Project Group 'Racing Car IT'

Documentation

Peer Adelt	Mahak Anand	Rishab Dhar
Waleri Heldt	Sven Henning	Benjamin Koch
Thorsten Koch	Tobias Nentwig	Christof Possienke
Manoveg Saxena	Sven Schönberg	Krishna Sudhakar
Sebastian Surmund	Bhavesh Talreja	Hemanth Venkatappa

University of Paderborn

Summer Semester 2012

Advisers:
Prof. Dr. Marco Platzner, Tobias Beisel, Sebastian Meisner, Lars Schäfers

Abstract

This document usage information and implementation documentation for the software that has been developed in the project group 'racing car it'. The software is tailored to the needs of the UPBracing Team, an association of students that builds up a new racing car every year. We believe that the software is useful for other applications that involve a CAN bus and/or microcontrollers of the Atmel AVR family.

The project groups has developed two tools:

- ~~The RemoteCockpit can be used to monitor messages on the car's CAN bus via a wireless or wired connection. Its interface is very easy, so it can be used by non-technical persons. The software can be used on a PC as well as an Android tablet.~~ This part is not published as open source software.
- The project group has developed code generators to reduce the complexity of the microcontroller software in the car. Programmers use graphical editors and configuration files instead of C code, which makes the programs easier to understand and change.

Both tools are used by the UPBracing Team to develop the software for their next car, the PX213. Therefore, we assume that our software improves the development process and helps the software developers. In the last chapter of this documentation, we outline some extensions of our tools that we are going to implement.

Contents

1 Introduction

1.1 Racing Team

The idea for this project group is based on the UPBracing Team. This team builds a racing car each year to take part the international racing series Formula Student.

Formula Student is a racing series for small single seated racing car prototypes up to 600ccm of cylinder capacity. It is an international racing series for student teams which take part in arranged student design competitions all over the world. The competition consists of presentations (static events) and driving the car on the race track (dynamic events). The static events are presentations about the racing car's design and cost. Examples for the dynamic competition are a short acceleration race and a long race (22 km) to prove the reliability of the car.

To compete with student teams from all over the world, students of the University of Paderborn founded a team, the UPBracing Team. The team consists of students from different area of studies. Of course, women and men who study mechanical engineering, electrical engineering or computer science are needed to build such a racing car. Furthermore, students of economic sciences manage the budget calculation or deal with public relation. The racing car is planed, designed and constructed only by students. The manufacturing of the car's parts is done by different companies or by the student themselves if this is possible. All assembling and repairing tasks are also done by students, so theoretical tasks are part of the work as well as practical tasks. All members of the team must work together to build the racing car and successfully compete against the other students in the competitions.

1.2 Racing Car Hardware

This section outlines the subject of research by introducing our latest racing car 'PX212' (shown in Figure 1.1), which is the primary target of the software we develop in the project group. At first, we list the basic characteristics of the car

followed by detailed information about the build in electronics to give an overview of the car parts relevant for this project. The next section (section 1.3) highlights the parts of the current system that are improved by this project group.

Figure 1.1: The current UPBracing Team's racing car

General car details:

Engine

- In-line engine from Suzuki GSR 600
- Engine performane: 92 hp
- Motor torque: 63 Nm at 9000 rpm
- Fuel: E85
- Custom modifications and self-developed air intake

Chassis

- Weight: About 200 kg
- Space frame made of steel with integrated CFK shear fields to improve rigidity
- Fairing out of two layers of CFK in sandwich style
- Self-manufactured rim bases from CFK

PX212 electronics

The racing car contains lots of electronics, essential for running the car competitively. For example the engine control unit called OpenSquirt is a significant electronic part in the car's mechatronic composition. It controls the fuel injection timings, the spark coils, the fuel pump and much more and therefore plays a key

role in running the car at the maximum performance possible during races. There are more electronic devices within the car which all have their specific function.

The various electronic devices within the racing car are interconnected by a CAN bus to enable device communication. The CAN bus has multiple advantages over conventional direct wiring. Firstly the wire harness is reduced drastically by requiring only a few wires to connect an almost arbitrary number of devices. Due to this fact the number of plug-in connectors decrease too. One of the most important advantages is that the integration and modification of existing elements connected to the CAN bus can be achieved more easily. More detailed information about CAN bus basics can be found in section 1.5.

The electronic devices build up a CAN network that allows the transmission of CAN messages and enables a function-specific processing of received data. Some of the messages within the CAN bus of the PX212 are sent periodically, some are sent event-driven. Messages carrying information about the system-state like sensor values (e.g. oil temperature, oil pressure or engine speed) are usually sent periodically. Control messages manipulating the system-state are usually sent event-driven (e.g. pressing a hardware button like shift up/down).

The CAN bus of the PX212 approximately contains 50 signals which are defined within 30 different messages.

In the following we give an overview of all electronic devices that are currently connected to the CAN bus of the racing car:

Engine Control Unit (OpenSquirt) The engine control unit called OpenSquirt controls the engine. The OpenSquirt can be programmed with custom software and the engine parameters like the ignition timing or the air/fuel ratio can be tuned to achieve optimal performance and efficiency. The OpenSquirt receives several sensor signals while its output is mainly for the actuator handling like controlling the spark coils, the injection valves, the fan or the fuel pump. Additionally, the OpenSquirt communicates the engine's sensor- and state-information over the CAN bus.

Electronical Shifter The shifting unit receives the shifting commands from the steering wheel[1] and activates the shifting actuator which is a linear actuator with solenoids. Using this device, the car can shift very fast. It cooperates with the engine solenoids. Using this device fast shifting by only pressing a button/pulling a paddle is possible.

[1] The drivers requests an upshift or downshift by pulling a paddle on the steering wheel.

Cockpit electronics The cockpit contains a colored LED bar graph which displays the actual engine speed. Next to this LED bar, there is a 7-segment display that indicates the current gear. Furthermore, there are some status LEDs which light up if some temperatures, pressures or the battery charge values are out of the usual range.

Steering electronics The steering wheel has two paddles on the back side for the electronic shifting. On the front of the steering wheel, there are various buttons and an OLED display. This display shows some important measurement values, such as the engine speed, temperatures, battery charge and pressures. For the display, a menu has been implemented which provides additional functions and which can be controlled by the pre-mentioned buttons. In the menu, you can choose the acceleration mode (automatic upshifting), adjust the cockpit brightness (LED brightness), watch all messages on the CAN bus[2], calibrate the electronic clutch[3] and choose some games for playing as gimmick (Tetris, Pong, Snake).

Sensorboard & various sensors In the racing car, there are some sensors which are not used for controlling the engine, so they aren't connected to the engine control unit. Nonetheless, they are interesting for the engineers. For those sensors there is the so called Sensorboard which is connected to the sensors and converts the sensor values with ADCs (Analog/digital converters) into transmittable discrete values which are then sent over the CAN bus.

Microcontrollers

The software in the car doesn't run on normal PC hardware, but instead the team uses microcontrollers that are small, robust and well suited for embedded deployment. For microcontroller development, you need a lot of specific knowledge and hardware (programming adapters and debugging interfaces (in-circuit emulators)), so it is a good idea to stay with one family of microcontrollers. In the racing team, this happens to be the AVR family. Those 8-bit microcontrollers are quite cheap and very popular among hobby developers, so a lot of information is available from the manufacturer, Atmel, and other sources.

More specifically, the team uses the AT90CAN128 processor which has a built-in CAN interface, so access to the CAN bus is quite easy. Using this microcontroller, only one additional integrated circuit is necessary to adapt the voltage level. By

[2] The data is displayed in 'raw', hexadecimal form, so the user needs to find out the signal values.

[3] The next car will have a mechanical clutch, so that feature won't be used anymore.

using this special microcontroller, CAN bus support can easily be added to self-developed printed circuit boards (PCBs).

It is possible to reprogram microcontrollers in the car by using the CAN bus. You only have to connect to the CAN bus of the car to reprogram all microcontrollers and you don't have to use special programmers to change the programs on the microcontrollers. This allows easy and fast changes on the program in the car.

1.3 Motivation and structure of this project group

The UPBracing Team improves their car each year, so the hardware and software becomes more sophisticated. This gradually changes the development process. The mechanical engineers need better insight into the state of the car. They want to know how the engine reacts on sudden load changes and they want to tune its behavior in real-time. The programmers also want real-time information about the car and the software in it because they use it to test their programs and track down errors.

As the car becomes more sophisticated, it also gets more complex. The mechanical engineers use their CAD system to handle that complexity. The software developers face ever-increasing demands on their software. They have good tools for communication and team work, but they are lacking tools that enable them to make their programs easy to understand despite their growing size and complexity.

In this project group, we tackle both concern. The project groups consists of two subgroups: The software subgroup aims to develop a system for real-time information about the car and the microcontroller subgroup aims to provide tools that support software developers.

Software subgroup

To be more detailed, the software subgroup develops a system which allows users to observe the current state of the racing car during a test run. This has great advantages because the users can analyze and interpret the real-time data for tuning the car's parameter like for example engine parameters (injection timing and amount of fuel). Therefore, some kind of a live view of important values is preferable.

As mentioned in the previous chapter, the car's current state can be read out by accessing the car's CAN bus which contains status messages by various electronic devices. Since this CAN bus is not easily accessible, especially while the car is running on a test track, the software subgroup comes to conclusion that the system to be developed should provide the functionality of reading the car's current state by Wifi. In this case the users can stand on the sideway of a race track while observing the car in real time. Furthermore the software subgroup wants to have the possibility to use various end devices and platforms, so the user can use a tablet or a PC which is running Windows, Linux or Android.

Microcontroller subgroup

Our team members work on the software for about two to three years. When they leave the team, someone else has to take responsibility for his programs. This means that the programs must be well structured and easy to understand, so a new team member can extend them without introducing severe bugs.

Furthermore, it should be easy to change the programs in a consistent way. For example, we sometimes change the identifier of a CAN message to adjust its priority. We need to replicate that change to all programs that send or receive that message and most likely we would forget to tell the mechanical engineers that they have to change the ID in their data logging tool. To enable easy modification, we try to store all information in exactly one place.

We already use the facilities that are provided by existing tools. For example, we use functions and libraries to structure our programs and facilitate code reuse. As our programs grow larger, we start to hit the limits of the C language. We achieve our goals by generating code from configuration files that are modeled to be easily understood and modified by the user. We also provide graphical editors and support for third-party tools where that makes sense.

1.4 Document Structure

This documentation is divided along two axes, so you can easily find the information that is relevant to you. First, the real-time monitoring system (RemoteCockpit) is described in a different part of the document than the software development tools. Second, each part is divided into a user documentation that describes the usage of the software and a technical documentation that explains

how you can improve and extend the programs. You can find the relevant chapters in Table 1.1.

	usage	**technical**	**Responsible subgroup**
RemoteCockpit	chapter 2	chapter 3	Software subgroup
tools for developers	chapter 4	chapter 5	Microcontroller subgroup

Table 1.1: Chapter overview

We draw some conclusions from the work of both subgroups in chapter 6 and provide future prospects and ideas for follow-up projects.

In addition to this document, we provide further documentation in additional documents and in the code. Some aspects of the programs are too complex to describe in this document. We provide additional documents that focus on one topic. We refer to these documents from related parts of this document. This document will eventually get out-of-sync as we further improve the programs, so the most detailed documentation is in the code itself. If you want to understand a particular piece of the program, find the code (we provide some pointers in this document) and read the comments. In particular, you should read the JavaDoc comments that describe the programming interface of each module.

Most parts of our implementation use the CAN bus in some way, so you should know some basic information about the CAN bus to understand this document. Therefore, we give a short introduction in section 1.5. Furthermore, you should know at least one programming language (preferably Java), unless you only want to use the RemoteCockpit.

1.5 Background to Controller Area Networks

This chapter introduces the Controller Area Network (CAN) bus system. It outlines the basics of the CAN protocol that are needed to understand this documentation and gives some additional interesting technical details about the event-based transmission handling.

1.5.1 Function and Layout of CAN

The CAN bus (Controller Area Network) is an asynchronous serial bus system which belongs to the field buses. It was developed by BOSCH in 1983 with the aim to shorten the wires used in cars to reduce the car's weight and complexity as you can see in Figure 1.2. The CAN bus allows the communication with multiple nodes via the same cable and therefore made the direct wiring between related components obsolete. It is widely used and initially developed for the automotive sector but also present in industrial automation systems.

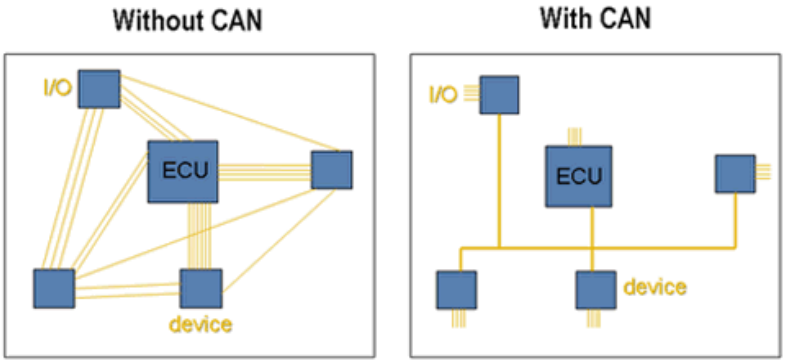

Figure 1.2: Comparison between a non-CAN network and a CAN network

The nodes are usually connected in a line (line topology) and can reach a transmission speed of up to 1 Mbit/s (depending on the wire length). The maximum amount of nodes connected to one CAN bus is not limited by protocol but in practice there are physical limitation starting at a node count of about 32 nodes. [4].

[4] Background: This number results from the fact that every bus sharing unit has got an electrical resistance. With a given signal voltage the current on bus can be calculated. This current must not exceed a well defined value because some electronically parts could then get damaged. For not exceeding this value the maximum number of bus sharing units must be limited.

One of the most important facts of the CAN bus is the priority allocation. At this point it should be noted that the CAN bus is a message-oriented, not a sender-oriented bus system. This means that all senders are equal bus participants [5] and that there is no cyclic message exchange necessary, although possible.

1.5.2 Contents of CAN messages

In this section we describe some details of the CAN protocol and message structure which is important to understand following chapters of this documentation that contain implementation details regarding the CAN communication. To get an idea of the message structure in a CAN network there is given an overview in Figure 1.3.

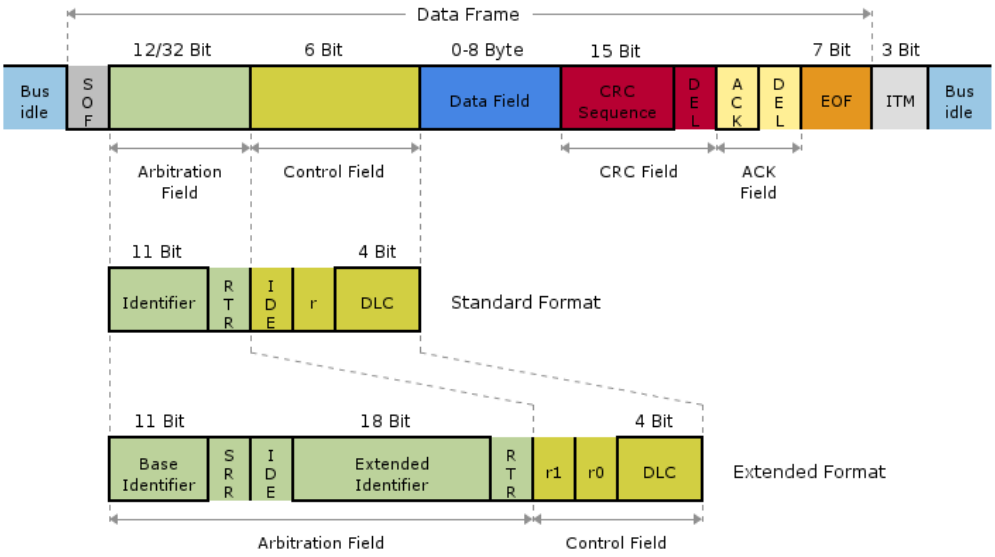

Figure 1.3: Content of a CAN message

The *ID Message Identifier* which can be seen in Figure 1.3 identifies a message. There are standard and extended message identifiers[6]. Each bus node uses the identifier to recognize received messages and process the transmitted data. Messages can be dropped if they don't contain valuable information for the receiving nodes or signals carried by the message can be extracted and handled by the node. The message priority corresponds to the identifier of a message. Lower identifiers will dominate higher ones, so the so called dominant bit in a CAN message '0'

[5] In this case equal means that every sender can send messages with arbitrary priorities. The decision whose message will be send first on bus depends only on the message priority. See subsection 1.5.3.

[6] standard: 11 bits for identifier, extended: 29 bits for identifier

will overwrite the recessive bit '1' of a lower prior message if they are sent at the same time. Further information on this you will get in subsection 1.5.3. So at this point it should be clear that there can only be one sender of a specific message to ensure a deterministic bit-arbitration that works in every situation (otherwise if there are two identical messages sent, one of them will get lost[7]).

The next important field of a CAN message is the data field whose size is defined in the DLC (Data Length Code) bits of the control field. The data field is used to send data over the bus. There is no restriction on the interpretation of transmitted data but the commonly used types of messages used by the UPBracing team are controlling messages, like nominal values or signals, and measurement data messages from sensors. The fact that there are no restrictions for the data interpretation, is mainly the reason why there has to be an uniform syntax of describing the message structure. Therefore we use the DBC (Data Base CAN) file format to store signal structures and message details in a central file to allow an easy interpretation of the CAN bitstream. Furthermore general information about the CAN communication can be specified in this DBC file. This is information like the determination whether standard or extended identifiers are used, which transfer rate is used and which nodes are part of the network (like e.g. actuators, logic units, etc).

1.5.3 Prioritization by bit-arbitration

For handling collisions on the wires the CAN bus uses a technique called CSMA/CR (Carrier Sense Multiple Access / Collision Resolution). Only one node can send a message to the CAN bus at the same time. To decide which node is allowed to send its message, the so called *bit-arbitration* is used:

Based on the identifier of a message it is decided which node can send its message, because the identifier specifies the priority of the message. An identifier with a low number has got a high priority and an identifier with a high number results in a low priority of the message. Each time multiple nodes wants to send a message the collision is solved by comparing the identifier of the message with the state of the CAN bus. How this comparison can solve the collision can be seen in the example below, shown in Figure 1.4.

In the given example there are three bus sharing units, also called nodes, which respectively want to send a message. Each of the messages has got a different

[7] Note: There can probably exist situations in whose it won't matter if a message get lost so various senders could send the same message, for example: Emergency stop message.

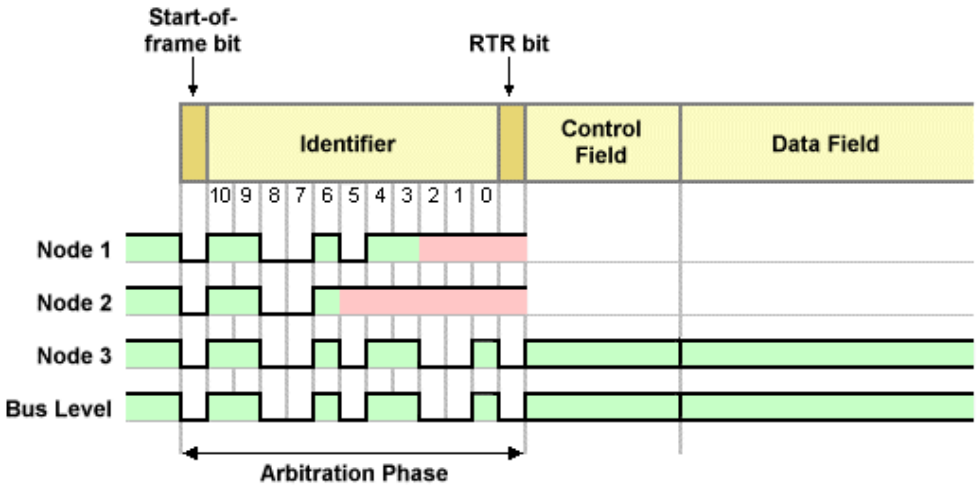

Figure 1.4: Example for CAN arbitration

identifier number. At the top of Figure 1.4 you can the see the content of a can message beginning with the identifier field. In the beginning the nodes transmit (from left to right) the same bits of their message identifiers until they get to the first difference at bit at 5. At the time when bit 5 is transmitted node 2 will notice that there is at least one other node which is sending as well because its recessive bit (1) is overwritten by the dominant bits (0) of node 1 and node 3. At this point node 1 and node 3 don't know that they are sending synchronously. At the time when bit 2 is transmitted there is the next difference between the identifiers. Node 1 will now notice that there is another node with a higher priority sending as well. Node 3 won't notice anything. For node 3 the sending process looks like there was no other node trying to send a message the whole time. Due to the fact that node 3 had the highest priority (and therefore the lowest identifier number) it begins to send its remaining data while node 1 and node 2 stop sending.

2 RemoteCockpit – User Guide

Not published as open source. Sorry...

3 RemoteCockpit – Technical Documentation

Not published as open source. Sorry...

4 Microcontroller tools – User Guide

Contents

4.1 Introduction

In our racing cars, we have five AVR microcontrollers. Each of them has its own program. Some programs only have 800 lines of code; others have more than 5000. The programs are still small enough to be read in one day, so we got away with manual programming using the available tools. However, we expect the programs to become larger as we add more features.

4.1.1 Motivation

Our team members work on the software for about two to three years. When they leave the team, someone else has to take responsibility for their programs. Most new team members don't have much experience with microcontrollers, so our programs should be so easy that an unexperienced programmer can understand them. For that, we need to structure our programs and document our code well enough, so the reader doesn't have to know every detail of the AVR datasheet. In particular, we have these goals:

Make our code easy to understand: On a microcontroller, we often use bit arithmetics to access hardware registers with obscure names. This is predetermined by the hardware manufacturer. We cannot rely on the programmer to describe every aspect of complex hardware access, so we try to provide tools that hide the hardware access and simplify both writing and understanding of the code.

Improve program structure for parallel tasks: Without an operating system, parallel tasks are hard to accomplish. In that case, parallel tasks would have to be called from the main loop, so they have to be split into small parts that return to the main loop quickly. That makes the program hard to understand and to change. Applications with many parallel tasks are virtually impossible to achieve without a task switching mechanisms. An operating system can preempt tasks, so the programmer doesn't have to split them.

Reduce redundancy: Writing applications which contain many repeating or similar code parts is error prone and time consuming. Even more important, you pay that cost many times while maintaining the program. Therefore, redundancy reduction is quite important. It also enables us to easily and consistently change some aspect of the program (e.g. the identifier or structure of a CAN message).

There are several approaches for reducing redundancy. The most common one is to use *libraries* which encapsulate functions for specific purposes. In our racing car, we use a custom library for drawing on the display in our steering wheel, so the drawing code is easier to understand and extend.

Another option for redundancy reduction would be to use a more sophisticated language than C. For the AVR processor family, there are several problems using higher-level programming languages like C++[1]: firstly, correctly working compilers are hard to find[2]. Secondly, extending the C++ language with *templates* is possible in general, but that system has limited potential (e.g. cannot read arbitrary files) and we cannot control the error messages, so the user will have a hard time understanding them. We decided not to use C++ for these reasons.

The third approach is to *generate* C code for repeating tasks. This is especially useful for creating large data structures like operating system Task Control Blocks (TCBs) or communication code for the CAN bus. We created several code generators for different purposes which we describe in the following section.

4.1.2 Code generators

At the moment we have these code generators:

CAN communication The generator loads the description of the CAN protocol from a CAN protocol description (DBC file) and generates code for sending and receiving messages. Received data is usually stored in global variables, but you can change that and get notified of received data. The DBC file format is used by many tools, so you may be able to reuse the CAN description file.

caRTOS config caRTOS is our real-time operating system developed for the AT90CAN microprocessor. The code generator for caRTOS is used to create Task Control Block (TCB) data structures, which we need for each OS task.

EEPROM We generate accessor methods (getters and setters) for variables in the non-volatile EEPROM of the microprocessor.

[1] The only languages that are practically used with AVRs (as to our knowledge) are C, C++ and Basic. Basic wouldn't improve our situation. There are some academical implementations of other languages, but they aren't suitable for our application or they aren't ready for practical use.

[2] The AVR-G++ compiler has too many limitations and bugs to use it for our purposes (e.g. with `__attribute__(progmem)`). Other compilers are available (e.g. IAR – `http://www.iar.com`), but too expensive.

Global variables We generate interrupt-safe accessors to global variables. Locking is only used, if it is necessary.

Pin names You can access a pin with self-explanatory syntax – instead of using `DDR`, `PIN` and `PORT` registers and bit arithmetic, you can write `HIGH(ERROR_OIL_TEMP)`. The generator makes the appropriate macros. If you use EAGLE, you can load pin names from the schematic to keep program and schematic in sync.

Statemachines We provide an Eclipse plugin that you can use to graphically design statemachines. The generator transforms them into code. The statemachines support timeouts and parallel execution which can be hard to achieve in plain C code.

Timer configuration We provide an Eclipse plugin for setting up timers. The tool does the hard calculations for you. You get immediate feedback because the tool shows how accurate the result will be.

You can also write your own generators. You can find the details in the technical documentation, subsection 5.2.3

4.1.3 Realtime operating system

Aside from the redundancy reduction, we also wanted to improve the program structure. Problems arise, when a single ECU shall run multiple tasks in parallel. One well-known concept to achieve pseudo-parallelism is by using an operating system. Since we did not find existing operating systems, that are free to use and and have a sufficiently small memory footprint for the AVR family, we decided to extend a minimal AVR operating system called caRTOS that was provided by a project group member.

This document explains the extensions we made to the system in section 4.6. However, it will not cover task switching concepts and kernel related implementation details. Please see the external caRTOS [2] documentation for further details.

4.1.4 Structure

The hands-on example in the next section will introduce you to using our code generators. You will build a working program that uses CAN, a statemachine and a timer.

In section 4.3 you will learn to configure the code generator framework according to your needs. Simple code generators like *pin names* or *global variables* are configured in the global configuration file. More sophisticated code generators like *statemachine* and *timer* have accompanying graphical editors that store their configuration in separate files, which in turn can be loaded from the global configuration file.

The code generator framework can be included in the make process of Eclipse. The procedure is described in section 4.4.

In section 4.5 we describe the usage of the different code generators for CAN, caRTOS, EEPROM memory, global variables, statemachines and timers in more detail.

We implemented mechanisms for resource management (semaphores) and Inter-Process Communication (IPC) for the caRTOS operating system. They are described in section 4.6.

4.2 Hands-on tutorial

In this section, you will learn how to use the code generators. You will build a working AVR application that uses CAN, timers and a statemachine.

We will help Alice and Bob who have some problems with their secure communication: Eve has hacked into their wifi and obviously she found a way to break their AES256 encryption. That way, she finds out about their secret meetings and turns up at their meetings. What a bother! Of course, Alice and Bob want to be alone!

Bob suggests that they use a CAN bus: “Eve knows everything about networks and encryption, but she doesn’t have a clue of electronics. She will never figure out how to eavesdrop on a CAN bus!” Alice is not so sure about this, but she agrees that they could at least try. Unfortunately, they don’t know how to use the CAN bus either. Alice say: “Hope is not lost. I’ve seen this cool code generator lately. Let’s try that!”

They agree on a very simple protocol: Alice sends a message with the time and location of the next meeting. A meeting is always on the next day, so they don't need a date. They use the 24-hour format, so they don't need a field for "am"/"pm".[3] They always go to the same few places, so they encode the location as an integer. Of course, we cannot tell you what the numbers mean – you might tell Eve! Bob replies to Alice's message and either acknowledges or rejects the meeting. "What if I want to meet you?", Bob asks. Alice replies: "Well, that's easy. You send me a message and I will set up the meeting. And I need a way to cancel a meeting, if I change my mind. Let's have a message for that, as well."

4.2.1 Requirements

You need a computer that can run Eclipse. This means that it should run Linux, Mac OS X or Windows and you should have half a gigabyte of free space on your hard disk.

We assume that you know how to use Eclipse (create a new project, build it, ...). If you normally use another IDE, you will be able to follow along, but sometimes you will have to look through the menus to find a command.

You need at least basic knowledge of the C language and AVR programming. You will be able to follow this walkthrough without that knowledge, but you won't understand the concepts. If you have never used a CAN bus, read our introduction about that topic in section 1.5.

If you want to try the program on hardware, you obviously need the hardware. We use two DVK90CAN1 evaluation boards. You also need a CAN cable (female DB-9 on both ends) and you should have a serial connection to one of the boards. If you use other hardware, you may have to change the programs accordingly. If you only have one board, you cannot use CAN, but you can still use the other generators. If you don't have any hardware at all, you can still use this tutorial, but you won't be able to run the program.

[3] In the 24-hour format, 'am' times remain the same and for 'pm' the hour is 12 plus the usual hour. For example, 15:07 means 3:07 pm.

4.2.2 CAN protocol description (DBC file)

On the following pages, you will often see a piece of text with a summary icon (like this one). It sums up the steps we are going to do next. If you are an experienced programmer, you may not need the long explanation. You can try to do it on your own. You might have to look at the API reference to do so. If you are going to read the long explanation, you should still read the summaries, so you always know what we are going to do.
At first, we will create a DBC file and describe the messages. If you use Windows, you can install Vector CANoe and use CANdb++. You can also just use a text editor.

There are several tools that can write DBC files. The most complete one is Vector CANdb++. It's a commercial tool, but you can get it for free. Unfortunately, it is only for Windows and you have to download and install Vector CANoe to get CANdb++. You can also try to use the copy from our Git.[4]

If you don't want to install CANoe, you can also just use a text editor. The file format is defined in `CAETEC-DBC_File_Format_Documentation.pdf`, but I suggest that you look at some examples instead.

Alice and Bob use two messages. To keep it simple, we only use unsigned 1-byte values:

purpose	sender	ID	length	data		
suggest meeting	Alice	42	3	hour	minute	location
accept meeting	Bob	43	1	accepted		
request meeting	Bob	44	1	location		
cancel meeting	any	45	1	reason		

There is no way to set the receiver for a message without signals. Therefore, we need at least one signal for each message.

This is how we put these messages in a DBC file:

Listing 4.1: DBC file: can.dbc

```
1  VERSION ""
2
3  BU_: Alice Bob
```

[4] You need to run `register_dbc.bat`. It will work, if you have all necessary DLLs. If you haven't, you're out of luck. In that case, you need to install CANoe.

```
4
5  BO_ 42 SuggestMeeting: 3 Alice
6   SG_ hour : 0|8@1+ (1,0) [0|0] "" Bob
7   SG_ minute : 8|8@1+ (1,0) [0|0] "" Bob
8   SG_ location : 16|8@1+ (1,0) [0|0] "" Bob
9
10 BO_ 43 AckMeeting: 1 Alice
11  SG_ hour : 0|8@1+ (1,0) [0|0] "" Bob
12
13 BO_ 44 RequestMeeting: 1 Bob
14  SG_ location : 0|8@1+ (1,0) [0|0] "" Alice
15
16 BO_ 45 CancelMeeting: 1 Vector__XXX
17  SG_ location : 0|8@1+ (1,0) [0|0] "" Alice,Bob
18
19 BO_TX_BU_ 45 : Alice,Bob
```

Firstly, we define the CAN nodes. We have one node for Alice and another one for Bob. Next, we define the messages and signals. At the end of the `BO_` line we state who sends the message. For each signal, we say who is receiving it.

The signals have some values that need explanation: The first part is "start-bit|length@endian sign". The signal `hour` starts at bit 0, has a length of 8 bit (1 byte), is stored as little endian ("`@1`") and cannot be negative (unsigned value). The other values are factor, offset, minimum, maximum and technical unit. We don't use them in the generator. They may be important for other tools. The message CancelMeeting is sent by more than one node. Therefore, we have to use the special token `Vector__XXX` in the message definition and specify the senders with `BO_TX_BU_`.

If you use big endian ("`@0`"), the start bit is different. We copy this quirk of the DBC format to be compatible with other tools. We suggest that you use a graphical editor, if you need big endian. If you do it in a text editor, add 7 to the start bit, before you put it in the DBC file.

If you cannot use the DBC file for other purposes, you may want to build the CAN model using JRuby. This is easier because you don't have to deal with the quirks of DBC. You can find the equivalent JRuby code in subsection A.3.3

Alice and Bob need some hardware to interface with the CAN bus. They both buy a DVK90CAN1 evaluation board. We need two AVR projects – one for Alice's board and the other one for Bob's.

We are going to build the software, now. You need an Eclipse with AVR development tools. Later we will need our timer and statemachine editors, so we set them up, as well. We create two AVR projects: for Alice and for Bob.

Please download Eclipse and install AVR eclipse.[5] You should also install our timer and statemachine editors because we will use them later. We provide them in the archive file `EclipsePlugins.zip` that you can use with the Eclipse update manager (Help → Install New Software...). You can find detailed instructions in section A.1.

Create two AVR Projects (menu item: File → New... → Project..., select "C Project", select "Empty Project" in "AVR Cross Target Application"). Call one of them Alice and the other one Bob. Copy the DBC file into both projects.[6]

Please open the project properties now. We have to change a few settings. You need to do this for both projects:

- Disable code analysis because it doesn't work well for AVR projects: C/C++ General, Code Analysis, Launching: disable "Run as you type"
- Enable HEX file generation: C/C++ Build, Settings, Additional Tools in Toolchain: select configuration "`[All configurations]`" and enable "Generate HEX file for Flash memory"
- Add an include directory: C/C++ Build, Settings, AVR Compiler, Directories: select configuration "`[All configurations]`" and add the `upbracing-common` folder
- AVR, Target Hardware: Check CPU type and frequency. The DVK90CAN1 has an AT90CAN128 at 8000000 Hz.
- AVR, AVRDude: Select your programmer. This depends on the programmer you use. Please consult the AVR Eclipse documentation, if you're unsure.

Now, we tell Eclipse to call the code generator during the build process. Please create the file `makefile.targets` as shown in Listing 4.2:

[5] `http://avr-eclipse.sourceforge.net/wiki/index.php/Plugin_Download`
[6] You can drag&drop the file into Eclipse. If you copy it in the file system, you have to refresh the projects (menu: File - Refresh).

Listing 4.2: Makefile extension: makefile.targets

```
# This is the path to the folder which contains run and run.bat.
# You can use an absolute path. If you use a relative path, you
# must add one more ".." because the root is not in the project
# folder but in the Debug or Release folder. Special characters
# or spaces in the path might be a problem, so avoid that.
# Normally, you should use the 'dist' folder which contains the
# de.upbracing.code_generation.jar file. If you use the project
# folder, you have to build it.
CODE_GENERATOR_DIR=../../../upbracing-AVR-CodeGenerator
include $(CODE_GENERATOR_DIR)/makefile.targets-inc
```

You can build the project, but the code generator won't do anything, if it doesn't find a config file. We also have to create a source file with a `main` method.

4.2.3 Configuration

We start with a very simple configuration file that only loads the DBC file and defines names for pins that are connected to buttons or LEDs. This file is almost identical for Alice and Bob because they use the same DBC file and their boards are similar. We only have to change the CAN node name (line 10). Please create the file `config.rb` in both projects using the text in Listing 4.3. Don't forget to change line 10 for Bob.

We will create a configuration file that loads the DBC file we have created. We use the '`DEPENDS ON`' annotation to make sure our program will be rebuild, when we change the DBC file. We also define pin names, so we can refer to the buttons and LEDs by name. The development board has five buttons and eight LEDs.

Listing 4.3: Simple configuration for Alice: config.rb

```
# load the DBC file
# The "DEPENDS ON" tells the code
# generator to rebuild the files,
# whenever can.dbc changes. The
# file name must be the only string
# on the next line.
#DEPENDS ON:
$config.can = parse_dbc("can.dbc")
# select messages for Alice
$config.use_can_node = "Alice"

# tell the program about the buttons
# This is only for the DVK90CAN1 board.
# If you use a different board, you need
```

```
15  # to change this.
16  pin("BUTTON_CENTER",     "PE2")
17  pin("BUTTON_CENTER_ALT", "PD1")
18  pin("BUTTON_NORTH",      "PE4")
19  pin("BUTTON_EAST",       "PE5")
20  pin("BUTTON_WEST",       "PE6")
21  pin("BUTTON_SOUTH",      "PE7")
22  # and some LEDs on PA0 through PA7
23  port("LED", "PA")
```

4.2.4 Alice's program

It's time to add the main program for Alice. Unfortunately, the board only has a few buttons and LEDs. Alice doesn't want some complex code to enter the data, so she uses two buttons with predefined time and location. Please create the file `alice.c` in Alice's project using the text in Listing 4.4.

We will make a simple program for Alice using the input and output facilities that are available on the development board. The main program sends a CAN message, when a button is pressed. We instruct the code generator to run pieces of our code, whenever it receives a message. We use this to change the state of an LED, when a message is received.

Listing 4.4: Alice's main program

```
1  #include <avr/io.h>
2  #include <util/delay.h>
3
4  #include <common.h>
5
6  #include "gen/can.h"
7  #include "gen/pins.h"
8
9  // return true, iff the button has been pressed recently
10 // old_state: state variable for this button
11 // new_state: current state of the button
12 inline static bool pressed(bool* old_state, bool new_state) {
13     // pressed now, but not before?
14     bool result = !*old_state && new_state;
15
16     // remember current state
17     *old_state = new_state;
18
19     return result;
20 }
21
```

```
22  void waiting_for_reply(void) {
23      // turn off LED7 - no meeting requested
24      LOW(LED7);
25
26      // turn off LED4 and LED5 - no reply received
27      LOW(LED4);
28      LOW(LED5);
29  }
30
31  int main(void) {
32      // turn on some LEDs
33      LED_OUTPUT();
34      SET_LED('A');
35
36      // configure buttons pins as input with pullup
37      INPUT(BUTTON_CENTER);
38      INPUT(BUTTON_CENTER_ALT);
39      INPUT(BUTTON_NORTH);
40      INPUT(BUTTON_EAST);
41      INPUT(BUTTON_WEST);
42      INPUT(BUTTON_SOUTH);
43      PULLUP(BUTTON_CENTER);
44      PULLUP(BUTTON_CENTER_ALT);
45      PULLUP(BUTTON_NORTH);
46      PULLUP(BUTTON_EAST);
47      PULLUP(BUTTON_WEST);
48      PULLUP(BUTTON_SOUTH);
49
50      // init CAN bus
51      can_init_500kbps();
52      can_init_mobs();
53      sei();
54
55      bool center_pressed, north_pressed, east_pressed,
56          west_pressed, south_pressed;
57      center_pressed = north_pressed = east_pressed = false;
58      west_pressed = south_pressed = false;
59      while (1) {
60          if (pressed(&center_pressed, !IS_SET(BUTTON_CENTER)
61                  || !IS_SET(BUTTON_CENTER_ALT))) {
62              // blink LED3 a few times
63              for (uint8_t i=0;i<5;i++) {
64                  HIGH(LED3);
65                  _delay_ms(100);
66                  LOW(LED3);
67                  _delay_ms(300);
68              }
69          }
70
71          if (pressed(&west_pressed, !IS_SET(BUTTON_WEST))) {
72              // suggest a meeting
73              TOGGLE(LED3);
74              // meeting at 11:15 am at location 7
75              send_SuggestMeeting_wait(11, 15, 7);
76              TOGGLE(LED3);
```

```
77
78              waiting_for_reply();
79          }
80
81          if (pressed(&east_pressed, !IS_SET(BUTTON_EAST))) {
82              // suggest meeting in another place
83              TOGGLE(LED3);
84              // meeting at 15:30 (3:30 pm) at location 2
85              send_SuggestMeeting_wait(15, 30, 2);
86              TOGGLE(LED3);
87
88              waiting_for_reply();
89          }
90
91          if (pressed(&south_pressed, !IS_SET(BUTTON_SOUTH))) {
92              // cancel meeting
93              TOGGLE(LED3);
94              send_CancelMeeting_wait(0);
95              TOGGLE(LED3);
96          }
97
98          _delay_ms(100);
99      }
100 }
```

Please note how easily we can send CAN messages. The code generator has created a function for each message we are allowed to send. We provide signal values as arguments and the function will put them into the message and send it.

You can build the project now. I suggest that you select the Release configuration because the `_delay_ms` function doesn't work well in Debug mode. If it builds fine, please flash it on the board (AVR → Upload Project to Target device).

Don't try to send any CAN messages, yet. This would block the program because CAN sends a message until at least one receiver acknowledges it. We don't have any receivers, so it would block forever. However, you can use the center button. It flashes LED3 five times.

Before we make Bob's part, we should let the program do something useful, if we receive a message from Bob. We haven't told the generator to do anything special, so it simply puts the signals into global variables. We could check the state of those variables repeatedly, but there is a better way: We can add our own code or replace parts of the generated code. Please add the code in Listing 4.5 to the `config.rb` file in Alice's project.

Please build the project again. You can see that the hook code is added to `can.c` in the appropriate places. We need the software for Bob to test it, so let's build

Listing 4.5: CAN receiver hooks for Alice

```
# We need a few header files for our handlers.
$config.can.addDeclarationsInCFile <<-END
#include <common.h>
#include "pins.h"
END

# We don't want to decode any signals, so we completely
# replace the code for the RequestMeeting message.
add_code("msg(RequestMeeting)", "rx_handler", "HIGH(LED7);")

# We need to know whether Bob has accepted the meeting, but
# we don't want to store that in a global variable. Therefore,
# we supply some custom code to handle the value.
add_code("signal(accepted, AcceptMeeting)", "put_value", <<-END_CODE)
  // We have an answer -> turn on LED4
  HIGH(LED4);
  // Show the status with LED5. The value of the
  // signal is stored in the local variable value.
  if (value)
    HIGH(LED5);
  else
    LOW(LED5);
END_CODE

# We treat CancelMeeting like a meeting that is not accepted.
# We might be interested in the reason, so we let the code
# generator produce its usual code which stores the reason
# in a global variable. Our code will be executed after the
# messages has been decoded and all values have been stored.
add_code("msg(CancelMeeting)", "after_rx", "HIGH(LED4); LOW(LED5);")
```

that now.

4.2.5 A simple program for Bob

Of course, Bob wants his program to be better than Alice's. Nonetheless, he starts with a simple program because Alice is impatiently waiting for him to reply to her messages.

You should already have the files `can.dbc`, `config.rb` and `makefile.targets` in Bob's project. If you haven't, you can copy them from Alice. Please remember to change `use_can_node` to `"Bob"` and remove the additional config parts for Alice. The main program for Bob is also quite similar to Alice's. You can find it in Listing 4.6.

We create a similar main program for Bob and test both programs.

Listing 4.6: Bob's simple main program

```
1  #include <avr/io.h>
2  #include <util/delay.h>
3
4  #include <common.h>
5
6  #include "gen/can.h"
7  #include "gen/pins.h"
8
9  // return true, iff the button has been pressed recently
10 // old_state: state variable for this button
11 // new_state: current state of the button
12 inline static bool pressed(bool* old_state, bool new_state) {
13     // pressed now, but not before?
14     bool result = !*old_state && new_state;
15
16     // remember current state
17     *old_state = new_state;
18
19     return result;
20 }
21
22 int main(void) {
23     // turn on some LEDs
24     LED_OUTPUT();
25     SET_LED('B');
26
27     // configure buttons pins as input with pullup
28     INPUT(BUTTON_CENTER);
29     INPUT(BUTTON_CENTER_ALT);
30     INPUT(BUTTON_NORTH);
31     INPUT(BUTTON_EAST);
32     INPUT(BUTTON_WEST);
33     INPUT(BUTTON_SOUTH);
34     PULLUP(BUTTON_CENTER);
35     PULLUP(BUTTON_CENTER_ALT);
36     PULLUP(BUTTON_NORTH);
37     PULLUP(BUTTON_EAST);
38     PULLUP(BUTTON_WEST);
39     PULLUP(BUTTON_SOUTH);
40
41     // init CAN bus
42     can_init_500kbps();
43     can_init_mobs();
44     sei();
45
46     bool center_pressed, north_pressed, east_pressed,
47         west_pressed, south_pressed;
48     center_pressed = north_pressed = east_pressed = false;
49     west_pressed = south_pressed = false;
50     while (1) {
51         if (pressed(&center_pressed, !IS_SET(BUTTON_CENTER)
52                 || !IS_SET(BUTTON_CENTER_ALT))) {
53             // blink LED3 a few times
```

```
54              for (uint8_t i=0;i<5;i++) {
55                  HIGH(LED3);
56                  _delay_ms(100);
57                  LOW(LED3);
58                  _delay_ms(300);
59              }
60          }
61
62          if (pressed(&west_pressed, !IS_SET(BUTTON_WEST))) {
63              // accept the meeting
64              TOGGLE(LED7);
65              send_AcceptMeeting_wait(true);
66              TOGGLE(LED7);
67          }
68
69          if (pressed(&east_pressed, !IS_SET(BUTTON_EAST))) {
70              // decline the meeting
71              TOGGLE(LED7);
72              send_AcceptMeeting_wait(false);
73              TOGGLE(LED7);
74          }
75
76          if (pressed(&south_pressed, !IS_SET(BUTTON_SOUTH))) {
77              // cancel meeting
78              TOGGLE(LED7);
79              send_CancelMeeting_wait(0);
80              TOGGLE(LED7);
81          }
82
83          if (pressed(&north_pressed, !IS_SET(BUTTON_NORTH))) {
84              // request a meeting
85              TOGGLE(LED7);
86              send_RequestMeeting_wait(42);
87              TOGGLE(LED7);
88          }
89
90          _delay_ms(100);
91      }
92  }
```

Now its time to test the program. Build both programs, flash program Alice to board A (if you haven't already done that) and flash program Bob to board B. Connect their CAN ports. In Table 4.1 you can see how we tested the programs. In addition to the results in the table, LED3 flashes for each CAN transmission. If it stays on, the other board doesn't receive the message. Check the connection if that happens.

[7] "o" means on, "-" means off, LED7 is left (as on the board): 76543210

Action	Meaning	LED pattern[7]
Turn on both boards	Start the programs	A: `-O-----O`, B: `-O----O-`
Press CENTER on A	Test button	LED3 on A flashes 5 times
Press NORTH on B	Request meeting	A: `OO-----O` (7 turned on)
Press WEST on A	Suggest meeting	A: `-O-----O` (7 turned off)
Press WEST on B	Accept meeting	A: `-OOO---O` (4 and 5 on)
Press SOUTH on B	Cancel meeting	A: `-O-O---O` (5 off)
Press EAST on A	Suggest meeting	A: `-O-----O` (4 turned off)
Press EAST on B	Decline meeting	A: `-O-O---O` (4 on)

Table 4.1: Test pattern

4.2.6 Protocol statemachine

At the moment, Bob can press the buttons whenever he feels like doing so. He may absent-mindedly request a meeting when they have already set a time. Alice will surely laugh at him, if he accepts a meeting, although Alice hasn't suggested a time. He won't let that happen!

Bob wants the system to be aware of its current state and stop him, if he attempts something stupid. "Current state", he ponders. "Could we use a statemachine for that?" Yes, of course!

He creates a new "Statemachine diagram" file and starts drawing. You can see the result in Figure 4.1. Please create the statemachine in Bob's project. In some text fields, you can insert a newline with CTRL+RETURN. You also have to set the base period of the statemachine. Rightclick on the white background and select "Show properties view". Set the base period to `100ms`. This means that we will call the `tick` function every 100 milliseconds. You can also use the properties view to change actions and transition labels, if you have trouble selecting the text field in the diagram.

We will create a statemachine that describes the protocol states. It encodes which messages Bob may send in each state. We trigger statemachine events by incoming messages and buttons that Bob presses on his board. The main loop will only call the statemachine. All actions happen inside the statemachine (except initialization).

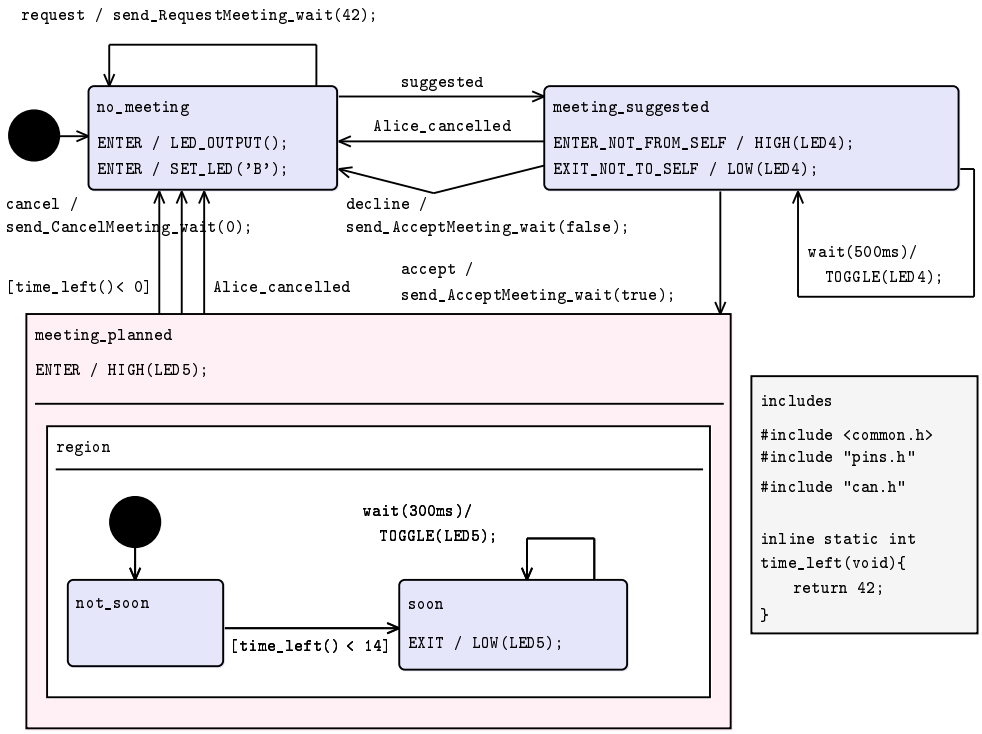

Figure 4.1: Bob's statemachine

The statemachine won't do anything by itself. We need to integrate it into our program:

- Load it in the config file
- Call its init function, when the program starts
- Call its tick function every 100 milliseconds
- Call event functions when a button is pressed or we receive a CAN message

We add a few lines to our config file:

Listing 4.7: Load Bob's statemachine: append to config.rb

```
1  # load Bob's statemachine
2  $config.statemachines.load("bob", "bob.statemachine")
3
4  # We need a few header files for our handlers.
5  $config.can.addDeclarationsInCFile <<-END
6  #include <common.h>
7  #include "pins.h"
8  #include "statemachines.h"
9  END
10
```

```
# We trigger statemachine actions for incoming messages.
add_code("msg(SuggestMeeting)", "after_rx", "event_suggested();")
add_code("msg(CancelMeeting)", "after_rx", "event_Alice_cancelled();")
```

Then we replace the main program. The new one is a bit simpler because we delegate most of the work to the statemachine. You can find the new source in Listing 4.8. Please put that into your C source file and build the project.

Listing 4.8: Bob's main program with a statemachine

```
#include <avr/io.h>
#include <util/delay.h>

#include <common.h>

#include "gen/can.h"
#include "gen/pins.h"
#include "gen/statemachines.h"

// return true, iff the button has been pressed recently
// old_state: state variable for this button
// new_state: current state of the button
inline static bool pressed(bool* old_state, bool new_state) {
    // pressed now, but not before?
    bool result = !*old_state && new_state;

    // remember current state
    *old_state = new_state;

    return result;
}

int main(void) {
    // configure buttons pins as input with pullup
    INPUT(BUTTON_CENTER);
    INPUT(BUTTON_CENTER_ALT);
    INPUT(BUTTON_NORTH);
    INPUT(BUTTON_EAST);
    INPUT(BUTTON_WEST);
    INPUT(BUTTON_SOUTH);
    PULLUP(BUTTON_CENTER);
    PULLUP(BUTTON_CENTER_ALT);
    PULLUP(BUTTON_NORTH);
    PULLUP(BUTTON_EAST);
    PULLUP(BUTTON_WEST);
    PULLUP(BUTTON_SOUTH);

    // initialize the statemachine
    bob_init();

    // init CAN bus
    can_init_500kbps();
    can_init_mobs();
```

```
44      sei();
45
46      bool center_pressed, north_pressed, east_pressed,
47          west_pressed, south_pressed;
48      center_pressed = north_pressed = east_pressed = false;
49      west_pressed = south_pressed = false;
50      while (1) {
51          if (pressed(&center_pressed, !IS_SET(BUTTON_CENTER)
52                  || !IS_SET(BUTTON_CENTER_ALT))) {
53              // blink LED3 a few times
54              for (uint8_t i=0;i<5;i++) {
55                  HIGH(LED3);
56                  _delay_ms(100);
57                  LOW(LED3);
58                  _delay_ms(300);
59              }
60          }
61
62          if (pressed(&west_pressed, !IS_SET(BUTTON_WEST))) {
63              TOGGLE(LED3);
64              event_accept();
65              TOGGLE(LED3);
66          }
67
68          if (pressed(&east_pressed, !IS_SET(BUTTON_EAST))) {
69              TOGGLE(LED3);
70              event_decline();
71              TOGGLE(LED3);
72          }
73
74          if (pressed(&south_pressed, !IS_SET(BUTTON_SOUTH))) {
75              TOGGLE(LED3);
76              event_cancel();
77              TOGGLE(LED3);
78          }
79
80          if (pressed(&north_pressed, !IS_SET(BUTTON_NORTH))) {
81              TOGGLE(LED3);
82              event_request();
83              TOGGLE(LED3);
84          }
85
86          // let the statemachine do its work
87          bob_tick();
88
89          // wait 100ms because that's what the statemachine expects
90          _delay_ms(100);
91      }
92  }
```

By now you now how to flash it to the board, don't you? Let's test the program again. We use the same test as before (see Table 4.1), but we also try to send some "dumb" messages. For example, you can start by cancelling a meeting.

Watch LED3 or the transmit indicator[8] to find out whether the board is sending a message.

When Alice suggests a meeting, LED4 should be blinking until Bob accepts or declines. If he accepts, LED5 is turned on until someone cancels the meeting.

You may have noticed that the `time_left` function isn't implemented properly. We're going to do that next.

4.2.7 Timers

This part is left as an exercise for the reader. You should do these things:

1. Create a timer configuration with mode "Clear on compare match" and set it to 1 second. You need a 16-bit timer because this period is too long for an 8-bit timer.
2. Load the timer configuration in the `config.rb` file.
3. Add an interrupt handler (`TIMER1_COMPA`) that increments a global variable (`clock`).
4. Implement `time_left`: It calculates the time that is left until the meeting (`getHour()*60+getMinute()- clock/60`).

4.3 Configuration

The configuration is the central part of code generation. It contains all the information that goes into the generated code. You can put all information into the main configuration file, but in most cases you will load additional files that you create with graphical editors.

The main configuration file is a Ruby script. It will usually be called `config.rb`. Ruby is a powerful scripting language, so you can use loops and conditionals. We suggest some applications later in this section. However, you don't have to learn Ruby to use our code generators. You can use all parts of our generators, if you know how to call a function and set properties. You will learn that in this section. You can also copy the example we provide which is even simpler.

[8] DVK90CAN1 boards have a dedicated LED that shows CAN traffic. It is next to the CAN connector (labelled U7).

In Listing 4.9 you can see a typical example configuration file. It loads a statemachine, defines some pin names and loads a CAN protocol description. It also adds some custom code to the function that receives the `OilPressure` CAN signal. Most configuration files look like that. In the 'Configuration' sections in section 4.5, we give at least one example for each code generator. You can copy the examples and adjust them to your needs.

Listing 4.9: Config file example

```
# An empty configuration object has been put into the global
# variable $config. We use its methods to put our configuration
# information into that object. There are also some helper
# methods in the global namespace, e.g. parse_dbc(...).

# load a statemachine
$config.statemachines.load("counter", "counter.statemachine")

# define some pin names
pin("ERROR_OIL_PRESSURE", "PA3")
pin("BUTTON_FLY", "PB7")

# load CAN protocol description and select node 'Cockpit'
$config.can = parse_dbc("../../can_final.dbc")
$config.use_can_node = "Cockpit"

# we need to include "pins.h" because we want to
# use it in our CAN handlers
$config.can.addDeclarations <<-END
#include "pins.h"
END

# update error LED whenever a new oil pressure is received
add_code("signal(OilPressure)", "after_rx", <<-END_CODE)
  if (getOilPressure() < 200)
    // turn on LED
    HIGH(ERROR_OIL_PRESSURE);
  else
    // turn off LED
    LOW(ERROR_OIL_PRESSURE);
END_CODE
```

If you do know a bit of Ruby, you can avoid redundancy. For example, you can automatically load all statemachines in a folder instead of adding each one to your config file. If your global variable names follow a pattern, write a loop instead of one line for each variable. You can get the information from all kinds of sources. We load pin names directly from EAGLE schematic files, so we don't have to copy the names into the configuration. You can also modify parts of the configuration that you load from other files. For example, we can add tracing code to our statemachines (state transitions logged to serial line). We do that

	JRuby	Java
set a property	`object.material = "Unobtainium"` or: `object.setMaterial("Wood")`	`object.setMaterial( "Unobtainium");`
print a property	`puts object.material` also: `puts object.getMaterial()`	`System.out.println( object.getMaterial());`
call a method	`object.someMethod(42, 7)` also: `object.some_method(42, 7)`	`object.someMethod(42, 7);`
local variable	`x = 7`	`int x = 7;`
comment	`# this is a comment`	`// this is a comment`
multi-line string	`$config.can.addDeclarations <<-END` `#include "common.h"` `#include "pins.h"` `END`	`config.getCan()` `.addDeclarations(` `"#include \"common.h\"\n"` `+ "#include \"pins.h\"")`

Table 4.2: Access Java object from JRuby (background information)

after loading the data for code generation, so we don't clutter the model we see in the editor. You can make similar extensions because all of that can be done by extending the config file.

The code generator main process runs the config file with JRuby. JRuby is a Ruby implementation for the JVM (Java Virtual Machine). Therefore, you can create and manipulate arbitrary Java objects. Indeed, the configuration is just a set of Java objects. In Table 4.2 you see how you can access Java objects from JRuby. You don't have to learn that because we always present the API from a JRuby perspective. Nonetheless, this knowledge will be useful, if you want to do advanced stuff. Ruby also has a nice syntax for multi-line strings.

For your convenience we put an empty configuration object into the global variable `$config`. You can access all parts of the configuration through that object. We also provide some helper methods that live in the global namespace. You will find the details in the sections about each code generator.

In the next section, we show you how to generate code for a configuration. If you want to try that with a working example, please look at section 4.2.

4.4 Build process integration

Calling the code generator is actually really easy. We have packed all the files into a runnable JAR file. However, you usually don't want to call it manually. It should be integrated into the build process. In this section, we outline the build

integration for AVR Eclipse projects. If you want to know the details and learn about other build options, you can look at section A.2.

AVR Eclipse has to call the compiler and linker to generate an executable file for your program. It uses GNU make as an intermediate layer. AVR Eclipse generates configuration files for make. Therefore, make knows that it has to compile the program (which generates objects files) before linking the object files. Make also knows that it doesn't have to compile a file, if you have neither changed that file nor any header that it uses. Thereby, it can speed up the build.

We extend the make configuration that AVR Eclipse generates. AVR Eclipse tells make to look for the special file `makefile.targets` in the project folder. Make will include it, if it exists. We include our extensions in this file.

For your convenience, our distribution contains all the files that you need. The files are in the `code-generation/dist` folder in our Git. We need the file `makefile.targets-inc`. We will tell you about the other files in a minute. We put a reference to `makefile.targets-inc` into the `makefile.targets` file in your project. Create that file in your project using the text in Listing 4.10. Change the first line to point to the folder of `makefile.targets-inc`. It will tell make and our makefile extensions where they can find the files. The next line includes the extensions.

Listing 4.10: Makefile extensions for AVR Eclipse: makefile.targets

```
1 CODE_GENERATOR_DIR=../path/to/dist/folder
2 include $(CODE_GENERATOR_DIR)/makefile.targets-inc
```

Those two lines enable the code generator for your project. If you want to know how that works, read on. You can skip to the next section, if you don't.

Let's have a look at the file that we have just included into our build script. You can see the file in Listing 4.11. We have removed some parts, so it is shorter and easier to read. Firstly, the file determines the path that is used to execute the code generator. It uses the variable `CODE_GENERATOR_DIR` that we have set in our project. It will execute a script that is called `run`. If you look into the `dist`, you will find `run` and `run.bat`. Windows will automatically append the `.bat` extension, so it will use the `run.bat` script. On Linux and Mac OS, the `run` script is used. The scripts will call the Java application that is contained in the file `de.upbracing.code_generation.jar`. This file contains all code generators and their dependencies, so we only need those four files to run the generators.

Listing 4.11: Makefile extensions for AVR Eclipse: makefile.targets-inc

```
# path of the code generator script
CODE_GENERATOR_BIN=$(CODE_GENERATOR_DIR)/run

# config file is at the root of your project
CONFIG := ../config.rb
# generated files are in a folder next to it
TARGET_DIR=../gen
# this is how we call the code generator
CODE_GENERATOR=$(CODE_GENERATOR_BIN) -C $(TARGET_DIR) $(CONFIG)

# before we build any objects (thus compile C files), we must
# generate the code
$(OBJS): $(shell $(CODE_GENERATOR) -w)

# rule to execute the code generator
CODE_GEN_DUMMY=$(TARGET_DIR)/last_gen_time
$(shell $(CODE_GENERATOR) -w): $(CODE_GEN_DUMMY)

$(CODE_GEN_DUMMY): $(CONFIG) $(shell $(CODE_GENERATOR) -D)
	$(CODE_GENERATOR) && touch $(CODE_GEN_DUMMY)
```

Then, the script sets the variable `CONFIG`. You can see that it expects the config file in the parent directory. Eclipse will execute the build in the `Debug` or `Release` folder, so the parent folder is the root of your project. The generated files will end up in the `gen` folder next the config file. The code generators will create the folder, so you don't have to do that.

We want to run the code generator before any files are compiled. Eclipse puts a list of all compiled C files into the `$(OBJS)` variable. We use this variable to add a dependency to the generated files, so make has to run the code generator before the compiler. We call the code generator with the argument `-w`, so it outputs a list of all the files that it will generate.

The code generator creates several files. Make cannot keep track of that easily, so we use a dummy file that is created at the end of code generation. Make will know that the generator has been run, when it sees this file. If make notices that the dummy file is newer than your config file, it knows that the generated files are up-to-date and it won't bother running the code generator.

The last two lines tell make how it should run the code generator. The part left of the colon ensures that the process will generate the dummy file. The part right of the colon lists the dependencies. Of course, make should refresh the generated files, if the config file changes. In addition, we want to regenerate them, if the code generator JAR or additional config files change (marked with "`DEPENDS ON:` in your config, see section 4.3). We call the code generator with `-D` to generate a list of these files. In the last line you can see the commands that make will

run to refresh the generated files: `$(CODE_GENERATOR)` calls the code generator and `touch $(CODE_GEN_DUMMY)` creates the dummy file, if the code generator doesn't return an error.

By now, you should know how the code generator is called by the AVR Eclipse build process. You can integrate it into other build processes in a similar way. Before you do that, you may want to read appendix A.2 which has some more details about build integration.

4.5 Code generators

By now, you know how to write a configuration and generate the code for it. In this section we present all the generators and how you use them. For each generator, we show several aspects:

- What can you do with the generator? When should you use it?
- How to use the graphical editor (if its has one)
- How to write the configuration (with examples)
- The runtime API: How do you use the generated code from your C code?

4.5.1 Statemachine generator

4.5.1.1 Motivation

A graphical representation of a statemachine is usually easier to understand than code, so we often use statemachine diagrams to illustrate the behavior of a piece of code. If the statemachine diagrams are precise enough, they can be transformed to executable code. We do that for parts of the program that can be expressed better with a statemachine than with code. In addition, our statemachines provide these benefits:

- Statemachines store their whole state in the state variable. Therefore, we don't need a stack for each one, if we run them in parallel. In practice this means that we call a special function (the `tick` function) at a regular interval, e.g. from the main loop, a timer interrupt or an OS task. That function will return quickly, so we can still use the main loop for other tasks. We can achieve similar behavior with operating system tasks, but they need more memory (stack) and processor cycles (task switches).

- The statemachines support time-triggered transitions. This is very useful for implementing timeouts and intended delays without blocking other tasks. This feature is implemented with a counter, so it uses few resources.[9]
- Events are processed immediately. You can trigger an event from your code by calling a special function. If an event is triggered by an interrupt, it will be processed in the interrupt handler.

To use a statemachine from your program, you regularly call its `tick` function and you call an `event_X` function whenever an event occurs. You can find the details in appendix A.4.1.

The next section describes the features of our statemachine implementation in more detail.

4.5.1.2 Features of the Graphical Editor

The Statemachine editor allows the user to model a statemachine, which runs at a specified base rate (in seconds or milliseconds), with one of the 4 supported states and transitions between any of these states. In addition, the editor also supports inclusion of global code in GlobalCode boxes. The notation of the basic statemachine elements – *InitialState, NormalState, FinalState, and SuperState, Transition, and GlobalCode* – is shown in Figure 4.2. As can be seen in the figure, there is another statemachine element, the Region, which is contained in SuperState. Only a SuperState can have a Region. The states, regions, and global code boxes are labeled with State/Region/GlobalCode names, while the transitions are labeled with one of the possible triggers, followed optionally by a list of actions – the set of instructions (such as lighting up an LED) that the statemachine performs – to be executed. The semantics of transition, action, and the 4 types of states are discussed in the following sections.

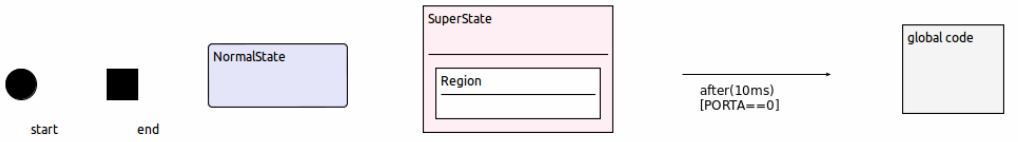

Figure 4.2: Basic elements of Statemachine Editor

[9] The downside is that the timing depends on the calls to the `tick` function. If you need exact timing, the calls to that function must match the base period setting exactly.

Actions

An action consists of a sequence of C language statements that are executed by the statemachine. Except the InitialState all states and transitions can have actions, although actions of transitions are slightly different (see section Transition). The state actions can be of 4 different types – ENTRY, EXIT, DURING, and ALWAYS. The ENTRY action is performed when the statemachine enters that state; the EXIT action is performed when the statemachine exits that state; the DURING action is executed when the statemachine is in that state during the tick, and ALWAYS action is executed when the statemachine enters that state or remains in it. The separator / is used to separate the C statements from the action type. Below are a few examples on actions,

ENTER/PORTA++
EXIT/PORTA = 0
ALWAYS/ increment_portb()

States

Among the 4 types of states, the SuperState is a special one because it can contain nested statemachines in a container called Region. Each of these regions and states must have a name that is a valid C identifier, besides having a parent, which decides the context of that region or state. Within each context each state/region must have a unique name to generate code properly. This is demonstrated with a small example below.

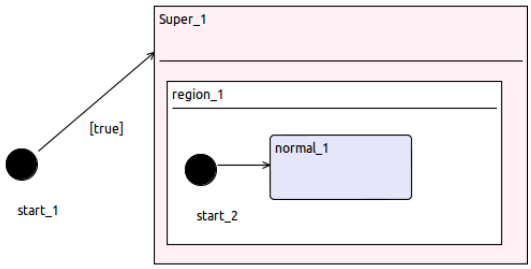

Figure 4.3: Example on State and Region parent

In Figure 4.3 the parent of start_1 and Super_1 is the State Machine, so these states are in State Machine context. The parent of the region_1 is Super_1, so it is in State Machine → Super_1 context. Finally the parent and context of

start_2 and normal_1 are region_1 and State Machine → Super_1 → region_1 respectively.

Transitions

Each transition has a source and a destination state. However, InitialState should not be a target of any transition and not be a source of more than one transition, and the FinalState cannot be a source of any transition. Each transition can have three type of triggers (except for transition from InitialState, which cannot have any triggers) : **event**, **waitType**, and **guard condition**. These triggers can be optionally followed by an *action*. The event name should be a valid C identifier. The general syntax for labeling a transition is shown below.

event :waitType(rate) [condition] / action

An event triggered transition takes place when that event occurs, and the statemachine is in the source state of that transition. Examples of events include a button being pressed or an interrupt. There are three possible types of waitTypes – *before*, *after*, and *wait*. Each waitType specifies when the transition should take place. The waitType should also include time information, such as wait(10ms) or after(1s), where ms or s are abbreviations for milliseconds and seconds. A condition triggered transition takes place when that condition evaluates to true.

A few examples of valid and invalid transition labels are shown in Table 4.3.

Valid Transition Labels	**Invalid Transition Labels**
turn_off : before(10ms) /PORTA=0	turn_off before(10ms) PORTA = 0
[PORTA == 0xff] / PORTA = 0	PORTA == 0xff / PORTA
increment : after(10ms) [PORTA != 0]	2increment : after(10ms) [PORTA != 0]
wait(10ms)	wait(10)

Table 4.3: Examples of Transition labels

4.5.1.3 Using the Editor

Installation

The graphical editor works out of the box once you install the statemachine plugin from the `EclipsePlugins.zip` archive. This archive contains two UPBracing programs, the `UPBracing AVR Timer-Configuration` and `UPBracingStatemachineEditor`. To work with the statemachine editor, you need to install `UPBracingStatemachineEditor`. The other software is used for creating timer configuration files, and so is of no use here. You can find a more detailed explanation in appendix A.1.

Modeling the statemachine elements

Before proceeding with the following section, you already need to create a general eclipse project with the requisite statemachine files `.statemachine` and `.statemachine_diagram` (to setup a project with them refer to section 1.4 in statemachine userdoc). The `.statemachine` file is the important file and is required for generating code, while the other file is for the convenience of the user to create the Data Model of the statemachine graphically. This file can be generated from `.statemachine` albeit with the structure of the statemachine lost.

The statemachine elements can be modeled both from the palette and the canvas. Since it is quite trivial to use the palette, we jump straight to modeling from the canvas (for details on modeling from palette refer to section 1.5 in statemachine user documentation). Except of Region, all statemachine elements can be modeled from the canvas. A description on how to model these elements is given below:

- **States and GlobalCodeBoxes** can be modeled by simply hovering the mouse over the canvas to bring the container shown in Figure 4.4 to front and selecting the desired statemachine element.

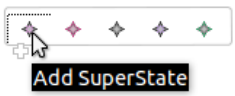

Figure 4.4: Pop up in the Canvas

- **Transitions** can be modeled in two ways from the canvas. In both these approaches you need to hover your mouse over one of the states (source/destination) to bring an arrow selection menu to front, select one of those arrows, and drag it to the target state. Once the arrow has been selected, you can either drag your mouse to the target state, or drag your mouse to the canvas to bring the menu shown in Figure 4.5 to front. From this menu, you can either select an existing state as your target state, or create a new one.

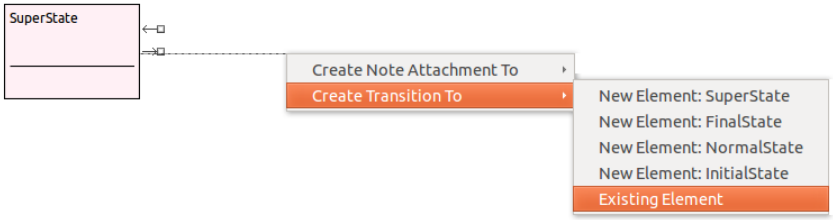

Figure 4.5: Transitions from menu in the canvas

Resizing States, Regions, and Globalcode boxes

The editor creates new instances of states, regions, or globalcode boxes in default sizes. However, these elements can be resized. Resizing most elements is quite trivial. In case of Region, however, if you haven't resized the SuperState already, it is better to resize it first (at the same time adjusting the position of region by selecting and moving it in the SuperState) to the point there are no more scrollbars as you see in figure Figure 4.6 and finally resize the region in the same way as the others are done.

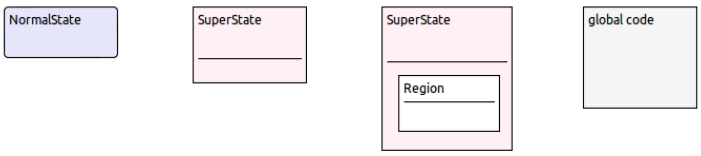

Figure 4.6: Resized NormalState, SuperState, and GlobalCode

Using multiline textfields for properties

The editor supports **multiline textfields** for *Actions* in the NormalState and the SuperState, for *conditions* on transitions, and for *Code* in GlobalCode. To use multiline texfields in any element, you must first have that property's textfield selected, in case you haven't, and then press **CTRL+ALT+ENTER** where you want to break to a new line (see Figure 4.7).

Figure 4.7: Multiline text fields for action in NormalState

Setting statemachine element properties

The editor supports setting the element properties both from the Canvas view and from the Properties view (*Window* → *Show View* → *Other..* → *Select Properties* (under General) from the Menu that pops up or *Right click statemachine element* → *Show properties view*). Setting property values from the canvas can be quite tedious, so using properties view is recommended in that case. Besides, some advanced properties can be set from the properties view, for e.g. statemachine base rate, with the added facility to set properties deleted from the view in canvas.

Below is a list of the properties which can be set for the statemachine elements, how to set them (especially canvas view), and advantages of using one view over the other.

- **Statemachine base rate** The statemachine base rate can be set only from the properties view, which can be brought to front by clicking anywhere on the canvas and setting the value of "Base Period" property as shown in Figure 4.8.

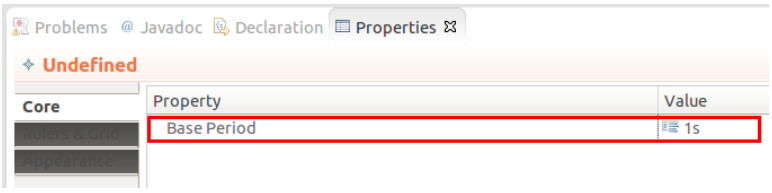

Figure 4.8: Setting statemachine base rate

- **State, Region, and GlobalCode box names** To set a name from from the Canvas view, you simply need to select the default name and change it to the desired name. Initial and Final states name have the default names start and end placed outside their graphical representation. All other states, the regions, and globalcode boxes have default name placed inside the container, which needs to be selected before moving on to change the name. To set a name from the properties view, you simply need to select the desired element to bring its properties view to front where you can change its name. One advantage that properties view does have is that you still change the name of the InitialState or FinalState in case they have been deleted from the canvas view.

- **TransitionInfo** The TransitionInfo can be set both from the canvas and properties view. For the canvas view, simply select the default value of TransitionInfo which is `[true]` and type in the desired transition information. The same can also be done from the properties view, but if you want multiline TransitonInfo, then the property value must be set from the Canvas.

Figure 4.9: Transition properties view

- **Actions and Code in GlobalCode boxes** To add actions to the states or code to GlobalCode boxes from the canvas view, first select the state, and then with your mouse close to the bottom of the element (in case of SuperState slightly above the partitioning line) click again to bring the cursor to front as in Figure 4.10 where you can type in the desired actions or code.

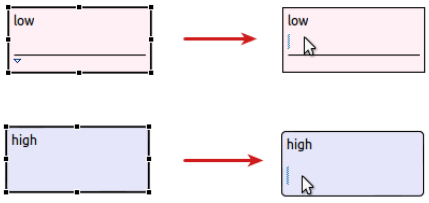

Figure 4.10: Add actions or code

While actions and code to the GlobalCode boxes can be added from the properties, multiline textfields are not possible from here. But the properties view does allow you to specify the "In header file" property in GlobalCode box as shown in Figure 4.11, which is not possible from the Canvas view.

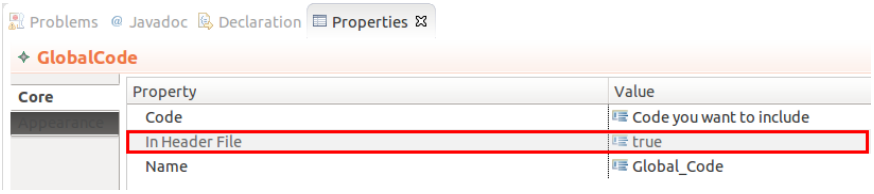

Figure 4.11: GlobalCode box properties view

4.5.1.4 Code generator configuration

Statemachines are created with the Statemachine editor and saved in '`*.statemachine`' files. The '`*.statemachine_diagram`' files only contain the layout. They aren't used for code generation. If you want to use the statemachine, copy Listing 4.12 and adjust the name and file path.

Listing 4.12: Statemachine configuration: config.rb

```
1 #DEPENDS ON:
2 counter_statemachine = "counter.statemachine"
3 # "counter" is the statemachine name used for the runtime API
4 $config.statemachines.load("counter", counter_statemachine)
```

You can also define statemachines in your config file. This is not the official way, but it can be useful. We provide a DSL (domain-specific language), so you don't have to use the Java classes. You can find an example in appendix A.3.1.

4.5.2 CAN communication generator

4.5.2.1 Motivation

The CAN code generator generates code for all parts of the CAN communication process. It generates methods for initialization and for sending and receiving messages. It can also create operating system tasks to send messages periodically.

Global variables, accessed through the operating system, are used for reading and writing the signal values when sending or receiving messages.

The specification of the messages and their signals can be parsed from a DBC file.

This section only covers the basics of the CAN code generator. For a detailed description please see the seperately shipped CAN Code Generator User Guide.

4.5.2.2 Configuration

The configuration is done with a JRuby script.

There are many points where custom code can be included in the generated code and many parts can be replaced by custom code. To be able to use custom code effectively, it is necessary to understand how the generated code works and how it is structured.

DBC

The specification for the messages, signals, etc. are stored in DBC objects. They are created from Java classes from the package `de.upbracing.dbc`. The objects are created by the DBC parser from a DBC file.

A description of the different DBC objects:

DBC The root object, containing the ecus, messages and valuetables. There must only be one DBC object.

DBCEcu Describes an ECU. It also contains the messages and signals that are sent and received by the ECU.

DBCMessage Describes a message. It also contains the signals and the ECUs, that can transmit the message.

DBCSignal Describes a signal. All important information about the structure of the signal, like length, endianness, factor and offset are stored here. It also contains the messsage it belongs to, a list of ECUs by which it can be received and the name of the value table if it uses one.

DBCValueTable Maps strings to other strings. It is used to map numeric values to meaningful strings, because signals can only contains numeric values.

As the DBC objects are simply Java objects, they could be created by hand in the JRuby config file, but usually they are created by parsing a DBC file.

To parse a DBC file, simply call the `parse_dbc()` function with the DBC filename as the argument and assign its return value to the configuration object. The CAN generator needs to know the CAN node name. It can be obtained by reading the ECU list and selecting the ECU. See Listing 4.13 for an example.

Listing 4.13: Calling the DBC parser and selecting ECU

```
1 $config.can = parse_dbc("can_final.dbc")
2 $config.ecus = read_ecu_list("ecu-list-example.xml")
3 $config.selectEcu("Lenkrad-Display")
```

Alternatively, you can set `use_can_node` directly to the node name. This way you don't need an ECU list file. See Listing 4.14.

Listing 4.14: Directly selecting the CAN node

```
1 $config.can = parse_dbc("can_final.dbc")
2 $config.use_can_node = "examplenode"
```

Configuration Files

The DBC objects only store the information that was parsed from the DBC file, but have no further configuration properties like code replacements. To configure them for the code generator, they are converted to configuration objects, which are subclasses of the DBC objects.

- DBC → DBCConfig
- DBCEcu → DBCEcuConfig
- DBCMessage → DBCMessageConfig
- DBCSignal → DBCSignalConfig

The DBC objects are automatically converted to configuration objects when the DBC parser output is assigned to `$config.can`.

Sending

The code generator creates a send method for each message. They are placed in the can.h file. You can see the definition of a generated send method in Listing 4.15.

Listing 4.15: Send method definition

```
1 inline static void send_<MessageName>(bool wait, <params>)
```

When the parameter `wait` is true, the method blocks until the message has been sent.

A message offers several properties to configure the sending. To access such a property in the JRuby file, use `getMessage()` or alternatively `msg()` and then simply set the property as in Listing 4.16.

Listing 4.16: Accessing a message property

```
1 $config.can.getMessage("MessageName").mobDisabled = true
```

You can also use the the `can_config()` method to set properties. You can call this method with `can_config(objspec, value_map)` or `can_config(objspec, key, value)`. The `objspec` argument can be used to specify a message, signal or MOb. See Listing 4.17 for an example.

Listing 4.17: Using `can_config`

```
1 can_config('msg(Shift_Up)', 'tx_mob', 'MOB_Shift')
2 can_config('msg(CockpitBrightness)', 'use_general_transmitter')
3 can_config('signal(Gang)', {"put_value" => "update_gear(value);"})
4 can_config('mob(MOB_Bootloader_1)', 'disabled')
```

A description of sending related properties of messages:

aliases A message can have aliases. They can be added by calling `addAlias("aliasname")` on the `DBCMessageConfig` object. The aliases are added in addition to the regular message name to three enums: `CAN_msgID`, `CAN_isExtended` and `MessageObjectID`.

mobDisabled Can be set to true or false. When set to true, the MOb (Message Object) for sending this message will not be initialized by the `can_init_mobs()` method. It will still reserve a MOb and create the MOb initialization methods.

noSendMesssage Can be set to true or false. When set to true, no send method will be generated, but a MOb is still reserved for sending (if not disabled). This is useful if you want to implement your own send method.

rtr Can be set to true or false. When set to true the RTR (Remote Transmission Request) bit is set for the message. This means that the ECU does not transmit any data with this message, but requests the other ECUs on the CAN bus to do so.

usingGeneralTransmitter Can be set to true or false. When set to true, the message will not have an individual MOb for sending the message, but use the general transmitter MOb. The general transmitter is not a hardware feature, but a regular MOb, which is shared by all messages where `usingGeneralTransmitter` is set to true. It is useful for messages where timing is not critical.

A signal also offers properties for configuration. To access a signal object and its properties in JRuby, call `getSignal()` on the message object. See Listing 4.18 for an example.

Listing 4.18: Accessing a signal property

```
1 $config.can.getMessage("MessageName").
2             getSignal("SignalName").noGlobalVar = true
```

A description of sending related properties of signals:

noGlobalVar When set to true, the code generator will not create a global variable for the signal. The code generator by default generates a global variable for each signal. It is used to store incoming data and read outgoing data. When you set this to true, you also have to replace the code parts that try to access global variables.

globalVarName Can be set with a string for the global variable name. When this value is not set, the global variable name will be the signal name.

In addition to configuration properties, the messages and signals offer special properties that allow additional custom code to be included in the generated code or to replace parts of it.

Please see Figure 4.12 for an overview of the send method and where the custom code is added or replaced.

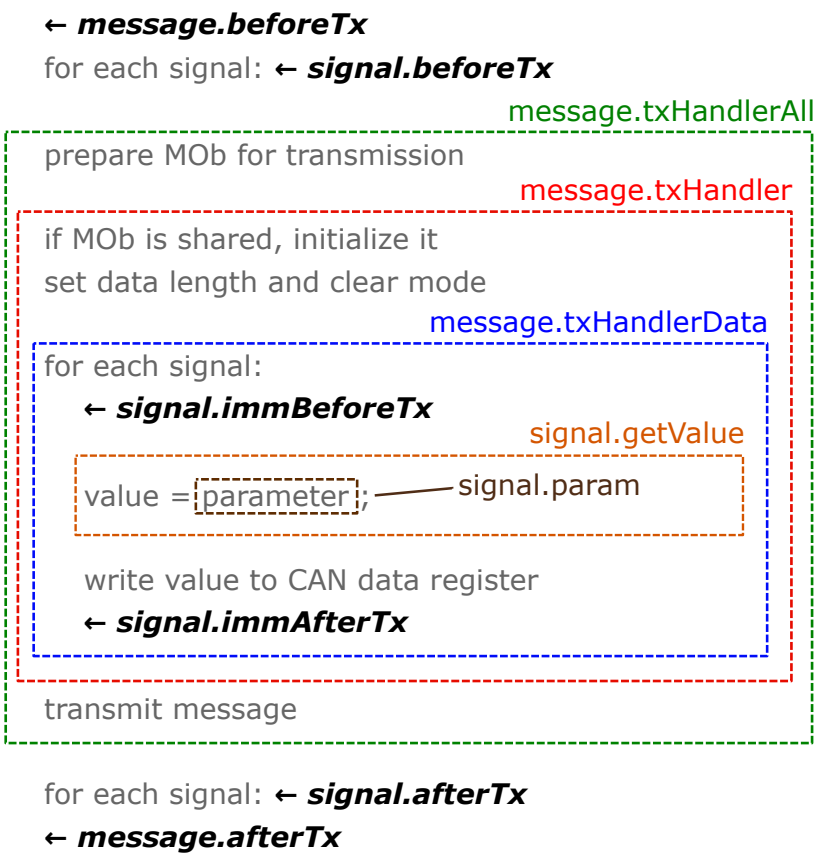

Figure 4.12: Overview of a send method and possible code replacements

Receiving

The generated code receives CAN messages with an interrupt service routine (ISR). It is one large method for all messages in which it checks which message has been received. It then extracts the signals and stores them in their corresponding global variables. The ISR is placed in the can.c file.

A description of reception related properties of messages:

rxMob This sets the name of the MOb that is used to receive this message. It is useful if you want to receive multiple messages with one MOb. By default, the message name is used as the MOb name.

A description of reception related properties of signals:

expected_factor and expected_offset Due to the limited value range of the signals, the actual physical value that the signal represents may be shifted

by an offset and/or multiplied by a factor. The offset and factor are specified in the DBC file for each signal.

The generated CAN code does not convert the signal value to the physical value. You have to take care of this yourself. You should specify the signal factor and offset that you expect with the `expected_factor` and `expected_offset` parameters.

The expected factor and offset are compared to the actual factor and offset from the DBC file for a set of test points, to make sure the error is below a certain threshold. If it is not, a warning generated.

The expected offset is of type `float`.

The expected factor can be either set with the type `rational` in JRuby, as in the example Listing 4.19, or by calling the Java setter `setExpectedFactor(10, 1)` with two integers for the numerator and denominator of the factor.

Listing 4.19: Setting the expected factor

```
1 can_config('signal(Temp_Wasser)',
2             'expected_factor', Rational(10, 1))
```

As with the send method, the generated ISR can also be heavily modified by code replacement properties. In addition to messages and signals, the MObs can also be modified.

Please see Figure 4.13 for an overview of the ISR and where the custom code is added or replaced.

Periodic Sending

The code generator offers a feature to set up messages for periodic sending. For each message, a task is created. When several messages have the same period, they share a task. The periodic tasks are placed in the can.c file.

To set up a message to be send periodically, simply set the `period` value of the message object. See Listing 4.20 for an example.

Listing 4.20: Setting up periodic sending

```
1 $config.can.getMessage("Radio").period = 0.003
```

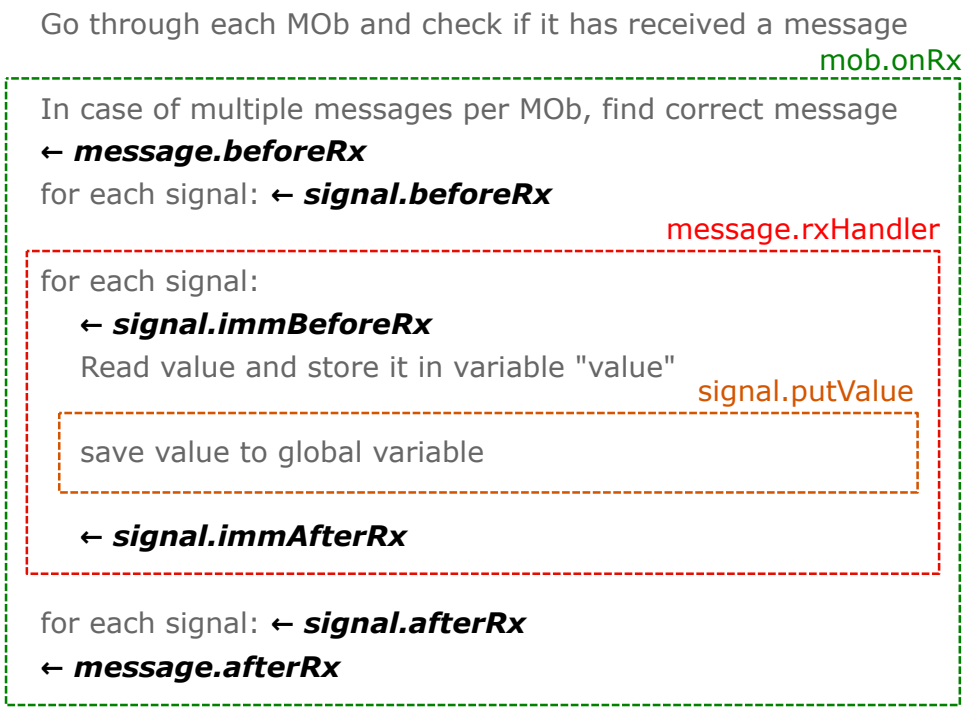

Figure 4.13: Overview of the ISR and possible code replacements

This will set the Message "Radio" to be send every 3 ms. The value can be of type `float`, in which case it is interpreted as seconds, or as a formatted string with a time unit (e.g. "3ms", "10s", etc.).

Please see Figure 4.14 for an overview of a periodic task and where the custom code is added or replaced.

4.5.3 Timer initialization generator

4.5.3.1 Motivation

Configuring timer hardware on the AVR microprocessor can be a time consuming and error prone task. Since many automotive applications require at least one timer per microprocessor (e.g. for triggering the OS counter), we decided to develop a graphical configuration editor in order to ease the configuration process for this repeating task.

To meet the requirements of the project group, all configuration options of the AT90CAN microprocessor family are available in the editor. This especially includes:

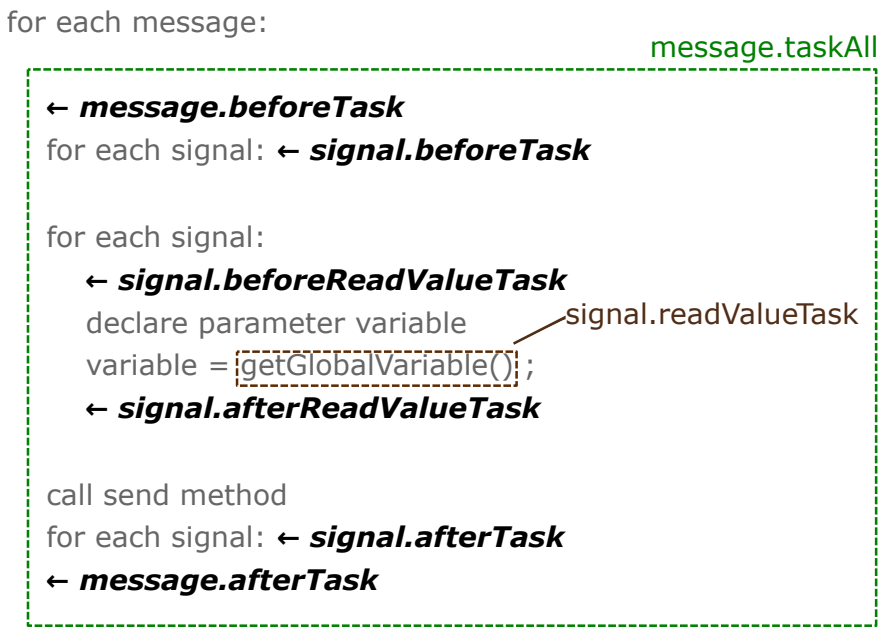

Figure 4.14: Overview of a periodic task and possible code replacements

Clear timer on compare match: In this mode, the timer resets at a configurable frequency. This can be used for triggering functions (e.g. OS counter) at a fixed rate. Furthermore, it allows to output square wave signals.

PWM: The purpose of the three PWM modes available in the AT90CAN processor is to output square wave signals with a variable duty-cycle and optionally a variable base rate (*Phase and Frequency Correct PWM* mode only. See [3] for details).

The next sections will describe the features of the editor in more detail and explain, how to generate C code for the timer hardware using the code generator framework and from within the editor window. Please see the API reference for all runtime functions generated in subsection A.4.3 for details.

4.5.3.2 Using the editor

Installation

The graphical editor is installed from the *EclipsePlugins.zip* archive. This archive contains the UPBracingStatemachineEditor and AVR Timer-Configuration plu-

gins. You need to install *AVR Timer-Configuration* to work with the editor. You can find a more detailed explanation in appendix A.1.

Creating configuration files

The AVR Timer Configuration Editor works on configuration files *(*.tcxml)*. To create a new configuration file, please choose *File ->New ->Other...* from either the Eclipse menu or in the context menu of the folder in which you want the file to be created.

Select *AVR Timer Configuration ->New Timer Configuration* and click *Next >*. Enter the destination folder and filename and click *Finish*. Eclipse automatically starts a new instance of the AVR Timer Configuration Editor after the file has been created.

CPU frequency and error tolerance settings

Figure 4.15 shows the editor view for the file *timer_config.tcxml*. By default, the processor frequency is set to 8MHz and the error tolerance to 5%. The processor frequency is required for register value calculation. The minimum and maximum frequencies are 1Hz and 16MHz. Other values will cause a validation error.

The error tolerance property tells the generator how much quantization error you are willing to tolerate. If the actual period deviates by more than that, this causes a warning.

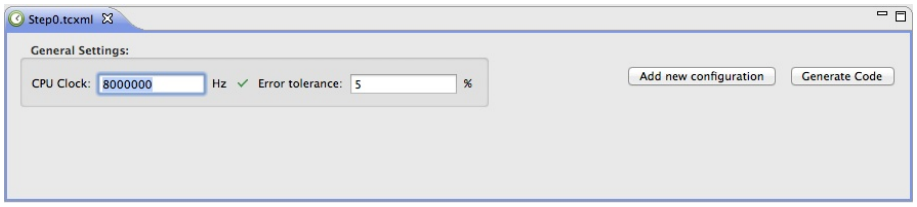

Figure 4.15: Initial configuration file

User input is automatically validated. Icons next to the input fields show their current *validation result*:

Ok ✔
: The user input is valid.

Warning ⚠
: Please move the mouse over the icon to see the tooltip text.

Error ✖
: Code generation is not possible, if the configuration contains errors. Please move the mouse over the icon to see the tooltip text.

Creating use case oriented timer configurations

The button *Add new configuration* in the top area of the editor window creates a new use case oriented timer configuration and displays it in the editor immediately. The configurations are shown in a vertical expand bar. To expand the newly created configuration, click on its title bar. Figure 4.16 shows an expanded timer configuration.

The red area (top) contains an input field for the name of this configuration, a dropdown menu for the timer to use and a another dropdown menu to select the mode of operation. The button *Delete configuration* deletes this configuration from the configuration file.

The green area (left) dynamically displays all configuration options, that are available for the currently selected mode and timer.

For each configuration mode, a descriptive text and image is dynamically displayed in the dark blue area (right). It explains the behavior of the timer in the current operation mode in a textual way.

A waveform preview shows the behavior of the timer in a graphical way. When using *Clear Timer on Compare Match* or *PWM* modes, it also shows the behavior of the output pins.

The following paragraphs show the available configuration options for the different modes of operation for the timer hardware.

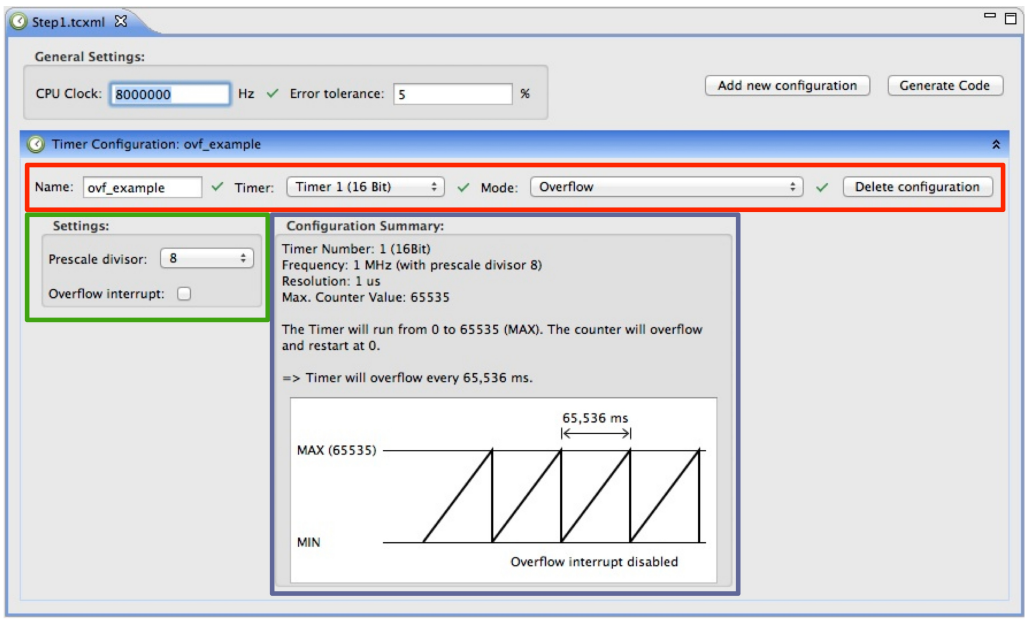

Figure 4.16: Overview of the editor elements

Overflow mode In Overflow mode, the timer repeatedly runs from 0 to MAX (8 Bit: 255, 16 Bit: 65535) and restarts. The overrun frequency can only be influenced by prescaling. Overflow interrupts can be enabled in the editor. Figure 4.17 shows an example overflow mode configuration for timer 1 (16 Bit). Prescaling is set to 8 and overflow interrupts are disabled. If you need these notifications, please enable the *Overflow interrupt* checkbox.

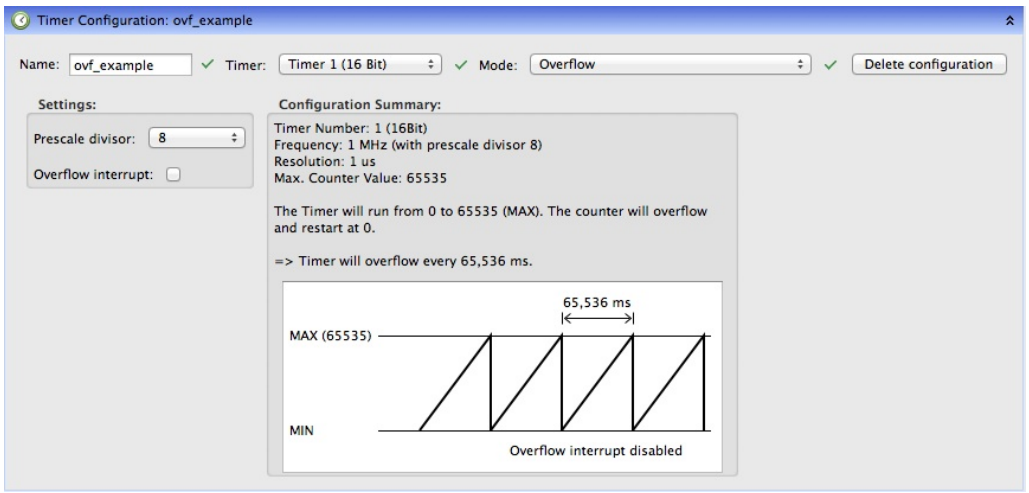

Figure 4.17: Detail view (*Overflow* mode)

The *Configuration Summary* on the right displays detailed information about the overflow configuration. The reset period is shown on top of the waveform. If over-

flow interrupts are enabled, this is displayed below the waveform. Additionally, blue bullets are shown on top of the waveform, if interrupts are enabled.

Clear on compare match mode In Clear on Compare Match (CTC) mode, the timer repeatedly runs from 0 to the user defined TOP value. The reset frequency can thus be influenced by the TOP value as well as by the prescale divisor.

Figure 4.18 shows an example CTC mode configuration for timer 1 (16 Bit). The desired TOP period was set to 10 ms. Although this value cannot be reached exactly (quantized to 9.984 ms) it is validated with *Ok* status, because the deviation from the desired value is less than 5%. The output pin associated with this channel is toggled on every compare match.

Compare channel B is expected to match 1.08 ms after timer reset. Because quantization leads to a value with an error percentage of more than 5, a *Warning* status is displayed.The output pin associated with this channel is cleared on every compare match.

Channel C is expected to match 5 ms after timer reset. Like the period of channel A, the desired value lies within the tolerated range of 5% around the optimal value. The output pin associated with this channel is set on every compare match.

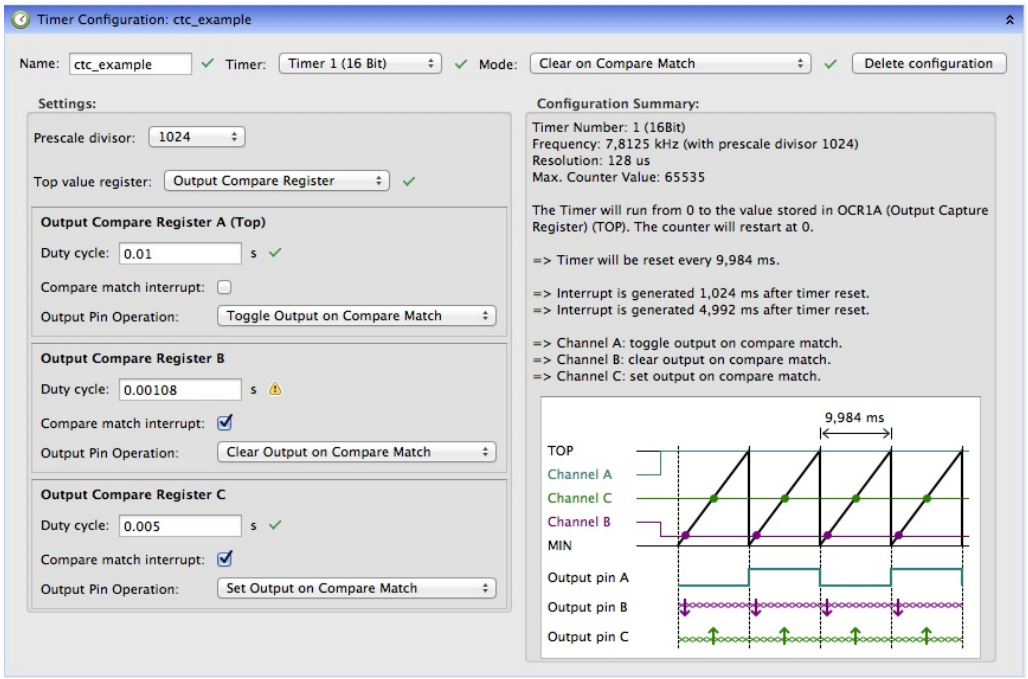

Figure 4.18: Detail view (*Clear on Compare Match* mode)

The *Configuration Summary* on the right displays detailed information about the overflow configuration. Channels A, B and C are displayed as horizontal lines on top of the waveform. Channels B and C generate compare match interrupts, indicated by green and purple bullets. The bottom area of the image shows previews for all the output pins.

Fast PWM mode In Fast PWM mode, the timer repeatedly runs from 0 to TOP. It depends on the timer, which values can be chosen as TOP values. For 8 Bit timers, only the fixed value 255 can be used. For 16 Bit timers, possible fixed values are 255, 511 and 1023. Additionally, *Input Capture Register* or *Output Compare Register A* can be used for defining a custom TOP value.

Figure 4.19 shows an example Fast PWM configuration for timer 2 (8 Bit). Prescaling was set to 8, resulting in a PWM base period of 256 µs (3.90625 kHz). Duty cycle for channel A is set to 100 µs.

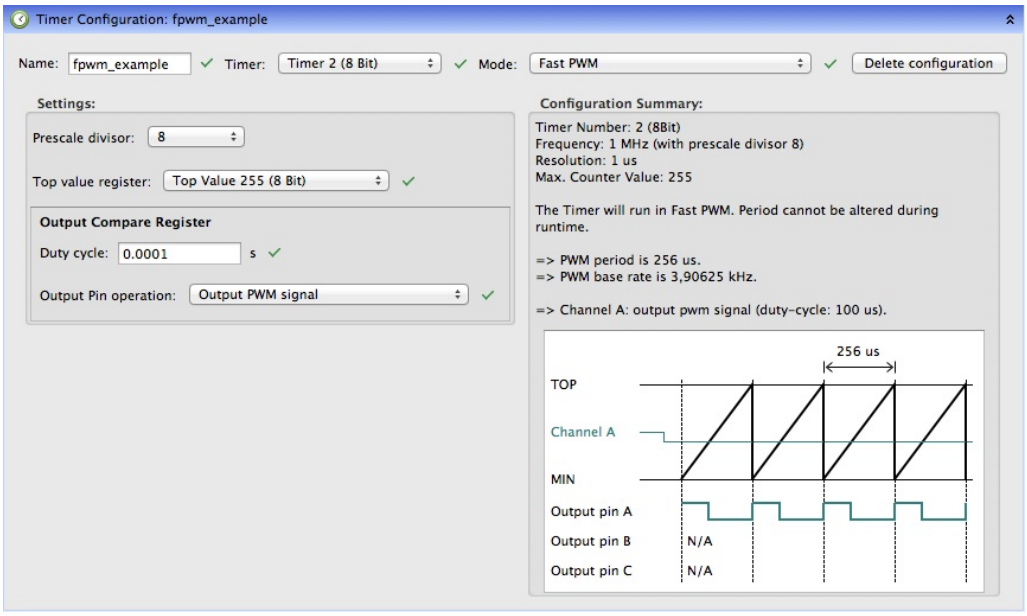

Figure 4.19: Detail view (*Fast PWM* mode)

The *Configuration Summary* on the right displays detailed information about the Fast PWM configuration. The image at the bottom shows the behavior of the timer in this mode. Channel A is displayed as a horizontal line on top of the waveform. The bottom area of the image shows previews for all the output pins. Since channels B and C are not available for this timer, "N/A" is displayed in the image.

Phase Correct PWM mode In Phase Correct PWM mode, the timer runs in dual-slope operation: it counts up from 0 to TOP and then decrements down to 0. This happens in an endless loop. Like in Fast PWM mode, 255, 511 and 1023 are predefined TOP values. Additionally, *Input Capture Register* or *Output Compare Register A* can be used for defining a custom TOP value.

Figure 4.20 shows an example Phase Correct PWM configuration for timer 3 (16 Bit). Prescaling was disabled. *Input Capture Register* was chosen for storing the TOP value. This frees all three compare match channels to output different signals simultaneously. In this example, the PWM base period is set to 1 ms.

The duty-cycle for channel A is set to 300 µs, Channel B ist set 550 µs and C to 800 µs. Channels A and C output non-inverted PWM signals, while the signal of Channel B is inverted.

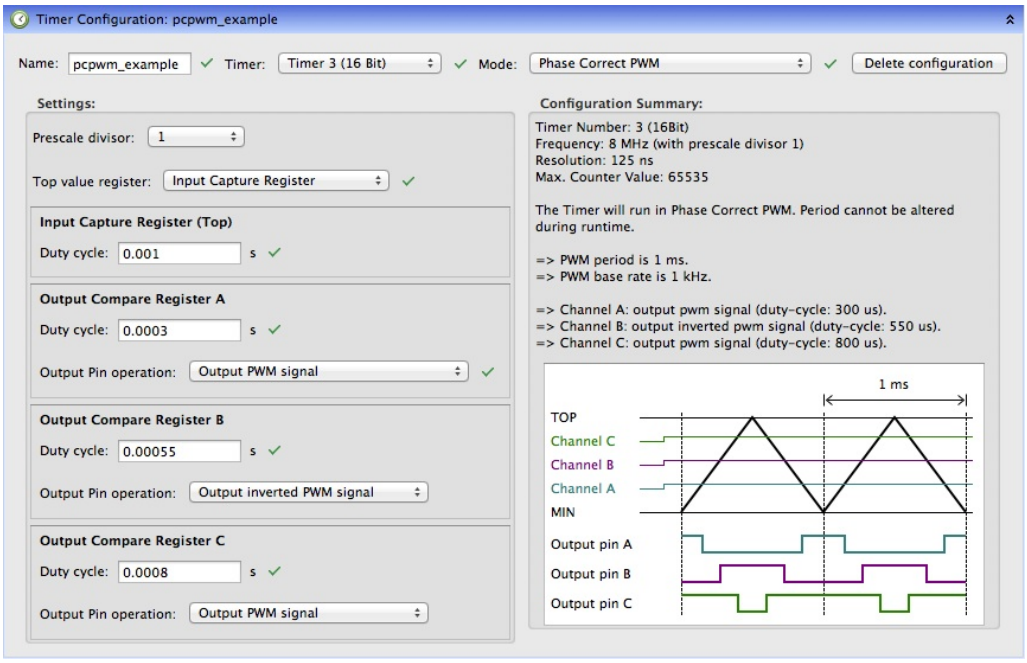

Figure 4.20: Detail view (*Phase Correct PWM* mode)

The *Configuration Summary* on the right displays detailed information about the Phase Correct PWM configuration. The image at the bottom shows the behavior of the timer in this mode. Channels A, B and C are displayed as horizontal lines on top of the waveform. The bottom area of the image shows previews for all the output pins.

Phase and Frequency Correct PWM mode The configuration process of Phase and Frequency Correct PWM mode is the same as for Phase Correct PWM

mode. The only difference between this mode and Phase Correct PWM mode is the number of API functions generated. While Phase Correct PWM mode does not support altering the base rate at runtime, this is possible with Phase and Frequency Correct PWM mode. API functions for altering the base rate are provided (see subsection A.4.3 for details).

This mode is only available for 16 Bit timers. Only *Input Capture Register* and *Output Compare Register A* can be used for storing the top value. No fixed values are usable.

4.5.3.3 Code generator configuration

Using a timer configuration (**.tcxml*) file in the code generation process is quite simple. You only need to specify the absolute or relative path to the timer configuration file:

Listing 4.21: Statemachine configuration: config.rb

```
1 # Timer configurations are created with the AVR Timer-Configuration
2 # editor and saved in '*.tcxml' files. If you want to use the timer
3 # configuration, you load it like this:
4 $config.loadTimerConfiguration("<path_to_config_file>")
```

4.5.4 caRTOS configuration generator

4.5.4.1 Motivation

In the caRTOS configuration files, we define the TCBs and data structures that are used by the operating system. The files are C source files, so any errors are either reported by the C compiler or cause wrong behavior at runtime. This is confusing for new users and experienced users prefer defining a task with one line of code instead of filling C structures. Therefore, we provide a code generator that creates the configuration files for you.

The operating system is the topic of section 4.6. We give a short introduction to its config files, so you don't have to read that section to understand the generator:

- You can enable or disable features of the OS, so you can choose your preferred trade-off of rich features and small size.

- The tasks set is static, so you cannot define new tasks at runtime. All tasks are defined in the configuration files.
- Tasks are enabled by an event. A periodic task has an event that regularly enables it.

The code generator doesn't produce any runtime API by itself because it only generates the caRTOS config file. It will create `Os_config_features.h`, `Os_config_application.h` and `Os_config_application.c`. If you change any setting in `Os_config_features.h` (e.g. enable a feature of the OS), you must recompile the caRTOS library.

4.5.4.2 Code generator configuration

The configuration consists of some global settings (e.g. processor frequency and OS features) and the list of tasks. You can find the details in Listing 4.22.

If you don't set a feature value (the values set with `setConfigValue`), it will have its default value. All of them have sensible default values, so you usually don't have to set them. At the moment, only the USART driver has some options (see Listing 4.22). You can find the most up-to-date list in class `CaRTOSConfigValueProvider` in the project `upbracing-AVR-CodeGenerator`.

The values for `clock` and `tick_period` are required because we cannot choose any sensible default value for your application.

Listing 4.22: caRTOS configuration: config.rb

```
1  # processor frequency is 8Mhz
2  $config.rtos.clock = 8000000
3  # caRTOS should schedule every 4ms
4  $config.rtos.tick_period = "4 ms"
5
6  # enable the USART driver (this is the default)
7  #NOTE This goes into the features header, so you have to recompile
8  #      caRTOS library, if you change it.
9  $config.rtos.setConfigValue("drivers/usart",
10     "USART_ENABLE_DRIVER", true)
11 # set a queue length of 10 bytes (also the default)
12 # This goes into the application config, so you can change it at will.
13 $config.rtos.setConfigValue("drivers/usart",
14     "USART_TRANSMIT_QUEUE_LENGTH", 10)
15
16 # define the task Update which is initially suspended
17 # and will be enabled every 8th timer tick (every 32ms).
18 $config.rtos.addTask("Update", SUSPENDED, 8)
19 # define two more tasks
20 # You can leave out the SUSPENDED argument
21 # because this is the default.
22 $config.rtos.addTask("Increment", 128)
```

```
23 $config.rtos.addTask("Shift", 24)
```

4.5.5 EEPROM accessor generator

4.5.5.1 Motivation

The EEPROM cannot be used like RAM - it must be accessed with special functions. You need a different function depending on the data type (or rather its size). You also have to manage the location of variables in EEPROM.

The code generator uses a struct, so the compiler determines a unique address for each variable. Access to those variable requires working with pointers, which the generator hides behind a simple API: The variable `NAME` can be read and written with `READ_NAME()` and `WRITE_NAME(value)`.[10] You must include the generated header `eeprom_accessors.h` before using the macros.

We use an ECU definition XML file (`ecu-list.xml`) that contains informations about all microcontrollers in our racing car. We read the variable definitions from that file. We are going to write a configuration tool that loads the ECU definition and provides a GUI to alter EEPROM values.

If you don't need EEPROM variable information in other tools, you should put your variable definitions into the main configuration file. We provide a simple JRuby API to define the variables. You will find more information in the next section.

If you want to use default values, you have to extract the EEPROM image from the ELF file and flash it to the microcontroller. This works similar to the FLASH image (the program). The EEPROM will be reset to its factory default value (0xff), if the processor is erased, unless you change the fuse bits. You can find additional information in most AVR tutorials. We suggest that your program treats 0xff (or 0xffff, depending on the data type) as a special value, so it behaves in a sensible way, if the user forgets to initialize EEPROM.

[10] EEPROM has a limited amount of write cycles. Therefore, the names are written in caps, so the developer doesn't confuse it with a global variable access. The prefix `READ` and `WRITE` (as opposed to `get` and `set`) also signify an IO operation.

4.5.5.2 Code generator configuration

Variables in main config file

Listing 4.23: EEPROM configuration: config.rb

```
$config.eeprom.add("var1", "uint8_t")
$config.eeprom.add("var2", "uint32_t")
# custom data type, size is 16 byte, default value is 42
$config.eeprom.add("var3", "very_long_int", 16, 42)
```

Variables in ECU definition file

Listing 4.24: EEPROM configuration in ECU definition file: config.rb

```
#DEPENDS ON:
ecu_list_file = "ecu-list.xml"
$config.ecus = read_ecu_list(ecu_list_file)
$config.selectEcu("Cockpit")
```

Listing 4.25: EEPROM configuration in ECU definition file: ecu-list.xml

```
<?xml version="1.0" encoding="utf-8"?>
<ECUs>
    <ecu name="Cockpit" type="AT90CAN">
        <eeprom>
            <value name="var1" type="uint8_t" default="0" />
            <value name="var2" type="uint8_t" default="255" />
        </eeprom>
    </ecu>
</ECUs>
```

4.5.6 Global variable generator

4.5.6.1 Motivation

One can simply access global variables by their name, but this will break, if the variable is more than one byte in size and an interrupt changes the value at an inappropriate time. Using manual locking is tedious, so we use a code generator to create wrapper methods: You can access the global variable `name` with `getName()` and `setName(value)`. You must include the generated header `global_variables.h` before using the functions.

The wrapper function disables interrupts while it reads/writes the value. It restores the interrupt state afterwards. The code generator doesn't disable interrupts for one-byte variables, so you can use it for all global variables without fear of a performance hit.

The data will be stored in a 'real' global variable. A random string is appended to its name, so you cannot access it directly.

The functions make sure that single reads and writes are interrupt-safe. This works well, if only one task thread (task, interrupt or main program) writes to the variable and other tasks only read. It doesn't protect a read-modify-write operation. If you write `setValue(getValue()+1)`, this will still be unsafe. If you need that, make sure that you do it a critical section.

How can a single read or write go wrong? Let's assume that an interrupt increments an 16-bit unsigned variable from `0x00ff` to `0x0100`. The main program reads the value and we expect it to get either of the two values. However, the main program might read the first byte before the interrupt modifies it and the second one after the modification. In that case, we will get the value `0x0000` which is not even near the value we would expect.

4.5.6.2 Code generator configuration

Listing 4.26: Global variable configuration: config.rb

```
1 $config.globalVariables.add("var1", "uint8_t")
2 $config.globalVariables.add("var2", "uint32_t")
3 # custom data type, size is 16 byte, default value is 42
4 $config.globalVariables.addDeclaration "#include <very_long_int.h>"
5 $config.globalVariables.add("var3", "very_long_int", 16, 42)
```

4.5.7 Pin name generator

4.5.7.1 Motivation

Code like `DDRC |= 0x40;` is hard to understand. You need knowledge of the processor, the board and the application:

- `DDRC` can be used to set the direction (input/output) of pins on port C.

- 0x40 is 0b01000000, so the 'bit-or' operation sets bit 6.
- The 'oil temperature high' LED is connected to pin PC6.
- We configure the pin as an output, so we can later turn on the LED with `PORTC |= 0x40;`. It will be off here because the default value of `PORTC` is 0x00.

The code generator enables a more comprehensible syntax: `OUTPUT(LED_OIL_TEMP_HIGH)`. The reader doesn't have to think about AVR registers and bit arithmetic at all and one can easily grasp what piece of hardware is connected to the pin. The code still doesn't say why you want it to be an output, but the programmer will likely put that into a comment since she doesn't have to describe the trivia.

The program doesn't mention the hardware pin at all. This means that you only have to change the configuration, if you modify the board and connect the LED to a different pin. If you have an EAGLE schematics file for your hardware, you can let the program read the names from that file. In that case you don't have to change the configuration at all.

In the next sections, we tell you how you set the pin names in your configuration file. This enables you to use the macros `INPUT(pin)`, `OUTPUT(pin)`, `HIGH(pin)` and `LOW(pin)` in your C code. You can find the details about the runtime API in appendix A.4.2.

4.5.7.2 Code generator configuration

You can define pin names in your configuration file as shown in Listing 4.27. If you use EAGLE to create your schematics, you can read the names from the schematics file. This option is discussed in appendix A.3.2.

Listing 4.27: Pin names in configuration: config.rb

```
1  # declare some pins, e.g. ERROR1 is 2nd pin on port C
2  pin("ERROR1", "PC1")
3  pin("ERROR2", "PA0")
4
5  # a pin can have more than one name
6  # (You can also call 'pin' more than once.)
7  pinAlias("ERROR_OIL_PRESSURE", "ERROR3")
8
9  # Have you connected 8 similar signals to a
10 # port? You can give them names all at once.
11 # similar to: pin("RPM0", "PB0"), ...
12 # It creates some additional helpers that can
13 # be used to modify all pins.
14 port("RPM", "PB")
15
16
```

```
17  # You can do the same things with the 'pins'
18  # function:
19  pins("ERROR1" => "PC1", "ERROR2" => "PA0",
20      "RPM" => "PB")
21  # Arguments may be processed in any order, so we
22  # must call the function a second time, if we
23  # use aliases (signified by a '=' prefix).
24  pins("ERROR_OIL_PRESSURE" => "=ERROR3",
25      "FIRST_RPM" => "=RPM0")
```

4.6 The Real Time Operating System

4.6.1 caRTOS

The RTOS that is used is called caRTOS (car Real Time Operating System). caRTOS is a static operating system, which means that all the usage of resources is known a priori. Also, no new task or resource can be added at runtime. The RTOS supports 4 task states: Suspended, Ready, Waiting and Running. The tasks are scheduled on the basis of the round robin algorithm. For tasks to communicate with each other, queues are used. For tasks to use the resources in a safe and secure manner, semaphores are present in this RTOS.

Let us now discuss the usage of semaphores and queues in caRTOS.

4.6.2 Semaphores

You have to use a semaphore in order to use a resource. The declaration and operations of the semaphore is explained in the following sections.

4.6.2.1 Semaphore declaration

To declare a semaphore you can use the following macro:

```
SEMAPHORE(name, initial_value, queue_capacity)
```

The parameters are explained as follows:

- Name: Name of the semaphore to be declared.
- initial_value: The number of resources available.
- Capacity: Total capacity of the queue of the semaphore.

An example declaration: `SEMAPHORE(example_semaphore, 1, 10)`

Here the semaphore name is example_semaphore. The number of resource this semaphore guards is 1. The length of the semaphore queue is 10. That means 10 tasks can wait for this semaphore in this queue.

A new type of semaphore is introduced here. When there is a need for multiple access to a resource, it is no longer needed to request for multiple waits on the semaphore. With this new type of semaphore, you can request for multiple access to the resource by specifying the number of access requests in one wait request on the semaphore. This type of semaphore is usually used with the usage of queues.

Here is how it is declared:

```
SEMAPHORE_N(name, initial_value, queue_capacity)
```

The parameters are explained as follows:

- Name: Name of the semaphore to be declared.
- initial_value: The number of resources available.
- Capacity: Total capacity of the queue of the semaphore.

The difference between this semaphore and the normal semaphore comes when wait and signal functions are being called.

In the rest of the document, this semaphore will be referred to as n semaphore.

An example: `SEMAPHORE_N(example_n_semaphore, 1, 10)`

4.6.2.2 Waiting for a semaphore

The following functions are used to wait for the semaphore.

Normal Semaphores

To wait on a normal semaphore use: `sem_wait(name)`

The parameters are:

- name: The name of the semaphore to be waited on.

An example: `sem_wait(example_semaphore);`

N Semaphores

The function used to wait on n semaphore is: `sem_wait_n(sem, n)`

The parameters are:

- name: The name of the semaphore to be waited on.
- n: The number of requested waits on the resource.

An example: `sem_wait_n(example_n_semaphore, 5);`

Basically this is like calling `sem_wait()` n times.

4.6.2.3 Signaling a semaphore

The following functions are used to signal a semaphore.

Normal Semaphores

To signal a normal semaphore use: `sem_signal(name)`

The parameters are:

- name: The name of the semaphore to be signaled.

An example: `sem_signal(example_semaphore);`

N Semaphore

To signal a n semaphore use: `sem_signal_n(name, n)`

The parameters are:

- name: The name of the semaphore to be signaled.
- n: The number of requested to be signaled.

An example: `sem_signal_n(example_n_semaphore, 5);`

4.6.2.4 Asynchronous operations on a normal semaphore

Here we have introduced a new concept of waiting for a semaphore asynchronously. Instead of the task being blocked after calling `sem_wait()`, with the use of asynchronous waiting, the task can carry on with executing without being blocked.

Start waiting

To start waiting on a semaphore use: `sem_start_wait(name)`

The parameters are:

- name: The name of the semaphore to be waited on.

This function returns a token of the type `sem_token_t`. This token is used to check up on the status of the semaphore.

An example: `token = sem_start_wait(example_semaphore)`

Continuing wait

This function is used to check if the semaphore has been granted to the task, using the token issued when `sem_start_wait()` was executed.

Usage is: `sem_continue_wait(name, token)`

The parameters are:

- name: The name of the semaphore to be checked on.
- token: The token issued when starting the wait on the semaphore

This returns `True` if the semaphore has been granted. In this case, this token must not be used again. If this returns `False`, you must try again later, because the semaphore has not yet been granted.

An example: `check = sem_continue_wait(example_semaphore, token)`

Finishing wait for a semaphore

After the completion of the semaphore, use this macro to finish the wait for semaphore. This function cleans up the semaphore queue. It is important to note that you have to signal the semaphore after calling this macro.

Usage of the macro is: `sem_finish_wait(name, token)`

The parameters are:

- name: The name of the semaphore to finish waiting for.
- token: The token issued when starting the wait on the semaphore

An example: `sem_finish_wait(example_semaphore, token); sem_signal(example_semaphore)`

Aborting a wait

In case you need to abort, during the wait for a semaphore, use this macro: `sem_abort_wait(name, token)`

The parameters are:

- name: The name of the semaphore to abort waiting for.
- token: The token issued when starting the wait on the semaphore

It is important to note that `sem_signal()` is called from inside this macro, so there is no need for an explicit call.

An example: `sem_abort_wait(example_semaphore, token)`

4.6.2.5 Asynchronous operation on n semaphores

The operations on n semaphores are usually dealt along with operations on a queue. In this part, we shall still have a look at the asynchronous operation on n semaphores.

Start waiting

Macro: `sem_start_wait_n(name,n)`

The parameters are:

- name: The name of the semaphore to finish waiting for.
- n: The number of requests to be requested.

The working of this macro is same as `sem_start_wait()`

Continue waiting

Macro: `sem_continue_wait_n(sem, token)`

The definition and working is the same as `sem_continue_wait()`

Finish waiting

Macro: `sem_finish_wait_n(sem, token)`

The definition and working is the same as `sem_finish_wait()`

Abort waiting

Macro: `sem_abort_wait_n(sem, token)`

The definition and working is the same as `sem_abort_wait()`

4.6.3 Inter-process Communication (IPC)

The Inter-process Communication is done through a circular queue. The declaration and operations of the queue are explained in the following sections.

4.6.3.1 Queue

To declare a queue you can use the following macro:

```
QUEUE(name, capacity, reader_count, writer_count)
```

The parameters are explained as follows:

- Name: Name of the queue to be declared.
- Capacity: Total capacity of the queue.
- reader_count: Number of tasks reading from the queue.
- writer_count: Number of tasks writing to the queue.

An example declaration: `QUEUE(example_queue, 50, 14, 9)`

Here the queue name is example_queue, the queue capacity is 50, 14 tasks will read from the queue, and 9 tasks will write to the queue.

4.6.3.2 Queue Enqueue

To write into a queue use the following macros.

Synchronous Enqueue

To write one byte into a queue use: `queue_enqueue(name, data)`

The parameters are:

- name: The name of the queue to be written in to.
- data: The byte to be written to the queue.

An example: `queue_enqueue(example_queue, "a");`

To write a stream of bytes to a queue use: `queue_enqueue_many(name, count, data)`

The parameters are:

- name: The name of the queue to be written into.
- count: The number of bytes to be written to the queue.
- data: The pointer to the data to be written.

An example: `queue_enqueue(example_queue, 5, *write_in);`

Asynchronous Enqueue

When you want to write to a queue but do not want the task to be waiting for the queue, use asynchronous enqueueing.

To enqueue asynchronously, you have to start the enqueue process by, `queue_start_enqueue (name, no_of_bytes)`, where `name` is the name of the queue, and `no_of_bytes` is the number of bytes you want to write into the queue. This returns a `token`, which is used in the rest of the enqueue process.

Then you have to check if the queue is ready to be written into. You do that with `queue_continue_enqueue(name, token)`, where `name` is the queue name you want to write into, and `token` is the token you obtained when you called `queue_start_enqueue()`. This returns a boolean. If it returns true, then the queue is ready to be written. If it returns false, try again in a bit.

To write into the queue, after waiting for it, use `queue_finish_enqueue(name, token, no_of_bytes, data)`, where `name` is the name of the queue to be written into, `token` is the token you obtained when you called `queue_start_enqueue()`, `no_of_bytes` is the number of bytes you want to write to the queue, and `data` is the data you want to write to the queue.

Example, to asynchronously enqueue 5 bytes of data into queue, test_queue, is given in Listing 4.28

Listing 4.28: Example of Asynchronous Enqueue

```
1
2  sem_token_t queue_token;
3  queue_token = queue_start_enqueue(test_queue, 5);
4  if ( queue_continue_enqueue(test_queue, queue_token) )
5      queue_finish_enqueue(test_queue, queue_token, 5, "tests");
```

To abort waiting for queue use `queue_abort_enqueue(name, token)`, where `name` is the name of the queue you initialized the wait on, `token` is the token you obtained when you called `queue_start_enqueue()`.

Example, to abort a queue enqueue asynchronously on queue, test_queue, is given in Listing 4.29

Listing 4.29: Example of Asynchronous Enqueue Abort

```
1
2  sem_token_t queue_token;
3  queue_token = queue_start_enqueue(test_queue, 5);
4  queue_abort_enqueue(test_queue, queue_token);
```

4.6.3.3 Queue Dequeue

To read from a queue use the following macros:

Synchronous Dequeue

To read one byte from the queue: `queue_dequeue(name);`

Parameter name is the name of the queue to dequeue from.

An example: `item = queue_dequeue(our_queue);` puts a byte of data into the variable item.

To dequeue a stream of bytes from a queue use: `queue_dequeue_many(name, count, data );`

Parameters:

- name is the name of the queue to be read from.
- data is the pointer to the buffer where the data should be written to.

An example: `queue_dequeue_many(our_queue, 5, write_out);` writes 5 bytes of data from the queue into the buffer write_out.

Asynchronous Dequeue

When you want to read from a queue but do not want the task to be waiting for the queue, use asynchronous dequeueing.

To dequeue asynchronously, you have to start the dequeue process by, `queue_start_dequeue (name, no_of_bytes)`, where `name` is the name of the queue, and `no_of_bytes` is the number of bytes you want to read from the queue. This returns a `token`, which is used in the rest of the dequeue process.

Then you have to check if the queue is ready to be read from. You do that with `queue_continue_dequeue(name, token)`, where `name` is the queue name you want to read from, and `token` is the token you obtained when you called `queue_start_dequeue()`. This returns a boolean. If it returns true, then the queue is ready to be read. If it returns false, try again in a bit.

To read from the queue, after waiting for it, use `queue_finish_dequeue(name, token, no_of_bytes, data)`, where `name` is the name of the queue to be read from, `token` is the token you obtained when you called `queue_start_dequeue()`, `no_of_bytes` is the number

of bytes you want to read from the queue, and `data` is the buffer where you want to write from the queue.

Example, to asynchronously dequeue 5 bytes of data into queue, test_queue, is given in Listing 4.30

Listing 4.30: Example of Asynchronous Dequeue

```
1
2  sem_token_t queue_token;
3  queue_token = queue_start_dequeue(test_queue, 5);
4  if ( queue_continue_dequeue(test_queue, queue_token) )
5      queue_finish_dequeue(test_queue, queue_token, 5, "tests");
```

To abort waiting for queue use `queue_abort_dequeue(name, token)`, where `name` is the name of the queue you initialized the wait on, `token` is the token you obtained when you called `queue_start_dequeue()`.

Example, to abort a queue dequeue asynchronously on queue, test_queue, is given in Listing 4.31

Listing 4.31: Example of Asynchronous Dequeue Abort

```
1
2  sem_token_t queue_token;
3  queue_token = queue_start_dequeue(test_queue, 5);
4  queue_abort_dequeue(test_queue, queue_token);
```

5 Microcontroller tools – Technical documentation

Contents

5.1 Introduction

In this document, we write down all important details about the internals of our tools. After reading it, you will be able to extend caRTOS (the car real-time operating system), improve existing code generators and also write new code generators that can be used with our framework. You don't have to read this document if you only want to use the tools.

We assume that you already know how to use caRTOS and the code generators. If you don't, please read the user documentation. Furthermore we assume good knowledge of Eclipse and Java. To work on some parts of the software, you also need to know Ruby and Eclipse plugin development. For parts of the software, you need additional knowledge. For your convenience, we provide a list of technologies we use (see below).

In section 5.3 we present our real-time operating system. It can run tasks in parallel and provides semaphores and queues for task synchronization and inter-task communication. In section 5.2 we describe our code generators. It starts with an overview of the code generation process and goes on to the existing code generators. You can learn how to write your own generators in subsection 5.2.3. Finally, we present our tests in section 5.4.

In this document, we refer to additional documentation and programs. All of them live in our version control system (Git): `programs.git` contains the software, `documentation.git` contains the documentation. You should have received a copy of both repositories together with this document. If you don't have them, please contact Benjamin Koch <bbbsnowball@gmail.com>.

5.1.1 Used technologies

We have build our tools on top of existing tools. You should know how to use these tools, if you are going to extend our programs. If you only work on some parts of our programs, you don't have to know all the tools. We give an overview here, so you know what to expect. We provide some links in case you want to learn about a topic.

[1] Datasheet: `http://www.atmel.com/Images/doc7679.pdf`
Tutorial: `http://hackaday.com/2010/10/23/avr-programming-introduction/`

AVR microcontroller family[1] We use AVR microcontrollers for most control units in our racing car. All the tools we present in this document work with these controllers. We always use the CAN bus bus to connect the units, so we (almost) exclusively use AT90CAN128 controllers.

GNU toolchain for AVR[2] We program the microcontrollers using the GNU toolchain, i.e. the compiler AVR-GCC, the AVR libc library and the binutils-avr linker.

Eclipse IDE[3] We use Eclipse to edit and build the source. You can use another IDE for most parts. We sometimes use AVR Studio for debugging. The code works well with it. However, you definitely need Eclipse for the graphical editors (statemachine and timer).

Eclipse Plugin Development The timer editor is an Eclipse plugin that uses the SWT[4] toolkit for its UI. The statemachine editor is generated by the Eclipse Modelling Tools (EMF, GMF and Eugenia[5]). The tools are really easy to use, so you probably can get away with little knowledge, unless you want to extend the editor in ways that the tools don't support.

Java Our tools are written in Java.

JRuby Ruby[6] is a dynamic programming language. JRuby[7] is the Ruby implementation for the Java virtual machine. You should be able to read Ruby scripts. You should know Ruby well, if you want to extend one of these parts: DBC parser, EAGLE pin name extractor, some helper methods for the configuration, the hardware test build environment.

GNU make[8] Eclipse uses 'make' to build C/C++ projects. It automatically creates the build scripts, so we only extend them a little bit to execute our code generators.

rake[9] 'rake' works like 'make', but it uses a Ruby configuration file. This enables 'intelligent' build scripts, so we use it for our hardware tests. You only need

[2] Compiler: `http://gcc.gnu.org/wiki/avr-gcc`
AVR libc: `http://www.nongnu.org/avr-libc/`
for Windows: `http://winavr.sourceforge.net/`
[3] `http://www.eclipse.org/`
[4] `http://www.eclipse.org/articles/Article-Your%20First%20Plug-in/YourFirstPlugin.html`
[5] `http://www.eclipse.org/epsilon/doc/articles/eugenia-gmf-tutorial/`
[6] `http://www.ruby-lang.org/`
[7] `http://jruby.org/`
[8] `http://www.gnu.org/software/make/`
[9] `http://rake.rubyforge.org/`

to know 'rake', if you want to want to extend this build environment. You don't need to know it to write or run hardware tests.

5.1.2 Directory structure

If you want to read our code or extend it, you should know where you find the program parts that you are most interested in. In Figure 5.1 we provide an overview of the most important parts.

```
programs.git
├── avr-programs ..... programs running on our control units
├── caRTOS ..... the real-time operation system
│   └── os ..... caRTOS source files
├── code-generation
│   ├── CodegenExample ..... example programs (Alice and Bob)
│   ├── dist ..... binary distribution files (JAR files)
│   ├── java-parser-tools ..... parser library
│   ├── StatemachineEditor
│   │   ├── Statemachine ..... statemachine model (EMF)
│   │   ├── Statemachine.diagram ..... statemachine editor (GMF)
│   │   └── StatemachineFeature ..... Eclipse Plugin description
│   ├── statemachine-test-pc ..... test random statemachines on PC
│   ├── TableGeneratorExample ..... example, see subsection 5.2.3
│   ├── tests ..... hardware tests
│   ├── tests-java-helpers ..... Java classes used by the hardware tests
│   ├── upbracing-AVR-CodeGenerator ..... main code-generator project
│   ├── upbracing-AVR-TimerConfigurationEditor ..... Eclipse editor
│   ├── upbracing-AVR-TimerConfigurationModel ..... Timer model
│   └── upbracing-AVR-TimerConfigurationModel-test ..... Timer tests
├── core ..... (part of RemoteCockpit)
├── gui-Android ..... (part of RemoteCockpit)
├── gui-common ..... (part of RemoteCockpit)
└── gui-pc ..... (part of RemoteCockpit)
```

Figure 5.1: Directory structure

If you want to build the programs, see appendix B.1 .

5.2 Code generators

Our code generators are written in Java and shipped in one big JAR file. We use these components:

Code-generation framework The framework calls all active generators and provides common services, e.g. integration with the build process.

Generator plugins Each generator is provided by a plugin. You can add a new one without changing the existing code.

Templates We use Java Emitter Templates (JET) to generate code. A generator may use more than one template.

Configuration We store the configuration in Java objects that are created when the config file is loaded.

JRuby The main configuration file is a JRuby script. It runs in the java process and builds the configuration objects.

In the next section, we give a more detailed overview of the system. We show how to create a new generator plugin in subsection 5.2.3 and the following sections have information about existing plugins.

5.2.1 The code generation process

The workflow for the user is like this:

- She creates the main configuration file (JRuby script) with a text editor and probably some additional files with other editors (e.g. our Eclipse plugins). The main file loads all additional files that are going to be used.
- The user starts the build process which passes the main config file to our code generator program. The code generators wave their wands and magically some header and source files appear on the harddisk.
- Those files are compiled and linked to other files provided by the user and some libraries (including caRTOS). This is part of the normal build process and uses the normal AVR toolchain.

In this section, we will shed some more light on the "magic" parts of the process. In Figure 5.2 you can see that there are four steps inside the code generator process:

Load Configuration Files We create a JRuby context, put some helpers and an empty configuration object into the context and execute the config file. It uses the provided helpers to load additional files and extends the configuration object.

Validate We check all user input and fail earlier. That way, we can avoid exceptions in the generator code and compiler errors.

Preprocess The generators can modify the configuration before it is sent to the templates. The config objects are designed for the user, so we may want to transform them to be more suited for the code generator. If part of the code is generated by another code generator, we will extend its configuration in this step.

Generate Code The preprocessed and validated configuration is passed to the templates. Each template invocation generates one file.

We demonstrate this process with a small example: Fred wants to heat up the bathroom before he gets out of bed. Next to the radiator he has an AVR board that can measure the temperature and control the valve. He doesn't trust wireless connects, so he replaces the wifi chipset in his mobile phone by a CAN transceiver and connects that to the AVR. The Android stuff is easy, but he needs our help with the AVR part. This obviously is a job for our CAN generator.

We create a simple CAN configuration (DBC file) with two messages (current temperature and commanded valve position) and reference it in this config file:

Listing 5.1: Use CAN configuration in config.rb

```
1 #DEPENDS ON:
2 dbc_file = "heat_control.dbc"
3 $config.can = parse_dbc(dbc_file)
4 $config.use_can_node = "bathroom"
5 add_code("message(ValveControl)", "after_rx", <<-END_CODE)
6   control_valve(getCommandedValvePosition());
7 END_CODE
```

Now, Fred clicks on "Build project" in Eclipse and this is what happens:

Dependency tracking

Eclipse uses GNU make to build the project. Most of its control files are generated by Eclipse, but we have some additions in `makefile.targets`. The most important part is this:

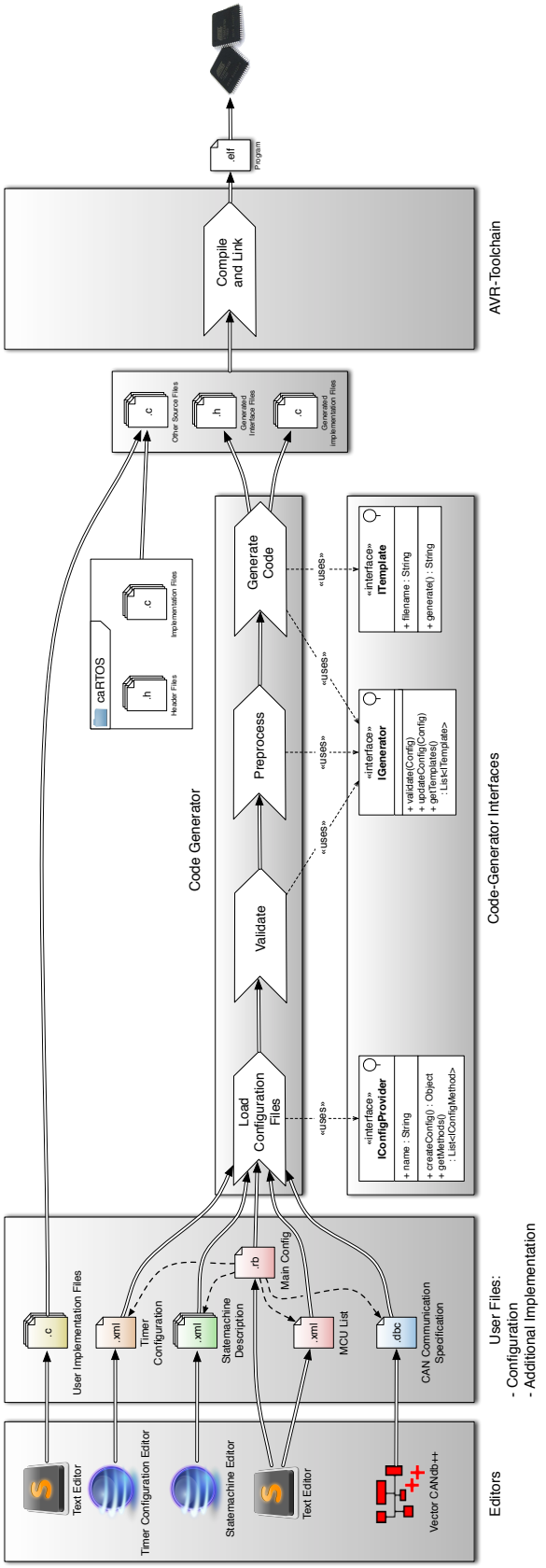

Figure 5.2: Code generation toolflow with internals

Listing 5.2: Makefile extension for GNU make: makefile.targets (simplified)

```
1 # before we build any objects (thus compile C files), we must generate
      the code
2 $(OBJS): $(shell $(CODE_GENERATOR) -w)
3
4 # rule to execute the code generator
5 CODE_GEN_DUMMY=$(TARGET_DIR)/last_gen_time
6 $(shell $(CODE_GENERATOR) -w): $(CODE_GEN_DUMMY)
7
8 $(CODE_GEN_DUMMY): $(CONFIG) $(shell $(CODE_GENERATOR) -D)
9     $(CODE_GENERATOR) && touch $(CODE_GEN_DUMMY)
```

`$(CODE_GEN_DUMMY)` is an empty file. It is used to track the time we last called the code generator. `$(CODE_GENERATOR)` is how we execute the generator including arguments for the target directory and where the config file lives.

When we call the generator with the `-w` flag, it will determine which files it will generate. It asks all code generators for a list of those files and prints them in a format suitable for `make`. When we call it with `-D`, it will track down all dependencies and print a list of them. Dependencies are all files that might cause the output of the generator to change:

- the configuration file
- files referenced by the configuration file[10]
- the code generator classes[11]

Dependency handling is provided by the code generator framework. The code generators only have to provide a list of generator files. The framework instantiates all generators and calls the appropiate methods on them. In the next section, we will see how the framework loads the generators.

The generator framework

If `make` decides that the generated files should be refreshed (or created), it calls the code generator without any dependency flags. The class `Main` parses the commandline arguments and stores them in `CodeGenerationMain.Arguments`. It then calls some method of `CodeGenerationMain` to do the heavy lifting.

[10] Referenced files must be marked with a "`#DEPENDS ON:`" annotation, as running the config would take too long. In the future, we may change the framework to automatically keep track of all dependencies and store them in the dummy file.

[11] This list can be very long, as it includes everything on the CLASSPATH. You can use the flag `-d` instead of `-D` to only get the other dependencies.

Next, the generators are loaded. We don't want a list of generators in some class because that would be hard to change. Instead, we use the ServiceLoader class.[12] It looks for a special file in all CLASSPATH entries[13] and loads all classes that it finds. This means that you can add another generator plugin by putting another JAR file on the CLASSPATH - you don't have to modify any existing code.

The classes implement a common interface. Most generators enhance the configuration object with `IConfigProvider` and register files for generation with `IGenerator`. You can see those interfaces in Figure 5.2. `IGenerator` provides an instance of `ITemplate` for each generated file.

After loading the generators, the code generation framework loads the configuration file.

Loading the configuration

The main configuration file is a JRuby script. This means that we don't have to provide our own parser. The script can do lots of interesting things, as JRuby is a full-featured programming language. Most importantly, the script can change Java objects and call functions that we provide.

We provide an empty configuration object (an instance of `CodeGeneratorConfigurations`) that the script can fill with information. It used to have some setter and getter methods for each generator, but this meant that we had to change it for each new generator. Instead, generators can extend the object at runtime. The framework calls the `extendConfiguration` method of all `IConfigProvider` instances and they use a `ConfigurationExtender` to add custom state, methods and properties. For the state, we use an instance of `ConfigState` instead of a name, so we can provide a type-safe interface and avoid name clashes. This also means that you cannot access a state, if the `IConfigProvider` doesn't give the `ConfigState` instance to you (information hiding).

Of course, we cannot extend a Java object at runtime. This means that we have to access the extensions with code like `config.getProperty("statemachines")` and `config.call("selectECU", "bathroom")`. This is unnatual and ugly, but fortunately we can do better in JRuby. `config-object.rb` registers all configuration extensions as

[12] provided by Java, see `http://docs.oracle.com/javase/6/docs/api/java/util/ServiceLoader.html`JavaDoc of `java.util.ServiceLoader`

[13] For the `IGenerator` interface, this is the file `META-INF/services/de.upbracing.code_generator.IGenerator`

JRuby methods, so you can use the usual syntax: `$config.statemachines` and `$config.selectECU("bathroom")`.[14]

For our "bathroom" example, this is what happens: The framework creates an empty config object and lets the generators register their extensions. It loads some additional helpers which have been implemented in JRuby. You can find them in the package `de.upbracing.code_generation.ruby`.[15] The config script calls `parse_dbc` to load the DBC file and `add_code` to add some code that will be executed, when a new valve position is received. It can also directly change the `$config` object.

The generation process will abort in this step, if the config script contains a syntax error or throws an exception. In our case everything is fine, so we continue with the next step.

Validation

We want to support the user with good error messages. This means that we want to avoid exceptions in the code generator (e.g. NullPointerException because of a missing value) or syntax errors in the generated code (caused by invalid input that is copied verbatim). To keep the code generators small, we have a seperate validation step.

The validation is actually quite simple: We call the `validate` method of all generators and they can either return `true` (success) or `false` (code generation not possible). Messages to the user (errors and warnings) are reported using the `Messages` object (`config.getMessages()`). It supports different severities, nested context and custom formatters. Please have a look at its JavaDoc to learn all the details. You can create a context for your part of the validation and later query whether there have been any errors in this context.

For our example, we make sure that there are enough hardware resources (Message Objects). There are some more checks. Fortunately, all of them succeed, so we continue to the next step.

[14] We didn't have to change even a single line of our JRuby code when we switched to the `IConfigProvider` system.

[15] At the moment, there is no way for a plugin to provide helpers at this level.

Preprocessing

During preprocessing, each generator has the chance to modify the configuration. This is useful for three reasons: Removing "syntactic sugar", building helper models and inter-generator cooperation.

A feature of a programming language is called "syntactic sugar", if it simplifies the expression of a particular solution compared to other languages. The code with "sugar" is often shorter and easier to read. For example, the condition `wait(100ms)` can be replaced by a counter variable. We provide it anyway, as it allows for code to be easier to write and maintain. We rewrite it in the preprocessing step, so the user can use sugar, but the templates don't need to become more complex because of that.

The config objects are designed for the user. We often want to add some information that is used by the template. For example, the template generates a function for each event in a statemachine, but in the statemachine model this information is only available in the transitions. We build that list during preprocessing, so we don't have to do it in the template.

The other reason is inter-generator cooperation: Some parts of the generated code is in the realm of another code generator. In our example, the CAN bus generator needs a global variable to store the received valve position. It could easily generate that variable itself, but the global variable generator can do it better (in an interrupt-safe way). Therefore, the CAN bus generator simply adds the variable to the global variable configuration.

We must make sure that preprocessing for the global variable generator happens after the CAN bus generator modifies its configuration. The CAN bus generator states that it needs the global variable generator (see `getUsedGenerators` method) and the framework calls them in the right order.

In addition, we must make sure that the global variable generator is present. This is really simple: When the CAN bus generator modifies the configuration, it has to use several classes of the global variable generator, so Java will complain, if it cannot find that generator.

The preprocessing itself is quite simple: The framework calls all `updateConfig` methods (in the right order, of course) and they can do arbitrary modifications to the configuration. The generators may return an arbitrary object that will be passed to the templates. This can be used to pass additional information without cluttering the config objects. For example, you can transform the configuration into

a new form that is more useful for code generation. The existing code generators don't need this and simply return `null`.

The configuration is validated again to make sure that preprocessing didn't break it. The second validation knows that it is the second one, so it can adjust the checks accordingly. If everything is fine, we can finally generate the code.

Generating code

Each generator provides a list of file names and templates. The framework calls the templates and writes the text into the appropriate files. The template simply returns a String. Most of our templates are implemented as JET templates, but you can use an ordinary Java class or another templating engine.

Most of the time only a few output files really change. The framework doesn't touch an output file, unless the new content is different. That way, the build process doesn't have to refresh dependent files.

If a generator hasn't been configured at all, we still generate the files. Of course, they are empty in that case. However, there is a good reasons to generate them: The user might want to temporarily disable certain aspects of the generator configuration. If we don't generate the empty files, the build process might still look for them and fail. Even worse, the build process might use an old file and produce an incorrect result. The user shouldn't have to change the build configuration, especially if the configuration contains some code that automatically enables or disables a generator.

In our example, we get two files for the global variables and three files for the CAN interface. They reside in the newly generated folder "gen". We compile and link to the source files and include the header files where we need them. As we are using Eclipse, we simply refresh the project and Eclipse picks up the new folder and adds all source files to the build script. We build the project a second time to include the new files.

5.2.2 Existing code generators

The user documentation describes the purpose and usage of each code generator. We won't repeat that here. For most of them, we only give you some pointers to the implementing classes.

You can find those classes in Table 5.1. Generators live in namespace `de.upbracing.code_generation.generators`, config objects and providers are in `de.upbracing.code_generation.config` and the templates are in the folder `templates`.

In the table, you can also see the files that are generated by the templates. All the generated files go into the target folder (provided by the user as a command-line argument). We usually call this folder `gen`.

Some generated code uses code in the project "upbracing-common". This code hasn't been developed in the project group. The CAN bus generator uses the CAN bus library in `can_at90.h` and `can_at90.c` and the pin generator includes `Pins.h` which defines some helper macros.

5.2.2.1 Preprocessing

The code generators can do some work in `updateConfig`. This is used to keep the template small and tell other generators to do part of the work.

The CAN bus generator uses the global variable generator to make variables for received signals. It also uses the caRTOS generator to create a task for sending messages. In `updateConfig` it also sorts the messages, assigns hardware resources (message objects) and resolves name clashes.

The global variable generator determines the internal variable names and the caRTOS generator assigns a unique ID to each task.

The statemachine generator build helper models in `updateConfig`. You can find the details in subsubsection 5.2.2.4 and in the code.

The other generators don't need a preprocessing step.

5.2.2.2 Timer Eclipse Plugin

The AVR timer configuration program is divided into three parts (see Figure 5.3 for a UML component diagram):

The *Configuration Model + Validator* package containts the subpackage *Data Model*, which stores the configuration data entered by the user. The model can be serialized into and deserialized from xml files. The file extension for timer configuration files is **.tcxml*. The *Validator* subpackage checks the model's data for correctness. Each property is validated and the results can be *Ok*, *Warning* or *Error*. Warnings can occur, if quantization errors lie above the tolerated range

CAN	Generator class	CANGenerator
	ConfigProvider	CANConfigProvider
	Main Config	DBCConfig
	Additional Config	DBC{Ecu,Message,Signal}Config, Mob
	ConfigProvider	ECUListProvider
	Template	can_cfile.jet → can.c
		can_header.jet → can.h
		can_valuetables.jet → can_valuetables.h
EEPROM	Generator class	EEPROMAccessorGenerator
	ConfigProvider	EEPROMConfigProvider
	Main Config	EEPROMConfig
	Additional Config	EEPROMVariable, CType, Variables
		Variable, VariableWithSize
	Template	eeprom_cfile.jet → eeprom_accessors.c
		eeprom_header.jet → eeprom_accessors.h
Global variables	Generator class	GlobalVariableGenerator
	ConfigProvider	GlobalVariableConfigProvider
	Main Config	GlobalVariableConfig
	Additional Config	GlobalVariable, CType, Variables
		Variable, VariableWithSize
	Template	global_variables_cfile.jet → global_variables.c
		global_variables_header.jet → global_variables.h
Pin names	Generator class	PinNameGenerator
	ConfigProvider	PinConfigProvider
	Main Config	PinConfig
	Additional Config	Pin
	Template	pin.jet → pins.h
RTOS config	Generator class	RTOSGenerator
	ConfigProvider	CaRTOSConfigValueProvider
	Main Config	rtos/RTOSConfig
	Additional Config	rtos/RTOS*
	Template	rtos_application_cfile.jet → Os_cfg_application.c
		rtos_application_header.jet → Os_cfg_application.h
		rtos_features.jet → Os_cfg_features.h
Statemachine	Generator class	StatemachineGenerator, fsm/{Validator,Updater}
	ConfigProvider	StatemachinesConfigProvider
	Main Config	StatemachinesConfig
	Additional Config	Statemachine(project)/model/statemachine.emf
	Template	generators/fsm/StatemachinesCFileTemplate.java → statemachines.c
		statemachines_header.jet → statemachines.h
Timer	Generator class	TimerGenerator
	Main Config	de.upbracing.shared.timer.model.ConfigurationModel
	ConfigProvider	TimerConfigProvider
	Additional Config	de.upbracing.shared.timer.model.*
	Template	timer_cfile.jet → timer.c
		timer_header.jet → timer.h
		timer/*.jetinclude

Table 5.1: Classes that implement the code generators

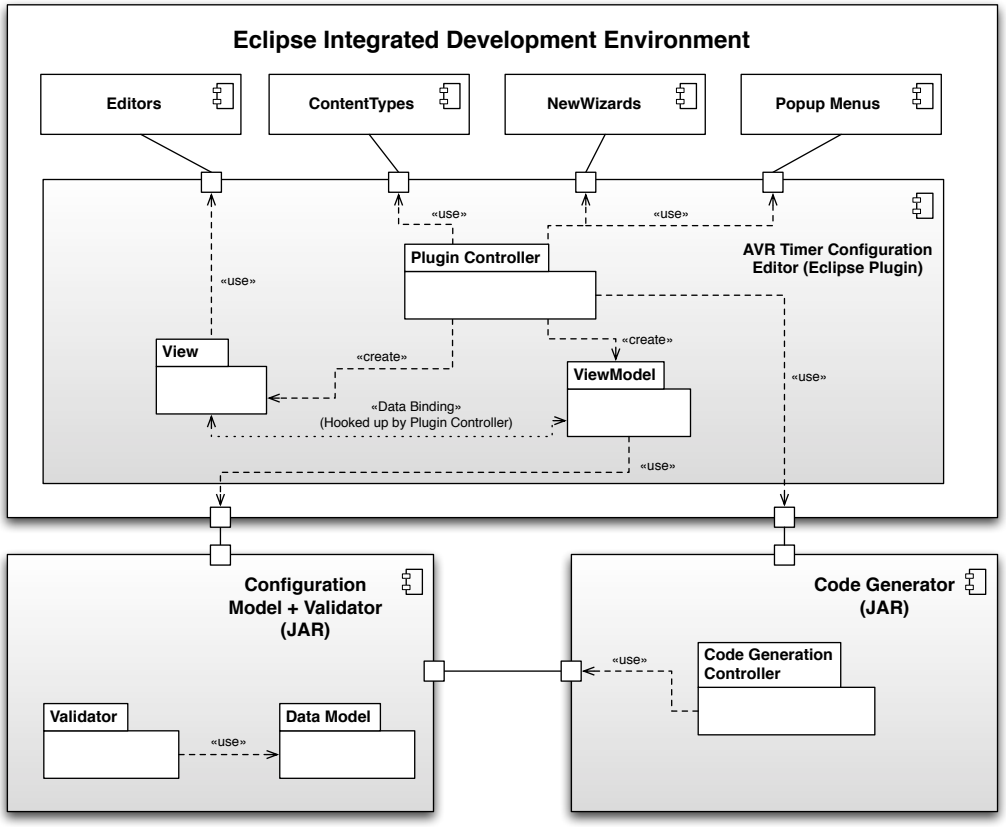

Figure 5.3: Architecture Overview (UML Component Diagram)

(see user documentation for details). Invalid combinations of timer and mode lead to errors, which prevent code from being generated.

You will find the Java implementation of the data model and validator in packages `de.upbracing.shared.timer.model.*`.

The editor is implemented as an Eclipse plugin. The subpackage *View* shows the graphical interface to the user. The *Plugin Controller* is responsible for registering menu entries, that are specific to **.tcxml* files, and creating editor instances. Furthermore, it provides the wizard for creating new **.tcxml* files. The *ViewModel* synchronizes the graphical widgets to the model's data. This synchronization is achieved with *java.beans* databinding techniques. Additionally, the ViewModel contains textual data like tooltips and descriptive texts, that are used in the editor but are not needed within the data model.

The *Code Generator* package translates the data model into compilable C code for the AT90CAN microprocessor. This package is used to allow code generation

from within the editor or the context menu of **.tcxml* files. It does not contain the whole code generator framework but only the single generator plugin and the templates, that are necessary for the timer code generation process (see section 5.2 for details).

The following paragraphs only give a short overview of the different package implementations. The interested reader will find more detailed information in the separately shipped technical documentation for the AVR timer configuration program.

Data model

The central class in the data model is called `ConfigurationModel`. It stores the *Frequency* of the processor and the integer value *ErrorTolerance*.

The actual configurations for specific timers of the AT90CAN microprocessor are stored in `UseCaseConfiguration` objects which in turn are stored in the list *Configurations* of the TimerConfiguration object. Each UseCaseConfiguration object stores:

- Name of the UseCaseConfiguration which is used for naming the generated API
- Mode of the timer hardware (Overflow, CTC, PWM)
- Timer that will be used (0, 1, 2 or 3)
- Desired (user-entered) period values for Input Capture Register (ICR) and all Output Compare Registers (OCRnA, OCRnB, OCRnC)
- Top register selections for CTC and various PWM modes
- Interrupt enable flags for all modes of operation
- Output pin modes for waveform generation in CTC and PWM modes

Validation

Both `TimerConfiguration` and `UseCaseConfiguration` objects can be validated. To achieve this, we have implemented the classes `ConfigurationModelValidator` and `UseCaseModelValidator`. The following properties of the data model are validated:

- Processor frequency: between 1 Hz and 16 MHz, error otherwise
- Names of UseCaseConfiguration objects: name collisions and invalid characters are errors

- Period errors: value specified is too large for current timer and prescaling setting
- Period warnings: quantization error larger than tolerated
- Top register selection errors: some top registers cannot be chosen in some modes
- Mode and timer errors: some combinations of mode and timer are invalid
- Output pin errors: not all waveforms can be generated on every output channel.

`Java.beans.PropertyChangeSupport` allows the editor to synchronize its validation widgets with the validation results from the `ConfigurationModelValidator` and `UseCaseModelValidator` classes. Please see the separate documentation for more details.

ViewModel

The ViewModel acts as a middleware connecting both view elements and data model properties. The technology used for these connections is called *databinding*.

Whenever new values are entered in the editor, they are written though to the model instantaneously. Additionally, the validator associated to the model is informed about the changes via its `updateValidation()` method. This will make the validator reevaluate the model. Since the editor is data-bound to the validation results, they are displayed immediately.

Eclipse Editor

The editor is composed of platform-independent Java SWT widgets. All widgets are databound to corresponding properties in the ViewModel. Databinding is used for more than just synchronizing text widgets. It is also used to show or hide view elements according to the mode of operation, that was selected for a timer configuration.

5.2.2.3 Statemachine Model and Editor

The Statemachine graphical editor is a GMF editor which is generated from the `Statemachine Metamodel`. Using this editor, programs can be modeled in the form of

statemachines. In other words, the statemachine helps the user to create the `Data Model` graphically that is acted upon by the `Validator`, `Updater`, and `Code Generator`. The Validator part ensures that the Data Model created meets the constraints defined for the metamodel elements, while the Updater refines the Data Model and handles warnings/errors from the Validator wherever possible. The Code Generator is responsible for translating the data model into compilable C code for the AT90CAN microprocessor. The subsequent sections elaborate more on each of these parts.

Statemachine Metamodel

The statemachine metamodel Figure 5.4 was created in Emfatic, and can be found in the Statemachine/model folder as `statemachine.emf`. The main class of the metamodel is the StateMachine class, which contains the entire configuration of the statemachine. Each state, region, and global code box in a statemachine must have a name, which helps in identifying the context for printing error messages and generating state specific code. The InitialState marks the entry point in the statemachine, and the other states contain embedded C code held in the action attribute.

The Region is akin to the StateMachine class and can contain a whole statemachine inside it. In addition, each transition class has transitionInfo and a priority which decide when or whether a transition takes place from a state. To provide the facility of including external header files or code, the statemachine contains the GlobalCode class.

Statemachine Graphical Editor

The Statemachine Graphical Editor is a GMF editor, which is generated from the `statemachine.emf` file (*EuGENia* $\longrightarrow$ *Generate GMF Editor* from the contextual menu). To beautify the statemachine elements in the GMF Editor various EuGENia annotations have also been used. Three plugins are generated in this process – `Statemachine.diagram`, `Statemachine.edit`, and `Statemachine.editor`. The editor that is provided as part of `Eclipse.zip` contains these generated plugins exported as JAR files. To provide the functionality of multiline textfields for States with actions or Transitions, some minor modifications were made to the files in the src folder of the `Statemachine.diagram` plugin. These files are `GlobalCodeCodeEditPart.java`,

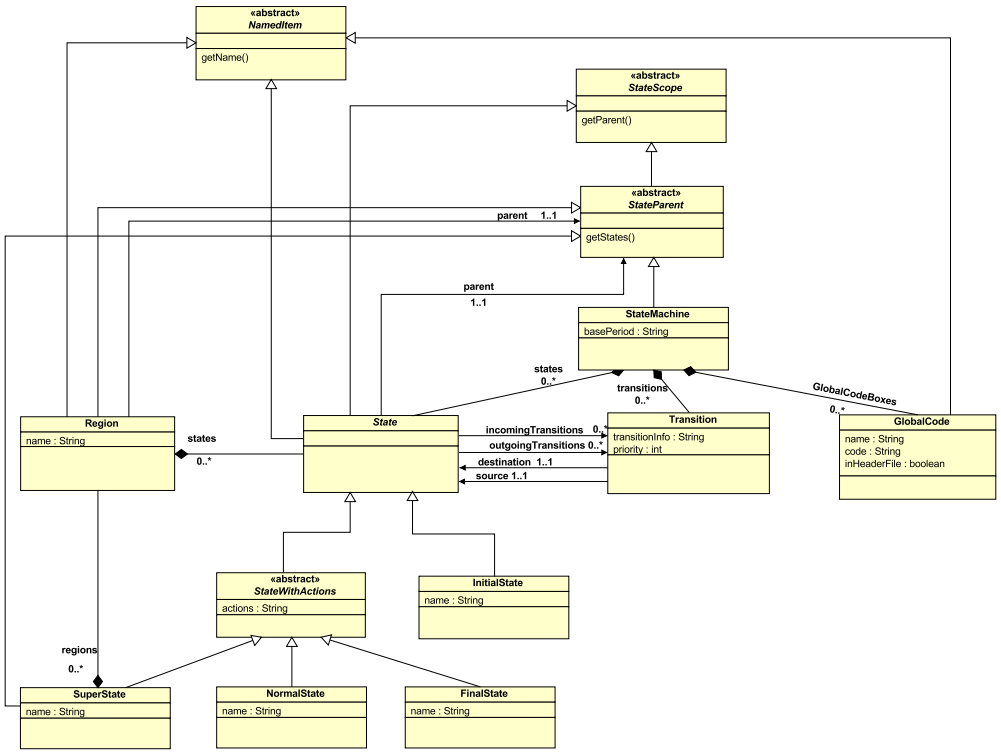

Figure 5.4: Statemachine metamodel (UML Class Diagram)

`NormalStateActionsEditPart.java`, `SuperStateActionsEditPart.java`, and `TransitionTransition InfoEditPart.java` (for this purpose Statemachine.diagram is already provided as part of the software package). Listing 5.3 shows the changes made to the `setLabel` method in GlobalCodeCodeEditPart.java. In each case almost identical changes are made to the same method. No testing is performed for this feature, as it was not considered necessary to test it, so you won't find any tests on it.

Listing 5.3: GlobalCodeCodeEditPart.java

```
1       /**
2        * @not-generated
3        */
4       public void setLabel(WrappingLabel figure) {
5           unregisterVisuals();
6           setFigure(figure);
7
8           if (figure instanceof WrappingLabel)
9               ((WrappingLabel) figure).setTextWrap(true);
10          else
11              System.err
12                      .println("WARN: GlobalCode has code that doesn't
                            support wrapping. I cannot make that a multi-
                            line label.");
13          defaultText = getLabelTextHelper(figure);
```

```
14          registerVisuals();
15          refreshVisuals();
16      }
```

Data Model

The Statemachine Data Model is created using the editor. Each of the visual elements are graphical representation of the concrete Metamodel classes. Data Model is saved in two files `.statemachine` and `.statemachine_diagram`. The .statemachine contains the graphical representation while the .statemachine_diagram contains the tree structure. The Code generation part works with the .statemachine file, and takes the Data Model to fill the template `StatemachinesCFileTemplate.java` to generate `statemachine.c` and `statemachine.h` files. For successful code generation from the Data Model to take place, the Data Model must be valid, which is checked by the Validator.

5.2.2.4 Statemachine generator

The statemachines are our most complicated generator. Therefore, we are explaining them in more detail than the other ones.

Validation and preprocessing happens in the dedicated classes `Validator` and `Updater` in namespace `de.upbracing.code_generation.generators.fsm`. Look at the `validate` resp. `updateConfig` method in those classes to get an overview of the process.

In the preprocessing step we parse transition labels and state actions and put the information into instances of `TransitionInfo` and `Action`. We use "java-parser-tools"[16] for our parsers because it is a parser combinator.[17] This means that we can extend the parser at runtime. For example, we build the parser for action types using `ActionType.values()`, so source code and grammar cannot get out of sync.

We don't want to store the additional information in the statemachine model that we use in the editor. Therefore, we use the wrapper class `StateMachineForGeneration` for code generation. It stores the parsed values and keeps track of some values

[16] `http://code.google.com/p/java-parser-tools/`
Unfortunately, this library is quite slow. We have optimized some bottlenecks to get useful performance. If you want to know the details, please look at the Git history.

[17] If you want to learn about parser combinators, you should look at Parsec. `http://www.haskell.org/haskellwiki/Parsec`

that we use for code generation, e.g. the set of event names that are used by a statemachine.

For a simple statemachine, we only need one state variable, but for a nested statemachine this is more complex. We use nested structs and unions. That way, we can make sure that we keep memory consumption small: If we can guarantee that two superstates cannot ever be active at the same time, the generator will put their variables into a union, so they will share the memory. We store that information in a tree of `StateVariable` instances. This system can be used by statemachine extensions. They can make a custom `StateVariablePurpose` and create variables for their own use. At the moment, only the code for `wait` uses it.

The generator itself is split into a lot of methods. JET cannot handle that well, so we use a normal Java class. You can find it in `de.upbracing.code_generation.generators.fsm.StatemachinesCFileTemplate`. The main method `generate` triggers the generation for each statemachine. Each statemachine has these parts: data declarations, actions functions, init function, tick function and event functions. Most of the code is generated by `printCode` (action code), `generateEventFunction` (event or tick function) and `generateEventSwitchCase` (appropiate action according to the current state). You can find more details in the JavaDoc.

5.2.2.5 Extension points

In addition to the plugin system we use for the generators, we support small extensions of one aspect of a code generator. Most of the extension points have been accounted for in the design of the generators, but only the most useful one has been implemented.

All extensions must implement a certain interface and be registered with the ServiceLoader framework. See subsection 5.2.3 for the details.

RTOS configuration

The caRTOS configuration has a lot of configuration options. You can add some new ones with this extension. You have to implement `RTOSConfigValueProvider` and create some instances of `RTOSConfigValue`. See the JavaDoc of `RTOSConfigValueType` to learn about the types of values you can use (e.g. a constant or a flag).

Statemachine events and actions

We have two useful extensions of statemachines: The `wait` statement (e.g. `wait (100ms)` and interrupt actions (`ISR(INT0)`). Those extensions need to do several things:

- Parser extensions: They can add alternatives to parser rules, e.g. allow "`wait(100ms)`" as an event name. We use a parser combinator library, so we can easily allow such extensions at runtime.
- Run code during preprocessing: We cannot allow extensions for all parts of the generator, so the extensions must rewrite the extended parts to use only normal statemachine features.
- Add some custom state, e.g. a counter for wait. This is possible using the `StateVariable` system (see subsubsection 5.2.2.4).
- Add some code to the generated file, e.g. an interrupt handler that calls an event function.

This extension hasn't been implemented, yet. However, the most important parts are in place. When we have some more extensions, we will implement the ServiceLoader indirection.

Support for other operating systems

Some generators can only be used with caRTOS. We should let the user select the OS and load an appropriate provider through a ServiceLoader. We won't implement that before we need another operating system.

5.2.2.6 The upbracing-common project

This project contains code that is used on more than one ECU in our car. The code generators replace existing code, so some of them rely on files in that project. In particular, we use the CAN library (`can_at90.c/h`), some type definitions from `common.h` and the macros in `Pins.h`. The other files are deprecated (especially the PHP code generators), but we keep them because they are used by some old software in pre-PX211 cars.

5.2.3 Writing a new plugin

In this section you learn how to write your own plugin. We continue the example of subsection 5.2.1: On the internet, Fred has found a response curve for his valve actuator. He needs this data in his AVR program, but he is too lazy to convert the table to C code. "Show me how to write a code generator", he demands. Well, that is easy.

The generated code should look like Listing 5.4.

Listing 5.4: Example generator output

```
typedef struct {
    int8_t voltage;
    uint8_t position;
} valve_data_t;

valve_data_t valve_data[] = {
    { -5, 0 },
    { 3, 40 },
    { 10, 55 },
    { 24, 100 },
};
```

You should start up Eclipse and create the generator while reading this section. If you are lazy, you will find the result in the project TableGeneratorExample in our Git.
You can try to only read the summaries and figure out the details yourself. If you are stuck, read the full text.

5.2.3.1 Configuration

Fred's file is a comma-separated text file. We decide that we will parse the file in Ruby, so the Java part can be used for other tabular data, as well. The configuration object mostly contains the data.

We create a dedicated object for the configuration. It contains all information that we need to generate the code. Each instance of the object corresponds to one table that we generate.

Firstly, we create a new Java project in Eclipse. In this example, we put all code into the namespace `fred`, so you should create that, as well.

We create an object that describes one table. The object will be used by JRuby, so we try to make an interface that looks good from a JRuby script.

JRuby can access arbitrary Java objects, so you can design the interface for Java. However, there are some things you can do to make a good interface for JRuby:

- Getters and setters look like properties to JRuby, so we use them extensively.
- If we create an object in JRuby, we need its full name. Create some helper methods to avoid that.[18]
- JRuby automatically provides `snake_case` variants of everything, so use `camelCase` and `PascalCase`, as usual.

Create the class `TableConfig` in namespace `fred` using the code in Listing 5.5. In the comments you can see how each method can be accessed from JRuby. In the example, we left out all error handling. You should check all user input in a real generator.

Listing 5.5: Configuration class for one table

```
1  package fred;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class TableConfig {
7      private String name;
8      private List<String> names = new ArrayList<String>();
9      private List<String> types = new ArrayList<String>();
10     private List<Object[]> data = new ArrayList<Object[]>();
11
12     // JRuby: tableconfig.name
13     public String getName() {
14         return name;
15     }
16
17     // JRuby: tableconfig.name = "my_name"
18     public void setName(String name) {
19         this.name = name;
20     }
21
22     // JRuby: first_name = tableconfig.names.first
23     // JRuby: tableconfig.names.add "position"
24     // JRuby: column_pos = tableconfig.names.index? "position"
25     public List<String> getNames() {
```

```
26          return names;
27      }
28
29      // JRuby: tableconfig.types.add "uint8_t"
30      public List<String> getTypes() {
31          return types;
32      }
33
34      // JRuby: tableconfig.data.first[0]
35      public List<Object[]> getData() {
36          return data;
37      }
38
39      // JRuby: add_column("voltage", "int8_t")
40      public void addColumn(String name, String type) {
41          this.names.add(name);
42          this.types.add(type);
43      }
44
45      // JRuby: add_data(3, 40)
46      public void addData(Object... row) {
47          this.data.add(row);
48      }
49  }
```

We use an `IConfigProvider` to add the property `tables` which is a list of `TableConfig` instances. For convenience, we add the method `newTable(String name)` that creates a new table.
We have to add a reference to the code generator project or JAR file before we can use the `IConfigProvider` class. If you haven't set-up the projects for development (see subsection B.1.1), you have to use the JAR file in the `dist` folder.
We add it to the file `META-INF/services/de.upbracing.code_generation.IConfigProvider`, so the `ServiceLoader` can find it.

We have to put our `TableConfig` into the configuration object. We could put an empty `TableConfig` into each configuration object and let the user fill it, but we want to support more than one table in each program. Therefore, we use a list.

All configuration extensions must be in an `IConfigProvider`, so we create one. Please create the class `TableConfigProvider` using the code in Listing 5.6. You will notice that Eclipse cannot find the class `IConfigProvider`. This is an appropriate time to add a reference to the code generator library. You can either use the code generator project (upbracing-AVR-CodeGenerator) or the JAR file (upbracing-AVR-CodeGenerator.jar). Please add it to the build path of your project ("Java Build Path" in the project properties).

Listing 5.6: Configuration class for one table

```
1  package fred;
2
3  import java.util.List;
4  import de.upbracing.code_generation.Messages;
5  import de.upbracing.code_generation.config.*;
6
7  public class TableConfigProvider implements IConfigProvider {
8      // STATE is public, so our classes and other generators can use it
9      public static final ConfigState<List<TableConfig>> STATE
10         = new ConfigState<List<TableConfig>>("tables");
11
12     @Override
13     public void extendConfiguration(RichConfigurationExtender ext) {
14         // add a list of TableConfig objects to each config object
15         ext.addListState(STATE);
16
17         // let the user access the list like a property
18         ext.addProperty("tables", STATE);
19
20         // add all methods marked with @ConfigurationMethod
21         ext.addMethods(TableConfigProvider.class);
22     }
23
24     @Override
25     public void initConfiguration(CodeGeneratorConfigurations config) {
26         config.setState(STATE, new ArrayList<TableConfig>());
27     }
28
29     @Override
30     public void addFormatters(Messages messages) {
31         // no custom formatters
32     }
33
34     @ConfigurationMethod
35     /// create and add a new TableConfig
36     public static TableConfig addTable(CodeGeneratorConfigurations
           config, String name) {
37         TableConfig t = new TableConfig();
38         t.setName(name);
39         config.getState(STATE).add(t);
40         return t;
41     }
42 }
```

Firstly, we add the list as a state variable. To do so, we create a `ConfigState` instance and register it using `addState`.[19] It acts as a name for our data, so we have to remember it. We can use `config.getState(STATE)` and `config.setState(STATE)` to access the value. We put the name into a public constant, so other generators

[19] Actually we use `addListState` which solves some Java typing quirks for us. It calls `addState(STATE, List.class)`.

can access the data. If another generator needs some table, it can add it to our configuration. We can provide `STATE.readonly()` instead, if we don't want them to set the value.

To expose the data to JRuby, we have to give it a name. We register a property with the name `tables`. We could use `addReadonlyProperty`, if we didn't want the config script to set the value, but we don't care.

The config script can add a new table: `$config.tables.add(fred.TableConfig.new)`. We want to provide a helper method to make it even easier. We can either use a custom list class for the `tables` property (`class Tables extends List<TableConfig> { ... }`) or add our method to the main config object. In this case, we do the latter. If we had a lot of methods or we had to keep some common state for tables, we would rather create a custom list class.

We can add individual methods, but there is a useful shortcut: We mark the methods with the annotation `@ConfigurationMethod` and use `addMethods` to add them all. This is the preferred way, even if we only add one method.

Now, we tell the ServiceLoader about our class. For that, we create the folders `META-INF` and `META-INF/services` in `src`. In the `services` folder, we create the empty text file `de.upbracing.code_generation.config.IConfigProvider` and add a line for our class: `fred.TableConfigProvider`.

We can write the configuration file, now. If you want to compile the resulting code, you should put the file into an AVR project. In this case, we simply put it into the generator project. The config in Listing 5.7 doesn't use a CSV file, but its output will look like Listing 5.4.

Listing 5.7: Example configuration file

```
1  t = $config.tables.add_table "valve_data"
2  t.add_column "voltage", "int8_t"
3  t.add_column "position", "uint8_t"
4
5  # Function calls don't need parentheses in Ruby. That
6  # way it looks more like a real csv table.
7  t.add_data -5,   0
8  t.add_data  3,  40
9  t.add_data 10,  55
10 t.add_data 24, 100
```

Let's run the program! Of course, we won't see any output, yet. Please open the "Run Configurations" dialog of Eclipse and create a new configuration for a "Java Application". Use the main class `de.upbracing.code_generation.Main` (you may have to

enable “Include inherited mains”) and these arguments: src/config.rb -C gen -T tmp

Please run the application, now. After some seconds it should output a list of generated files. If it prints an error, read this section again and make sure that you haven't left out any step. If the program complains that it cannot find “`tables`”, make sure that you have the ServiceLoader part right.

Now we can load our configuration and we have a bunch of generated files, but none of them is ours. Let's change that! Time to write the generator.

5.2.3.2 Template

Every generated file corresponds to an instance of `ITemplate`. The generator framework uses the template to generate the file's contents. We use JET here, but you may use any templating engine you like best or simply create the template in Java code. It only has to implement the `ITemplate` interface.

We create a JET template that produces the code of Listing 5.4. We use a skeleton to make the template implement the `ITemplate` interface.

At first we add the JET nature to our project, so we can use Java Emitter Templates. Invoke “New -> Other... -> Java Emitter Templates -> Convert Projects to JET Projects”, select the project and click “Finish”. This will create a `templates` folder in our project. You should open the project properties and set the source container to “src” in the JET properties.

If you cannot find the JET properties in the project properties, you may have to install the JET plugin. You can do that by installing EMF using “Install modelling components” (if available) or the update site `http://download.eclipse.org/modeling/emf/emf/updates/releases/`.

You may want to put the generated files into another source folder. We use the folder “src-gen”. You have to create the source folder (not a normal folder!) before you can set it in the JET properties.

Our JET templates must implement the `ITemplate` interface. We can adjust the JET code with a skeleton which is some Java code that looks like the class we

want to have. Please add the file `templates/ITemplate.skeleton` using the code in Listing 5.8.

Listing 5.8: ITemplate.skeleton

```
1  public class CLASS implements de.upbracing.code_generation.ITemplate {
2    /* (non-javadoc)
3      * @see IGenerator#generate(Object)
4      */
5    public String generate(de.upbracing.code_generation.config.
         CodeGeneratorConfigurations config, Object generator_data) {
6      return "";
7    }
8  }
```

Next, we add the template itself. Add the file `templates/tables.jet` using the text in Listing 5.9. After you create the file, you may see an error message because it doesn't have a JET header. This message will vanish, when you copy the text into the file.

Listing 5.9: tables.jet

```
1  <%@ jet
2      package="fred"
3      class="TablesTemplate"
4      skeleton="ITemplate.skeleton"
5      imports="" %>
6  /*
7   * tables.h
8   *
9   * This file defines the features the the Timers should have.
10  *
11  * NOTE: This file was generated and should not be altered manually!
12  */
13
14 <%
15 for (TableConfig table : config.getState(TableConfigProvider.STATE)) {
16 %>
17
18 typedef struct {
19 <% for (int i=0;i<table.getNames().size();i++) { %>
20     <%= table.getTypes().get(i) %> <%= table.getNames().get(i) %>;
21 <% } %>
22 } <%= table.getName() %>_t;
23
24 <%= table.getName() %>_t <%= table.getName() %>[] = {
25 <%
26 for (Object[] data : table.getData()) {
27     stringBuffer.append("\n\t{ ");
28     for (int j=0;j<data.length;j++) {
29         if (j > 0)
30             stringBuffer.append(", ");
```

```
31              stringBuffer.append(data[j]);
32          }
33          stringBuffer.append(" },");
34      }
35      %>
36      };
37
38      <%
39      }
40      %>
```

In a JET file, you can add Java code inside angle-bracket-percent markers (`<% ... %>`). If you add an equal sign, the code is evaluated as an expression and appended to the output (`<%= table.getName()%>`). You can also append to the output using the `stringBuffer` object. Any text outside of special markers will go into the output verbatim.

When you add the template file, it will be converted to a Java class. You can find the `TablesTemplate` class next to the other classes in the `fred` namespace.

You may find it easier to write new code in the converted Java class and later add it to the Java class, as you have syntax highlighting and code completion for Java, but not for JET. However, be very careful, as Eclipse overwrites the Java file without a prompt! Especially Eclipse does this when you save the Java file!

We do not register the template as a service. Instead, we create an `IGenerator` class that uses an instance of our template. Let's do that now.

5.2.3.3 Generator

Now, we create the generator. We use `AbstractGenerator` instead of implementing the `IGenerator` interface ourselves.

Our generator must implement the `IGenerator` interface. We subclass `AbstractGenerator` which does some of the work. It provides default implementations for all optional methods (e.g. validation), so we only have to implement the constructor. We pass some information to the super constructor. This information is used by `getFiles` and `getUsedGenerators`.

Please create the class `TablesGenerator` using the code in Listing 5.10.

Listing 5.10: Generator class

```
1 package fred;
2
3 import de.upbracing.code_generation.generators.AbstractGenerator;
4
5 public class TablesGenerator extends AbstractGenerator {
6     public TablesGenerator() {
7         super("tables.h", new TablesTemplate());
8     }
9 }
```

We have to register the generator as a service. The process is similar to what we did for the `TableConfigProvider`: Create the file `META-INF/services/de.upbracing.code_generation.IGenerator` and add the line: `fred.TablesGenerator`

Now we can run our project again. This time, the output should say that `gen/tables.h` has been generated (among many other files) and we find the file in the `gen` folder (after refreshing the project). It should look similar to Listing 5.4.

5.2.3.4 Making it real

Now we have a very simple code generator. If we were to use it in a real project, we should improve it. We leave this as an exercise to you. However, we give you some pointers to what should be changed:

- Put the config.rb file into an AVR project and compile it.
- Split it into a header and a C file.
- Implement or use a CSV parser. You can find a simple example in section B.2.
- Implement the `validate` method of the generator.
- Implement `updateConfig`: Convert all values to strings. If the value already is a String (and/or the column type is `char*`), add quotes and escape special characters.
- Use the `de.upbracing.code_generation.Table` class to align the values.
- Write some tests that check the output of the generator. You can use JUnit and our `GeneratorTester` class to do that. Use our tests as examples. The project needs a dependency to JUnit. Eclipse normally adds it for you. If it doesn't, you can use the quick fix "Fix project setup..." to do so.

- Write some tests on hardware to make sure that the generated code not only looks right but also works well. Please confer the files FIRST_STEPS and README in the tests directory for further details.

You should have a look at our code generators. When we implemented them, we might have solved some of the problems you are facing. They also show how most of the API should be used. An overview of those generators is in subsection 5.2.2.

5.3 caRTOS - a real-time operating system for AVR

Most of the OS implementation follows standard procedure. You can find the algorithms for round-robin scheduling and the description of the semaphore concept in [6] and [5]. Therefore, we will only describe the most important aspects in this sections:

- task states
- 'many' semaphores
- asynchronous semaphores
- building the library

Inter-process communication is implemented with queues. They use our semaphore extensions, but apart from that they follow the implementation in [4, chapter 10.1] with additional error handling.

We decided to document the caRTOS project in a separate document in more detail. The interested reader can find more information regarding the operating system in [2].

5.3.1 Task state model

There are four task states defined. A task is *suspended*, when it either has not run at all or has terminated itself via `TerminateTask()`. A suspended task will be transferred into the *ready* state, if `ActivateTask(<id>)` is called with the unique ID of the task as the parameter.

The operating system scheduler is responsible for switching tasks between the ready and *running* state. A task can be blocked on attempting to access a resource (`WaitTask()`), which is currently in use. The requesting task is interrupted

and put into state *waiting*. The task is made ready for execution again, when the resource becomes free again (`SignalTask(<id>)`).

You can see the task model of caRTOS and the state transitions in Figure 5.5.

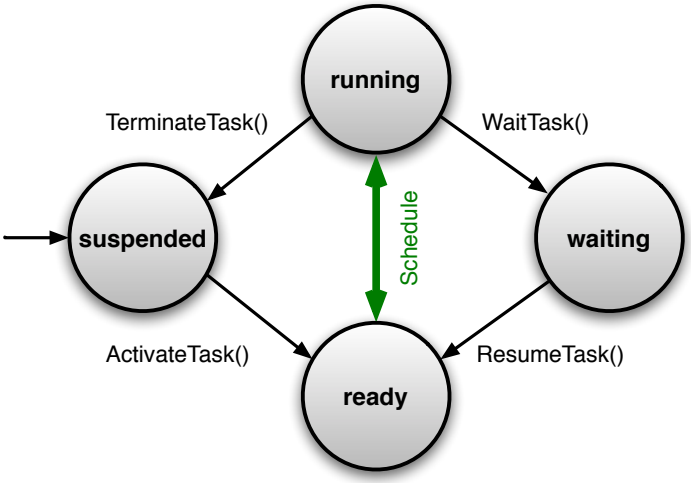

Figure 5.5: Task states and transitions

5.3.2 Semaphore extensions

The user doesn't have to manipulate task states. Task synchronization is implemented with semaphores which use `WaitTask()` and `SignalTask()`. The semaphore concept is well known and described in [6, section 2.2.5].

In this section we describe our extensions to the standard semaphores: 'many' semaphores and asynchronous semaphores.

We provide the 'many' extension which enables an atomic waiting operation for more than one piece of the guarded resource. This is useful for the queue implementation because it often has to put more than one byte into the queue and the bytes should be consecutive.

The implementation of 'many' is straightforward: We store the request size in the waiting queue (in addition to the task ID) and only signal a task, if the whole request can be granted.[20]

Semaphores with the 'many' extension need more memory at runtime, so we provide two implementations. The functions and types with the extension have

[20] This implementation also needs the `ready_counter`.

an '_n' suffix. They are implemented in the file `semaphore.c.inc` that produces either one depending on a macro. That way, we don't have to maintain twice the code.

Statemachines mustn't block the calling process, so they cannot use synchronous semaphores. Therefore, we provide asynchronous semaphores, so statemachines can interact with tasks. An asynchronous 'waiter' gets a unique ID that we use in place of the task ID. It must present that ID with each subsequent call. We can distinguish task IDs and tokens because a token always has a higher value than the highest task ID which we know in advance. In most cases, tokens are treated the same as processes, but of course we mustn't awake them when they become ready. Instead, we wait for the 'waiter' to ask for its status.

For both extensions, we need the `ready_count` which complements the semaphore value. It holds the amount of the semaphore that we could use to wake up processes. In a normal semaphore, we always wake up a process immediately, so it doesn't need an additional counter. In a 'many' semaphore `ready_count` will grow until the first request can be granted.[21] We cannot 'awake' an asynchronous token, so we store the available pieces in `ready_count`. We use that value to tell the waiting task whether it can access the resource.

5.3.3 Compiling the OS library

The OS is provided as a library. The library name includes the MD5 hash of `Os_cfg_features.h` because it has to be rebuild, if this file changes. That way you cannot link to the wrong library and you can provide libraries for more than one config without name clashes. Code that depends on settings in `Os_cfg_application.h` is not linked into the library. It is copied into `Os_application_dependent_code.c` which will be compiled for each application. You can mark such code with a special `#ifdef` (see `Os_application_dependent_code.c`) and run `Os_application_dependent_code.c.rb` to update the file.

5.4 Tests

Of course, we test every part of our code to make sure that it works. Our main tests run on DVK90CAN1 evaluation boards, but we also have unit tests which don't need any hardware. For parts of our tools, we use additional tests.

[21] Example: The first tasks in the waiting queue request 2 and 5 pieces of the resource. Another task calls `sem_signal_n(6)`. We can grant the first request and we have 4 pieces left. That's not enough for the second request, so we put that into `ready_count`. If we get another `sem_signal_n(2)`, we awake the second task and have 1 left in `ready_count`.

5.4.1 Hardware tests

We have several guidelines for our tests:

Repeatable We want to make sure that refactorings and new features don't break old code, so we repeat all tests on new code. We document all prerequisites for the test and we don't use any non-standard hardware.

Automated Extensive testing takes a lot of time, so we let the computer do the work. We can build and run all tests with one line of code. If a test needs a manual step (e.g. pressing a button on the board), the test prints appropriate instructions and waits for the user. This reduces the manual work and potential for errors.

Non-redundant Each test has a text file that describes its purpose, so we don't test an aspect of the program more than is necessary (and we don't forget to test some feature).

Incremental We only use features that have been tested earlier in the test run. This means that we can be sure that a test failure is caused by the feature that it should test.

Correct We make sure that a test fails, if the tested feature breaks.

Verbose The tests sends a lot of information to the PC and the PC determines whether it is correct. If we checked that on the MCU and only sent "OK", false positives would be much more likely. This is also very useful for debugging a failed test.

On hardware We want our test environment to be close to the production environment. This means that the code has to run on real hardware. It is not enough that the code runs well in a simulator or "looks right".

You can find those tests in the folder `code-generation/tests`. To run all of them, call the rake script for your platform with the argument "run-all". Before you do that, you must connect at least one board, a programming adapter and a serial line. The jumpers on the evaluation board must be set to "USART1". You need a second board for the CAN bus tests. You must connect their CAN bus ports with an appropriate cable (DB-9 female on both ends). In our tests we didn't need a terminator resistor, but you might want to add them. If you have a second programming adapter, connect it to the second board. If you don't have one, the tests will tell you to connect the other adapter to the second board at the appropriate time. You do not need a second serial line. You also have to

create `userconfig.rb` which contains information about your test setup (serial port, programming adapter(s)). Please see `userconfig.rb.example` in the test folder.

5.4.2 Unit tests

Although the hardware tests are the only way to be really sure that the software is correct, we use additional tests. We use unit tests for several purposes:

- Testing the config API
- Testing helpers, e.g. the message reporting API (`Messages`)
- Testing parts of the code generator, e.g. validation and timer calculations
- Testing details that we couldn't test on hardware, e.g. comments and whitespace in generated files
- Quickly checking local modifications, if the hardware is not available

Our unit tests are implemented with JUnit, so we can run them from Eclipse. We have a test suit that can be used to run all tests. We (almost) always do that before committing changes to our version control system. We have at least one test for each generator and helper module.

5.4.3 Additional timer tests

The timers can control a lot of pins and the code is not trivial, so we wanted to test it for every pin. The tests are involved because every signal must be checked with an oscilloscope. That can hardly be automated, so we have done the test once after the timer generator was finished. The test procedure and results are documented in [1], so the test can be repeated, if we change the generator or port it to another microcontroller.

5.4.4 Additional statemachine tests

The statemachine code is not at all tied to the AVR microcontroller family (except for the interrupt extension), so we can run the code on a PC. We used that to run some very extensive tests. The project `statemachine-test-pc` builds huge random statemachines and generates code for the statemachines and tests. Those tests run the real statemachine code, but they don't need the hardware. They also can be much bigger than any real statemachine you could use with an AVR.

For each statemachine, the program generates a random way through the statemachine along with some test code that executes the actions and checks whether the statemachine behaves in the right way. This code uses a different, simpler algorithm than the generator, so we can be quite sure that we don't have any common errors.[22] We suppose that the generated statemachines are extensive enough to trigger almost any bug that might be in the generator.[23]

[22] Why don't we use that algorithm for the generator, if it is simpler? The algorithm determines the actions it needs for the way, but the generator needs to determine the right way (next state) using the events and conditions.

[23] We have found several bugs using the random tests. The hardware tests haven't uncovered any further bugs, yet.

6 Conclusion

In the project group, we strived to simplify software development in the UPB-racing Team. In particular, we wanted to make the programs easier to write, maintain and chan ge. We have started to use the tools and we see that our software becomes more structured and we can implement advanced features like timeouts which were too much work wit hout the code generators.

We have tests on hardware for all our tools, so we are quite confident that they are reliable and won't introduce errors into the UPBracing software.

While we were writing the documentation, we noticed that we can simplify parts of the configuration, if the generator is used outside the UPBracing Team. For example, we have added the ability to use the CAN generator without an ECU XML file. We believe that our generators are a good tool for AVR programmers inside and outside the UPBra cing Team, so we will release them as open source software.

Appendix

A Microcontroller tools – User Guide

A.1 Installing our Eclipse Plugins

The graphical editor works out of the box once you install the statemachine plugin from the `EclipsePlugins.zip` archive. A step-by-step description for installing the plugin is given below.

1. Download Eclipse and install it. If you already have Eclipse Indigo or Juno,[1] you can skip this step. You can select any variant because you can install additional plugins later. We suggest that you use the one for C/C++ Developers.[2] You can find the downloads here: `http://www.eclipse.org/downloads/`
2. Start Eclipse.
3. Open the Eclipse update manager with menu item: Help → Install New Software...
4. You can add the repository of our plugins on the Install window by (Add → Archive). A file explorer is opened, in which you can navigate to the directory containing the EclipsePlugins.zip archive and import it like in Figure A.1. This archive contains two UPBracing plugins for Eclipse. Select both plugins (statemachine editor and timer configuration editor). If you are sure that you only need one of them, of course, you can only install that one.

If you have performed the above steps successfully, it means you have successfully set up the graphical editor. You can now move on to the tutorial in section 4.2.

[1] We develop the tools with Eclipse Indigo and we occassionally use them on Juno. Both versions should work fine.

[2] If you are going to compile the statemachine editor, you will need the "Eclipse Modelling Tools" variant. You don't need it to use our tools (including the statemachine editor).

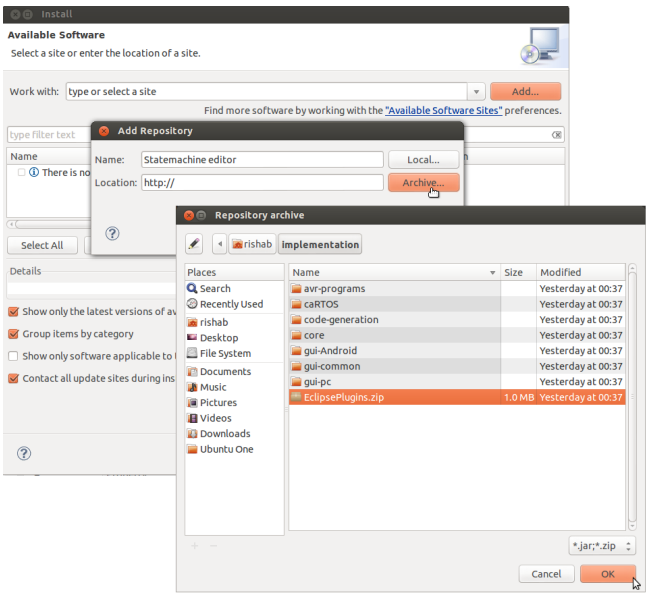

Figure A.1: Adding update site in EclipsePlugins.zip archive file

A.2 Calling the code generator

Calling the code generator is actually really easy. We have packed all the files into a runnable JAR file. However, you usually don't want to call it manually. It should be integrated into the build process. Firstly, we'll have a look at the parameters. Then we show some ways of calling it automatically.

The code generators are distributed in a JAR file.[3] We also provide "run" scripts (`run` for Linux and Mac OS X, `run.bat` for Windows). You should always use the scripts because they can be extended and changed more easily.[4] The distribution folder contains one more file that we will use later.

You can run the script with argument `--help` to get a description of all parameters you can use. You can see the output in Listing A.1. In most cases you only need `--directory` (target directory) and the main configuration file. If you want a really simple build integration, you can tell your build system to run the code generator before compiling.[5] It will run every time and delay your build by a few seconds.

[3] In our Git, you can find the distribution folder at `code-generation/dist`.

[4] During development we use different scripts, so we can use the generators without packing them into a JAR. If you have additional generator plugins, you need to add them to the classpath. You can simply change the script to do that.

[5] You should give the path to the run script because that works on all platforms. Windows will automatically append the `.bat` extension.

Listing A.1: Code generator arguments

```
usage: code_generator [options] config_file
 -c,--check-config            Only load the configuration file
 -C,--directory <arg>         Generate files in directory
 -D,--all-dependencies        like -d, but includes the class path of
     the
                              generator
 -d,--dependencies            Print the names of all files that could
                              influence the code generation. Don't
                                  create
                              any files.
 -h,--help                    Print this help
 -T,--temp-directory <arg>    Put intermediate files in this directory
 -w,--which-files             Print the names of all files that would be
                              generated. Don't create any files.
```

If you're using GNU make, you can use its dependency management to only run the code generator, if an input file has changed. You can tell the code generator to print its input and output file in a format that is suitable for make. We provide the `makefile.targets-inc` file that you can include into your `Makefile`. It looks for the file `../config.rb`. If your configuration file has a different name, you should adjust it.

The code generator won't run `config.rb` to determine the dependencies because that would take too long. Therefore, you must mark all dependencies with `#DEPENDS ON:`. The name of the dependent file must be a double-quoted string on the next line. It must be the only string on that line. You can find a lot of examples in this document.

We use AVR Eclipse, so we provide special support for that. The makefiles generated by Eclipse includes some user-provided files, if they exist. This means that you only have to create the file `makefile.targets` with the two lines in Listing A.2. You can also remove the `CODE_GENERATOR_DIR` assignment and set the path in the project properties (as an environment variable). If you want to use a workspace path, create a "Build Variable" and use that as value for the environment variable.

Listing A.2: Makefile extensions for AVR Eclipse: makefile.targets

```
CODE_GENERATOR_DIR=../path/to/dist/folder
include $(CODE_GENERATOR_DIR)/makefile.targets-inc
```

If you have a lot of projects, you might not want to setup an Eclipse project for each one. We faced that challenge when we created our hardware tests. We are

using rake which works like make but uses a Ruby configuration. With our rake configuration, we can build a lot of projects at once and we can also build and run all tests with just one command. If you want to know the details, read the text files in `code-generation/tests` in our Git.

A.3 Special configuration options

Some code generators have additional configuration options that aren't discussed in the main documentation. You can find the information about them in this appendix.

A.3.1 Statemachine DSL

You can also define statemachines in your config file. This is not the official way, but it can be useful. We provide a DSL (domain-specific language), so you don't have to use the Java classes. We only provide an example in this documentation. If you want to use the DSL, have a look at `statemachine-dsl.rb` in project `upbracing-AVR-CodeGenerator`. You can specify most things in more than one way. The example shows a few of them. You can find some more in the tests (See JUnit tests in project `upbracing-AVR-CodeGenerator`).

Listing A.3: Statemachine configuration with DSL: config.rb

```
1  $config.statemachines << statemachine("counter") do
2    self.base_period = "1ms"
3
4    initial :stopped, "PORTA = 0"
5
6    state :stopped, :normal do
7      enter "DDRB = 0xff", "PORTB++"
8      action :always => "wdt_reset()"
9    end
10
11   state :running, :normal do
12     always "wdt_reset()"
13
14     transition_to :running. "wait(100ms) / PORTA++"
15     transition_to :stopped, "[PORTA >= 128]"
16
17     action "ENTER/DDRA = 0xff", "EXIT/DDRA = 0x00"
18   end
19
20   transition :running => :stopped, :t_info => "startstop_pressed"
21   transition :stopped => :running, :t_info => "startstop_pressed"
22   transition(:stopped => :stopped) { "reset / PORTA = 0" }
23   transition :running, :stopped, "ISR(INT0)"
24
25   global_code :includes, <<-EOF
26     #include <avr/io.h>
27     #include <avr/wdt.h>
28 EOF
29 end
```

A.3.2 Loading pin names from EAGLE

EAGLE is a tool for printed circuit boards (PCB) design. We use it to draw schematics and layout the boards of all custom electronic components in our car.

Listing A.4: Pin names from EAGLE: config.rb

```
# load the schematic file
# (only loads the names)
load_pins_from_eagle(
  #DEPENDS ON:
  "CockpitPX212-L5970D-V2.sch")

# define some names
# for any AT90CAN whose name begins with "IC"
# with name GEAR_A, GEAR_B, ...
# and with name ERROR0, ERROR1, ..., also ERROR514
# It uses regular expressions. If you don't know POSIX
# regular expressions (or you want to use special Ruby
# features), search for a Ruby regex tutorial.
eagle_pins(/^IC.*/, /^GEAR_[A-G]$/)
eagle_pins(/^IC.*/, /^ERROR[0-9]+$/)

# GEAR_ANODE is connected to two pins, so we
# need to say which one we want.
eagle_pins(/^IC.*/, "GEAR_ANODE", /OC3B/)

# include ALL pin names (NOT RECOMMENDED)
# You shouldn't do it. This will fail or generate
# invalid code, if you have ANY signal name that
# is not a valid C identifier name!
eagle_pins(//, //)

# You can also do the equivalent of port(...)
# This will fail, if the regex doesn't match
# all pins of exactly one port.
eagle_port("RPM", /^IC.*/, /^RPM[1-8]$/)

# aliases work as usual
pinAlias("ERROR_OIL_PRESSURE", "ERROR3")
```

A.3.3 CAN protocol description DSL

If you don't want to use a DBC file, you can put the CAN protocol description into your JRuby config script (`config.rb`). In this appendix, we only provide an example. If you want to know the details, you should look at the file `can-create.rb` in project `upbracing-AVR-CodeGenerator`. In Listing A.5, you can see some JRuby code that creates the same model that is used in subsection 4.2.2. You can replace the line `parse_dbc("can.dbc")` by this code to use it in the tutorial. If you compare the generated code, you will notice that the exact same code will be generated by both options.

Listing A.5: CAN protocol description without a DBC file: config.rb

```
1  $config.can = empty_dbc do
2    alice = create_ecu "Alice"
3    bob   = create_ecu "Bob"
4
5    # create_message(name, id, :standard/:extended, length)
6    create_message "SuggestMeeting", 42, :standard, 3 do
7      create_signal "hour"
8      create_signal "minute", :start => 8
9      create_signal "location", :start => 16
10
11     sent_by alice
12     all_signals_received_by bob
13   end
14
15   create_message "AcceptMeeting", 43, :standard do
16     sent_by bob
17
18     create_signal "accepted" do received_by alice end
19   end
20
21   create_message "RequestMeeting", 44, :standard do
22     sent_by bob
23
24     create_signal "location" do received_by alice end
25   end
26
27   create_message "CancelMeeting", 45, :standard do
28     sent_by alice, bob
29
30     create_signal "reason" do received_by alice, bob end
31   end
32 end
33
34 # select messages for Alice
35 $config.use_can_node = "Alice"
```

A.4 Runtime API

A.4.1 Statemachines

Most methods belong to one statemachine. The `<name>` part of their names will be replaced by the name of that statemachine. You have to include the generated header `statemachines.h`.

The function `<name>_init()` initializes the statemachine. You must call it before calling any other function of that statemachine. It runs initial actions and makes sure that the statemachine is in the initial state. You mustn't call it more than once.

You must call the `<name>_tick()` function regularly. If the statemachine uses timing specifications, make sure that the timing of your calls matches the base rate of the statemachine.

If an event occurs, you should call `<name>_event_<eventname>()` to make the statemachine process the event. When the function returns, all transitions and actions that are caused by the event will have been executed. If you want to defer actions,[6] you can add a dummy state and do the real work in an unconditional transition to the real state.[7] Unconditional transition are always executed by the tick function.

If you want to trigger an event in all statemachines, you can use the `event_<eventname>()` function that calls the event function for all statemachines that have an event with that name.

A.4.2 Pin names

You must include `<common.h>` and the generated header file `pins.h`, if you want to use the API.

Listing A.6: Runtime API for pin names

```
1 // You get some macros for each pin,
2 // e.g. for pin("NAME", ...) you get...
3
4 # include header files
5 #include <common.h>
```

[6] This is recommended, if your actions take a lot of time and you call the event function from an interrupt handler.

[7] `A -- event/hard_work()--> B` becomes `A -- event --> DUMMY -- [true] / hard_work()--> B`

```
6  #include "gen/pins.h"
7
8  // change data direction
9  OUTPUT(NAME);    // DDRx |= ...
10 INPUT(NAME);     // DDRx &= ~(...)
11
12 // change state or enable/disable pullup resistor
13 LOW(NAME);           // PORTx |= ...
14 HIGH(NAME);          // PORTx &= ~(...)
15 PULLUP(NAME);        // same as HIGH(NAME);
16 NO_PULLUP(NAME);     // same as LOW(NAME);
17 TOGGLE(NAME);        // PORTx ^= ...
18     // toggle: bit will be 0, if it was 1;
19     //         bit will be 1, if it was 0
20     // It is a bit slower than LOW or HIGH.
21 SET(NAME, condition);
22     // HIGH(NAME), if (condition) is true;
23     // LOW(NAME) otherwise
24
25 // check state of pin (read PINx)
26 // It will return 0, if the bit is not set. It
27 // will return a non-zero value, otherwise. Don't
28 // rely on the particular value.
29 if (IS_SET(name)) ...
30
31 // If your code needs the value on a particular
32 // pin, you should check this macro with #ifdef
33 // and either fail (with #error) or use different
34 // code (e.g. emulate in software instead of using
35 // a timer).
36 // For this example we assume that NAME is
37 // connected to pin PA2.
38 #define NAME_IS_PA2
39
40 // For port("NAME", ...) you can use all the macros for
41 // pin("NAME0", ...) ... pin("NAME7", ...). In
42 // addition you get these macros that modify all pins:
43
44 NAME_OUTPUT();               // DDRx = 0xff;
45 NAME_INPUT();                // DDRx = 0x00;
46 NAME_TOGGLE_INPUT_OUTPUT(); // DDRx = ~DDRx;
47
48 SET_RPM(x);                  // PORTx = (x);
49 GET_RPM()                    // (PINx)
50
51 // For pinAlias("ALIAS", ...) you get the same macros you
52 // get for pin("ALIAS", ...).
53
54 // Some macros (e.g. OUTPUT) are defined in Pins.h in
55 // project upbracing-common. You may want to look at that
56 // file. The code generator defines several more macros
57 // that are used by those macros. These macros are not
58 // meant to be used in your code. Their names may change
59 // in future versions of the generator.
```

A.4.3 Timer hardware

This section describes all API functions being generated for the supported use cases.

A.4.3.1 Common functions

The following functions are applicable to all use cases.

Name:	void **timer_<config>_init**(void)
Description:	This function initializes the timer module for the configuration named <config>. No parameters are passed to this function, since all configuration details are hard-coded. There is no return value, because there are no conditional statements in the initialization function, which could result in different execution paths.
Parameters:	none
Return value:	none
Remarks:	none

Name:	void **timer_<config>_start**(void)
Description:	This function starts the timer module for the configuration named <config>. The timer counts upwards from its current value. Note, that the timer has to be initialized before. If the timer was already started, no action shall take place.
Parameters:	none
Return value:	none
Remarks:	none

Name:	void **timer_<config>_stop**(void)
Description:	This function stops the timer module with assigned configuration <config>. The timer value is not reset. If the timer was already stopped, no action shall take place.
Parameters:	none
Return value:	none
Remarks:	none

Name:	[INT] **timer_<config>_getCounterValue**(void)
Description:	This function returns the current counter value of the timer with assigned configuration <config>. The return type [INT] depends on the type of timer. For timers 0 and 2, an *uint8_t* variable is returned. Functions for sixteen bit timers 1 and 3 return a *uint16_t* variable.
Parameters:	none
Return value:	[INT]: current counter value
Remarks:	none

Name:	void **timer_<config>_setCounterValue**([INT] t)
Description:	This function sets the counter value of the timer with assigned configuration <config> to t. The datatype [INT] of t depends on the type of timer. For timers 0 and 2, an *uint8_t* is required. The function for the sixteen bit timers 1 and 3 requires a *uint16_t* variable.
Parameters:	[INT] t: new counter value
Return value:	none
Remarks:	none

A.4.3.2 Clear on Compare Match

The following API functions are specific to Clear on Compare Match mode.

Name:	[INT] **timer_<config>_getPeriod_ChannelM**(void)
Description:	This function returns the current period of the M-th channel of the timer with configuration <config>. The return type [INT] depends on the type of timer. For timers 0 and 2, an *uint8_t* variable is returned. Functions for sixteen bit timers 1 and 3 return a *uint16_t* variable.
Parameters:	none
Return value:	[INT]: current time period value in ticks
Remarks:	none

Name:	void **timer_<config>_setPeriod_ChannelM**([INT] p)
Description:	This function sets the period of the M-th channel of the timer with configuration <config> to p. For timers 0 and 2, the datatype of p is *uint8_t*. For timers 1 and 3, the datatype is *uint16_t*. Timers 0 and 2 only have one channel (A), while timers 1 and 3 have three channels (A, B, C). (See datasheet of AT90CAN128 for details.)
Parameters:	[INT] p: new time period value in ticks
Return value:	none
Remarks:	none

A.4.3.3 PWM

The following functions are applicable to all PWM modes.

Name:	[INT] **timer_<config>_getPWM_ChannelM**(void)
Description:	This function returns the current PWM duty-cycle value of the M-th output compare channel of the timer with configuration <config>. The return type [INT] depends on the type of timer. For timers 0 and 2, a *uint8_t* variable is returned. Functions for sixteen bit timers 1 and 3 return a *uint16_t* variable.
Parameters:	none
Return value:	[INT]: current PWM duty-cycle in ticks
Remarks:	none

Name:	void **timer_<config>_setPWM_ChannelM**([INT] p)
Description:	This function sets the PWM duty-cycle value of the M-th output compare channel of the timer with configuration <config> to p. For timers 0 and 2, the datatype of p is *uint8_t*. For timers 1 and 3, the datatype is *uint16_t*. Timers 0 and 2 only have one channel (A), while timers 1 and 3 have three channels (A, B, C). (See datasheet of AT90CAN128 for details.)
Parameters:	[INT] p: new PWM duty-cycle in ticks
Return value:	none
Remarks:	To avoid phase shifts, you should use Phase Correct or Phase and Frequency Correct PWM mode if you are planning to alter the duty-cycle(s) of a timer at runtime.

Phase and Frequency Correct PWM

The following API functions are specific to Phase and Frequency Correct PWM mode.

Name:	[INT] **timer_<config>_getPWM_BaseRate**(void)
Description:	This function returns the current PWM base rate value of the timer with configuration <config>. The return type [INT] depends on the type of timer. For timers 0 and 2, an *uint8_t* variable is returned. Functions for sixteen bit timers 1 and 3 return a *uint16_t* variable.
Parameters:	none
Return value:	[INT]: current PWM base rate in ticks
Remarks:	none

Name:	void **timer_<config>_setPWM_BaseRate**([INT] p)
Description:	This function sets the PWM base rate of the timer with configuration <config> to p. For timers 0 and 2, the datatype of p is *uint8_t*. For timers 1 and 3, the datatype is *uint16_t*.
Parameters:	[INT] p: new PWM base rate in ticks
Return value:	none
Remarks:	none

B Microcontroller tools – Technical documentation

B.1 Development setup and build

In this section, you will learn how to setup your Eclipse for development and build our software from source. You may not have to do that. If you only want to make your own generator, any Java IDE is fine. Add `de.upbracing.code_generation.jar` to your project and you're set.

If you want to build our editors and generators from source, go on. We provide download links for all the software. If you have access to our AFS folder, you can get most of it from there.

B.1.1 Setting up your system and Eclipse

1. Install a Java Development Kit (JDK), a Java Runtime Environment (JRE) is not enough. On Linux, you can use the package manager to install OpenJDK. Sun/Oracle JDK[1] will work on all platforms.
2. Install Eclipse with GMF and Eugenia
   a) Download the "Eclipse Modelling Tools" package. At least for Indigo, you cannot add GMF and Eugenia to any of the other packages. We use Eclipse Indigo[2], so that should work best. We also test it on Juno[3], so that should work, as well.
   b) Extract the files and start Eclipse.
   c) Start the modelling components dialog using the menu: "Help → Install Modelling Components" (only available in "Eclipse Modelling Tools" package) and install these components:

[1] JDK: `http://www.oracle.com/technetwork/java/javase/downloads/index.html`
[2] Indigo: `http://www.eclipse.org/downloads/packages/release/indigo/sr2`
[3] Juno: `http://www.eclipse.org/downloads/`

- Java Emitter Templates (JET)
- Graphical Modelling Framework (GMF)
- Eugenia
- Emfatic

3. Install the AVR toolchain. On Windows, you have to download and install WinAVR[4]. On Linux, install the packages for AVR-GCC, AVR binutils and AVR libc. For Debian and Ubuntu, this will work: `apt-get install gcc-avr`

4. Install AVR-Eclipse: Open the install dialog using the menu: "Help → Install New Software...". Add their update site "`http://avr-eclipse.sourceforge.net/updatesite`" and install the plugin. You can find detailed instructions on their web site.[5]

5. Import most projects in the code-generation folder (menu "File → Import..."; choose "Existing projects into workspace"). You need to add these projects which are located in the `code-generation` folder in our Git:

   - java-parser-tools
   - Statemachine (in folder StatemachineEditor)
   - StatemachineFeature (in folder StatemachineEditor)
   - Statemachine.diagram (in folder StatemachineEditor)
   - tests
   - tests-java-helpers
   - upbracing-AVR-CodeGenerator
   - upbracing-AVR-TimerConfigurationEditor
   - upbracing-AVR-TimerConfigurationFeature
   - upbracing-AVR-TimerConfigurationModel

6. Reset `code-generation/upbracing-AVR-CodeGenerator/.jetproperties` to the version in Git because Eclipse messes it up, when you import the project. You can use your favorite graphical tool or do it in a terminal. Open a terminal[6], change to the Git folder and run:

   ```
   git checkout code-generation/upbracing-AVR-CodeGenerator/.jetproperties
   ```

   You need to update the project in Eclipse: "Refresh" in the context menu of the `upbracing-AVR-CodeGenerator` project.

[4] WinAVR: `http://winavr.sourceforge.net/download.html`
[5] AVR-Eclipse installation: `http://avr-eclipse.sourceforge.net/wiki/index.php/Plugin_Download`
[6] The `git` executable must be accessible. Use `Git-Bash` on Windows. On other platforms, this will work automatically, so don't worry.

7. Delete the folder `de` in project `upbracing-AVR-CodeGenerator`, if it exists. This folder is created by Eclipse before we have a chance to reset `.jetproperties` in the previous step.

8. Build project `upbracing-AVRTimerConfigurationModel`
   If it cannot find the Java compiler, please make sure that Eclipse uses a JDK and not a JRE. If the file `de.upbracing.timer.configurationmodel.jar` already exists in the `dist` folder (and is up to date), you can skip this step and the next one.

9. Refresh the project. There should be the folder `dist` with a JAR file. If this file doesn't exist, you need to fix the project setup and repeat the previous step. The project is using an external ANT builder. You can also run ANT manually to build the file.

10. Find the file `model/statemachine.emf` in project `Statemachine` and generate the editor from its context menu: "Eugenia → Generate GMF editor"

11. You may have to run "Build project" on Statemachine to get rid of the error messages.

12. If you want multi-line text fields in the statemachine editor, you must make sure that it uses a few modified files. You can find them in the Git at `code-generation/StatemachineEditor/Statemachine.diagram`. Reset them to the state in the Git and repeat the previous step because Eclipse sometimes overwrites the files, when you run the build for the first time. The modified files contain calls to the `setTextWrap` function.

13. The other projects should be fine now. All projects should build without errors. If there are errors, refresh and clean all projects that have errors. If this helps for some projects, repeat until there are no more errors.

14. Import the `caRTOS` project (in folder `caRTOS/XMEGAOs/os`)

15. Open the `config` folder of that project. You may have to create a few files, if they don't exist:

    - `Os_application_dependent_code.c`: Run the Shell script (only on Linux) or Ruby script with the same name to generate that file.
    - `Os_cfg_features.h`: Copy the example file.

16. Build the project to create the Makefiles. You have to build both configurations (Debug and Release), e.g. from the context menu: "Build Configurations → Build All".

17. You will need EAGLE[7] for some hardware tests. You can download packages for all platforms on their web site. In addition, it might be available from the package manager (in Ubuntu it is). We use EAGLE 5. Starting with Version 6, EAGLE uses a new file format. Our tools will work with the new version and the new file format, but the tests use the old file format. I suggest that you install EAGLE 5 because that's what we use and test.

18. If you want to run the hardware tests, make sure that all tools are in the `PATH` environment variable. If you aren't using Windows, this will simply work, so we focus on Windows here.

    AVR toolchain folders `bin` and `utils/bin` of WinAVR

    Java `bin` folder with `java.exe` and `javac.exe`. In addition, the `CLASSPATH` variable must be right. The installer will set it, so it should be right.

    EAGLE `bin` folder

    You can test it by running these commands. All of them must work:

    - `java -version`
    - `avr-gcc --version`
    - `md5sum --version` (part of WinAVR)
    - `eagle -?`

B.1.2 Running tests

The JUnit tests are in the projects `upbracing-AVR-CodeGenerator` (source folder `tests`) and `upbracing-AVR-TimerConfigurationModel-test`. You can run them in Eclipse ("Run As → JUnit Test"). If you want to execute the random statemachine tests, import the project `statemachine-test-pc` and run it as a C program.

For the hardware tests, you need some hardware:

2x DVK90CAN1 evaluation board Our tests run on a DVK90CAN1[8] evaluation board. You can buy them from Atmel or a distributor. You could use a custom board that has all the hardware that the tests use (CAN bus, serial line, 8 LEDs, 5 buttons), but we don't support this. You can do most tests with only one board, but for the CAN bus tests we need a second board. This means that two boards are required to run the complete test suite. Of course, you also need a power supply for the boards.

[7] EAGLE: `http://www.cadsoftusa.com/download-eagle/`
[8] DVK90CAN1: `http://store.atmel.com/PartDetail.aspx?q=p:10500184`

1x or 2x programming interface The tests load test programs to the microcontrollers, so we need at least one programming interface. You can use any adapter that is supported by avrdude[9], so you can use cheap 3rd-party interfaces. We recommend that you use two programming adapters, so you don't have to repeatedly plug them to the other board during the tests. Avrdude must be able to select the right programmer, so use programmers that can be selected by connection port or use different programmers.[10] We recommend that you have at least one debugging interface (e.g. AVR JTAG ICE) as this is very useful to track down bugs.

1x serial connection You must connect the serial line of one board to your PC. If your PC doesn't have serial ports, you need an USB-to-serial converter.

1x CAN bus cable You must connect the board's CAN bus ports. You need a cable with female DB-9 connectors on both ends. We didn't have to terminate the data lines because they were really short, but you might want to add a 120 Ohm terminator resistor at each end of the connection.

CAN-USB Interface (optional) We recommend a CAN-to-USB interface, so you can view messages on the CAN bus bus during the test. The tests won't use the interface, so you only need it, if you investigate a bug. We use the CAN Debugger[11] because it is cheap.

Now, we can run the tests:

1. Connect the programming interface(s) to the boards and turn on the boards.
2. Make sure that the fuses are set to sensible values. Our tests assume that the 8Mhz crystal on the board is used as the MCU clock. The other fuses should be on their default values. Make sure that no lock bits are set. We use AVR Studio to set the fuses. Of course, you can use any tool you are familiar with.
3. Make sure that you can flash a simple test program using avrdude. If you don't have a test program, at least check the connection. If you leave out the `-u` argument, avrdude will only read the signature. This must work before you can continue. Remember the command that you use to run avrdude.

[9] avrdude: `http://www.nongnu.org/avrdude/`
[10] Read the avrdude manual. Please note that different programmers are NOT different to avrdude, if both of them emulate a ISP mkII programmer.
[11] CAN Debugger: `http://www.kreatives-chaos.com/artikel/can-debugger`

4. Connect the serial line of the first board to your PC. Find the name of the port it is connected to (COMn, /dev/ttySn, /dev/ttyUSBn or /dev/ttyACMn). Test the connection with an example program. The tests configure the serial line as 8N1 with 9600 or 115200 baud. If you use a self-made adapter, make sure that it supports these modes.

5. Create `code-generation/tests/userconfig.rb` using the example file `userconfig.rb.example` in the same folder. Please change the information to match your programming interface(s) (avrdude command you use) and serial connection.

6. Make sure that you have successully done all the steps in subsection B.1.1. All the tools must be available and you must have build the caRTOS library in Eclipse to create the Makefile.

7. Now, you can run the tests. Make sure that the boards are on and connected. If you only use one programmer, connect it to the first board. Check everything again and then open a terminal. Go to the `code-generation/tests` folder and run the tests: `rake-Linux test-all` (adapt to your platform).

8. The script will call rake[12] and run the file `Rakefile`. It builds all the projects that are in subfolders of the `test` folder. Most of them have a text file that describes the test, so have a look at them, if you're intersted. If you only want to build the projects, you can call the script with the argument `build-all`. You can also build or run individual projects. Call the script with the argument `-T` to get a complete list of targets.

9. After building a project, the test will be run. Most tests load a program onto the microcontroller and check the data it sends over the serial line. Some test scripts need your help. They will tell you what to do and they will wait for you. The messages should be self-explanatory, so you know exactly what you should do. If you don't understand a message, file a bug report.

10. If a build fails, make sure that you have done all the steps in subsection B.1.1. You can see the build commands and output in the terminal to help you solve problems.

11. If a test discovers an error, it will abort the test process and print the error. In the terminal, you can see messages from the test and everything it has

[12] JRuby and rake are in JAR files in the library folder of `upbracing-AVR-CodeGenerator`, so you don't have to install them. You cannot run the file with a normal rake because it uses Java objects. That won't work, if it is called from a normal (non-Java) Ruby.

sent and received via the serial line. You can usually see what went wrong. If the error isn't obvious, you can go to the test's project folder and run the ELF executable in a debugger.

12. If all tests run without problems, everything is fine. Congratulations!

B.1.3 Updating generated files

For your convenience, we have put some generated files into the Git. This means that you don't have to generate those files after checking out the code. However, you must regenerate them, if you change certain aspects of our programs.

We run all programs from the output folders of our Eclipse projects. This means that we have to put those folders and all dependencies on the CLASSPATH. If we run the projects in Eclipse, it will build the CLASSPATH for us. However, we need to run some programs from the terminal or in build scripts. In that case, we need a script that builds the CLASSPATH. Those scripts are generated from the `.classpath` files of our Eclipse projects. If you change the dependencies of a project, please run the script `make-run-scripts.rb` to update the scripts.

The file `config/Os_application_dependent_code.c` in the caRTOS project accumulates pieces of code from several other files in that project. You can generate it with a shell script (only on Linux) or Ruby script that is in the same folder.

We need the timer model in the Eclipse editor and in the code generator. We found only one way to cater to both targets: The model is in a normal Java project. We cannot add this to the editor as a dependent project, so we have to put it in a JAR file and add that to the editor project. You start the export with the "Export" command in the context-menu of the project `upbracing-AVR-TimerConfigurationModel`. As you can see in Figure B.1, you export it as a normal JAR file (not a "Runnable JAR file").

B.1.4 Building distribution files

Eclipse automatically builds most of our code. During development, we run every program using the class files that Eclipse builds for us. This ensures that our modify-build-test cycles remain short. However, we don't require our users to build the programs from source. Instead, we distribute them in JAR and ZIP files. In this section, you will learn how to build those JARs.

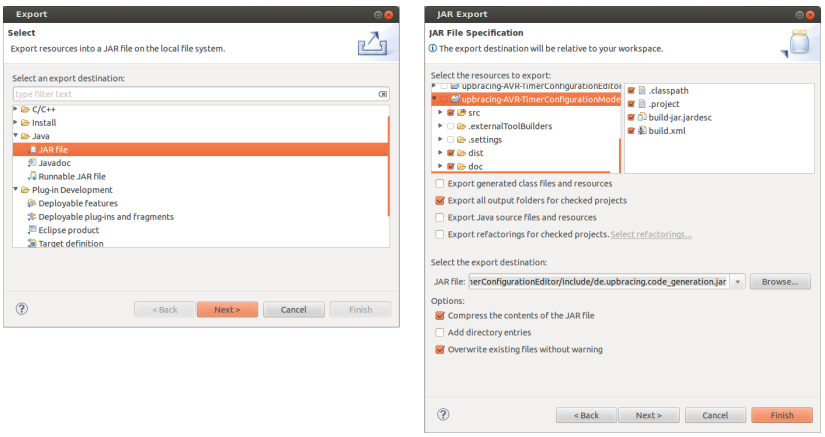

Figure B.1: Export JAR file of timer model, used by timer editor

Distributing your own generator

Export a JAR from your Eclipse project. Make sure that you ship it with all libraries. You shouldn't include our JAR in your JAR because the user might want to use a newer version of our generators.

You can do it like this: In the context menu of your project, choose "Export" and select "JAR file". Select your project and all files that it needs in the JAR file. All output folders of your project (class files) should be included. Select a location and name for the generated JAR and click "Finish". Distribute the generated JAR file together with the files in our `dist` folder (the JAR file and run scripts). You should adjust the run scripts to include your JAR on the classpath.

If you put the source code of your generator into our Git, put your JAR into the `code-generation/dist` folder.

Distributing the code generators

You can find the distribution files in the folder `code-generation/dist`. That folder contains everything that a user needs to run the code generators. If you change the sources, you should also update these files.

The folder contains these files:

JAR file The file `de.upbracing.code_generation.jar` contains the code generation framework and all code generators. You must update it, if you change the framework or the existing generators. If you add a generator, you should package

it in a seperate JAR (see previous section). The regeneration process is outlined below.

Makefile extensions The file `makefile.targets-inc` contains the makefile extensions. You usually don't have to change it. It is a copy of the file in `upbracing-AVR-CodeGenerator`.

Run scripts Those scripts are used to run the code generator. They are provided for the convenience of the user. The user could simply run the JAR file. You usually don't have to change the scripts. In particular, don't copy the run scripts in one of the projects because they serve a different purpose (run the generators without packaging them for distribution).

We distribute our generators in a JAR file. If you want to update that file, do it like this: Firstly, make sure that you can run the generators from Eclipse's output folders and that they pass all tests (including all hardware tests). Then, open the context menu of the `upbracing-AVR-CodeGenerator` project and choose "Export". Select "Runnable JAR file". Choose "Package required libraries into generated JAR" and export it into the `dist` folder. You can see the steps in Figure B.2.

You may have to use "Extract required libraries into generated JAR", if JRuby complains that it cannot find a file, class or gem. However, you have to make sure that you don't violate any license, before you distribute such a JAR file.

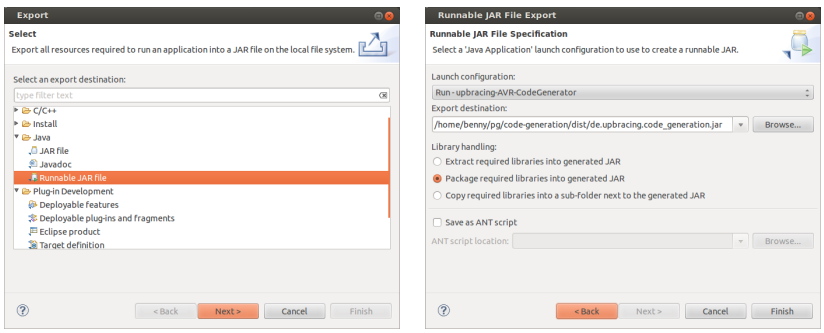

Figure B.2: Export JAR file of our code generators and the framework

Distributing the Eclipse plugins

Usually Eclipse plugins are distributed with an update site that lives at a HTTP URL. We don't want to host a webserver, so we package the update site as a ZIP archive. Nonetheless, it can be used like a normal update site.

You invoke the export command from the Eclipse menu: "File → Export...". You can see the further steps in Figure B.3, Figure B.4 and Figure B.5. It is very important that you add the category file, as the user won't see the projects, if they don't have category information. The generated ZIP file should be placed at the root of our Git and it should be called `EclipsePlugins.zip`.

If you export it several times, use a different name each time. Eclipse will cache the contents of the file and it will ignore your updates, if you don't use a different file.

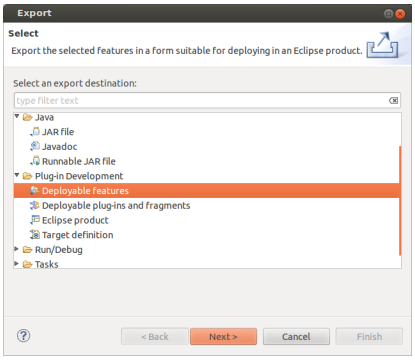

Figure B.3: Export Eclipse plugins into update site archive - Step 1

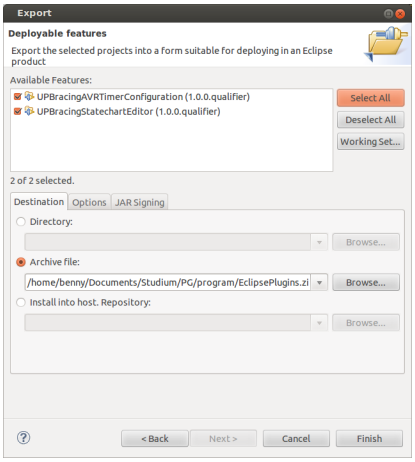

Figure B.4: Export Eclipse plugins into update site archive - Step 2

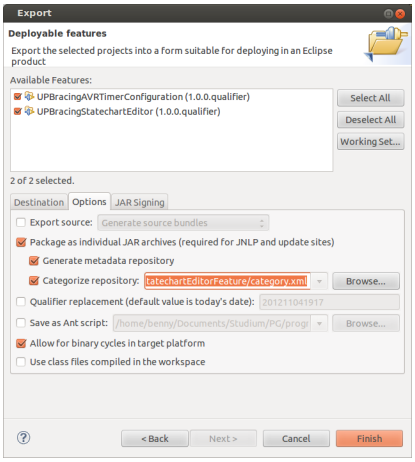

Figure B.5: Export Eclipse plugins into update site archive - Step 3

B.2 CSV parser for the Tables generator

You can put this into your config file to parse a CSV file. Fortunately, Ruby provides a CSV parser. You can find its documentation online.[13]

Listing B.1: Simple CSV parser

```
require 'csv'

def load_csv(name, file, types = {})
  table = $config.add_table name
  CSV.foreach(file) do |row|
    if table.names.empty?
      # This is the first row
      row.each do |name|
        # remove whitespace
        name = name.strip

        # get type or use default
        type = (types[name] || "int")

        # add column
        table.add_column name, type
      end
    else
      # remove whitespace
      row = row.map { |x| x.strip }

      # add the row
      table.add_data *row unless row.empty?
    end
  end
end

# voltage is an int8_t, position gets default type (which is int)
load_csv("valve_data", "valve_data.csv", "voltage" => "int8_t")
```

Listing B.2: Example CSV data

```
voltage, position
-5,         0
 3,        40
10,        55
24,       100
```

[13] http://ruby-doc.org/stdlib-1.9.2/libdoc/csv/rdoc/CSV.html

Glossary

ANT is an automatic build system comparable to make, but implemented in Java. 145

API is a set of functions provided by a library as an interface to other software components. 27, 44, 47, 63, 71, 72, 75, 102, 118, 122, 136

AVR is family of microcontrollers produced by Atmel. We use several AVR microcontrollers in our racing car, especially the AVR AT90CAN128. 23, 26, 44, 62, 72, 75, 88, 91, 99, 108, 122, 144, 156

CAN bus Controller Area Network (CAN) is a serial bus commonly used in embedded systems. CAN plays an important role in nowadays cars. 11, 24, 88, 97, 99, 121, 146, 147

CAN protocol description (DBC file) is a file that describes nodes, messages and signals in a CAN network. It is supported by many vendors and tools that support CAN communication. 24

CAN-USB Interface is used to connect a computer or tablet pc to a CAN bus. This allows to inject or listen to messages. 147

CSV file is a file format which stores tabular data as plain ASCII text. The columns are separated by commas and each line is a row. 113, 154

DB-9 is a D shaped connector commonly used for serial line connections. It is also used by the evaluation board DVK90CAN1 for connecting the CAN bus. 27, 121, 147

EAGLE is an editor and autorouter for printed circuit boards (PCB) layouts. We use it to design the boards in our car. 25, 43, 75, 89, 133, 145

ECU An Electronic Control Unit (ECU) is an embedded computer system. Modern cars are shipped with a large number of ECUs which are responsible for communicating with sensors and actuators in the vehicle. 108

EEPROM is a piece of memory in an AVR processor that can be used to permanently store configuration data. It won't loose its contents, if the processor is turned off or looses power. In constrast to the flash (program memory), single bytes can be erased and rewritten and EEPROM supports more write-cycles before it fails. The EEPROM is not mapped into the IO space of the processor, so you need to access it in a special way. In general, the term EEPROM refers to a permanent (non-volatile) memory that can be electrically deleted and rewritten. 24

ELF is a binary file format for executable files and libraries on (among others) Linux. 72, 148

fuse is a setting for AVR microcontrollers, that is set by the programming device. The microcontroller offers several fuses to configure e.g. the clock source or whether a bootloader is used or not. 72, 147

Git is a revision control and source code management system. We use it for our source code and documentation. 28, 45, 88, 106, 109, 130, 149

Inter-Process Communication (IPC) refers to communication between processes. Usually the operating system makes sure that processes cannot (accidentally) access each others memory and it provides well-defined a API to facilitate communication and cooperation of processes. In our system, we cannot enforce memory policies, but nonetheless we provide an queues for IPC because they are often easier to use than shared memory. 26

JAR file is an archive file containing Java class files and resources. It bundles Java applications and libraries in one file. 44, 90, 94, 104, 111, 130, 145, 148

Java Development Kit (JDK) is a set of tools needed for developing Java applications. It contains the Java compiler, debugger, a JAR file packer, etc. It also includes the Java Runtime Environment (JRE). 143

Java Runtime Environment (JRE) is a set of programs needed to run Java applications. It contains the Java Virtual Machine that is executing the Java application. 143, 156

make is a tool to automatically build software from source files. It is configured with a makefile and can detect which source files have changed since the last build and only recompile necessary parts. 26, 45, 89, 92, 130, 155

MD5 is a hashing algorithm, that produces a 16 byte long hash value of the input data. 120

printed circuit boards (PCB) is the base of electronic devices. Electronic components are soldered to the PCB and connected by tracks (wires) on the PCB to build an electronic circuit. 8, 133, 155

real-time operating system is an operating system which focuses on response times instead of throughput or user experience. In a hard real-time system deadlines are always met, in a soft real-time system deadlines are usually met. 24

semaphore is a tool to synchronize tasks. A task can request access to a shared resource and if necessary, it will be blocked, until the resource is available. 26

task is a thread of execution in a real-time operating system. It is similar to processes and threads in desktop operating systems. A task is usually executed regularly by the operating system. However, our operating system also supports tasks that run very long or don't ever finish. 157

Task Control Block (TCB) is a structure in memory that holds the state of a task. 24

Bibliography

[1] Peer Adelt. AVR Timer Configuration - Report of manual tests. University of Paderborn: RacingCarIT Project Group Documentation, 2013.

[2] Peer Adelt and Benjamin Koch. caRTOS - AVR Operating System - Concepts and usage. University of Paderborn: RacingCarIT Project Group Documentation, 2013.

[3] Atmel Corporation. AT90CAN Microprocessor Datasheet, 05 2012. (available online at `http://www.atmel.com/Images/doc7679.pdf`).

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms. The MIT Press, 2 edition, 2001.

[5] Andrew S. Tanenbaum. Modern Operating Systems (3rd revised edition). Prentice-Hall, December 2007.

[6] Andrew S. Tanenbaum and Albert S. Woodhull. Operating Systems Design and Implementation (3rd edition). Prentice Hall, July 2008.