HOCHSCHULE KONSTANZ TECHNIK, WIRTSCHAFT UND GESTALTUNG
UNIVERSITY OF APPLIED SCIENCES

A Task Language with Analysis Tools for Mbeddr

Jan-Philipp Jägers

Konstanz, 30.06.2014

BACHELORARBEIT

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science (B. Sc.)

an der

Hochschule Konstanz

Technik, Wirtschaft und Gestaltung

Fakultät Informatik

Studiengang Software-Engineering

Thema: **A Task Language with Analysis Tools for Mbeddr**

Bachelorkandidat: Jan-Philipp Jägers, Mainaustraße 73, 78464 Konstanz

1. Prüfer: Prof. Dr. Michael Mächtel
2. Prüfer: Dipl.-Ing. (FH) Bernd Kolb

Ausgabedatum: 01.04.2014
Abgabedatum: 30.06.2014

Zusammenfassung

Thema:	A Task Language with Analysis Tools for Mbeddr
Bachelorkandidat:	Jan-Philipp Jägers
Firma:	itemis AG
Betreuer:	Prof. Dr. Michael Mächtel Dipl.-Ing. (FH) Bernd Kolb
Abgabedatum:	30.06.2014
Schlagworte:	mbeddr, task language extension, static execution time analysis, dynamic execution time analysis, schedulability analysis

Es sind mehrere Werkzeuge notwendig um die Einhaltung der zeitlichen Anforderungen eines Realzeitsystems zu beweisen. Diese Arbeit befasst sich mit der Integrierung und Implementierung aller notwendigen Komponenten um durch eine einzige Entwicklungsumgebung die Einhaltung der zeitlichen Anforderungen analysieren zu können. Es wird eine domänenspezifische Programmiersprache definiert um eine Abstraktion für Tasks bereitzustellen. Ein Programmierer kann mit dieser Task Sprache die Tasks seines Systems definieren und konfigurieren. Die eingegebenen Informationen werden dann verwendet um die Einhaltung der zeitlichen Anforderungen statisch zu überprüfen. Die Verwendung des mbeddr Projektes als Basis erlaubt eine enge Integration der Task Sprache in die mbeddr C Programmiersprache und Entwicklungsumgebung. Es wird ein externes Analysewerkzeug integriert um die maximale Ausführungszeit der definierten Tasks zu berechnen. Um die Ergebnisse des Analysewerkzeuges zu überprüfen, wird die Möglichkeit dynamische Ausführungszeiten zu messen in mbeddr implementiert. Die Fähigkeiten der implementierten statischen Überprüfung werden anhand eines für die AVR 8-Bit Hardwareplattform kompilierten Beispiels gezeigt.

Abstract

Multiple tools are necessary to prove the satisfaction of timing constraints a real-time system has. This thesis addresses the integration and implementation of all components required to perform static schedulability analysis in a single IDE. A domain specific language is defined to provide an abstraction for tasks. With this task language a programmer can define and configure the tasks of his system. The entered information is then used for static schedulability analysis. The usage of the mbeddr project as a basis allows a tight integration of the task language with the mbeddr C programming language and the IDE. An external tool is integrated to get the worst-case execution time of the defined tasks. To validate the results of the worst-case execution time analyzer, a dynamic execution time analysis capability is implemented for mbeddr. The features of the schedulability analyzer are demonstrated by an example compiled for the Atmel AVR 8-bit hardware platform.

Ehrenwörtliche Erklärung

Hiermit erkläre ich, *Jan-Philipp Jägers, geboren am 07.07.1989 in Frechen*, dass ich

(1) meine Bachelorarbeit mit dem Titel

A Task Language with Analysis Tools for Mbeddr

bei der itemis AG unter Anleitung von Prof. Dr. Michael Mächtel selbständig und ohne fremde Hilfe angefertigt und keine anderen als die angeführten Hilfen benutzt habe;

(2) die Übernahme wörtlicher Zitate, von Tabellen, Zeichnungen, Bildern und Programmen aus der Literatur oder anderen Quellen (Internet) sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Konstanz, 30.06.2014

(Unterschrift)

Acknowledgements

First of all I want to thank my family for the great support during my studies. Thank you! I wouldn't have been able to study in Konstanz without your support.

A big thank you goes to the mbeddr team at itemis AG and to my supervisor Bernd Kolb. Thank You!

Special thanks goes to my colleague Kolja Dummann. Without your patient support and your comments, this thesis would not have been possible. Thank you!

Additionally, I want to thank Professor Michael Mächtel for encouraging me to investigate this interesting topic. Thank you!

Next I want to thank Marco Eilers and Johannes Häußler for giving great comments on the text of the thesis. Thank you!

Contents

Abbreviations

CPU Central Processing Unit

DSR Deferred Service Routine

FCFS First-Come, First-Served

GCC GNU Compiler Collection

HAL Hardware Abstraction Layer
HTTP Hypertext Transfer Protocol

IDE Integrated Development Environment
I/O Input / Output
ISP In-System Programming
ISR Interrupt Service Routine

MPS Meta Programming System

OS Operating System

RAM Random-Access Memory

TCB Task Control Block

WCET Worst-Case Execution Time

Symbols

a	arrival, start or release time
d	relative deadline
e	maximum or worst-case execution time
d	relative deadline
p	periodicity or minimum separation of releases

Ti_t	Task with index i of type t where t is p for periodic or s for sporadic

1 Motivation

The majority of computers are used in embedded systems. Their production and importance is increasing. With more and more electronic devices controlling our environment the importance of safety is rising. This means that the embedded software running on the electronic devices must function correctly in any situation.

Today most tests regarding the functionality are non-formal. The test results are mostly collected during run-time of an embedded system. Only in the development process of safety-critical systems static functionality verification is used. Most static verification is done by model checking. At first, a model, for example a state machine, is defined. Secondly, the model is checked formally. Finally the model is used to generate source code templates, which can then be completed by the programmer.

This approach includes the problem of handling many different tools during the development process. Additionally, after the generated code has been edited, conflicts with model changes will come up.

The mbeddr project is a new approach which uses language oriented programming paradigms for developing embedded systems [Ward 1995] [Dmitriev 2004]. The mbeddr IDE allows to add and customize programming languages and makes it easy to create a programming language which supports a tight integration of external tools with the sourcecode editor. A custom language can provide the information which external tools need. For example, the addition of a state machine syntax allows to extract the state machine information for a verification tool directly from the source code. The external verification tool can then analyze the state machine statically.

The advantage is that the code, programmed with a custom, domain specific programming language, is entered at the same place as the code programmed with existing languages. The

tight integration has many benefits, for example the ability to integrate type checking of a custom language with an existing one.

The motivation for this thesis is to provide developers with a productive tool helping to increase the safety of embedded systems.

1.1 Objectives

The main objective is to build a language extension for mbeddr which provides an abstraction for tasks. The new language syntax should provide the configuration and definition of tasks as well as the concepts to control these tasks during run-time.

The language extension should also support schedulability analysis by using the worst-case execution time (WCET) of the tasks. The tasks' execution time is calculated statically by an external tool and the results are imported by the mbeddr IDE for further calculations. The integration of the WCET calculation tool has to be as generic as possible, so that different tools can be supported in the future.

The extension must be able to support different hardware and software platforms. It should be possible to adapt the extension for several operating systems as well as custom schedulers. To demonstrate the features of the task language a simple task scheduler shall be implemented. For the demonstration the AVR 8-Bit hardware architecture is used.

For achieving these goals multiple separate components will be created. The component structure and the separation of concerns is covered in Chapter 4.

1.2 Work Units

The work is separated into different units. This section gives a short overview of these work units. Some of the results are independent of each other and can be used separately as generic features of mbeddr. Subsequent chapters cover the work units and results in detail.

At first, the requirements for the task language are determined. This includes the schematic syntax of the language and the functionalities it should provide. Moreover the implementation of the language concepts is included in this work unit.

The second unit is the selection, evaluation and integration of a WCET calculator tool. The integration includes a generic interface, so that additional tools may be supported in the future. Independently of the remaining work units the integration provides WCET calculations of a program or parts of a program. The calculation results are presented in the source code editor.

The third unit is the implementation of a dynamic execution time measurement capability for mbeddr. The results of the dynamic execution time measurement are compared to the results of the static execution time calculator to estimate the quality of the static analysis. The dynamic execution time measurement capability is also independent of the other work units. The analysis results are annotated to source code elements, such as function calls.

The fourth unit is the proof of concept implementation of a platform, the mbeddr AVR platform, which can be used with the task language. This platform includes a simple task scheduler.

The fifth unit is the creation of the schedulability analyzer for scheduler of the mbeddr AVR platform. The analyzer checks if all tasks meet their deadlines in all situations. Additionally a time span of the worst-case schedule of the tasks is visualized by presenting a diagram in the mbeddr IDE.

The sixth unit is the verification of the mbeddr scheduler by measuring the task activations on a running system.

1.3 Thesis Content

The following content is structured into seven chapters. The chapter *Context* describes the basic technologies and concepts used. The chapter *Task Language and Analysis Tools Requirements* presents the requirements determined for the topics presented in this thesis. It includes the requirements of the work units one to five. The chapter *Component Structure* gives an overview of the implemented components. The chapter *Task Language* describes the task language and the mbeddr AVR platform in detail. The results of the work units one and four are presented here. The chapter *Execution Time Analysis* covers the work units two and three and the chapter *Schedulability Analysis* the work units five and six. The final chapter *Summary* concludes with the criticism of the results.

2 Context

This chapter describes the basic principles and technologies on which the work for this thesis is based.

2.1 Embedded Systems

Embedded systems are computer systems which often have a dedicated single purpose. They mainly have especially designed hardware. Their software footprint is reduced to a minimum, to minimize the hardware resource costs. They are often part of a larger system.

A typical embedded system has got sensors for capturing some environmental data and actuators for reacting according to the input data. It is then coupled to its environment, which can imply the need of real-time computation constraints [Stallings 2008, pp. 594 - 596].

2.2 Real-Time Systems

A real-time system is a computational system with timing constraints. The computation of real-time system tasks must be finished within set deadlines. An engine controller, for example, must calculate the fuel injection time point for every driving shaft revolution. As the injection time point depends on the revolution speed, it must be calculated between the start of the new revolution and the actual injection time point.

For the different tasks in a real-time system the timing constraints are defined as requirements. The definition allows a formal verification of the systems. Often run-time tests are

used for gathering the timing behavior. The output values are measured, taking into account as many input parameters as possible, so that most possible situations are covered.

A different verification technique is the static analysis of the tasks in a system. The computation time of a task can be estimated by analyzers. The results are then compared to the defined requirements.

To define the timing requirements of a task it must be characterized. The following attributes constitute the timing parameters of a task [Cheng 2003, 1 85]:

a: arrival, start or release time This parameter describes the time point at which a task is ready to be executed. The actual start of the execution can be delayed by the scheduler if a resource e.g. the CPU, is already busy.

e: maximum or worst-case execution time The worst-case execution time (WCET) is the maximum time a task needs for its computation. For the WCET calculation the control flow with the maximum execution time must be taken.

d: relative deadline The deadline defines the time point at which the execution of the task must be completed. The relative deadline is specified relative to the arrival time of the task.

p: periodicity, minimum separation of releases This parameter describes the period of a task. For aperiodic tasks the parameter describes the period of the minimum separation between two consecutive releases.

The tasks of a system can be divided into two different groups:

sporadic task A sporadic task is not released at pre-defined time points. Its arrival time is typically triggered by an external event, for example when a driving shaft revolution is completed. A sporadic task has a minimum separation between two consecutive releases, which is often determined by the physical characteristics of the environment e.g. the maximum driving shaft revolution speed.

periodic task A periodic task arises periodically. The arrival time of the task is known before its release. This task type is often used to get environmental data from a sensor, such as a temperature sensor.

Consequently, a task is defined by Ti_t, where t is the tasks type (s for sporadic or p for periodic) and i the task's index. A periodic task has repetitive releases, thus the multiple arrival times of task Ti_p are $ai_n = ai_0 + pi * n, n \in \{0, 1, \cdots\}$.

The CPU scheduler of a system is responsible for the execution of the tasks. It picks a task from a task set according to its algorithm and assigns the chosen task to the CPU. For designing real-time systems it is essential to analyze the time a task is assigned to the CPU together with the applied scheduler technique and algorithm, because the information is required to verify the satisfaction of timing constraints. The analysis is done by using the estimated WCET of all tasks in a system. According to the various scheduler algorithms different verification formulas exist.

A visualized worst-case scenario schedule of a task set is depicted in figure 2.1. The following tasks are included in the task set:

$$T1_p: \quad a1_0 = 0,\ e1 = 786,\ p1 = 8192$$
$$T2_p: \quad a2_0 = 0,\ e2 = 1186,\ p2 = 4096$$
$$T3_p: \quad a3_0 = 2048,\ e3 = 986,\ p3 = 3072$$

$T1_p$ has the highest priority (0) and $T3_p$ the lowest. The schedule diagram shows the arrival times and which task is assigned to the resource (CPU) at a time point. The three tasks are ordered top down by the tasks priority. The x-axis shows the execution time in CPU ticks. The arrival time points are marked with an upward pointing arrow. A gray box indicates that a task is assigned to the CPU by the scheduler.

To make a task available for the scheduler it must be registered via the task control interface. The task control interface also provides the functionalities for controlling the tasks in a system at run-time.

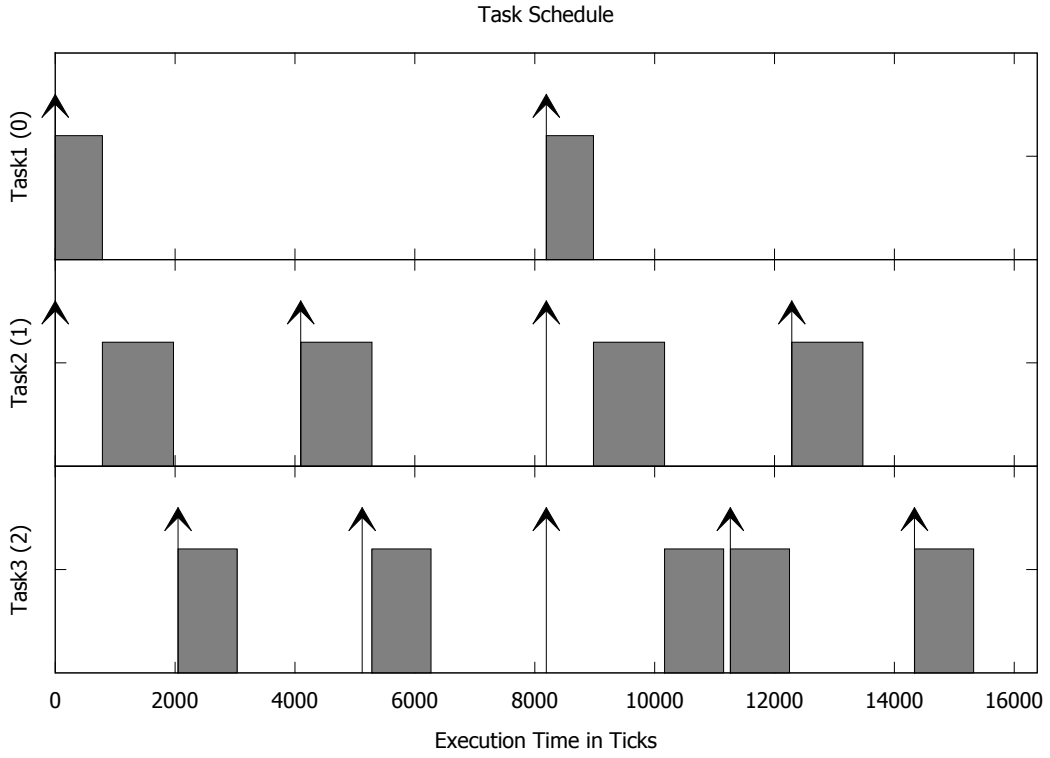

Figure 2.1: The figure illustrates a worst-case schedule of three periodic tasks. The arrival time points are marked with an upward pointing arrow. The gray boxes show the time spans within which a task is executed.

2.3 Mbeddr

The majority of embedded systems are programmed by using only a few common programming languages. As these languages have their origin in the distant past, like the C programming language, they do not support modern programming language paradigms. Paradigms like object oriented programming can only be implemented circumstantially. This makes it very difficult to build developer tools which can provide typical modern editing features. For example, auto-completion of C source code is very difficult to implement, because the source code symbols, which can be chosen in a specific context, often depend heavily on the C preprocessor.

Mbeddr[1] solves these problems. It is a modern integrated development environment (IDE) for programming embedded systems. Additionally a C-like base programming language, including some improvements to C, is provided. Mbeddr uses and supports language oriented

[1] `http://mbeddr.com`

programming paradigms[2]. This means that the base language can be easily customized and extended by domain specific languages. The mbeddr IDE is shown in Figure 2.2.

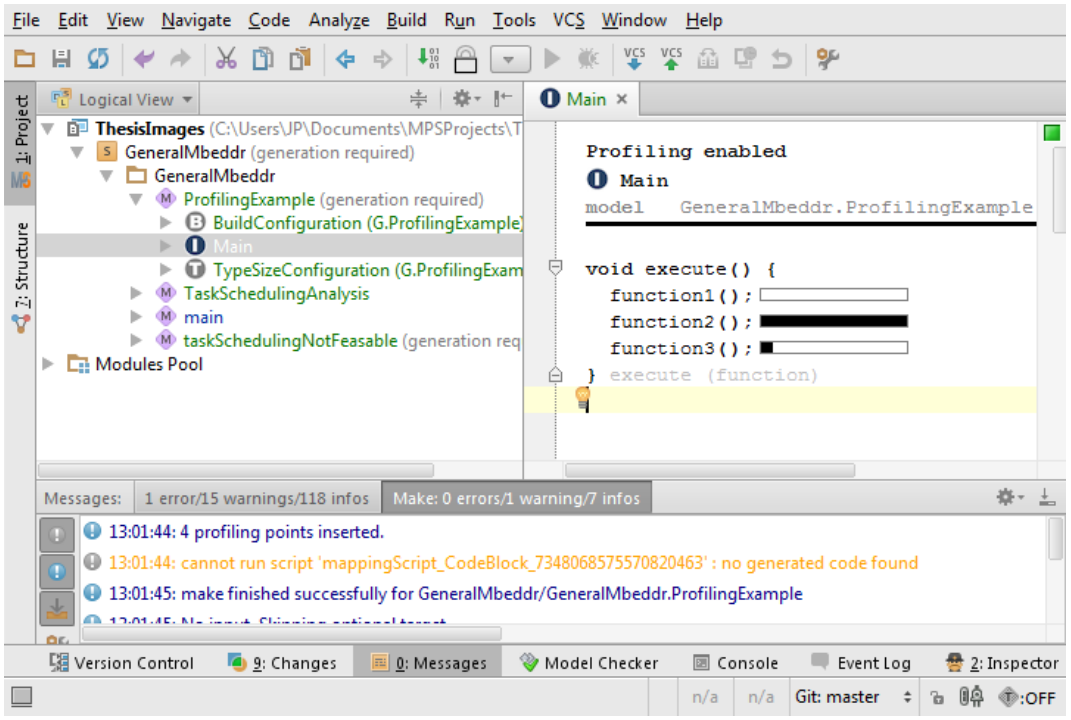

Figure 2.2: The mbeddr IDE. On the left side the logical view of projects and their contents is shown. On the right side an implementation module of the mbeddr base language is shown. An implementation module represents a .c and .h file in the C programming language.

The different languages are separated into language modules. They can depend on or coexist next to each other. Figure 2.3 shows different language modules. The mbeddr C core, including the mbeddr C base language, has no dependencies to other modules. But the components, physical units, state machines and further languages cannot be used without the base language. New languages could be added on top of one or multiple existing languages.

The mbeddr base language is similar to C. A few concepts of the original C programming language are changed to increase the consistency and safety. The C preprocessor, for example, is replaced by language extensions which explicitly support the functionalities the preprocessor is used for. Different source code variants can be generated by using the mbeddr presence condition language extension. This extension allows to configure different product lines and to decide which part of the source code is included in a binary. Many more language extensions are already included in mbeddr [Voelter, Ratiu, Schaetz, et al. 2012].

[2]More on language oriented programming Ward 1995, Dmitriev 2004, Völter et al. 2013

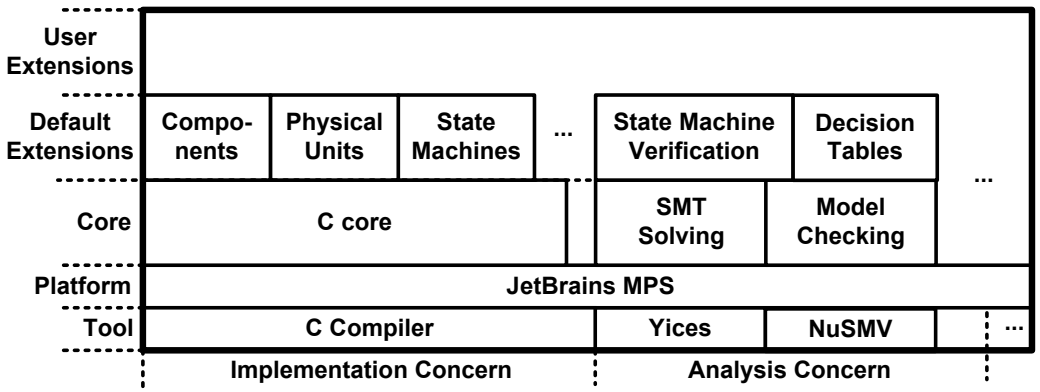

Figure 2.3: The mbeddr components, structured into layers, are shown. Mbeddr relies on the JetBrains MPS Language Workbench. The C core includes the mbeddr C base language. Multiple default languages, extending the base language, can be used by default. Additionally analysis capabilities are included. External tools are the C compiler and analysis tools. *Note:* Modified from [Voelter, Ratiu, Kolb, et al. 2013, p. 11]. Modified with permission.

Figure 2.4 shows some source code written in the mbeddr IDE. The code is written inside an implementation module. An implementation module is the representation of the .c and .h files. If a symbol is marked as exported, like the `function1()` in the figure, it can be referenced from other implementation modules. Other implementation modules only have to import the implementation module where the symbol is defined.

Source code written in the mbeddr IDE is generated into C99 code. The generation is done incrementally. For each language a generator must be specified. The generator is executed before a generator of a depending language. This is done by specifying the dependencies for each generator explicitly. Figure 2.5 depicts the generation process. Code written in the language A is reduced to mbeddr C base language code. The conversion and reduction rules are defined in the language A generator. In the next iteration the code using the mbeddr C base language is reduced to C99 textual code.

Mbeddr is based on the projectional editor Meta Programming System (MPS)[3]. With projectional editors the structured model of the entered code is edited via a visual projection rather than editing the source code as text. Traditional editors must parse the entered text in order to analyze the code statically. In contrast, the projectional editor does not have

[3] `http://www.jetbrains.com/mps`

to parse the source code because it was entered via the projections directly into the abstract code representation - the abstract syntax tree [Völter et al. 2013, p. 68]. The great advantage of MPS is that it allows to easily create new languages and to extend existing ones.

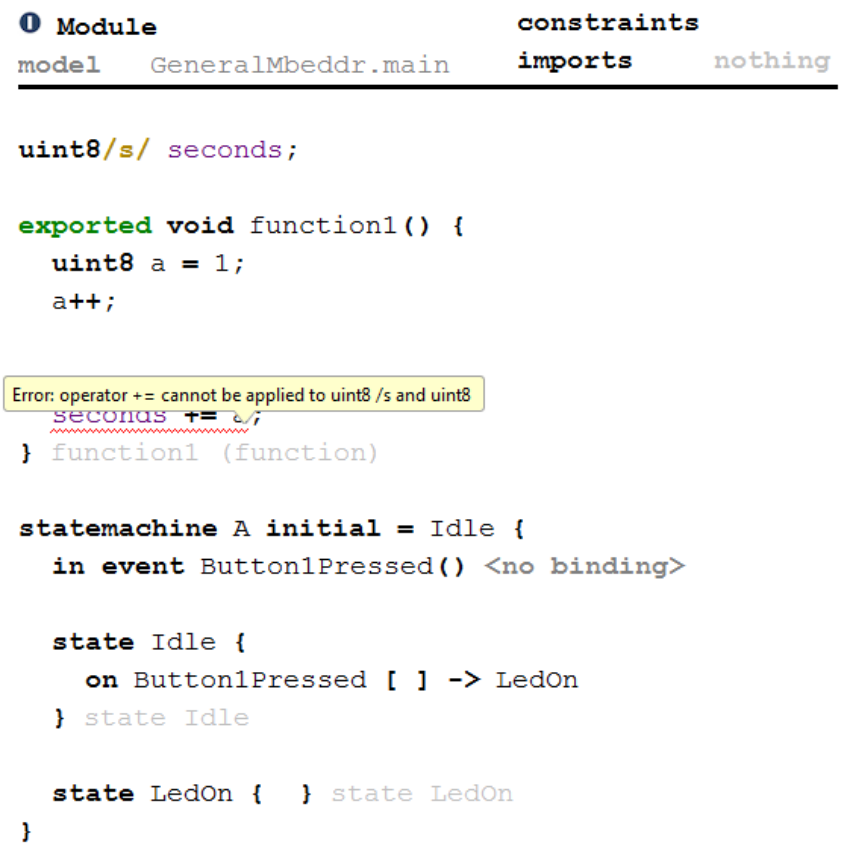

Figure 2.4: An implementation module with code using different languages is shown. By using the physical unit language new types can be defined. Type checking ensures that only values of the same physical unit can be assigned to each other. A variable with the physical unit second is defined at the top. A state machine is defined by using a domain specific language at the bottom.

2.4 Components Integration

The work of the thesis is the implementation of extensions for mbeddr. This includes extensions for the mbeddr C base language and the IDE are created. The extensions are separate components which a user of the IDE can include to an mbeddr project. The installation of external tools is required for the static execution time analysis and the diagram creation.

In Figure 2.6 the added components are marked blue. The component details are described in the next chapters.

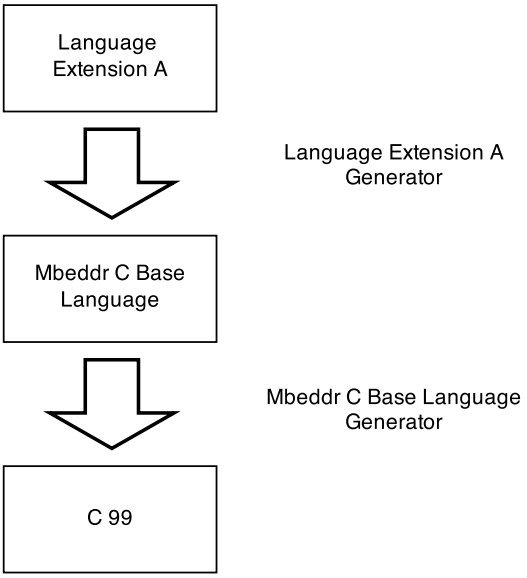

Figure 2.5: The mbeddr code generation process is shown. In the first iteration code written with language A is reduced to code using the mbeddr C base language. In the next iteration C99 textual code is generated.

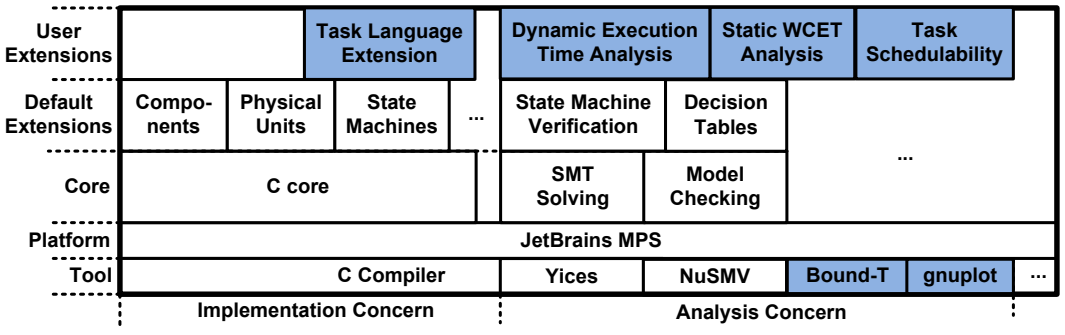

Figure 2.6: The blue components shown in this figure are added to mbeddr. *Note:* Modified from [Voelter, Ratiu, Kolb, et al. 2013, p. 11]. Modified with permission.

3 Task Language and Analysis Tools Requirements

This chapter describes the determined requirements for a task language and the necessary components for the schedulability analysis.

At first, existing task control interfaces and their usage were examined to define the requirements for a task language. Different operating systems and operating system interfaces were regarded, because one of the goals is the ability to simply add the support for additional task control interfaces.

Secondly, the requirements for the integration of the WCET analysis into the mbeddr IDE are listed. The possibility of supporting multiple external execution time analyzers is one objective.

Thirdly, the features which the schedulability analyzer should provide are noted. The analyzer should determine if a schedule of a task set is feasible and present a time span of the schedule as a diagram to the mbeddr IDE user.

Fourthly, the requirements for the mbeddr AVR platform, which can be used with the task language and schedulability analyzer, are pointed out.

3.1 Existing Task Control Interfaces

Most operating systems define different execution contexts. The execution context for a task is chosen according to its purpose. Often the execution context thread and interrupt service

routine (ISR) are provided by a platform. A thread is controlled by the scheduler and the execution sequence of multiple threads is specified by the scheduler algorithm.

An ISR is executed spontaneously when an interrupt occurs. Some platforms allow multiple ISRs with different priorities for one interrupt. Particular platforms allow to define deferred service routines (DSRs). A DSR has an ISR counterpart and is executed after the associated ISR by the scheduler, so other ISRs can interrupt the execution of a DSR if the respective interrupt occurs.

The execution context also defines which platform interface functions are allowed to call. For example, blocking platform interface functions must not be called in an ISR, because this could lead to a blocking of the whole system.

The task control interfaces of the operating systems eCos[1] and Femto OS[2] were examined. Additionally the interfaces RTAI[3] and Posix[4] were regarded. It should be possible to add the support for these task control interfaces to the task language.

eCos The Embedded Configurable Operating System is an open source real-time operating system. The key feature is its configuration system. A customized operating system can be built by choosing the necessary components from the provided features of eCos. eCos supports the different task execution contexts thread, ISR and DSR. An execution context limits the operating system interface functions that can be called. Blocking task synchronization functions, for example, must not be called in ISRs. The tasks are registered with the scheduler at run-time.

Femto OS This small operating system is designed for the Atmel AVR 8-bit processor family. It supports threads and ISRs. The tasks are registered statically.

RTAI The real-time application interface introduces real-time capabilities to the linux kernel. It is designed as an additional hardware abstraction layer (HAL). By applying some modifications to the linux kernel, the kernel can be interrupted by RTAI. This allows RTAI to execute real-time tasks by using its own scheduler.

[1] `http://ecos.sourceware.org`
[2] `http://www.femtoos.org`
[3] `https://www.rtai.org`
[4] `http://www.pasc.org`

Posix The portable operating system interface defines an interface, which many operating systems support. Consequently a program can be ported from one operating system to another with little effort. Posix also has a defined interface for controlling threads. A mechanism similar to interrupts is provided with Posix signals. All threads and signals are registered at run-time.

3.2 Task Language

The following paragraphs point out the requirements for a task language. Some examples are given by using pseudo C code.

The task language must provide a language syntax which allows the definition and configuration of tasks. The task definition contains the execution statements of a task. The task configuration includes the properties of a task, e.g. the scheduler priority. Furthermore, controlling the tasks at run-time has to be supported for most underlying task control interfaces.

3.2.1 Portability

The configuration and definition of tasks in a system should be very easy. Moreover the configuration method should be similar for different underlying task control interfaces. This allows to define tasks independently of the underlying platform, similar to the basic idea of Posix. The task language must be able to provide different configuration possibilities. It must be possible, for example, to write program code for different task execution contexts e.g. it must be possible to write DSR code for eCos, but not for Femto OS.

3.2.2 Task Configuration Properties

A separation of the task configuration and the definition of the task is of advantage. It should be possible to place the configuration of several tasks at the same location. Posix threads, for example, are configured by passing a configuration object to the `pthread_create()` function. Thus the different scheduling properties like the priority of threads can be spread all over the source code. By configuring the tasks in one place in a task configuration list,

the different configuration values of all tasks can be efficiently surveyed and changed. This idea is often implemented in C by placing configuration properties as C defines, like priority of tasks, in one location. The task language should pursue this approach.

3.2.3 Task Definition and Task Configuration Separation

By separating the task execution statements from the configuration of a task, the execution context is chosen with a task configuration item. Thus it is possible to write the executable code of a task, the task definition, first and decide during entering its configuration if the task should be executed in, for example, a thread or an ISR context.

Figure 3.1 shows the proposed relation between task configurations and task definitions. A task configuration list includes multiple task configurations. A task configuration references a task definition. This relation allows multiple task configurations to reference the same task definition. For example, the configuration of two threads which process data the same way in parallel, but on different CPU cores, can reference the same task definition. The CPU core, to which a tasks should be assigned, can be defined in the task configurations.

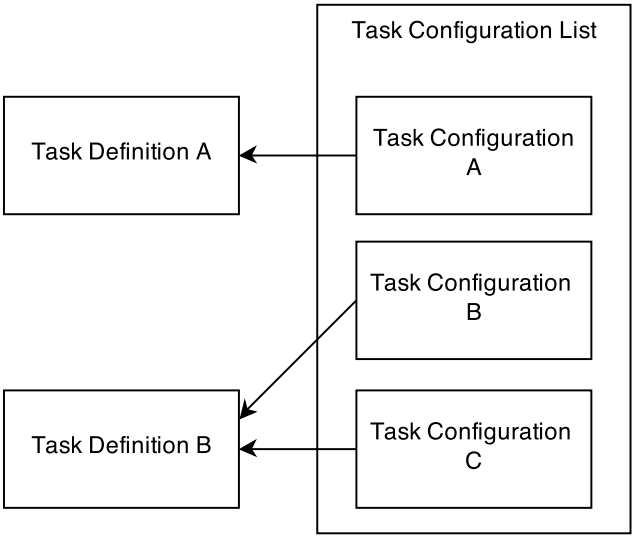

Figure 3.1: This figure shows the proposed relation between the task configurations and task definitions. A task definition contains execution statements, a task configuration allows to configure a task and to reference a task definition.

3.2.4 Multiple Task Configuration Lists

It should be possible to include multiple task configuration lists in one executable, because libraries which bring their own tasks must be supported. A HTTP server library, for example, could have its own tasks.

3.2.5 Task Startup and Registration

The run-time task interactions, which a task control interface may provide, must be supported. To define requirements the way in which task control interfaces are used, must be examined. Typical embedded system tasks are registered with the operating system during the startup of the system. Threads are started and interrupts are registered via the task control interface after a basic setup of the system.

3.2.6 Different Thread Types

A task definition must support the same functionalities which traditional program code, that is defining a task, can. When a thread is started it often allocates a resource, for example a serial port to communicate with a sensor, and subsequently enters an infinite loop reading and processing the sensor value periodically. A schematic example of this scenario is given in Listing 3.1. To inject the dependency of the specific resource, the necessary information is often passed to a thread via a function argument.

Another option is to allocate the resource before the start of the thread and to pass the resource itself as an argument. Listing 3.2 depicts the second scenario. The difference between the first and the second scenario is the time when the resource is allocated. In the second scenario the resource is allocated at the startup of the system and could be used by multiple threads, for example a memory region. In contrast, the resource in the first scenario is allocated at a time point which the scheduler algorithm chooses.

Further scenarios can be built by changing the resource type from a sensor to an actor. Often, these types of threads are waiting for a resource to produce data, which the thread processes and then passes to an actor.

```
thread_t thread_sensor1;

void system_startup() {
    start( &thread_sensor1,
        &read_sensor_thread,
        "sensor1" );
}

void read_sensor_thread( void * thread_argument ) {
    sensor_t * sensor_resource =
                    allocate( thread_argument );

    while( true ) {
        uint8_t sensor_value = read( sensor_resource );
        process_sensor_value( sensor_value );

        wait_until_next_period();
    }
}
```

Listing 3.1: The listing shows the creation and start of a thread which allocates a resource inside the thread execution function `read_sensor_thread()`. The resource name is passed as an argument by the `start()` function.

```
thread_t thread_sensor1;

void system_startup() {
    sensor_t * sensor_resource = allocate( thread_argument );
    start( &thread_sensor1,
        &read_sensor_thread,
        sensor_resource );
}

void read_sensor_thread( void * thread_argument ) {
    while( true ) {
        uint8_t sensor_value = read( thread_argument );
        process_sensor_value( sensor_value );

        wait_until_next_period();
    }
}
```

Listing 3.2: The listing shows the start of a thread. A resource is allocated before the thread start and passed to the thread execution function `read_sensor_thread()` as an argument. The allocated resource could be passed to multiple threads.

```
data_received_context_t interrupt_context;
interrupt_t data_received_interrupt;

void system_startup() {
    register_isr( &data_received_interrupt,
                  &data_recv_isr,
                  &interrupt_context );
}

void data_recv_isr( data_received_context_t * isr_arg ) {
    isr_arg->buffer[isr_arg->index] =
                read_register( DATA_REGISTER_1 );
    isr_arg->index++;
}
```

Listing 3.3: The listing shows the registration of an ISR. A context data object is passed to the ISR as an argument every time the corresponding interrupt is triggered.

As shown, different functionalities must be provided by a task definition programmed for a thread execution context. It should be possible to define tasks including an infinite loop statement, to place statements before the infinite loop and to pass variables to the task function.

3.2.7 Interrupts

Other important task definitions are ISRs. Usually their instructions do not contain an infinite loop, so that they are run-to-completion tasks. An ISR often needs some context variables, which data is kept beyond a single ISR execution. For example, a data buffer is often needed in an ISR which handles incoming serial data. Listing 3.3 gives an example scenario.

The example scenario shows that run-to-completion task definitions must be available. Additionally it must be possible to have context variables containing data that is stored beyond a single task execution.

3.3 Static Execution Time Analysis

This section describes the requirements for the possibility to analyze the WCET of code segments. The concrete calculation should be done by an external tool.

3.3.1 Different Tools

One goal is to make the integration of the external tool as generic as possible, so that multiple analyzers can be supported. Therefore an interface for the integration must be implemented. The user of the mbeddr IDE should have the option to choose from different supported analyzer tools.

3.3.2 Assertions

The control flow of a program cannot always be determined. Sometimes the number of loop repetitions can only be known at run-time, for example, if the number of repeats depends on an externally measured value. To solve this problem it must be possible to annotate the source code with additional information, such as the maximum repetition value for loops. These assertions must then be passed to the external analyzer tool.

3.3.3 Analysis Results

The results of the analyzing process should be presented in the source code editor. Additionally they must be available for further processing, for example for the schedulability analyzer.

3.4 Dynamic Execution Time Analysis

For this thesis dynamic execution time analysis is used to estimate the quality of the static analysis. The main requirement is the possibility to measure the execution time of function calls. The analysis must be applicable together with restricted hardware resources like an AVR 8-bit micro-controller.

3.5 Schedulability Analysis

The schedulability analyzer must decide if a schedule of all tasks in a system is feasible. The task configuration lists and task definitions should be used to gather the information for the analysis.

3.5.1 Analysis Algorithm Selection

As the analysis algorithm heavily depends on the scheduler, it must be possible to provide different analysis algorithms.

3.5.2 Analysis Results

The schedulability analysis results should be presented in the mbeddr IDE. The result whether the schedule of a task set is feasible must be presented as true or false. Additionally a time span of the tasks schedule should be presented as a diagram. If a schedule is not feasible, a diagram, including the first passed deadline, should be presented.

3.6 Mbeddr AVR Platform

For a proof of concept the task language, WCET analyzer and the schedulability analyzer should be adapted to one example platform. The implementation should be done for the AVR 8-bit hardware. The software platform, in this case only a scheduler, must also be programmed.

The custom scheduler for the platform has to support run-to-completion periodic tasks. The relative deadlines of the tasks are set to their periodic value.

It should be possible to add support for interrupts to the platform, but for the proof of concept they are not implemented.

4 Component Structure

Several components are implemented for this thesis. The separation is done according to their concerns. Figure 4.1 gives an overview.

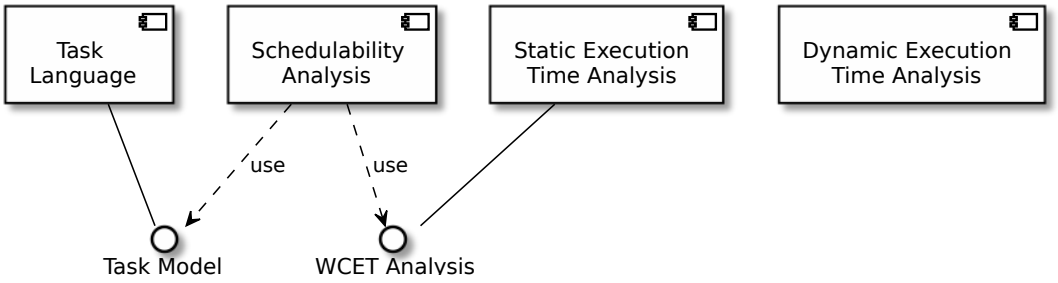

Figure 4.1: This figure depicts the four components implemented for this thesis. The Schedulability analysis component uses the interface of the task language and static execution time analysis component to gather the information needed.

The task language provides language concepts for defining and configuring tasks. It provides a model of all tasks in an mbeddr project, which can be requested via an interface.

The WCET of program parts can be requested from the interface of the static execution time analysis component. This component hides the way an external tool is integrated into the mbeddr IDE.

The dynamic execution time analysis capabilities are implemented in a separate component. With this component the execution time of program parts can be measured during run-time. It is used to estimate the quality of the external static analysis tool.

The schedulability analysis component provides an interface for schedulability analysis algorithms and enables the creation of schedule diagrams. For the mbeddr AVR platform an implementation of the interface is presented. The schedulability analyzer depends on the

execution time analysis component, because it uses its interface for gathering the WCET of program parts.

The implementation descriptions in the following chapters are referencing the corresponding requirements. The reference is given by the section number where a requirement is defined in the format (Requirement Section 3.2.1).

5 Task Language

The task language provides language concepts for the definition and configuration of tasks. The concepts are designed with respect to different underlying task control interfaces of operating systems.

The language concepts and components of the task language are described in detail in the following sections. An overview is given in figure 5.1.

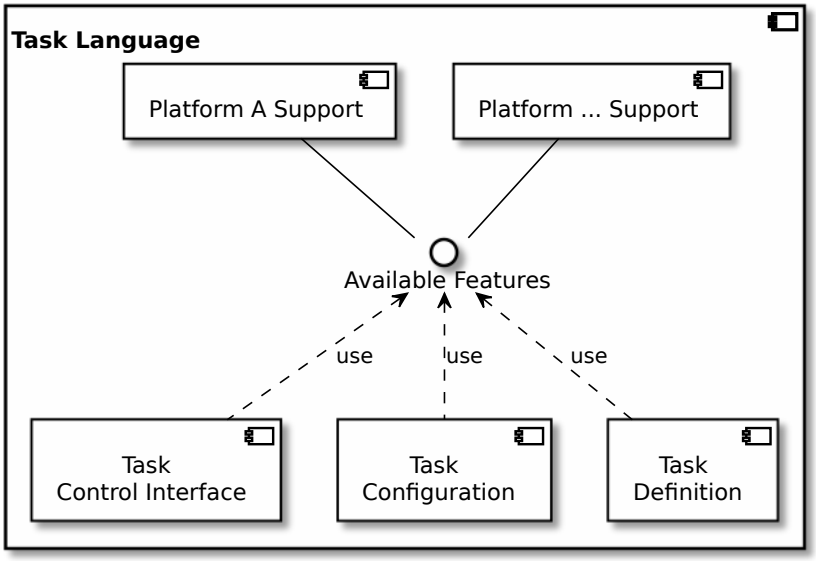

Figure 5.1: This figure shows the components of the task language. A platform support component can restrict or extend the features available for the mbeddr user. The task control interface, task configuration and task definition components request the restriction and extension information via an interface.

The first component provides language concepts for task configuration lists. A task configuration list allows the user to configure tasks in an mbeddr project. The second component provides concepts for task definitions. A task definition is a layout, into which the concrete executable instructions of a task can be inserted (Requirement Section 3.2.3).

The third component includes language concepts of the task control interface. The concepts allow interactions with the functionalities exposed by the task language. These interactions, for example, can be used to register tasks with the scheduler and to control tasks during the execution of a system from different code locations.

Remaining components are responsible for the support of different platforms. Each component provides an interface where the available features of its platform can be requested. The other components use this interface to query which options they display to the mbeddr user.

Each platform support component includes code generator definitions which cause the generation of platform specific code. In this way the functionalities of the task language are mapped to the underlying task control interface of the used platform.

5.1 Platform Support Component

A platform support component provides the platform specific implementations for the task language. Additional task configuration items may be supplied. Furthermore, all constraints a task configuration item may have for the task definition, referenced by the task configuration, are defined. Additionally, the platform support component defines which task definition layouts and task control interface interactions are available in the code.

The platform support component, that should be used for a project, can be chosen in the mbeddr build configuration. The platform is selected via the configuration item `task platform`. Figure 5.2 shows a part of an mbeddr build configuration. Here the `mbeddr avr` platform support component is chosen.

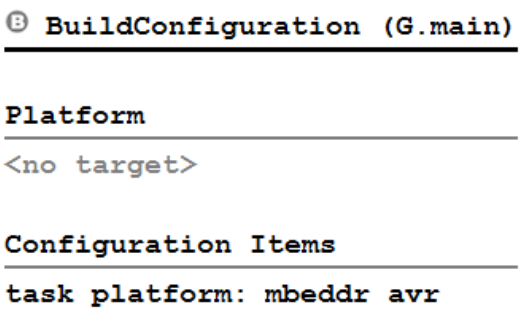

Figure 5.2: To be able to select a platform support component a build configuration item is provided for each platform.

A platform support component is responsible for the code generation. Figure 5.3 illustrates the platform specific code generation schematically. The generator input consists of the instances of the language concepts, i.e. the task configuration lists, task definitions and task control interface interactions. The generator transforms this input to platform specific mbeddr base language code.

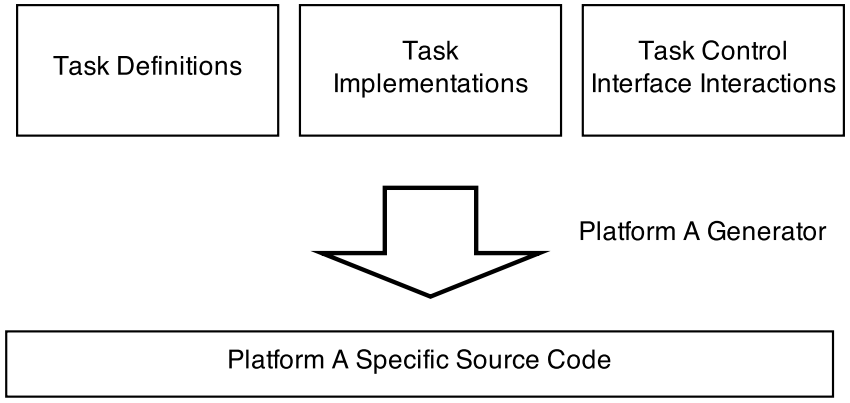

Figure 5.3: The generator of a platform support component converts the task definitions, task configurations and task control interface interactions into platform specific mbeddr base language code.

The implementation details of the mbeddr AVR platform are covered in Section 5.5.

5.2 Task Configuration List

A task configuration list provides an overview of the tasks and their configuration in a system. In a list tasks are configured with multiple properties (Requirement Section 3.2.2). A task configuration list can be placed anywhere in an mbeddr module. Thus multiple lists can be created (Requirement Section 3.2.4).

In Figure 5.4 a task configuration list is shown. The list contains two task configurations. Each task configuration has a name, so that it can be referenced from different source code locations. This is used by the task control interface. The task definition reference property holds a reference to a task definition, which contains the executable instructions. Additional properties depend on the platform and may also have constraints for the referenced task definition. If, for example, the ISR context is chosen, no infinite loop is allowed in the task definition. Another example for a constraint is that every task configuration must have a

priority value defined. These specific properties and constraints are provided by the chosen platform support component (Requirement Section 3.2.1).

```
TaskConfigurations          constraints
model   GeneralMbeddr.main  imports     nothing

task configuration list SystemTasks {
  configuration Task1
    task definition: Task1Definition
    priority: 1
    period: 1000
  configuration Task2
    task definition: Task2Definition
    priority: 2
    period: 500
}
```

Figure 5.4: The presented task configuration list has two items. The different configuration properties are set for each configuration item.

5.3 Task Definition

A task definition is a template where the task instructions can be inserted. Multiple different layouts exist, so that thread and interrupt scenarios are covered (Requirement Section 3.2.6, Section 3.2.7). The chosen platform support component defines which layouts are available. The following paragraphs describe a task definition template. Additional templates can be provided by platform support components. In the described template three different sections for instructions are available. The sections are listed below.

Context: In the optional *Context* section the context variables of the tasks can be defined. Their values are stored in the heap memory and thus have a life cycle beyond a single task execution. They are accessible in the *Init* and *Execution* section.

Init: In the *Init* section the context variables can be set. The values of the optional arguments are passed from the responsible task control interface interaction. This could be, for example, the `start` interaction, which executes this section.

Execution: According to the computational behaviour of a task, different *Execution* sections are available. The selection of an *Execution* section is mandatory. The possible

layouts are platform dependent and thus defined by the chosen platform. The run-to-completion *Execution* section can be used if no infinite loop task instruction is needed and the task computation time is finite. A loop *Execution* section can be used if the instructions should be repeated infinitely.

A run-to-completion task definition is shown in figure 5.5. A task definition can be placed in an mbeddr implementation module together with additional code. The additional code elements can be referenced from the *Execution* and *Init* sections of the task definition. For example, a function could be called or a global variable could be referenced out of these sections.

```
Task1                              constraints
model   GeneralMbeddr.main         imports      nothing

task definition Task1Definition {
  context {
    uint8 a;
  }

  init(uint8 argument1) {
    context->a = 0;
    context->a += argument1;
  }

  execution {
    for (uint8 i = 0; i < context->a; i++) {

    } for
  }
}
```

Figure 5.5: The figure shows a run-to-completion task definition with a *Context*, *Init* and *Execution* section.

5.4 Task Control Interface

The task control interface provides interactions with the functionalities exposed by the task language. These interactions are available at various code locations. As different platforms support different interactions with the tasks, the available interactions are restricted

by the chosen platform. The platform can also add platform specific interactions (Requirement Section 3.2.5).

Right now two different interactions, which a platform may support, are defined.

startAll This interaction can be used to start all tasks of a task configuration list. The *Init* sections of the referenced task definitions are invoked and the tasks are registered with the scheduler. For all arguments defined at the *Init* sections of the referenced task definitions, values must be supplied.

start This interaction starts a specific task of a task configuration list. The *Init* section of the referenced task definition is invoked and the task is registered with the scheduler. For the arguments defined in the *Init* section of the referenced task definition values must be specified.

Figure 5.6 shows the `startAll` interaction. `Task1` and `Task2` are task configurations in the task configuration list `SystemTasks`. The *Init* section of the referenced task definition of `Task1` takes one argument, that of `Task2` two. If no referenced task definition defines arguments at their *Init* sections, no arguments need to be specified for the `startAll` interaction.

```
TaskControlInterface        constraints
model   GeneralMbeddr.main  imports     TaskConfigurations

void function1() {
  uint8 argument1 = 0;
  uint8 argument2 = 0;
  uint8 argument3 = 0;

  SystemTasks.startAll( Task1: { argument1 }, Task2: { argument2, argument3 } );
} function1 (function)
```

Figure 5.6: The figure shows the `startAll` task control interface interaction. For the initialization of the task `Task1` and `Task2` values must be specified.

Figure 5.7 shows the `start` interaction. This interaction could be used, if after the execution of an ISR a task should process some information. The task configuration and the arguments of the referenced task definition's *Init* section must be specified for the `start` interaction.

```
TaskControlInterface2        constraints
model   GeneralMbeddr.main   imports      TaskConfigurations
```

```
void function2() {
  uint8 argument1 = 0;
  SystemTasks.start( Task1: { argument1 } );
} function2 (function)
```

Figure 5.7: The figure shows the `start` task control interface interaction. It can be used to start a single task of a task configuration list.

5.5 Mbeddr AVR Platform

The mbeddr AVR platform support component provides simple task management and a scheduler for AVR 8-bit micro-controllers. It also implements the necessary features to be used together with the task language and the schedulability analyzer (Requirement Section 3.6).

The task scheduler of this platform can schedule periodic, non-preemptive threads with static priorities. Thus only run-to-completion task definitions are allowed. The support of ISRs and sporadic tasks may be added in the future. Up to now only the task control interface interaction `startAll` can be used to start all tasks at the same time point. This simplifies the schedulability analysis.

5.5.1 Task Configuration Properties

Three task configuration items, which are added by this platform, are listed below. The first two properties are mandatory for a task configuration used together with this platform.

Priority This property specifies the static scheduler priority of a task. Each task must have a different priority. The priority numeration starts with zero, which represents the highest priority.

Period The period of a task in ticks. The scheduler executes the task each time the period expires.

Delay A start delay for the task in ticks. After the start of the task is requested by using the `startAll` interaction, the scheduler ignores the task until the delay expires.

5.5.2 Scheduler

The chosen scheduler algorithm is very simple, so that the schedulability analysis is simplified. The scheduler uses a single stack and can handle non-preemptive threads with static priorities and periods. The algorithm is based on the first-come, first-served (FCFS) scheduling principle. The static priority is only used to decide which thread should be executed first if multiple threads have the same arrival time. The source code is inspired by [Miller, Vahid, and Givargis 2013] and [Krishna 2014].

To reduce the thread dispatching time a single stack solution is chosen. Thus no thread can interrupt another one. The support for interrupts and sporadic tasks can be added in the future.

For all task configurations in a system a task control block (TCB) is generated (see Section 5.5.3). The TCBs are arranged in an array, which is sorted by the priority defined in the task configurations. This way the thread with the highest priority is selected for execution, if multiple threads have the same arrival time. The basic scheduler algorithm is shown schematically in Listing 5.1. The `dispatchTasks()` function is called from an infinite loop. The `getTicksSinceLastDispatch()` function returns the time passed between the last two calls. Subsequently, in order of priority, each thread is checked, whether it should be executed. If a thread is executed, the time until the next execution is set to the period of the thread.

5.5.3 Generator

The generator maps the language concepts of the task language to mbeddr C language concepts. For example, the three sections *Context*, *Init* and *Execution* of a task definition are reduced by the generator to a C structure and two functions.

The reduction rule for a task configuration list is shown in Figure 5.8. Each task configuration in the list is reduced to a global variable declaration where the type of the variable is the C struct, generated from the *Context* section of the referenced task definition. This way the context variables of each task configuration are stored on the heap. When the scheduler invokes the *Init* section of the referenced task definition, it passes the context variable as a function argument.

```
void dispatchTasks() {
    uint32 schedulerTicks = getTicksSinceLastDispatch();

    while( schedulerTicks > 0 ) {
        schedulerTicks--;

        for each thread {
            if( thread->delay == 0 ) {
                thread->delay = thread->period;
                thread->executionFunction( thread->context );
            }

            thread->delay--;
        }
    }
}
```

Listing 5.1: The listing shows the scheduler of the mbeddr AVR platform schematically.

```
reduction rules:
  [concept    TaskConfigurationList
  inheritors false
  condition  <always>]
      --> content node:
          Temp                                                    constraints
          model   com.mbeddr.ext.tasks.generator.template.main    imports
                                                                      nothing

          exported struct taskContext {
            uint8 a;
          };
          <TF [ $LOOP$[exported ->$[taskContext] $[taskContextHolder];]] TF>
```

Figure 5.8: In this figure the reduction rule for a task configuration list is shown.

If the mbeddr AVR platform is used, the generator adds the scheduler code to the mbeddr project the generator is invoked for. A part of the scheduler generator template is shown in Figure 5.9. For all task configuration items a TCB is added to the scheduler code. These TCBs are saved in an array on the heap. The order of the TCBs accords to the priorities defined in the task configurations.

MbeddrScheduler **constraints**
model com.mbeddr.ext.tasks.generator.template.main **imports** nothing
package mbeddrTasks

```
#constant TASKCOUNT = $[1];

#constant SCHEDULER_TICKS_DIV = 1000;

TaskControlBlock[] Tasks = {
  $LOOP$[{null, null, $[1], $[1], Stopped}]
};

exported enum TaskState {
  Stopped;
  Runnable;
}

exported struct TaskControlBlock {
  (void*)=>(void) taskFunction;
  void* taskContext;
  uint32 period;
  uint32 delay;
  TaskState taskState;
};
```

Figure 5.9: This figure shows a part of the mbeddr AVR platform scheduler. For each task configuration item a TCB is inserted into the `Tasks` array by the generator.

6 Execution Time Analysis

For real-time systems it is essential to satisfy timing constraints. To prove the satisfaction of these constraints, the WCET of execution paths must be known. The methods of getting these execution times are categorized into two different types: static and dynamic.

The static analysis method is based on the calculation of the execution time by using the compiled source code, the binary file. The analyzer knows the instructions for a hardware platform and constructs a graph representing the program with all possible execution paths. The execution time of an execution path is calculated by adding up the execution times of the instructions on the path. Similarly the WCET of a program part is calculated by taking only the path with the longest execution time into account.

The dynamic execution time analysis method is based on the execution of the program parts of interest. A time stamp before and after the execution of a program part is fetched. The difference is the execution time.

The work for this thesis includes the integration of both methods into mbeddr. For the static analysis method an external tool is used. The results are then imported into the mbeddr IDE. The dynamic method is implemented directly in mbeddr. The following sections cover the implementation details. The static method is used for the static schedulability analysis.

6.1 Static Execution Time Analysis

As static execution time analysis depends on the hardware architecture, many tools only support few architectures. Lists of different analysis tools can be found in [Rochange, Sainrat, and Uhrig 2014] and [Wilhelm et al. 2008]. To be usable with the mbeddr AVR platform, the external tool must support the AVR 8-bit hardware. The following execution time

calculators for the AVR 8-bit hardware platform were evaluated:

METAMOC The Modular Execution Time Analysis using Model Checking tool[1] uses models which describe the hardware, like the instructions and the pipelining capabilities [Dalsgaard et al. 2010], to calculate the WCET. Unfortunately, the models for the AVR hardware platform are not publicly available.

LLVM The LLVM project[2] provides compiler and tool-chain technologies. Source code written in many different languages can be compiled to LLVM assembly code. Tools of the LLVM project can then optimize and compile the LLVM assembly code to an executable binary for the target hardware platform. For the LLVM assembly code static execution time analysis tools exist. The time predictions of these tools differ greatly from the real execution time, because the analyzed code is not the binary code which is executed on the hardware. The second compilation introduces hardware dependent optimizations, which cannot be taken into account by the analyzer [Fachini 2011].

Bound-T The Bound-T time and stack analyzer[3] can compute the WCET of binary code for different hardware platforms. Among others the ARM7 and AVR 8-bit hardware platform are supported. Unfortunately, binary files for the AVR platform are only fully supported, if they are compiled by the IAR compiler[4].

For the proof of concept implementation the Bound-T analyzer is chosen for its precise WCET calculation capabilities. Since 2014 the tool has been free of charge and supplied under an open source license.

6.1.1 Assertions

Although execution time analyzers try to estimate the program flow from the binary code, a flow graph cannot always be created. This is the case, for example, if a loop breaking condition depends on an external value, such as a hardware register. For still being able to calculate the WCET the programmer has to define assertions which assist the analysis tool.

[1] `http://metamoc.dk`
[2] `http://llvm.org`
[3] `http://www.bound-t.com`
[4] `http://www.iar.com`

Up to now assertions can be annotated at loop statements and dynamic function calls. But the current annotations can be extended according to the needs of different external tools.

Figure 6.1 shows a loop assertion in mbeddr. The function argument `argc` cannot be calculated, because it depends on how many arguments are passed to the program. Thus the loop repetition cannot be known and must be defined by the programmer.

```
WCET                                constraints
model   GeneralMbeddr.main          imports      nothing

exported int32 main(int32 argc, string[] argv) {

  // WCET Assertion: Loop repeats 2 times
  for (int32 i = 0; i < argc; i++) {
    function1();
  } for

  return 0;
} main (function)
```

Figure 6.1: This figure shows an assertion annotated to a loop statement. The repetition of the loop cannot be calculated because it depends on the `argc` function argument.

Furthermore, a dynamic function call cannot be resolved by an analysis tool. Thus the functions, which could be called, must be specified by the programmer.

6.1.2 Analysis Results

The WCET analysis results are presented in the source code editor. Figure 6.2 shows how the results are annotated to the functions in the source code. Three time values, measured in CPU ticks, are presented. The first time is the WCET spent in the annotated function and all called functions. The *self* value is the time spent in the annotated function. The *callees* value represents the time spent in the functions which were called.

If the worst case-execution time could not be calculated, error messages are annotated to the source code in the same way as the results.

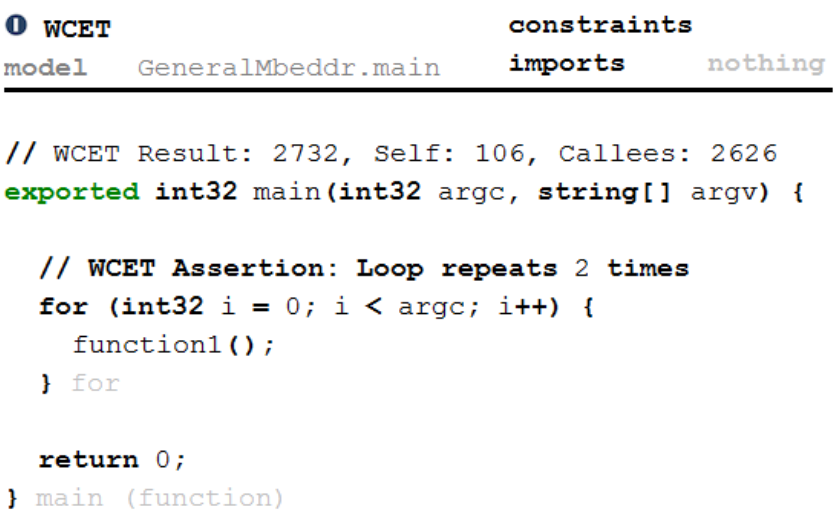

Figure 6.2: This figure shows how the results of a WCET analysis are presented in the code.

6.1.3 Implementation

A generic interface is implemented for the assertions, the external tool execution and the result presentation. Multiple different methods for providing the assertion information to the external tool are available for a tool support implementation. For each different execution time analysis tool an implementation of the interface must be provided. The implementation transforms the assertion information into a format the external tool can work with. After the execution of the external tool the results are converted into an object structure. The results are then added to the source code in the mbeddr IDE (Requirement Section 3.3.1, Section 3.3.2, Section 3.3.3).

When the mbeddr generator is executed, the annotations are reduced to C source code comments by the Bound-T support implementation. A WCET analysis request initiates the gathering of assertion information. The generated source code is scanned for the assertion comments. This way the source code line numbers of the statements, for which the assertions are inserted, are determined.

The Bound-T analyzer needs a specific additional file for assertions. Loops and dynamic function call assertions are assigned to a statement by specifying the enclosing function, the source code file and the source code line number.

6.2 Dynamic Execution Time Analysis

To estimate the quality of the static execution time analyzer, dynamic execution time analysis is used in this thesis. Instrumentation code, providing the measurement, is added to the program for the dynamic analysis.

6.2.1 Compiler Assisted Instrumentation

Some compilers support code instrumentation by adding code automatically if a compiler option is set. Compiler assisted instrumentation means that the way of measuring and storing measurement data must be implemented manually, but only a compiler option must be set and the compiler inserts the measuring points during the compilation process.

The GCC compiler, for example, adds a call to the two instrumentation functions shown in Listing 6.1 inside each function, if the `-finstrument-functions` compiler option is set. The first function call is inserted at the beginning and the second at the end of a function. Thus, after the insertion, a function body is surrounded by a call to the two instrumentation functions as shown in Listing 6.2. The instrumentation functions' bodies, i.e. the time measuring and storing of the data, must be implemented manually.

```
__cyg_profile_func_enter(void * called_func, void * caller)
__cyg_profile_func_exit(void * called_func, void * caller)
```

Listing 6.1: The two instrumentation functions inserted by the GCC compiler are shown in this listing.

```
void func1() {
    __cyg_profile_func_enter(&func1, ADDR_OF_CALLER_FUNC)

    // function1 body

    __cyg_profile_func_exit(&func1, ADDR_OF_CALLER_FUNC)
}
```

Listing 6.2: The two instrumentation functions are inserted by the GCC compiler as presented in this listing.

An example of compiler assisted instrumentation for the AVR platform which uses the GCC compiler is presented in Willmann 2012. The two arguments `void * called_func` and `void * caller` of the instrumentation functions shown in Listing 6.1 are used to identify a measuring point. Here a measuring point represents a function call.

To minimize the measured time inaccuracy, which would be introduced by outputting a time stamp at every call to the measuring functions via a serial port, the gathered data is stored into RAM and outputted at the end of a measuring session. As the RAM in the used microcontroller is limited, a time stamp is not stored every time the two instrumentation functions are called. Instead, for each measuring point statistical data is saved, such as the minimal, maximal, accumulated execution time and the function invocation count.

A function call, i.e. a measuring point, is identified by the called and the caller function addresses in this case. To combine multiple visits of a measuring point during run-time, the location where preceding visits for this measuring point were stored must be found. This is accomplished by using a binary search algorithm.

6.2.2 Mbeddr Instrumentation

The implemented mbeddr instrumentation, which is used for the dynamic execution time analysis, is based on the idea presented in the thesis of Willmann [Willmann 2012]. The difference is that the mbeddr generator is used to insert the instrumentation functions. Moreover, for all function calls to be analyzed, represented by measuring points, a unique identifier is generated. For every measuring point memory, where the measured data is stored, is reserved in an array. The identifier is used as an index for the array. Thus, no search algorithm must be implemented to find the storing location for measured data. The storing process takes a short and constant time, which is a strong advantage for time measurements.

Another advantage of the mbeddr instrumentation is that it is independent of the C compiler. No compiler specific instrumentation functions must be implemented.

A time provider language is used to get the time stamps. This language maps different time related functionalities to platform dependent code. Thereby the time determination is abstracted. The mbeddr AVR platform implementation uses a 16 bit hardware timer with

the frequency set to the CPU frequency. The mbeddr instrumentation functions use the CPU tick value for measuring time spans.

The instrumentation can be enabled for mbeddr modules, or inserted manually at any code location. When the generator for an mbeddr project is executed, the instrumentation functions are inserted. Figure 6.3 depicts a module with enabled instrumentation. The mbeddr instrumentation offers the option to define when and how the measured data should be outputted.

The measured data can be imported by the mbeddr IDE for displaying purpose as shown in Figure 6.3. A bar displays the average execution time relative to all measuring results. By clicking on an annotated bar the detailed results, the minimal, maximal, average execution time and the visit count are presented.

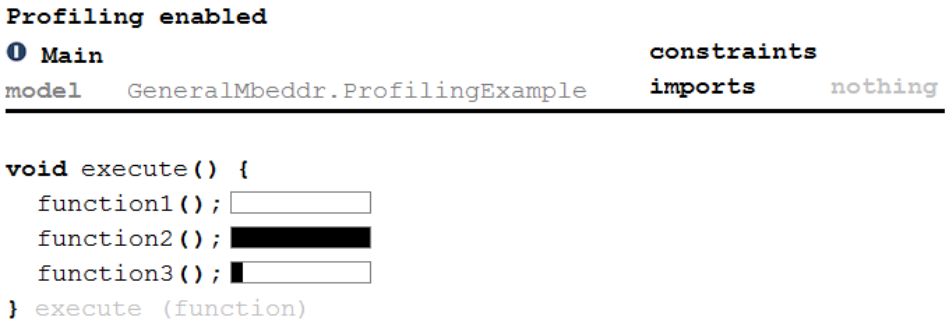

Figure 6.3: The average execution time is displayed by a bar like shown in this figure. By clicking on the bar the detailed information are presented.

6.3 Comparison

A test program is chosen to compare the static WCET results to the dynamic execution time results. The comparison should confirm that the Bound-T analyzer can calculate the WCET of program parts precisely enough to be utilized by the schedulability analyzer.

The code shown in Figure 6.4 is used for the test program. Three simple functions with a static execution flow are chosen for comparison.

To define test result expectations the following issue must be considered when comparing the static and dynamic analysis results. The measured execution time of a function, gathered by the implemented mbeddr instrumentation capabilities, includes an inaccuracy. As a time

```
ExecutionTimeAnalysisExampleCode        constraints
model   GeneralMbeddr.main              imports      nothing

exported void execute() {
  function1();
  function2();
  function3();
} execute (function)

void function1() {

} function1 (function)

void function2() {
  for (uint8 a = 0; a < 200; a++) {

  } for
} function2 (function)

void function3() {
  for (uint8 i = 0; i < 200; i++) {
    function1();
  } for
} function3 (function)
```

Figure 6.4: This figure shows the three simple functions of the test program.

stamp is taken before and after the function call, the execution time which is needed to get the time stamps is partially included in the measured value. To quantify the maximum additional time added by one measuring point e_{pm}, the static WCET of a function with and without included measuring points are compared.

It is expected that the statically analyzed WCET e_s is greater than the dynamic execution time e_d minus the maximum additional time added by one measuring point e_{pm} in this case, because in the test program no nested measuring points are used (see Equation 6.1).

$$e_s > e_d - e_{pm} \tag{6.1}$$

Further expectations are that the insertion of measuring points into the `execute()` function has no influence on e_{pm} and no influence on e_s of the function calls `function1()`,

`function2()` and `function3()`.

6.3.1 Set-up

The test hardware consists of an Atmel STK500 demo board [5], an Atmega168 micro-controller [6] and an AVR Dragon ISP [7]. For the test project the IAR compiler [8] is used, so that the WCET can be determined by the Bound-T analyzer. The compiled program is transferred to the micro-controller via the AVR Dragon ISP, controlled from the IAR Embedded Workbench IDE. The newly integrated static execution time analyzer is used to get the static execution time of the three functions. The dynamic analysis data is transferred via a serial connection after the execution of the test functions.

6.3.2 Methods

At first, the static execution times of the functions `execute()`, `function1()`, `function2()` and `function3()` are calculated via the mbeddr static execution time analysis capability, previously implemented.

Three data series are acquired by the static analysis. The first without inserted measuring points, the second with one measuring point assigned to the `function1()` function call inside the `execute()` function, and the third with measuring points assigned to all function calls inside the `execute()` function.

The dynamic analysis is executed with measuring points assigned to all function calls inside the `execute()` function.

6.3.3 Results

The acquired data is presented in Table 6.1. All statically analyzed data for the `execute()` function follow the equation $e_{pm} = 530$.

[5] `http://www.atmel.com/tools/stk500.aspx`
[6] `http://www.atmel.com/devices/atmega168.aspx`
[7] `http://www.atmel.com/tools/avrdragon.aspx`
[8] `http://www.iar.com/Products/IAR-Embedded-Workbench/AVR`

The value of $e_s - e_d$ is equal to 163 for the measuring points assigned to `function1()` and `function2()` but not for the measuring point assigned to `function3()`.

measuring point	static, 0 measuring points	static, 1 measuring point	static, 3 measuring points	dynamic
`execute()`	3843	4373	5433	
`function1()`	4	4	4	167
`function2()`	1008	1008	1008	1171
`function3()`	2812	2812	2812	2575

Table 6.1: In this table the results of the dynamic and static execution time analysis are listed. The execution time is measured in CPU ticks.

6.3.4 Conclusion

All previous expectations are met. Thus, the Bound-T analyzer's ability of calculating the WCET of simple program parts is shown.

Interestingly $e_s - e_d$ is not a constant. This means that the WCET value is influenced by function calls, as they are included in function `function3()`. This unexpected result could be examined by investigating the binary code in the future.

7 Schedulability Analysis

Tasks in a real-time system have timing constraints. To prove the satisfaction of these constraints the schedulability of the task in the system must be shown.

After an analysis request from an mbeddr user the items from the task configuration lists are collected and passed to the analysis algorithm. The concrete analysis algorithm depends on the scheduler. Thus the platform support component, which knows the scheduler, has to provide the algorithm. If the algorithm is implemented, the schedulability analysis component can create schedule diagrams and can present them to the mbeddr user (Requirement Section 3.5.1, Section 3.5.2).

In the following sections the analysis implementation for the mbeddr AVR platform scheduler and the examination of the scheduler during run-time are described. The timing behaviour of the scheduler is investigated during its execution on an AVR micro-controller. The predicted results of the schedulability analyzer are then compared to the results gathered during run-time.

7.1 Mbeddr AVR Platform

The mbeddr AVR platform schedulability analyzer can handle non-preemptive periodic tasks with a static priority. Its base principle is the FCFS scheduling algorithm. Only if two tasks have the same arrival time, the task with the higher priority is executed first.

When an analysis request for a task set is issued, the task set to be analyzed is passed to the analysis algorithm. The analysis algorithm creates a schedule based on the WCET of the dispatcher e_{dis} and the WCET of the *Execution* sections of the task definitions ei. The time span of the created schedule starts at time point 0 and ends when the schedule becomes

repetitive. A schedule of n tasks is periodic from the least common multiple of the tasks periods $lcm(p1, p2, ..., pn)$ added to the latest first arrival time of all tasks $max(a1_0, a2_0, ..., an_0)$ [Goossens and Macq 2001, p. 5]. The least common multiple of the periods of the tasks is also known as hyper-period.

The schedule is created by the same algorithm the scheduler is using. The WCET of the dispatcher e_{dis} is added to the WCET of a task ei.

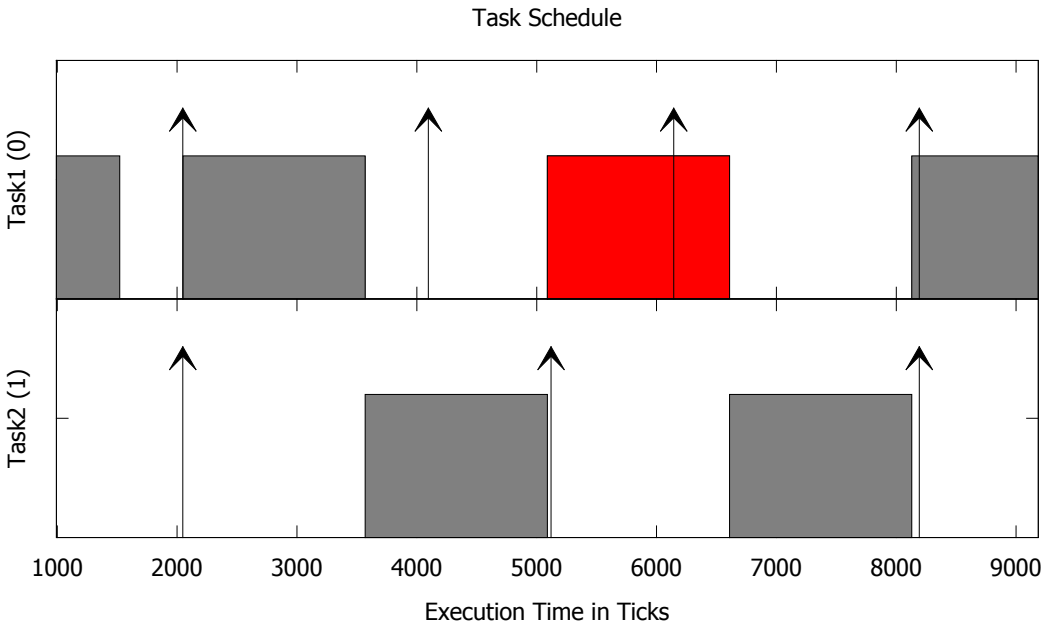

Figure 7.1: The diagram shows a task schedule which is presented to the mbeddr user if a deadline is not met.

After the schedule is created all deadlines are checked. For the mbeddr AVR platform scheduler the relative deadlines are set to the period of a task $di = pi$. If the schedule is feasible, a message is presented to the mbeddr user and a diagram of the schedule from time point 0 to $max(a1_0, a2_0, ..., an_0) + 2 * max(p1, p2, ..., pn)$ is generated with the tool gnuplot [1]. If the schedule is not feasible, a message is shown to the mbeddr user and the part of the schedule where the first deadline is missed is presented in a diagram as shown in Figure 7.1.

7.2 Run-Time Analysis

This section describes the run-time analysis of the mbeddr AVR platform scheduler. The main goal of the analysis is to check if the predicted schedule of the schedulability analyzer

[1] `http://www.gnuplot.info`

matches the measured behaviour of the executed scheduler.

7.2.1 Set-up

In the mbeddr IDE a test project is created. The task language is used together with the mbeddr AVR platform to define the following task set.

$$T1_p: \quad a1_0 = 0,\ p1 = 20480$$
$$T2_p: \quad a2_0 = 10240,\ p2 = 40960$$

Figure 7.2 depicts the diagram generated by the implemented schedulability analyzer for the task set.

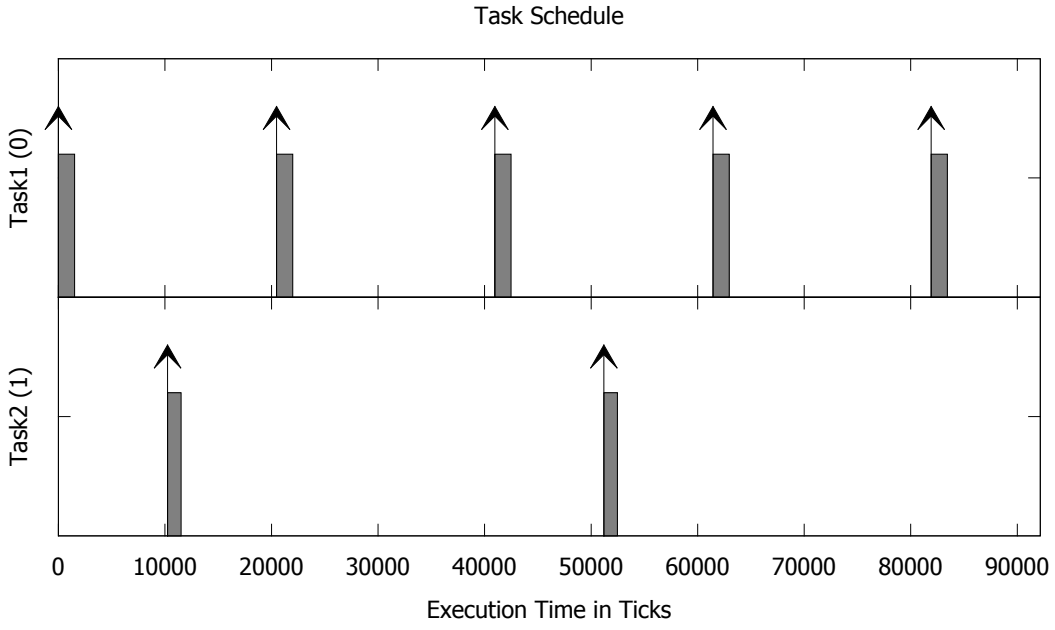

Figure 7.2: The diagram shows a part of the worst-case schedule of the task set used for the run-time analysis.

The execution sections of the task definitions contain only a loop, surrounded by the activation and deactivation of an I/O pin. This ensures a static WCET of the tasks, which makes the measurement easier. The loop of task $T1_p$ counts until 200 and the loop of task $T2_p$ counts up to 150. The activation and deactivation of the I/O pin allows to gather the release

time point and the execution time of the tasks. Figure 7.3 shows the task definition used by $T1_p$.

```
task definition Task1Definition {
  context {
    << ... >>
  }

  <no init>

  execution {
    PORTB |= hex<01>;
    for (uint8 a = 0; a < 200; a++) {

    } for
    PORTB &= hex<fe>;
  }
}
```

Figure 7.3: The task definition used by the task $T1_p$ is shown in this figure.

The test hardware consists of an Atmel STK500 demo board, an Atmega168 micro-controller, an AVR Dragon ISP and a digital oscilloscope. Two channels of the oscilloscope are connected to the two I/O pins which are toggled by the tasks.

For the compilation of the test project the IAR compiler is used, so that the WCET can be determined by the Bound-T analyzer, previously integrated into mbeddr. The compiled program is transferred to the micro-controller via the AVR Dragon ISP, controlled from the IAR Embedded Workbench IDE. Before the execution on the target hardware, the implemented schedulability analyzer is used to gather the static analysis data.

7.2.2 Methods

The static data is calculated by the worst-case execution time tool, when the schedulability analysis is started from the mbeddr IDE. In the IDE the results are outputted as text and the schedule diagram is presented.

The dynamic data is gathered by the oscilloscope, using its time measurement function. The measuring error, introduced by the measuring resolution and jitter is also noted. The scheduler does not always recognize immediately when a task becomes ready for execution

because of the scheduler algorithm. This introduces the jitter; the deviation of the measured periodic value. Figure 7.4 shows an image taken by the oscilloscope during measuring.

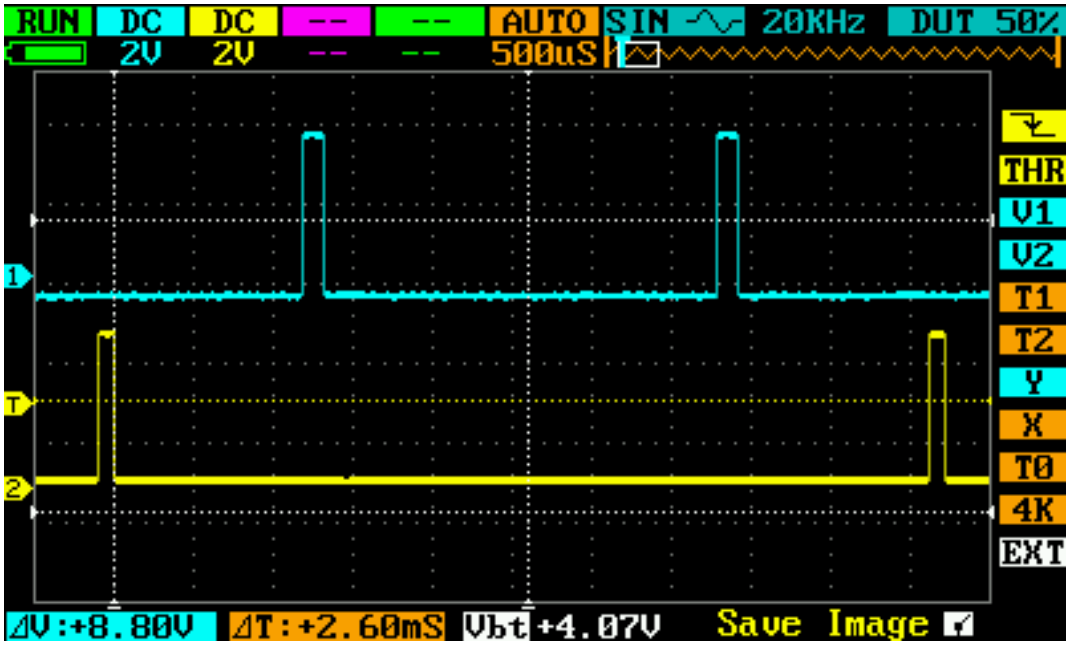

Figure 7.4: The figure shows an image created by an oscilloscope during measuring. The blue track shows the output of the I/O port controlled by task $T1_p$ and the I/O port controlled by task $T2_p$ is represented by the yellow track.

7.2.3 Results

The test results measured in CPU ticks are presented in Table 7.1. The dynamic analysis results are converted into CPU ticks with the following method. The CPU frequency $f_{cpu} = 8\,\text{MHz}$ equals a CPU cycle time $T_{cpu} = 0.125\,\text{us}$. A measured value t is noted in microseconds. Thus the CPU tick value is calculated by $t_{ticks} = t/T_{cpu}$.

	static	dynamic
$e1$	1010	1024 ± 16
$e2$	760	768 ± 16
$p1$	20480	20960 ± 560
$p2$	40960	41360 ± 560
$a2_0$	10240	10640 ± 560
e_{dis}	512	

Table 7.1: In this table the results of the static schedulability analysis and the measured data are listed. All values are measured in CPU ticks.

7.2.4 Conclusion

The test shows that the scheduler is working and that the predicted results of the schedulability analyzer are met. The schedulability analyzer predicted the results with a maximum inaccuracy of 5,5 %.

Interestingly the static measuring results are at the lower bounds of the measuring error. This could mean that there may be an implementation defect in the scheduler or the static analysis method. An investigation would try to reduce the measuring error by using tasks with a longer execution time. Further tests should be done with a task set causing a higher CPU load.

8 Summary

The main goal of this thesis was to develop a task language which can be integrated in normal source code and provides the ability to perform static schedulability analysis. The mbeddr project is perfectly made for the integration of new languages into existing ones. It comes with MPS, a great editor for language oriented programming.

At first, the new task language requirements were determined. This included the language syntax and the features to be supported. The main objective of the task language was to provide an easier definition and configuration and to provide the necessary information for the schedulability analysis. The schedulability analyses test set-up has shown how easy tasks can be defined and configured with the new language.

To achieve the main goal static execution time analysis capabilities had to be integrated into the mbeddr IDE. As the task of calculating the WCET is not easy and many tools exist, the integration of an external tool was chosen. A requirement was the possibility to add further execution time analysis tools. To fulfill this requirement a generic interface was created. The Bound-T analyzer integration showed that the integration is possible and works perfectly. The WCET of programs can now be presented directly in the source code editor.

The next goal was to compare the calculated execution time with results measured during run-time, because the schedulability analyzer has to rely on the calculated results. This led to the implementation of a generic and compiler independent dynamic execution time analysis capability for mbeddr. Although this thesis uses the dynamic execution time measurement only for a comparison, for mbeddr it is a very useful extension. With a test set-up the results of the dynamic and static analysis methods were successfully compared. The Bound-T analyzer was used for the schedulability analysis, because of the good results.

One further step was the implementation of the mbeddr AVR platform together with a simple FCFS scheduler for the AVR 8-bit hardware. The scheduler can handle periodic non-preemptive tasks. All required interfaces for the usage with the task language were implemented.

The main goal was achieved by implementing the schedulability analyzer infrastructure and the concrete algorithm for the mbeddr AVR platform. The analyzer can determine if the schedule of a task set, defined with the task language, is feasible and generates diagrams for the visualization.

To analyze the implemented scheduler and to compare the predicted static schedulability analysis results with run-time data, a second test-setup was made. The run-time data was gathered by an oscilloscope and compared with the data outputted by the implemented schedulability analyzer. The results of the run-time data verified the usefulness of the data, predicted by the schedulability analysis.

8.1 Future Work

The implementation results of this thesis can be extended by many features in the future. The support of more different platforms, like RTAI or eCos, could be implemented. This would include the integration of a static execution time analysis tool for the hardware platform, the platform support component for the task language and the specific schedulability algorithm.

Another interesting extension to the schedulability analysis would be the consideration of critical sections and shared resources. This part was not regarded in this thesis, because a coworker at itemis AG is creating a language for handling with critical sections and shared resources. A fusion of both work results is planned.

8.2 Discussion

The implementation of an own scheduler could have been avoided by using an existing operating system. An operating system with a preemptive scheduler could have been chosen,

because most embedded systems rely on them. The problem is that for the AVR 8-bit hardware only a few operating systems exist, that they heavily depend on the compiler and that most operating systems with a preemptive scheduler for the AVR 8-bit hardware are only supporting the GCC compiler. On the other hand, the static execution time analyzer is restricting the compilers. Thus, the selection of the hardware platform led to the restriction of the operating systems for the proof of concept implementation.

In my opinion for the AVR 8-bit hardware with restricted resources a scheduler with minimal overhead is best. Moreover the implemented mbeddr AVR platform can solve many problems an embedded system has to face.

Since only one platform support has been added to the task language so far, the ability of being able to support as many different task control interfaces as possible has not yet been shown. The same applies to the interface for the integration of static execution time analysis tools.

Overall I am very excited about the results implemented for this thesis, because no IDE I know of has the ability of editing C code in the same editor where the schedulability of the programmed tasks can be proven. Additionally, the simple way the analysis can be accomplished makes the static analysis more attractive, not only for real-time systems. In the end, this leads to a more productive build process of safety critical systems.

Bibliography

Cheng, Albert M. K. (2003). *Real-Time Scheduling and Schedulability Analysis*. John Wiley & Sons, Inc. ISBN: 9780471224624.

Dalsgaard, Andreas E. et al. (2010). "METAMOC: Modular Execution Time Analysis using Model Checking". In: *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Ed. by Björn Lisper. Vol. 15. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 113–123. ISBN: 9783939897217.

Dmitriev, Sergey (2004). *Language Oriented Programming: The Next Programming Paradigm*. URL: http://www.onboard.jetbrains.com/is1/articles/04/10/lop/mps.pdf.

Fachini, Guilherme James de Angelis (2011). *Modular and Generic WCET Static Analysis with LLVM Framework*. Work completion of graduation. Universidade Federal do Rio Grande do Sul. Instituto de Informática. Curso de Ciência da Computação: Ênfase em Ciência da Computação: Bacharelado. URL: http://hdl.handle.net/10183/31020.

Goossens, Joel and Christophe Macq (2001). "Limitation of the Hyper-Period in Real-Time Periodic Task Set Generation". In: *In Proceedings of the RTS Embedded System (RTS01*, pp. 133–147.

Krishna, K. Sai (2014). *Scheduler - avrtutorials2*. URL: https://sites.google.com/site/avrtutorials2/scheduler.

Miller, Bailey, Frank Vahid, and Tony Givargis (2013). "RIOS: A Lightweight Task Scheduler for Embedded Systems". In: *Proceedings of the Workshop on Embedded and Cyber-Physical Systems Education*. WESE '12. Tampere, Finland: ACM, 9:1–9:7. ISBN: 9781-4503-17658.

Rochange, C., P. Sainrat, and S. Uhrig (2014). "Time-Predictable Architectures". In: FOCUS Series. Wiley, p. 33. ISBN: 9781118790267.

Stallings, William (2008). *Operating Systems: Internals and Design Principles*. 6th. Upper Saddle River, NJ, USA: Prentice Hall Press. ISBN: 0136006329, 9780136006329.

Voelter, Markus, Daniel Ratiu, Bernd Kolb, et al. (2013). "mbeddr: instantiating a language workbench in the embedded software domain". In: URL: http://mbeddr.com/files/voelteretal-mbeddr-final.pdf.

Voelter, Markus, Daniel Ratiu, Bernhard Schaetz, et al. (2012). "Mbeddr: An Extensible C-based Programming Language and IDE for Embedded Systems". In: *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*. SPLASH '12. Tucson, Arizona, USA: ACM, pp. 121–140. ISBN: 9781450315630.

Völter, Markus et al. (2013). *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org. ISBN: 9781481218580. URL: http://www.dslbook.org.

Ward, M. P. (1995). "Language Oriented Programming". In: *SoftwareConcepts and Tools*. Vol. 15, pp. 147–161.

Wilhelm, Reinhard et al. (2008). "The Worst-case Execution-time Problem & Overview of Methods and Survey of Tools". In: *ACM Trans. Embed. Comput. Syst.* 7.3, 36:1–36:53. ISSN: 1539-9087.

Willmann, Daniel (2012). *Diploma thesis: Optimization of µDTN*. Diploma thesis. Technische Universität Braunschweig. URL: https://totalueberwachung.de/blog/files/opt_mudtn.pdf.