Project Group 'Racing Car IT'

AVR Timer Configuration
Technical Documentation

Peer Adelt

University of Paderborn

Summer Semester 2012

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

Contents

1 Introduction **3**

2 Software Architecture **4**

2.1 Configuration Model Package 5

2.1.1 Data Model 5

2.1.2 Validation 7

2.2 Eclipse Editor 10

2.3 Code Generator Package 10

1 Introduction

This document introduces the reader to the technical details of how the *AVR Timer Configuration* editor was developed. It describes the underlying *data model*, the *editor* itself and the *code generator*, which translates the data model into compilable C code for the AT90CAN microprocessor family. The code can then be compiled using the AVR-GCC compiler.

It is assumed, that the reader is familiar with the timer operation modes of the AT90CAN microprocessor family. This document does not cover hardware details. See the processor's datasheet for details.

2 Software Architecture

This section will explain how the AVR Timer Configuration program was created. The program itself is divded into three parts. An overview of the program's architecture is shown in Figure 1. The light gray elements are those that have been developed for the project group.

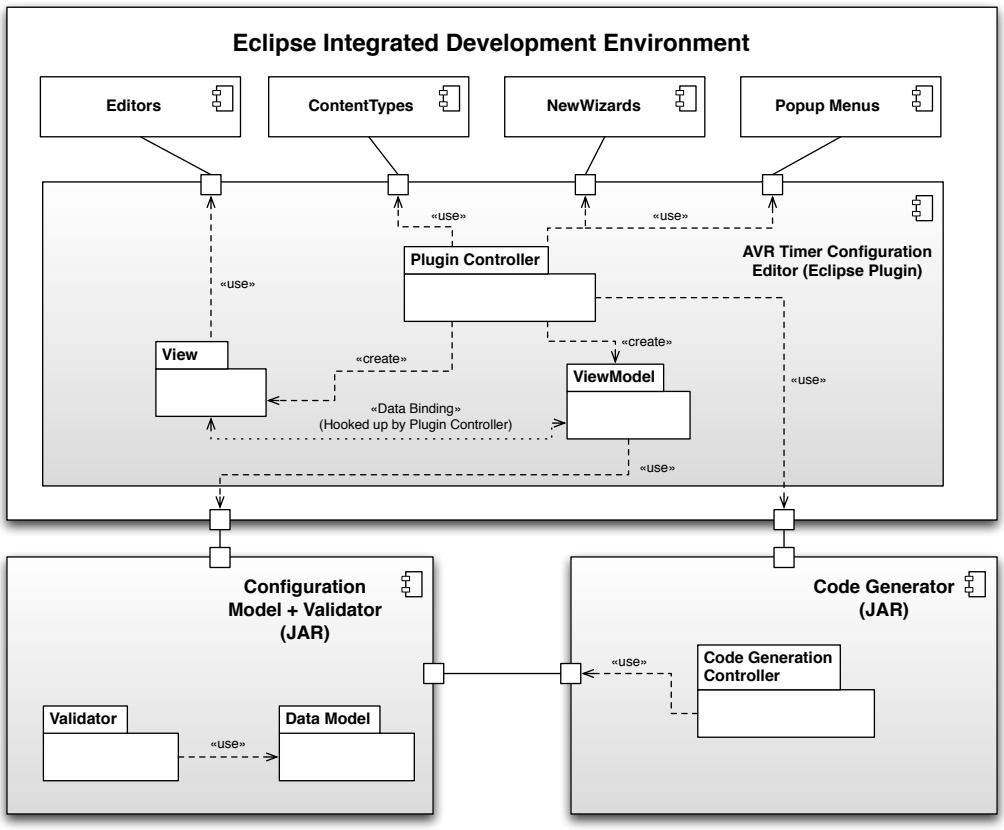

Figure 1: Architecture Overview (UML Component Diagram)

The most fundamental component for the AVR Timer Configuration is the *Configuration Model + Validator* Package (bottom left). It contains a subpackage *Data Model*, which holds the actual configuration data for the AT90CAN microprocessor. The subpackage *Validator* checks this Data Model for correctness. The package is delivered as a shared Java JAR file to allow it being used by the editor and the code generator separately. See subsection 2.1 for further details.

The graphical user interface of the program is delivered as a plugin for the Eclipse IDE (top). Its *View* package implements the graphical editor that is displayed to the user. The *Plugin Controller* is responsible for registering the custom file extension **.tcxml* to the custom editor within the Eclipse IDE. Furthermore, the

Plugin Controller implements a Wizard for creating new **.tcxml* files in a running Eclipse project. Lastly, a Popup Menu entry is created that applies to all **.tcxml* files, allowing code generation to be triggered from there. The *ViewModel* package is responsible for binding properties of the data model to the GUI elements of the Eclipse editor, ensuring that both always stay in sync.

Apart from the Configuration Model + Validator package, the Eclipse plugin also uses the Code Generator package. It is explained in more detail in subsection 2.2.

The *Code Generator* package is responsible for translating the data model into compilable C code for the AT90CAN microprocessor. The code generator is a separate project, which is described in detail in the project group's main documentation. It is embedded into the AVR Timer Configuration program as a Java JAR file. This allows the Eclipse Editor to trigger code generation from inside the GUI. See subsection 2.3 for further details.

2.1 Configuration Model Package

The Editor, Configuration Model and the Code Generator packages work with **.tcxml* files, which can be created, loaded and modified with the AVR Timer Configuration Editor (see subsection 2.2 for details). The following sections will explain, how the timer configuration data is organized in the data model, how it is validated for correctness and how saving and loading the **.tcxml* files is achieved.

2.1.1 Data Model

The data model is shown as a UML class diagram in Figure 2.

The central class in the data model is called **ConfigurationModel**. It stores the *Frequency* of the processor and the integer value *ErrorTolerance*. The frequency is of interest, when it comes to calculation of actual register values from time periods entered by the user. The error tolerance value is used by the *Validator* (validation will be explained in the next section).

Each TimerConfiguration object is able to save its contents to harddisk. The *Save* method accepts a file path as a parameter. When invoked, it uses the *java.beans* XML functionality[1] to produce the **.tcxml* file. Similarly, the static

[1] see `http://docs.oracle.com/javase/1.4.2/docs/api/java/beans/XMLEncoder.html` for details

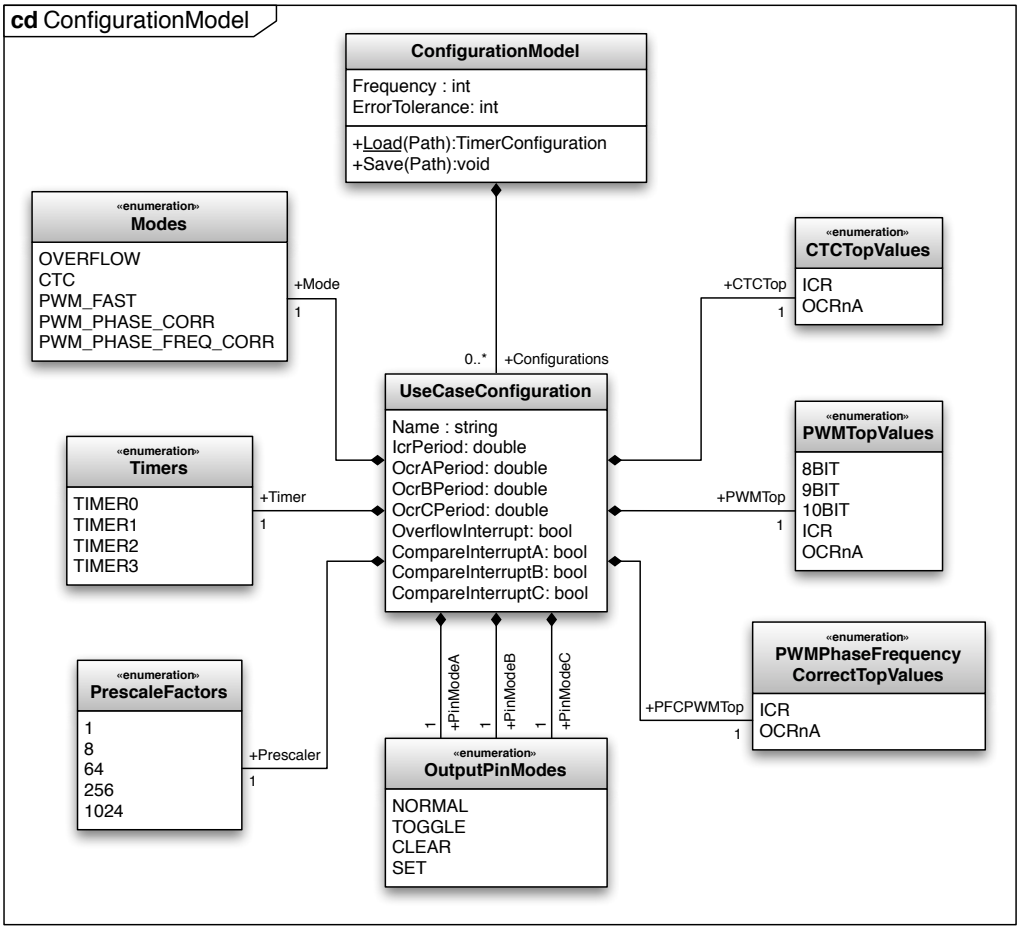

Figure 2: Timer Configuration Model (UML Class Diagram)

Load function parses the **.tcxml* file at a given path into a TimerConfiguration object.

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.

Every UseCaseConfiguration has a *Name* which needs to be a valid C identifier. The *Mode* determines, whether the timer should run in Overflow mode (OVERFLOW), Clear Timer on Compare Match mode (CTC), Fast PWM mode (PWM_FAST), Phase Correct PWM mode (PWM_PHASE_CORR) or Phase and Frequency correct PWM mode (PWM_PHASE_FREQ_CORR). Which timer is to be used is stored in the *Timer* property. Prescaling is configured with the *Prescaler* enumeration.

The flag *OverflowInterrupt* determines, whether the processor shall generate in-

terrupts on timer overflow or not. The properties *CompareInterruptA*, *CompareInterruptB* and *CompareInterruptC* indicate, whether interrupts will be generated when the timer reaches the compare match values stored in the Output Compare Registers A, B or C.

Initial register values for Input Capture Register and the (up to) three Output Compare Registers can be configured in the *IcrPeriod*, *OcrAPeriod*, *OcrBPeriod* and *OcrCPeriod* properties.

In the data model, desired periods in seconds are stored instead of actual register values. Those are calculated from the periods during code generation.

In Clear Timer on Compare Match mode, two different registers can be used for storing the timer's top value. Similarly, the different PWM modes can be configured to use different top value registers or even predefined, static top values. The properties *CTCTop*, *PWMTop* and *PFCPWMTop* store this kind of information. The modes *Fast PWM* and *Phase Correct PWM* share the same options for the top value (PWMTop). *Phase and Frequency Correct PWM*, on the other hand, was designed to allow base rate changes at runtime. Hence, no static values are defined by Atmel for this mode of operation (PFCPWMTop).

All PWM modes as well as the CTC mode allow to drive up to three output pins (depending on which timer is used). The way in which they are driven is dependent on the mode of operation and the settings in the *PinModeA*, *PinModeB* and *PinModeC* properties. (See AT90CAN datasheet for details.)

2.1.2 Validation

Since the configuration process itself is error prone, automatic validation was developed for the TimerConfiguration data model. The validation results are shown graphically by the Eclipse editor next to the input fields where errors are detected. The results are also used by the code generator.

Both TimerConfiguration and UseCaseConfiguration objects can be validated. To achieve this, two validator classes were created. See Figure 3 for a class diagram.

The abstract class ***AValidator*** implements *PropertyChangeSupport* provided by java.beans. With PropertyChangeSupport, the validators allow other objects to attach to them and listen to changes on their properties. The editor uses this

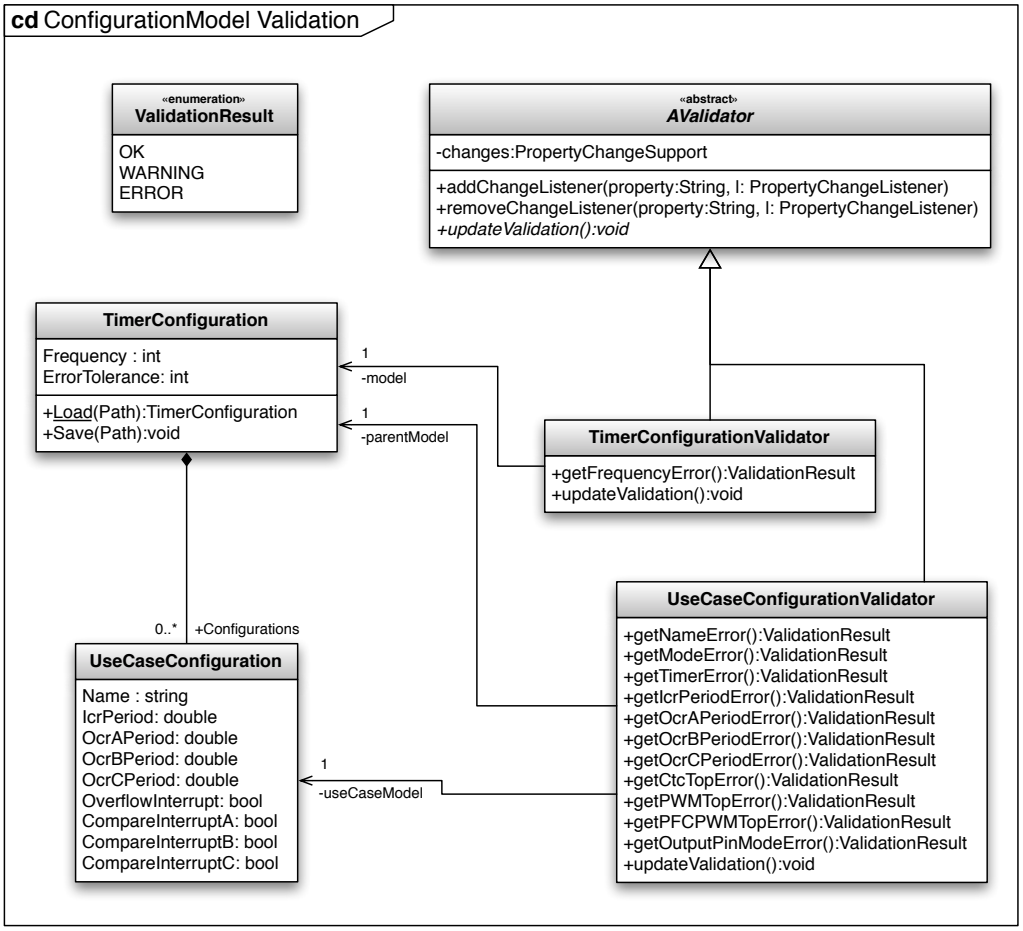

Figure 3: Timer Configuration Model Validation (UML Class Diagram)

feature to update the graphical widgets that display the current validation status. Objects can attach and detach with *addChangeListener(...)* and *removeChangeListener*. The abstract method *updateValidation* is implemented individually in the two validators described below. The purpose of this function is to make a validator reevaluate the model and update all validation results. New evaluation results are propagated by property change events.

TimerConfigurationValidator objects only need to verify the processor frequency. It only allows values ranging from 1 to 16,000,000 (Hz). The private field *model* provides access to the configured frequency in the data model. If the processor frequency in the model was set to an invalid value, *getFrequencyError()* will return *ValidationResult.ERROR*.

The class **UseCaseConfigurationValidator** tests all properties of a UseCaseConfiguration object for correctness. The field *useCaseModel* points to the UseCaseConfiguration object the validator refers to. The reference to the *parent-*

Model is necessary, because otherwise the UseCaseConfigurationValidator object could not know the processor frequency and error tolerance value.

The method *getNameError* checks, whether or not the value is a valid C identifier and if it is taken by another UseCaseConfiguration. It returns *ValidationResult.ERROR* if the name is not a valid C identifier or equal to another name.

Since not all combinations of timer, mode of operation and top value are allowed (see AT90CAN datasheet for details), the methods *getModeError*, *getTimerError*, *getCtcTopError*, *getPWMTopError* and *getPFCPWMTopError* are provided.

Depending on the timer, not all output pin modes are allowed for the first output compare pin. Whether the configuration for this pin is valid is determined by *getOutputPinModeError*.

The validation results of the period values depends on the processor frequency, the prescaler, the timers maximum value (8 or 16 Bit) and the desired period in seconds. To determine the validation status, the register value corresponding to the time period is calculated. It is then checked, whether the value is below the timer's maximum and top value. It is an *ValidationResult.ERROR*, if the register value exceeds either the top or maximum value. The register value is then calculated back into a (quantized) time period. The validator checks in a second step, whether the quantization error lies within the tolerated range (see TimerConfiguration's ErrorTolerance property). If the error tolerance is exceeded, the status is *ValidationResult.WARNING*. If there was no error and quantization led to a tolerable value, the validation status is *ValidationResult.OK*. The periods are validated with *getIcrPeriodError*, *getOcrAPeriodError*, *getOcrBPeriodError* and *getOcrCPeriodError*.

Whenever a property in the ViewModel is changed (e.g. by user input in the Editor window), two things happen. Firstly, the new value is copied from the ViewModel to the Data Model. Subsequently, the Validator is informed about the changes made to the Data Model via its `updateValidation()` method. The Validator will then recalculate *ValidationResult*s for every monitored property in the Data Model. The Editor will show the new validation results in the last step.

2.2 Eclipse Editor

The Eclipse Editor plugin consists of three subpackages. The *Plugin Controller* creates a *View*, which displays the graphical widgets on the editor surface. Furthermore, the plugin controller creates a *ViewModel* and is responsible for registering the **.tcxml* file type, a context menu entry for triggering code generation and a wizard for creating new **.tcxml* files.

Is is assumed, that the reader is familiar with writing Eclipse plugins. Please see one of the widely available introductory webpages (ex: `http://www.vogella.com/articles/EclipsePlugIn/article.html`).

The View consists of Java SWT standard widgets. Data binding techniques were used to keep the ViewModel and the View synchronous. They are provided by java.beans and not explained in further detail here. To focus user attention, widgets are displayed or hidden according to the current configuration. Some widgets are not applicable in specific situations, so the visibility of most SWT widgets is data bound as well.

The ViewModel provides all necessary values to display in the View. Like the model's validator, it provides java.beans' PropertyChangeSupport. This allows the View to bind all widget values to corresponding values in the ViewModel. Each ViewModel refers to one configuration model and one validator. When values in the graphical widgets are altered, the new value is automatically updated in the ViewModel due to the use of data binding. The ViewModel writes the new value to the Model and triggers the configuration model's validator afterwards to update its validation results. Since the View is also data bound to the model's validator, all warning and error indicators in the editor are updated automatically.

2.3 Code Generator Package

The code generator package embedded into the editor only includes the timer generator. All other generators were left out in the JAR shipped with the AVR Timer-Configuration plugin. The actual generation process is the same as described in the project group's main technical documentation (please see section 5.2 in that document).

The only difference is the way the generator is configured. In the description of the process in the main documentation, you have to set up the configuration process with a configuration file manually. This configuration step is unnecessary when calling the generation process from within the plugin, since the editor calls the timer generator directly. The data model is passed to the generator without saving and loading **.tcxml* files in between.