Project Group 'Racing Car IT'

CAN Code Generator
User Guide

Sven Schönberg

University of Paderborn

Summer Semester 2012

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

Contents

1 Introduction

This userguide for the CAN code generator, explains how to configure the code generator and how to modify the generation process to customize the generated code to your specific needs.

The code generator generates C code, intended to be compiled with the AVR-gcc compiler.

The generated code covers 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. The configuration can be 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.

This document assumes, that the reader is already familiar with the hardware details of the AT90CAN microprocessor family and the basics of how CAN communication works. Furthermore, the reader must know the C programming language.

2 DBC Objects

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.

This section briefly explains the purpose of each object and their relation to each other. For a detailed description of their properties, please have a look at the source code.

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 send 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.

2.1 Parsing a DBC file

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 1 for an example.

Listing 1: Calling the DBC parser and selecting ECU

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

Listing 2: Directly selecting the CAN node

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

3 Configuration Objects

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.

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

Each configuration objects is a sub class of the respective DBC object, so it inherits all its properties and adds configuration properties. The DBCValueTable has no corresponding configuration object, because there is nothing to configure.

Most parts of the configuration are either for sending or for receiving messages. They are discussed in detail in the next two chapters.

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

3.1 Adding Declarations

When you want to call external methods within the source (can.c) or header (can.h) file from custom code, you may need to declare other header files. The configuration object offers two methods for adding such declarations.

For adding declarations to the header file, use the `addDeclarations()` method and for adding declarations to the source file use `addDeclarationsInCFile()`. See Listing 3 for an example.

Listing 3: Adding a declaration to the header and to the source file

```
$config.can.addDeclarations "#include example.h"
$config.can.addDeclarationsInCFile "#include example.h"
```

4 Sending

The code generator creates a send method for each message. This section explains the possible configurations and code customizations.

4.1 Structure

The send methods are placed in the can.h file. You can see the definition of a generated send method in Listing 4.

Listing 4: Send method definition

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

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

Two additional send methods are created which call the send method with a preset value for wait. See Listing 5.

Listing 5: Additional send methods

```
inline static void send_<MessageName>_wait(<params>) {
  send_<MessageName>(true, <params>);
}
inline static void send_<MessageName>_nowait(<params>) {
  send_<MessageName>(false, <params>);
}
```

4.2 Configuration

4.2.1 Messages

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 6.

Listing 6: Accessing a message property

```
$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 7 for an example.

Listing 7: Using `can_config`

```
can_config('msg(Shift_Up)', 'tx_mob', 'MOB_Shift')
can_config('msg(CockpitBrightness)', 'use_general_transmitter')
can_config('signal(Gang)', {"put_value" => "update_gear(value);"})
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.

4.2.2 Signals

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 8 for an example.

Listing 8: Accessing a signal property

```
$config.can.getMessage("MessageName").
                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.

4.3 Code Replacements

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.

These properties expect the code as a string. Tab-characters are added by the generator for proper indentation.

Please see figure 1 for an overview of the send method and where the custom code is added or replaced.

4.3.1 Messages

beforeTx This code is added before anything else in the send method for this message.

afterTx This code is added after everything else in the send method for this message.

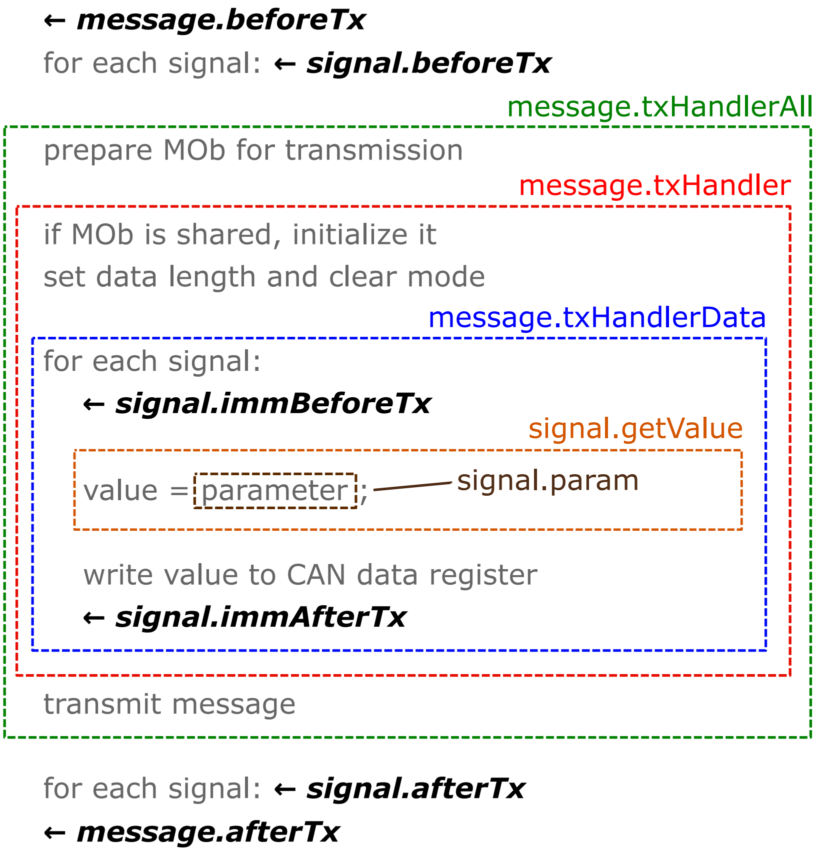

Figure 1: Overview of a send method and possible code replacements

txHandler This code replaces most of the tx handler in the send method. The correct MOb is still selected and it waits for an ongoing transmission to finish. After the tx handler, the transmit function is called to transmit the message.

txHandlerAll This code replaces the whole tx handler in the send method. This includes selecting the correct MOb, waiting for ongoing transmissions to finish and transmitting the message.

txHandlerData This code replaces the part, where the data for the signals is written to the CAN data register.

4.3.2 Signals

beforeTx This code is added at the beginning of the send method for each signal after the beforeTx code of the message.

afterTx This code is added at the end of the send method for each signal before the afterTx code of the message.

immBeforeTx This code is added immediately before the data for the signal is written to the CAN data register.

immAfterTx This code is added immediately after the data for the signal is written to the CAN data register.

getValue This code replaces the part where the value of the signal is read from the parameter. The code must put the value into the variable `value`.

param This code replaces only the assignment of the variable value. It has to be an expression without a semicolon.

5 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.

This section explains the possible configurations and code customizations.

5.1 Structure

The ISR is placed in the can.c file.

5.2 Configuration

5.2.1 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.

5.2.2 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 9, or by calling the Java setter `setExpectedFactor(10, 1)` with two integers for the numerator and denominator of the factor.

Listing 9: Setting the expected factor

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

5.3 Code Replacements

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 2 for an overview of the ISR and where the custom code is added or replaced.

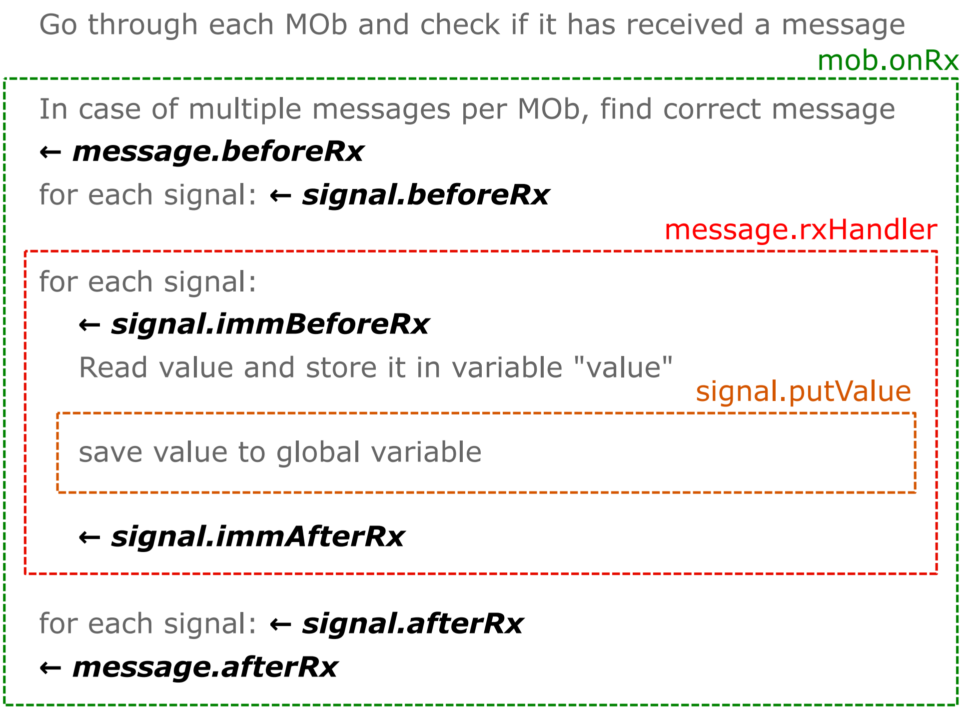

Figure 2: Overview of the ISR and possible code replacements

5.3.1 MObs

onRx This code replaces the complete receive handler of the MOb. If the MOb is configured to receive multiple messages, this has to be handled as well.

5.3.2 Messages

beforeRx This code is added before anything else in the receive handler for this message.

afterRx This code is added after everything else in the receive handler for this message.

rxHandler This code replaces the receive handler for the message. Only the MOb related parts like selecting the correct MOb and checking which message was received are still done. All message and signal related parts have to be done here

5.3.3 Signals

beforeRx This code is added at the beginning of the receive handler for each signal after the beforeRx code of the message.

afterRx This code is added at the end of the receive handler for each signal before the afterRx code of the message.

immBeforeRx This code is added immediately before the data of the signal is read from to the CAN data register.

immAfterRx This code is added immediately after the data of the signal has been read and saved in the global variable.

putValue This code replaces the call to the setter method of the global variable. The received signal value has already been read and is available through the variable `value`.

6 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.

6.1 Structure

The periodic tasks are placed in the can.c file.

6.2 Configuration

To set up a message to be send periodically, simply set the `period` value of the message object. See Listing 10 for an example.

Listing 10: Setting up periodic sending

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

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.

The time can be formatted in various ways:

- 5.2s → 5.2 seconds
- 10ms → 0.01 seconds
- 1:30:02.7 → 5402,7 seconds (1 hour, 30 minutes, 2.7 seconds)
- 1/3 day → 28.800 seconds

For detailed information about the possible time formats have a look at the source of `de.upbracing.code_generation.common.Times`.

6.3 Code Replacements

As with the sending and receiving methods, the code generated for the task, can be modified by adding and replacing code.

Please see Figure 3 for an overview of a periodic task and where the custom code is added or replaced.

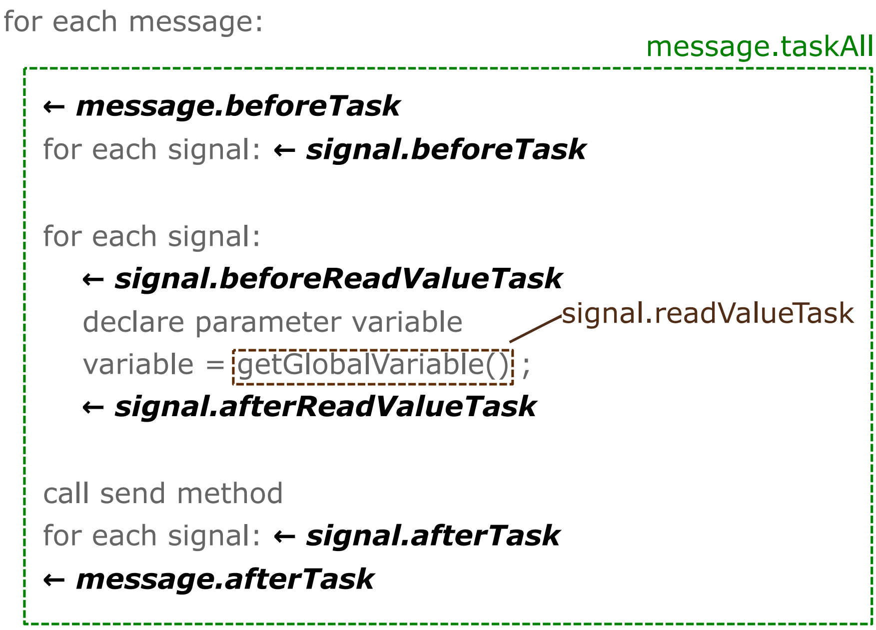

Figure 3: Overview of a periodic task and possible code replacements

6.3.1 Messages

taskAll This code replaces all parts of the task for this message.

beforeTask This code is added before the generated code of the task for this message.

afterTask This code is added after the generated code of the task for this message.

6.3.2 Signals

beforeTask This code is added for each signal before the generated code of the task, after the beforeTask of the message.

afterTask This code is added for each signal after the generated code of the task, before the afterTask of the message.

beforeReadValueTask This code is added directly before the value for this signal is read from the global variable.

afterReadValueTask This code is added directly after the value for this signal is read from the global variable.

readValueTask This code replaces only the assignment of the parameter for this signal. It can be used if it is not desired to use the global variable, but use another value instead. It has to be an expression without a semicolon.

References