = getInputValue
      s"""|$outType $outVal = 4096 - (($in - 110) * $gain); // Speed gain
          |if ($outVal <= 0)
          |  $outVal = 0;""".stripMargin
    }
  }

  /** Custom C component to invert a [[bool]] value and return an [[uint8]] value. */
  case class Not() extends CFct[bool, uint8]() {
    override val description = "Inverter to uint8"
    private val outVal = outValName()
    override def getOutputValue = outVal

    /* Code generation */
    override def loopCode = {
      val outType = getTypeString[uint8]
      val in = getInputValue
      s"$outType $outVal = ($in == 0) ? 1 : 0;"
    }
  }

  runDotGeneratorTest()
  runCodeCheckerTest()
  runCodeOptimizer()
  runDotGeneratorTest(optimizedVersion = true)

  runCodeGenTest()
}
```

Listing G.1 – Fan speed regulation appplication

Corresponding dataflow graph :

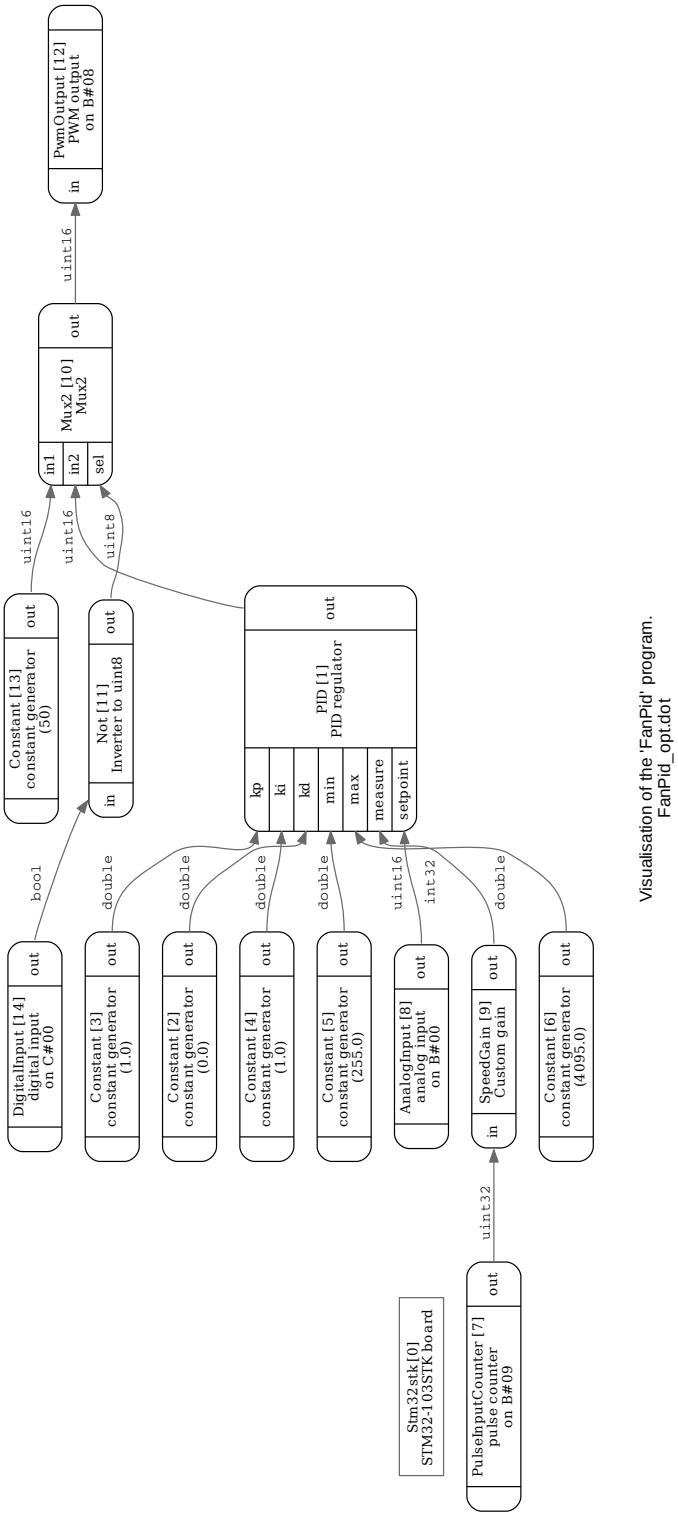


Figure G.1 – Generated graph of the application G.1

Generated code of the application G.1 :

```
// FanPid application
// Code generated automatically from the DSL program 'FanPID'.

//*// Section 00
#include "analoginput.h"
#include "pulseinput.h"
#include "digitalinput.h"
#include "demo/pid.h"
#include "pwmoutput.h"
//*// ----------

//*// Section 01
AnalogInput in_cmp08('B', 0, 8); // OutputPort[0] 'out' of Cmp[08] 'AnalogInput'
PulseInput in_cmp07('B', 9);     // OutputPort[0] 'out' of Cmp[07] 'PulseInputCounter'
DigitalInput in_cmp14('C', 0);   // OutputPort[0] 'out' of Cmp[14] 'DigitalInput'
pid_t pid_cmp01;                 // OutputPort[0] 'out' of Cmp[01] 'PID'
PwmOutput out_cmp12('B', 8);     // InputPort[0] 'in' of Cmp[12] 'PwmOutput'
//*// ----------

//*// Section 02
uint16_t getlAnalogInputB0() {
    return in_cmp08.read(); // Start an A/D conversion and wait for the result
}
//*// ----------

//*// Section 03
void initOutputs() {
    out_cmp12.initialize();
}

void init() {
    in_cmp08.initialize();
    in_cmp07.initialize();
    in_cmp14.initialize();
    pid_init(&pid_cmp01, 1.0, 1.0, 0.0, 255.0, 4095.0);
}
//*// ----------

int main() {
    initOutputs();
    init();

    //*// Section 04
    //*// ----------

    //*// Section 05
    while(1) {
        // 1) Read inputs
        uint16_t in_B0 = getlAnalogInputB0();
        uint32_t in_B9 = in_cmp07.get();
        bool in_C0 = in_cmp14.get();

        // 2) Loop logic
```

```
54
55          // -- User input code of ‘SpeedGain‘
56          int32_t out_cmp09 = 4096 - ((in_B9 - 110) * 50); // Speed gain
57          if (out_cmp09 <= 0)
58              out_cmp09 = 0;
59          // --
60
61          // -- User input code of ‘Not‘
62          uint8_t out_cmp11 = (in_C0 == 0) ? 1 : 0;
63          // --
64
65          pid_cmp01.state.setpoint = in_B0;
66          uint16_t out_cmp01 = pid_step(&pid_cmp01, out_cmp09);
67          uint8_t sel_cmp10 = out_cmp11;
68          uint16_t out_cmp10;
69
70          if(sel_cmp10 == 0)
71              out_cmp10 = 50;
72          else
73              out_cmp10 = out_cmp01;
74
75          // 3) Update outputs
76          out_cmp12.setPeriod(out_cmp10);
77      }
78      //*// ----------
79
80      //*// Section 06
81      //*// ----------
82  }
83  // END of file 'FanPid.cpp'
```

Listing G.2 – Generated code of the application G.1

List of tables

List of figures

List of programs