Code Generation for STM32F4 Boards with Modelica Device Drivers

[Work in Progress]

Lutz Berger
Berger IT-COSMOS GmbH
Puchheim, Bavaria, Germany
berger@it-cosmos.com

Martin Sjölund
Linköping University
Linköping, Sweden
martin.sjolund@liu.se

Bernhard Thiele
Linköping University
Linköping, Sweden
bernhard.thiele@liu.se

ABSTRACT

With the modeling, simulation and code generation of mixed continuous/discrete models in Modelica, a development approach becomes feasible which addresses one of the core challenges in cyber-physical systems. The aim is to achieve a simulation of the embedded system model in a physical environment model, before its deployment on real hardware. Based on the already existing support for Arduino boards, an effort has been started for supporting code-generation from Modelica models for STM32F4 boards. In this paper the concept will be explained. Basically Modelica's C-interface and the experimental low-footprint embedded code generation of OpenModelica is used for generating code for the main function. The support for the STM32F4 family is realized with the C-interface functions using the hardware abstraction layer (HAL) application programming interface (API) STM32F4CUBE from ST Microelectronics.

CCS CONCEPTS

• **Computing methodologies** → **Simulation types and techniques**; • **Computer systems organization** → **Embedded software**; Real-time system architecture; • **Software and its engineering** → *Source code generation*;

KEYWORDS

Modelica, Embedded, STM32F4

ACM Reference Format:
Lutz Berger, Martin Sjölund, and Bernhard Thiele. 2017. Code Generation for STM32F4 Boards with Modelica Device Drivers: [Work in Progress]. In *EOOLT'17: 8th International Workshop on Equation-Based Object-Oriented Languages and Tools, December 1, 2017, Wessling, Germany.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3158191.3158204

1 INTRODUCTION

The Modelica Language supports discrete-time and continuous-time models. While the continuous-time part allows modeling of the physical behavior via differential and algebraic equations, the discrete-time part allows modeling of digital control systems based on the synchronous data-flow paradigm, difference equations, state machines, and digital support logic. Since digital control applications play a major role in embedded systems, efforts have been started to extend the OpenModelica Compiler (OMC) with support for generating low-footprint code for embedded targets directly from Modelica models. The basic approach consists of generating target-agnostic ANSI C code by the OMC and *injecting* target specific functionality by suitable Modelica libraries. The long-term goal of this work-in-progress is to support code generation from hybrid models in which the continuous-time part facilitates the efficient implementation of advanced control and diagnosis functions based on physical models.

As of OpenModelica v1.11.0 the OMC embedded target is in an experimental stage and the scope of supported Modelica language elements is still rather restricted. Nevertheless, there have been efforts for supporting embedded code generation for the Atmel AVR boards Arduino UNO and the SBHS [1] board (a teaching board for control theory) using OpenModelica together with the Modelica_DeviceDrivers (MDD) library for integrating the target specific modules [4]. Based on the same concept as used for the AVR microcontrollers, the development for STM32F4 Boards could be started. In this paper we will show the results of the development of a new embedded target in the MDD library; no changes were made to the OMC to facilitate the new library although OMC-generated code is discussed in the paper. For achieving full support of the STM32F4 family, the STM32F4Cube HAL was chosen as the interface on which the Modelica module is based on. Starting from an example in STM32F4Cube allowed a quick realization of the first proof of concept as will be demonstrated via a basic Blink example.

2 THE OPENMODELICA EMBEDDED TARGET

The embedded target[1] for OpenModelica is an experiment to see if it is possible to create a much simpler code generator which is able to generate code for very restricted platforms such as the Atmel AVR 8-bit microcontrollers. The regular C-code generator creates huge data structures and contains much debugging information while the run-time system contains many numerical solvers and is around 6MB in size (of which 0.5MB is textual strings for error messages). The code is intended to run on powerful desktop CPUs where the code size does not matter much and it proved difficult to try to strip out unnecessary code when targeting embedded systems.

The embedded target code generator was designed to generate code that is easy to compile and the whole code generator was

[1]The embedded code generation target for OpenModelica can be activated by passing the option "`--simCodeTarget=ExperimentalEmbeddedC`" to the OMC.

written around a single workday as a proof of concept. It does not support arrays, strongly connected components or initialization, but somehow it still works fine for a lot of code since the OpenModelica compiler will make many array equations into scalar equations and so on [4]. Instead of having a big runtime system that is linked in (as is the case for the regular code generator), the code generator will generate C-functions corresponding to the Modelica function used. This results in the compiler for the embedded target not trying to compile functions that end up unused in the model.

Most of the logic instead takes place in a carefully designed Modelica library (in this case the MDD library), which uses constants and constant evaluation of functions to hint to the compiler what it can optimize away. The end result is a small, easy to compile, executable that can integrate a model (using forward Euler). The compiler itself (in this case OpenModelica) does not know anything about Atmel AVR or STM32F4, but generates C-code which the user can compile with the corresponding tool-chain and upload to an embedded target which makes the process flexible and extensible. This can be contrasted to common approaches to embedded system targets in modeling tools, where the modeling tool knows about the intrinsics of the embedded target and it is only possible to compile for these supported targets.

3 MODELICA APPROACH FOR CODE GENERATION

This section explains the code generation approach by walking through the building blocks of a basic blinking light-emitting diode (LED) example. The emphasis is on discussing problems like clock configuration, initialization of resources, and real-time synchronization. Starting from an overview of the library structure and a general approach of extending Modelica for embedded targets, it is shown how the full STM32F4 board's family can be supported and how the resolution of the scheduled frequency of the main routine can be increased. Finally, current limitations in generating code from Modelica models are discussed.

3.1 Library Structure

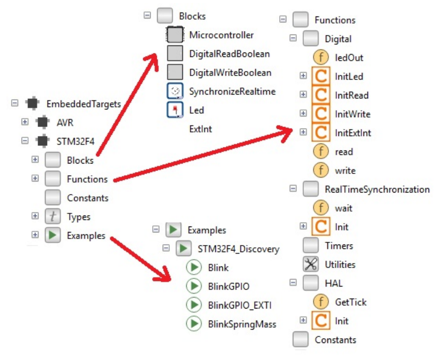

Figure 1: Structure of the `EmbeddedTargets` package within the Modelica_DeviceDrivers (MDD) library [4].

Figure 1 shows the embedded targets part of the MDD library [4]. Within the `EmbeddedTarget` package we are interested in the `STM32F4` subpackage, which is further divided into the packages `Blocks`, `Functions`, `Constants`, `Types` and `Examples`. The `Examples` package contains complete examples which can be translated to STM32F4 target code using the approach discussed in this paper. The `Blocks` package contains drag-and-drop blocks, the `Functions` package contains function definitions which provide an interface to external C code functions.

3.2 Embedded Target Extensions for Modelica

There are three main areas that need to be addressed when adding support for a new embedded target using the Modelica library-based approach advocated in Section 2: *initialization*, *synchronization*, and *cyclic execution* (simulation steps).

Required initialization code can be executed by using Modelica's *external object* facilities. Modelica specifies that the constructors of external objects are called exactly once before the first use of the object. Hence, external object constructors can be exploited for injecting required initialization C code. This approach is used for the STM32F4 target extension (see Figure 1): Blocks defined in the `Blocks` subpackage instantiate external objects which call constructor functions from the `Functions` subpackage. Microcontroller units (MCUs) make heavy use of general-purpose input/output (GPIO) pins. GPIOs can be used from Modelica by defining blocks which call to external C code functions which in turn read from (or write to) the corresponding ports. Again, these GPIOs need to be initialized before they can be used, which can be achieved by using external object constructor functions.

Time in the context of Modelica models usually refers to simulated time which is uncoupled from the wall-clock time (real-time). However, in the context of reactive MCUs it is typically required that the simulation time is *synchronized* with some real-time clock. This can be achieved by calling a blocking function in each simulation step which blocks the further program execution until the simulated time is synchronized with the real-time clock. Hence, the program *waits* until real-time has caught up with the simulation time (of course, an essential prerequisite is that the maximum computation time for one simulation step is actually *faster* than real-time). Each MCU needs to be configured appropriately in order to enable a correct real-time synchronization (e.g., using timers for the Arduino boards and clock configurations for the ARM-Cortex family). As before, necessary configuration options can be set in respective external object constructor functions. The STM32F4 `Blocks` package in Figure 1 contains the `SynchronizeRealtime` block which provides the real-time synchronization function and allows to configure the necessary clocks.

The code which is executed in every simulation step is generated from the Modelica equations which describe the system's behavior over time. The embedded code generation target for OpenModelica performs an automatic time-discretization of (continuous-time) differential equations using the forward Euler method (additional methods might be implemented in future versions).

3.3 STM32F4 specific considerations

Each MCU needs to be configured. As resources are limited, code has to be kept tight and thus only parts which are necessary for the specific application are enabled. First of all, the clock has to be configured, after that all the necessary timers, GPIOs' interrupts, etc. need to be enabled.

3.3.1 Clock Configuration. The boards have different oscillators: HSI, HSE, LSI, and LSE. "HS" stands for high speed and "LS" for low speed. "I" and "E" for internal and external. Certain registers have to be set for oscillator selection and enabling. It has to be decided whether to use the phase-locked Loop (PLL) unit and which clock source should be used as input for the PLL. Without using PLL the clock output is the frequency of the chosen oscillator, e.g, HSI 16 MHz, HSE 8 MHz on the STM32F4-Discovery board. With the PLL it is possible to achieve a higher processor speed, e.g., for STM32F4-Discovery up to 168 MHz. For the clock frequency (PLL) the parameters PLLM, PLLN, PLLP, and PLLQ have to be set in registers resulting in clock frequencies according to following formulas [2, Section 6.3.2]:

$$f_{\text{VCO-clock}} = f_{\text{PLL-clock-input}} \cdot (\text{PLLN}/\text{PLLM}) \tag{1}$$

$$f_{\text{PLL-general-clock-output}} = f_{\text{VCO-clock}}/\text{PLLP} \tag{2}$$

$$f_{\text{USB-OTG-FS,SDIO,RNG-clock-output}} = f_{\text{VCO-clock}}/\text{PLLQ} \tag{3}$$

where $f_{\text{PLL-clock-input}}$ is the oscillator input frequency, $f_{\text{PLL-general-clock-output}}$ is the system clock frequency, and $f_{\text{USB-OTG-FS,SDIO,RNG-clock-output}}$ is the frequency for the USB-OTG-FS (universal serial bus - on the go - full speed) clock, the SDIO[2] (Secure Digital Input Output) clock and the random generator clock.

From the system clock, the high speed clock (HCLK) frequency is derived with the advanced high performance bus (AHB) prescaler[3]. HCLK is used to clock the CPU. Peripheral clocks PCLK1/PCLK2 are derived from the HCLK to clock the advanced peripheral buses APB1/APB2 using the APB1/APB2 prescalers. From these buses 16 and 32 bit timers are derived. Timers are for example connected to the APB buses, where one of them can be used for real-time synchronization as described in Section 3.2. Using a lower frequency for the CPU leads to a lower power consumption. Thus, this possibility of adjustment is important for realizing low power systems. Care has to be taken that the resulting frequencies for PCLK1/PCLK12 are not exceeded for different MCUs as described in [2] page 166 and 229.

3.3.2 Configuration of other Components. If other components, e.g., UART (universal asynchronous receiver-transmitter), SPI (Serial Peripheral Interface), i^2C (Inter-Integrated Circuit), etc. shall be used they have to be enabled and configured. Further information can be found in [2] (register architecture) and in [3] (STM32F4Cube HAL interface).

[2]I/O extension of the SD (secure digital) card slot. Can be used for GPS, camera, WiFi, Ethernet, barcode readers, and Bluetooth.

[3]Prescalers are used to reduce a high frequency electrical signal to a lower frequency by integer division.

3.4 Blinking LED Model

The approach will be explained using a basic blinking LED model "`BlinkSpringMass`" which is one of the example models below the `Examples` package (see Figure 1). The model is assembled using drag-and-drop blocks from the `Blocks` package. Its diagram view is depicted in Figure 2. The relevant model components are:

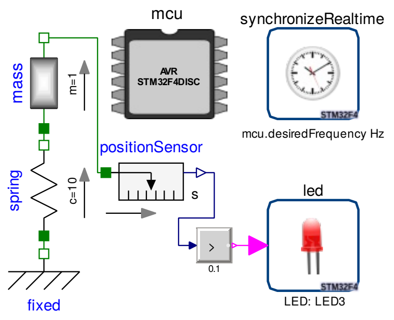

Figure 2: Blinking LED model for STM32F4 Discovery board (`Examples.STM32F4_Discovery.BlinkSpringMass`).

mcu acts as a shared configuration object using Modelica's **`inner/outer`** mechanism and can be accessed like a global variable. Parameters for setting the desired sampling period, as well as for specifying the STM32F4 platform type (here the STM32F4-Discovery board) can be given. Instantiation of the `mcu` block entails the creation of an external object `mcu.hal`. This external object is the gateway to the external C code which is responsible for initializing the MCU. The external C code uses the HAL interface for programming the STM32F4. Indeed, the only C function which needs to be called within the external object constructor is the HAL function `HAL_Init()` which is described in the respective manual [3, Section 2.12.2.1].

synchronizeRealtime allows the configuration of the clock of the board for ensuring real-time synchronization of the application using the desired sampling period. This block supports many clock related configuration parameters (see Section 3.3.1).

led supports toggling an LED on the evaluation board corresponding to the Boolean value of the input signal. Initializes the GPIO which corresponds to the LED.

The remaining components model a spring-mass system using translational mechanics elements from the Modelica Standard Library (MSL). The system starts in a deflected position. Its oscillations are monitored and generate the Boolean signal which toggles the LED's state.

In some cases it is important to establish a correct order of function calls in the generated code. In Modelica, the calling sequence of function calls associated to component declarations is based on data-flow dependencies and not on the sequential appearance in the source code like in typical imperative programming languages. If the required execution order between functions is not automatically ensured by intrinsic data-flow dependencies between those functions, it is possible to introduce artificial "*dummy*" dependencies for the sake of ensuring a desired ordering. This pattern is used in `Blocks.SynchronizeRealtime` and `Blocks.Led`. Both blocks have internal dependencies to the (globally accessible) `mcu.hal` object in order to ensure that the `HAL_Init()` function is called before other HAL API functions.

3.5 The C Code

The external C code used in the MDD library is provided as header-only files below the library's standard include directory Resources/Include. The STM32F4 related files in that directory are MDDSTM32F4-Digital.h (Digital GPIO and LED related functions), MDDSTM32F4-HAL.h (HAL initialization), and MDDSTM32F4RealTime.h (clock configuration and real-time synchronization). Analog read and write is not yet implemented.

As outlined in Section 3.3.1, the clock configuration can be a rather complex topic. Therefore, in a first step the clock configuration used in the SMT32F4-Discovery GPIO_EXTI example from the STM32F4Cube software suite was adopted and hard-coded into the initialization function for the real-time synchronization logic. Later, the clock configuration was extended for supporting the parameters discussed in Section 3.3.1. This extension brought two improvements: (1) support for the full STM32F4 family, and (2) a higher flexibility in the clock configuration regarding power consumption. The synchronization logic uses the `HAL_GetTick` function which returns the ticks after initialization in a milliseconds resolution. Ongoing work is trying to improve the synchronization code, since the MCU allows for a far better resolution. The intention is to use a timer interrupt which has the potential to improve the possible resolution to $1\,\mu$s.

Another limitations at present is the experimental state of the embedded code generator (see Section 2). The experimental code generator assumes a hard-coded step size of 0.002 s for the cyclic execution loop which leads to a mismatch between simulation time and real-time synchronization if the cyclic execution loop runs with a different step size.

Besides the Blink example described in this paper there are two other examples available in the MDD library: one example with digital input/output and one using an external interrupt line for toggling a connected LED (Figure 3 shows the SMT32F4-Discovery board running this example). The procedure for building and flashing the examples is described in the `STM32F4` package documentation. Extending the examples with other system components requires changes in the build environment such as definition of macros and adding resources to the respective Makefiles.

It should be noted that using the described code-generation facilities (at least at the current state) does not omit the requirement of having a good knowledge about the embedded target and manually created code sequences will be needed to complement any more serious designs. At present, there is no solution for how code can be generated for a configured interrupt, e.g., for a GPIO input. The code handling, when the interrupt is fired, must be manually written. It is an interesting questions how such scenarios can be supported and future work might propose a solution.

4 CONCLUSIONS

The paper presented a code-generation approach for for STM32F4 boards in which the target specific code is provided as part of the Modelica_DeviceDrivers (MDD) library, while the generic application code is generated from a target-agnostic experimental code-generator which is part of OpenModelica. OpenModelica's experimental embedded code-generator is still in its infancy and showed respective limitations, nevertheless the first results within this work suggest that the underlying approach is feasible. The longer-term vision consists of enabling more integrated approaches to cyber-physical system development which combine Modelica's excellent simulation capabilities with suitable approaches for automatic code-generation for embedded systems from Modelica models. This promises a more efficient development process since low-level manual coding tasks are reduced and early design validation and verification by simulation and formal methods is facilitated.

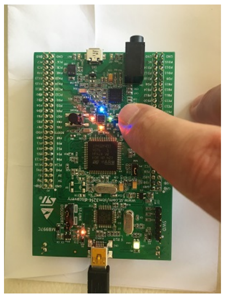
Figure 3: STM32F4-Discovery board with a blinking LED example variant running.

ACKNOWLEDGEMENTS

This work has been supported by Vinnova in the ITEA3 OPENCPS project and in the Vinnova RTISIM project. Support from the Swedish Government has been received from the ELLIIT project, as well as from the European Union in the H2020 INTO-CPS project. The OpenModelica development is supported by the Open Source Modelica Consortium.

REFERENCES

[1] Inderpreet Arora, Kannan M. Moudgalya, and Sachitanand Malewar. 2010. A low cost, open source, single board heater system. In *4th IEEE International Conference on E-Learning in Industrial Electronics (ICELIE)*. https://doi.org/10.1109/ICELIE.2010.5669868

[2] STMicroelectronics 2017. *RM0090 Reference manual,STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced ARM®-based 32-bit MCUs*. STMicroelectronics. http://www.st.com/content/ccc/resource/technical/document/reference_manual/3d/6d/5a/66/b4/99/40/d4/DM00031020.pdf/files/DM00031020.pdf/jcr:content/translations/en.DM00031020.pdf.

[3] STMicroelectronics 2017. *UM1725 User Manual, Description of STM32F4 HAL and LL drivers*. STMicroelectronics. http://www.st.com/content/ccc/resource/technical/document/user_manual/2f/71/ba/b8/75/54/47/cf/DM00105879.pdf/files/DM00105879.pdf/jcr:content/translations/en.DM00105879.pdf.

[4] Bernhard Thiele, Thomas Beutlich, Volker Waurich, Martin Sjölund, and Tobias Bellmann. 2017. Towards a Standard-Conform, Platform-Generic and Feature-Rich Modelica Device Drivers Library. In 12th *Int. Modelica Conference*, Jiří Kofránek and Francesco Casella (Eds.). Prague, Czech Republic. https://doi.org/10.3384/ecp17132713