# Mapping of Timing Definition Language (TDL) Components to Distributed Platforms

**Johannes Pletzer**

Department of Computer Sciences
University of Salzburg

Advisor: Prof. Dr. Wolfgang Pree

A dissertation submitted for the degree of
*Doctor technicae (Dr. techn.)*

Salzburg, Juli 2012

# Abstract

This dissertation deals with the mapping of real-time system components specified using the Timing Definition Language (TDL) to distributed embedded platforms. TDL enables the specification of the timing behavior of such components independently of their functionality and thereby abstracts from the hardware platforms on which they are eventually executed on. The main contributions are (1) a runtime system suitable for the distributed execution of TDL components and (2) a code and schedule generation framework whose aim is to provide fully automatic deployment of components to any hardware platform using customizable plug-ins. Both the runtime system and the code generation framework support recent extensions to TDL such as the ability to specify asynchronous activities. We provide framework plug-ins for a heterogeneous distributed system using the FlexRay communication protocol to demonstrate the applicability of our work. A comparison of the TDL tool chain to the workflow and tools currently employed in the automotive industry rounds out the thesis.

# Acknowledgements

First of all I thank my advisor Prof. Wolfgang Pree for giving me the opportunity to be part of his excellent research group and to write this thesis. Your positive energy and bold visions always inspired and encouraged me!

I also thank my co-advisor Dr. Josef Templ for countless software design discussions and pair-programming sessions. You formed the basis of my software engineering skills!

Furthermore, my gratitude goes to Claudiu Farcas and Prof. Ingolf Krueger from UC San Diego for reviewing the thesis. Your perspective helped me to advance my work!

Many thanks also go to my colleagues at the Department of Computer Sciences, especially Andreas Naderlinger, Peter Hintenaus, Emilia Farcas, Gerald Stieglbauer, Patricia Derler and Stefan Resmerita, for all those discussions about TDL and other topics. Without your input I never would have come this far!

Adriana Pratter was a great help for all administrative stuff. Thanks for your cheerful words when progress was not as fast as expected!

A lot of thanks go to my family and friends for both distracting me from the thesis and also for reminding me that it is a good thing to continue working on it. Thank you all for being there no matter what!

The longest lasting support of all was provided by my parents Annemarie and Johann Pletzer. Thank you for always believing in me without a doubt!

My deepest gratitude goes to my love Gloria Dürnberger for always supporting me, despite the fact that the thesis consumed a lot of time we otherwise could have spent together. Thank you for encouraging me until the very end!

Finally, special thanks go to our cat Beijing. You showed me how to relax and how to enjoy the simple things in life!

# Table of Contents

**Abstract** ..... **3**
**Acknowledgements** ..... **5**
**Table of Contents** ..... **7**
**List of Figures** ..... **11**
**1. Introduction** ..... **13**
1.1. Motivation and Context ..... 13
1.2. Objectives and Contributions ..... 14
1.3. Structure of the Dissertation ..... 16
**2. Timing Definition Language (TDL)** ..... **17**
2.1. The Logical Execution Time (LET) Abstraction ..... 17
2.2. TDL Language Constructs ..... 18
2.3. Transparent Distribution ..... 22
2.4. Execution of TDL Programs ..... 23
2.5. Extensions for Asynchronous Activities ..... 26
2.6. Visual TDL Tools ..... 28
2.7. TDL Tool Chain ..... 30
2.8. Related Work ..... 33
**3. TDL Runtime System** ..... **37**
3.1. TDL Machine ..... 41
3.1.1. Initialization ..... 42
3.1.2. Step Function ..... 43
3.1.3. Non-Preemptive Dispatcher ..... 44
3.2. Synchronization Mechanism for Asynchronous Activities ..... 45
3.2.1. Asynchronous Activities ..... 45
3.2.2. Threading and Synchronization ..... 47
3.2.3. Quantitative Analysis of Runtime Behavior ..... 53
3.2.4. Related Work ..... 53
3.3. TDL Comm Layer Framework ..... 54
3.3.1. Initialization ..... 57
3.3.2. Frame Handling ..... 58
3.3.3. Communication between Asynchronous Activities ..... 59

3.3.4. Platform-Specific Plug-Ins ..... 60
**4. Code and Schedule Generation Framework ..... 63**
4.1. Framework Foundations ..... 66
4.2. Node-Level Code Generation ..... 74
4.2.1. C Platform Plug-In ..... 77
4.2.2. Embedded C Platform Plug-In ..... 81
4.2.3. Stub Module Generation ..... 83
4.2.4. Communication Layer ..... 85
4.3. Cluster-Level Code Generation ..... 89
4.3.1. Comm Scheduler ..... 91
4.3.2. Iterative Frame Generator ..... 97
4.3.3. Genetic Frame Generator ..... 99
4.3.4. Comm Scheduler Plug-In ..... 101
**5. Platform-Specific Adaptations for FlexRay ..... 105**
5.1. The FlexRay Protocol ..... 105
5.2. Hardware Platforms ..... 107
5.2.1. Node Renesas ..... 107
5.2.2. MicroAutoBox ..... 108
5.3. TDL Comm Layer Framework Plug-Ins ..... 110
5.4. TDL:VisualDistributor Interfaces ..... 111
5.5. Node Platform Plug-Ins ..... 114
5.5.1. Node Renesas Platform ..... 114
5.5.2. MicroAutoBox Platform ..... 117
5.6. FlexRay Implementation ..... 122
5.6.1. FlexRay Communication Layer ..... 123
5.6.2. Node Renesas Communication Layer ..... 128
5.6.3. MicroAutoBox Communication Layer ..... 129
5.6.4. Cluster Platform Plug-In ..... 130
5.7. Case Study ..... 137
**6. TDL Workflow ..... 141**
6.1. Introduction ..... 141
6.2. AUTOSAR ..... 142
6.3. Current Workflow and Tools in the Automotive Industry ..... 144
6.3.1. EB Designer Pro ..... 144
6.3.2. DaVinci Tool Suite ..... 146
6.3.3. Evaluation ..... 147
6.4. The TDL Approach and its Impact on the Workflow ..... 148

6.4.1. TDL Tools ..... 148
6.4.2. Evaluation ..... 149
6.4.3. Workflow Advantages ..... 149
6.4.4. Transition from Today's Workflow ..... 150
**7. Conclusion and Future Work ..... 151**
**References ..... 153**

# List of Figures

Figure 1. Logical Execution Time (LET) ..... 18
Figure 2. Timing and data flow of the producer-consumer example ..... 18
Figure 3. Slot selection ..... 21
Figure 4. Physical timing and communication window ..... 22
Figure 5. Integration of asynchronous activities ..... 26
Figure 6. TDL:VisualCreator user interface ..... 29
Figure 7. TDL:VisualDistributor user interface ..... 30
Figure 8. TDL tool chain overview ..... 31
Figure 9. TDL system layers ..... 37
Figure 10. TDL Runtime System include relationships ..... 38
Figure 11. Assumed task model ..... 46
Figure 12. Threads and critical regions ..... 48
Figure 13. Stub module data flow ..... 55
Figure 14. Transmission of a port value via the network ..... 56
Figure 15. TDL Comm Layer frame buffers ..... 57
Figure 16. Framework collaboration diagram ..... 64
Figure 17. Framework foundation classes and interfaces ..... 66
Figure 18. Interface Platform ..... 67
Figure 19. Class ModuleDecl representing the Abstract Syntax Tree (AST) ..... 68
Figure 20. Abstract class AbstractPlatform ..... 69
Figure 21. Interface NodePlatform ..... 69
Figure 22. Abstract class AbstractNodePlatform ..... 71
Figure 23. Interface ClusterPlatform ..... 72
Figure 24. Class CommSchedule ..... 73
Figure 25. Node platform abstraction levels ..... 75
Figure 26. Communication layer class diagram ..... 85
Figure 27. Detailed framework collaboration diagram ..... 90
Figure 28. Class CommScheduler ..... 91
Figure 29. Sample binding of several messages to the same frame ..... 95
Figure 30. Sample mapping of frame windows to frames ..... 96

Figure 31. Interface CommSchedulerPlugin ..... 101
Figure 32. FlexRay cycle layout ..... 106
Figure 33. Node Renesas hardware overview ..... 107
Figure 34. dSPACE MicroAutoBox ..... 109
Figure 35. TDL:VisualDistributor property page example ..... 111
Figure 36. TDL:VisualDistributor data model classes ..... 112
Figure 37. TDL:VisualDistributor interfaces ..... 113
Figure 38. Prototyping hardware node platforms ..... 113
Figure 39. Node Renesas node property page ..... 115
Figure 40. Node Renesas platform output device mapping dialog..... 115
Figure 41. TDL:VisualDistributor interrupt assignment ..... 118
Figure 42. MicroAutoBox platform input device mapping dialog ..... 119
Figure 43. FlexRay-related classes..... 123
Figure 44. Node Renesas communication layer class diagram ..... 128
Figure 45. FlexrayPlatform class diagram ..... 131
Figure 46. FlexRay Property Editor ..... 134
Figure 47. FlexRay cluster property page ..... 134
Figure 48. Legacy case study data flow ..... 137
Figure 49. Mapping of a TDL sensor to a FlexRay signal ..... 138
Figure 50. Case study oscilloscope plot ..... 138
Figure 51. Automatic platform deployment ..... 142
Figure 52. EB Designer Pro Main User Interface ..... 144
Figure 53. EB Designer Pro workflow overview (white: OEM, gray: supplier)..... 145
Figure 54. DaVinci Tools workflow overview (white: OEM, gray: supplier)..... 146
Figure 55. DaVinci System Architect user interface ..... 147
Figure 56. TDL tools workflow overview (white: OEM, gray: supplier) ..... 149

# 1. Introduction

This chapter gives an overview of the motivation and main objectives of the thesis and summarizes its main contributions.

## 1.1. Motivation and Context

Today, embedded systems and their software are ubiquitous in modern life, with examples ranging from consumer electronics to medical and transportation systems. An impressive example of their growing importance is their application in the automotive and avionics industry. For example, a premium-class car today is contains millions of lines of code scattered across 70 to 100 networked electronic control units (ECUs) [1]. Software complexity escalates to the point that current development processes and tools can no longer ensure sufficiently reliable systems at affordable cost [2], also leading to a steadily rising cost of software-related warranty cases. This explains the increasing demand for improved software engineering, capable of handling the development and maintenance requirements faced by the industry. The growing complexity of today's embedded systems, together with the dropping cost of silicon, paves the way for the introduction of new abstractions in the field of embedded software engineering.

Traditionally, embedded software construction is platform dependent and not compositional, especially when it comes to its timing properties. This leads to increased efforts required for integration, validation and maintenance. The timing behavior is typically not specified explicitly but rather is a result of system load and the occurrence of sometimes unpredictable events at runtime and so developers often rely on intensive testing, although that can never proof that their design works as required under all circumstances. The concept of the Logical Execution Time (LET) introduced in the realm of the Giotto project [3] aims to overcome this shortcoming by abstracting from the physical execution time of tasks and, in the distributed case, from network communication. The LET abstraction specifies that the inputs of a task, which can be values obtained from sensors or from other tasks in the system, are read at the beginning of the LET period and the outputs provided to other tasks or actuators are only updated at the end of a task's LET. The LET programming paradigm enables the platform independent description of the timing behavior, which is guaranteed to be equal on any hardware platform, provided that it is fast enough. Thus, the LET abstraction leads to a significant reduction of complexity as it allows embedded software developers to focus on the functionality of a software component without having the target platform in mind.

Apart from Giotto, there are some other research projects which base on the promising LET concept. xGiotto [4] is one successor of Giotto which has events as its main structuring principle. It allows the definition of LETs for synchronous and asynchronous tasks and guarantees time-safety mainly by constraining the

occurrence rate of events and therefore bounding the time it takes until an event is processed. The Hierarchical Timing Language (HTL) [5] enables the hierarchical refinement of so-called abstract task invocations for the purpose of compact representation and simplified program analysis and schedulability tests.

The Timing Definition Language (TDL) also facilitates the LET abstraction and aims at supporting the development of deterministic, portable software for embedded real-time systems. It goes beyond Giotto in a number of aspects, most notably by the introduction of a component model and the integration of asynchronous activities. While Giotto is basically an abstract mathematical model of a time-triggered language with a rather simple tool chain that primarily proofs that it can be implemented, the TDL project aims at providing a comprehensive tool chain that makes Giotto's concepts available for real-world industrial projects. This is something which also xGiotto and HTL fail to provide, but is a necessity in order to eventually tackle the aforementioned challenges the industry is facing today.

## 1.2. Objectives and Contributions

The development of TDL started in 2003 in the context of the MoDECS (Model-Based Development of Distributed Embedded Control Systems) project [6]. Our work primarily bases on [7], which laid the foundation for a portable TDL runtime system, and [8], which proposed the transparent distribution of TDL modules. Recently, the TDL language has been extended by a number of essential features, above all the combination of the time-triggered and event-triggered paradigms by adding support for asynchronous activities. It allows specifying event-triggered (alias asynchronous) activities, which are triggered by the occurrence of an external hardware interrupt or other events, and which many real-time systems execute in addition to strictly time-triggered (alias synchronous) activities. Integrating asynchronous activities and other additional features require adaptations along the complete TDL tool chain, affecting the compiler, the runtime system, and code and schedule generation for nodes and communication networks. The extensions were driven by the vision of the application of TDL in industrial environments, and consequently we incorporated many hints from our industry partners which features they need and what would ease the integration of TDL in their existing development workflow. This not only led to numerous language adaptations and platform implementations, but also to extensions that require thorough theoretical research.

The following motivates and summarizes all major thesis contributions, which all are related to specific advancements of the TDL language and tools in recent years.

### Code & Schedule Generation Framework

Software modules developed with TDL are envisioned to be deployed on a broad range of different hardware platforms, including diverse communication protocols and operating systems. For that purpose, it was required to come up with a generic code generation framework.

One of the major contributions of this thesis is a flexible, LET-based code generation framework for potentially distributed real-time systems. A *distributed system* is as system consisting of multiple, interconnected computing nodes. The framework covers both code generation on node level as well as the generation of a communication schedule for the communication bus connecting such nodes. For that purpose task and communication schedules must be generated. Those schedules influence each other and consequently our framework deals with their interdependence by pioneering an iterative scheduling approach. The framework is extensible by the use of plug-ins, which implement support for specific node

platforms and communication protocols and therefore guarantee a clear separation of platform independent from platform dependent concerns. We present the plug-in interfaces and describe sample plug-ins. We also show that the proposed framework is not limited to TDL but can be applied to other languages that use the LET abstraction and that it is substantially more general and powerful than the methodology and tools presented in [8]. We provide empirical comparisons of various scheduling approaches and also investigate the use of genetic algorithms for this specific scheduling problem.

**TDL Runtime System Advancement**

Based on the work of Farcas [7], we redesigned the TDL runtime system so that it supports all recent TDL extensions such as cyclically imported modules, structured data types and global output ports. A special focus is on developing a communication layer with a well-defined interface in order to support specific communication protocols and a corresponding implementation compatible with the FlexRay protocol.

The most significant advancement of the TDL Runtime System is the integration of asynchronous activities. While synchronous activities are coordinated by the so-called TDL Machine, asynchronous activities are not as time critical and therefore are executed in the background so that they do not affect the timing behavior of synchronous activities. However, it is allowed that synchronous and asynchronous activities exchange data in a properly synchronized way. We describe the TDL language extensions and put special focus on the threading model and synchronization algorithm handling the data flow between the synchronous TDL Machine and asynchronous activities. We implemented the TDL Runtime System support by means of platform plug-ins for a number of target platforms, including distributed systems.

**FlexRay Support**

Another main objective of the thesis is to prove the applicability of TDL in real-world control applications. The FlexRay protocol is a state-of-the art communication protocol targeted at automotive applications and significantly gained importance, which eventually led to its adoption in automotive series production since 2006 [9]. While previous TDL prototype implementations supporting distribution used the CAN protocol, we then chose to apply TDL to the FlexRay communication bus and corresponding prototyping hardware commonly used in the industry, such as the MicroAutoBox from dSPACE and the NODE<RENESAS> by DECOMSYS (now Elektrobit). The FlexRay protocol is especially suitable for TDL as it has a static, time-triggered part, which we use for communicating synchronous TDL activities, and a dynamic, event-triggered part, which fits well for handling asynchronous TDL activities. We implemented FlexRay support by means of a code generation framework plug-in which calculates all required cluster and node parameters and ensures that the FlexRay cluster startup is performed correctly. In addition, we provide the ability for what we call incremental scheduling, i.e. the augmentation of an existing FlexRay schedule of a legacy system. This enables to share the bus between legacy devices and nodes running TDL while also allowing their interaction via the exchange of messages.

**TDL Workflow Analysis**

TDL has a significant impact on the development process. In an effort to demonstrate the differences between the TDL approach and the conventional way embedded software is designed, we compare the workflow when using the TDL language and tools with two state-of-the-art commercial tools for the design of

distributed embedded systems. For this comparison, we focus on the automotive industry and compare TDL to Elektrobit's DESIGNER PRO [10] and Vector's DaVinci Tools based on the AUTOSAR standard [11].

Furthermore, we analyze the current development workflow in the automotive industry, which mainly is characterized by the relationship between the original equipment manufacturer (OEM) and its suppliers. We question if and how it is possible to fit TDL in this workflow and evaluate possible ways to integrate it. We also describe what can be done in order to ease the paradigm shift, as for example the application of incremental scheduling which enables the combination of TDL with legacy systems and therefore allows for a smooth transition between the different development approaches.

## 1.3. Structure of the Dissertation

The thesis is divided into the following chapters: Chapter 2 introduces the Timing Definition Language (TDL) by describing its syntax, semantics, recent extensions, graphical front-ends, and tool chain. Chapter 3 details a generic TDL Runtime System for potentially distributed systems programmed in C. The static runtime system requires code which must be dynamically generated, for which the code and schedule generation framework described in chapter 4 is used. Chapter 5 presents plug-ins for this framework for a number of supported target platforms using the FlexRay communication protocol and demonstrates the usage of the framework and plug-ins by means of concrete examples. Chapter 6 takes an in-depth look at the development workflow TDL introduces and compares it to the one currently employed in the automotive industry. The thesis concludes with an outlook on future work in chapter 7.

# 2. Timing Definition Language (TDL)

The Timing Definition Language (TDL) is a high-level textual notation which allows the explicit specification of the timing aspects of a real-time system. Such a system typically performs periodic activities which consist of three phases, namely the reading of sensors, followed by a computation tasks and finally the setting of actuators [12]. In contrast to the traditional real-time system engineering approach, where the timing properties are typically a result of platform-dependent or non-deterministic factors such as the CPU clock speed and the occurrence of interrupts, in TDL the timing of tasks is exactly specified and is preserved on every hardware platform. Therefore, TDL allows developing and testing a software component only once and then deploying it to any supported and powerful enough hardware platform. This leads to significant advantages in system development, especially concerning testing, simulation, deployment and maintenance.

In 2001, the Giotto project at the University of California in Berkeley laid the scientific foundation for TDL by introducing a new programming abstraction called the Logical Execution Time (LET) [3], which decouples the logical timing of computation tasks from their physical execution. The development of TDL started in 2003 in the context of the MoDECS (Model-Based development of Distributed Embedded Control Systems) project at the University of Salzburg. Based on the project's results a spin-off company named preeTEC was founded in 2005 with the goal to make the TDL language available to real-world industry applications. In collaboration with partners from the industry, the development of the TDL language and tools and continued in a sequence of steps until now, whereas a significant milestone was marked by the integration of asynchronous activities in 2008.

In this chapter we present the LET concept and the constructs and syntax of the TDL language, including numerous extensions that have been added recently. Furthermore, we will present an overview of the tool chain enabling the application of TDL to real-world systems.

## 2.1. The Logical Execution Time (LET) Abstraction

The TDL language is based on the concept of Logical Execution Time (LET), which was introduced in the realm of Giotto [3]. It aims to resolve typical shortcomings of embedded software construction, such as platform dependency and lack of compositionality. These are caused primarily by the fact that timing behavior is not specified explicitly but rather is a result of system load and the occurrence of unpredictable events at runtime. The LET abstraction offers a solution by abstracting from the physical execution time of tasks and, in the distributed case, from network communication. It does so by specifying that the inputs of a task, which can be values read from sensors or outputs of other tasks, are read at the beginning of the

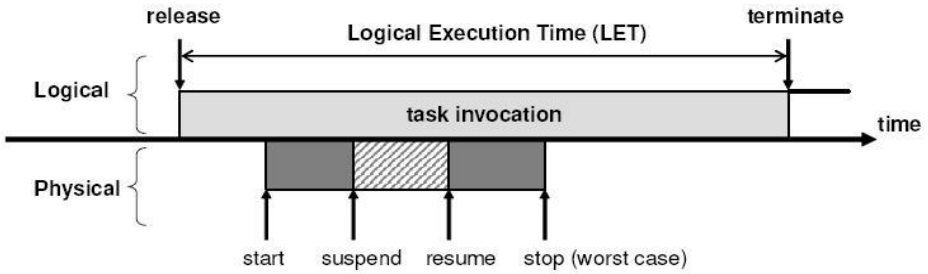


**Figure 1. Logical Execution Time (LET)**

LET period and the outputs provided to other tasks or actuators are only updated at the end of a task's LET. As shown in Figure 1, we call the beginning of the LET the release event and its end the terminate event. Between these, the outputs have the value of the previous execution. It is always defined which value is in use at which time instant and there are no race conditions or priority inversions involved. LET provides the cornerstone to deterministic behavior, well-defined interaction semantics between parallel activities and platform abstraction. As long as physical task execution at runtime and potential network communication take place within the LET of a task, the software will exhibit exactly the same observable behavior on any platform - no matter if it is fast, slow or even distributed.

In TDL, tasks interface with other tasks and sensors and actuators solely via so-called ports. Arranging the data flow by copying from one port to another, e.g. reading input ports and writing output ports, is considered a Logical Zero Time (LZT) operation. On the other side, the execution of the task's body is considered to be a long running operation that cannot simply be ignored.

## 2.2. TDL Language Constructs

TDL provides language constructs for the specification of systems based on the LET abstraction. This section introduces those constructs by means of a simple producer-consumer example. Figure 2 illustrates the data flow between two LET-based components called *Sender* and *Receiver*. *Sender* contains a task *produce* with a LET of 5 ms and *Receiver* runs a task *consume* with a LET of 10 ms. In this example, *consume* receives the output value of *produce*. The vertical arrows in the figure

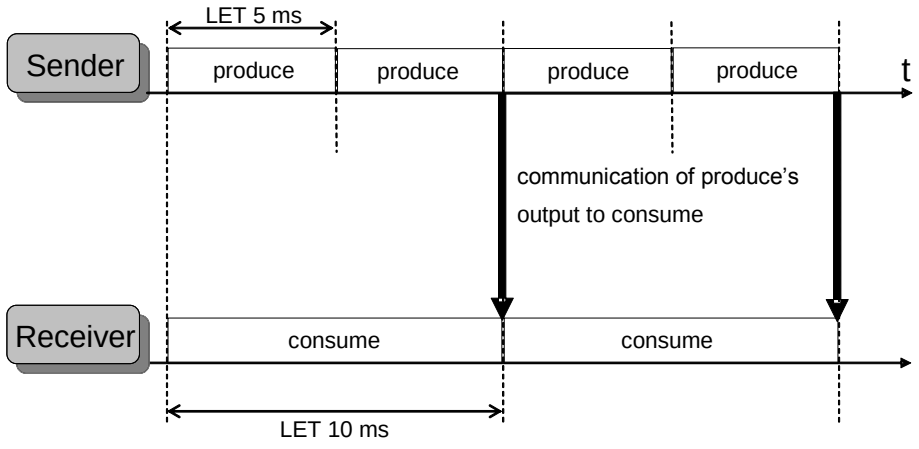


**Figure 2. Timing and data flow of the producer-consumer example**

indicate when the results of *produce* are communicated to *consume*, which is exactly at the terminate event at the end of *produce's* LET. The value is then available for *consume* at its release event at the start of its next LET period.

In the following, we present the TDL code of the *Sender* module step-by-step:

```
module Sender {
```

A TDL *module* represents a component of the system. Modules are the top-level structuring concept of TDL and serves multiple purposes: (1) a module is a named program unit and allows the decomposition of large systems by providing a name space and an export/import mechanism, (2) modules enable the parallel composition of a system as the timing behavior of a module is not affected by other modules, (3) modules serve as units of loading, i.e. a runtime system may support dynamic loading and unloading of modules, and (4) modules serve as unit of distribution because data flow within a module (cohesion) will typically be much larger than data flow across module boundaries (adhesion). All modules of a system are logically executed in sync and the data flow semantics is defined according to the LET abstraction.

In the TDL language a module is specified by the **module** keyword followed by its name and an enclosing curly bracket containing all constructs of the module, as can be seen in the two example modules *Sender* and *Receiver* above.

```
  sensor boolean switch uses getSwitch;
  actuator int display uses setDisplay;
```

The first constructs in the *Sender* module are *sensor* and *actuator* definitions. They are defined by a data type (in this example `boolean` and `int`), an identifier, and the name of a functionality code function followed by the `uses` keyword. This function contains the code which actually implements the reading of a sensor (getter function) and the setting of an actuator (setter function).

```
  public task produce {
    output int o := 10;
    uses produceImpl(o);
  }
```

TDL *tasks* are declared by specifying a task's inputs, outputs and implementation function which is again indicated by the `uses` keyword. The output port `o` of task `produce` is initialized with a value of `10`. Note that a task declaration contains no information on the LET and timing of the task or if it is even executed at all.

```
  start mode main [period=10ms] {
    task
      [freq=2] produce(); // LET = 10ms/2 = 5ms
    actuator
      [freq=1] display := produce.o; // updated every 10ms
    mode
      [freq=1] if exitMain(switch) then freeze;
  }

  mode freeze [period=10ms] {}
}
```

The timing of tasks is defined by so-called *task invocations*, which are one of the activities specified in a TDL *mode*. TDL supports multiple modes of operation for

every module where only one mode of a module can be active at a time. A mode specifies the exact timing of mode activities which are (1) task invocations, (2) actuator updates, and (3) mode switches. The *Sender* module has two modes: A normal mode of operation called `main` and a dead end mode called `freeze`. The conditions and time instants for mode switches are specified in the `mode` section of a mode. Note that `freeze` has no mode section and therefore it is not possible to leave this mode, meaning that after switching to it the module is effectively halted indefinitely. Every mode has a period after which its timing pattern repeats itself for as long as the mode is active. The frequency of an activity inside a mode is indicated by the `freq` attribute followed by an integer value which specifies how many times the activity is carried out per mode period. As for example the frequency of task *produce* is 2, its LET is 10 ms divided by 2 and therefore 5 ms. Note that the actuator `display` is only updated every 10 ms in this example.

```
module Receiver {

  import Sender;

  actuator int display uses setDisplay;

  task consume {
    input int i;
    output int o;
    uses consumeImpl(i, o);
  }

  mode main [period=10ms] {
    task
      [freq=1] consume(Sender.produce.o);
    actuator
      [freq=1] display := consume.o;
  }
}
```

The module *Receiver* makes the *Sender* module accessible by use of an import/export mechanism which is comparable to general purpose programming languages such as Java or C#. By specifying the name of a module after the `import` keyword, all constructs declared as `public` in the given module (e.g. the task `produce` in the *Sender* module) are accessible by the importing module. Also cyclic import relationships are allowed. Constructs of imported modules are referenced by using a dot notation. An example would be the use of `Sender.produce.o` as argument for the task `consume` in the mode declaration of the *Receiver* module.

### Further language constructs

The example above only uses a basic subset of all the available language and syntax features of TDL. In the following, we present a number of additional language constructs by means of modifications to the demo application above. Detailed discussion of those constructs can be found in [13] and a complete feature list and EBNF grammar of the language in the TDL Language Report [14].

**Global output ports.** In addition to ports assigned to specific tasks, it is also possible to define *global output ports.* Such ports can be written to by different tasks, but only by one task per mode. The following is an adapted version of parts of the

*Sender* module above, using the global output port `globalO` instead of the task output port `o`:

```
public output int globalO := 10;

public task produce {
  uses produceImpl(globalO);
}

start mode main [period=10ms] {
  task
    [freq=2] produce(); // LET = 10ms/2 = 5ms
  actuator
    [freq=1] display := globalO; // updated every 10ms
  [...]
}
```

**Slot selection.** By default, the period of a mode and the frequency of a task invocation define consecutive invocations whose LETs equal to the mode period divided by the frequency. TDL *slot selection* allows a more fine-grained specification of the timing task invocations. We call the intervals resulting from the division of the mode period by the frequency *slots*. Slot selection allows the programmer to specify which slots to use for the LET of a task invocation. Figure 3 illustrates the resulting slots for three different combinations of frequency and slot annotations. Note that `slots=1*` also is the default if the slot annotation is omitted, meaning that every slot is used as LET for an invocation of the task specified. In the following example code the mode period of 10 ms is divided into 4 slots with length 2.5 ms each by the setting a frequency of 4. The slot annotation `slots=1-2|4` results in a task invocation LET of 5 ms at the beginning of the mode period and another invocation with LET 2.5 ms from 7.5 ms to 10 ms at the end of the mode period.

```
start mode main [period=10ms] {
  task
    [freq=4, slots=1-2|4] produce();
  [...]
}
```

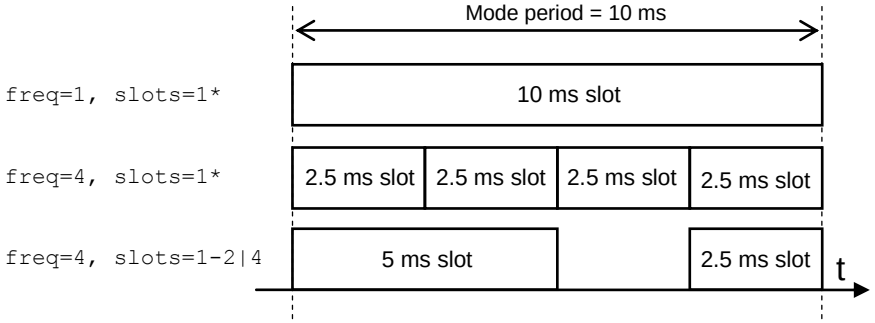


**Figure 3. Slot selection**

**Task splitting.** Typically, a TDL task is associated with a single external function that represents the task's body. However, for some applications it is beneficial to split up this function into two parts, a method which we call *task splitting*. These two parts consist of one simple function (fast step) which is executed in Logical Zero Time (LZT) when the task is released, i.e. at the task's LET start, and another long running function (slow step) which is executed within the LET of the task. The latter may update the task's internal state by some advance calculations such that the next

call of the LZT function can be done fast. This can be utilized e.g. for digital controllers which need to evaluate a polynomial as the core of their implementation. The following code shows task splitting applied to the `produce` task of the *Sender* module above:

```
public task produce {
  output int o := 10;
  uses [release] produceImplFast(o);
  uses produceImplSlow(o);
}
```

**Task sequences.** So-called *task sequences* allow setting actuators immediately after a task invocation. Such a sequence consists of a task invocation followed by a set of actuator updates. When combined with task splitting, this can be used to update actuators right after the fast task part at the LET start. This feature can be helpful in digital controller applications, where as a rule of thumb the reaction time of a controller should be below 10% of the sample time in order to achieve stable controller behavior. The syntax for using a task sequence is to enclose the task invocation and the actuator updates in curly brackets, as the following example code shows:

```
start mode main [period=10ms] {
  task
    [freq=2] {produce(); display := produce.o;}
  [...]
}
```

## 2.3. Transparent Distribution

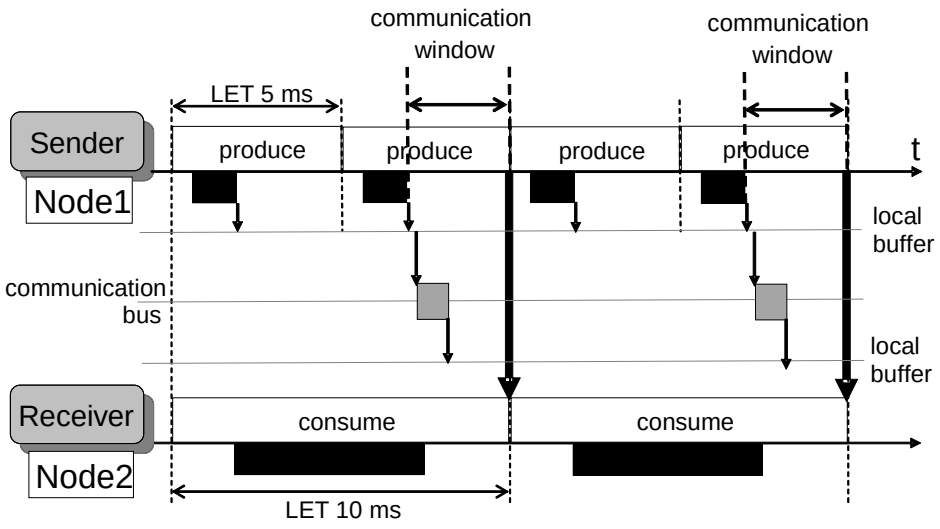


**Figure 4. Physical timing and communication window**

As briefly mentioned above, the behavior of TDL modules is also preserved when they are distributed across multiple nodes of a distributed system. This is done by accounting for the time it takes to communicate values via a network inside the LET of the task which produces them. In order to illustrate this, let us look at the producer-consumer example as described above. Figure 2 shows a logical view of our example system as it does not contain any information on how the two components are deployed on a platform. In contrast, Figure 4 also indicates the physical timing,

assuming that the *Sender* and *Receiver* modules are executed on a distributed system. The *Sender* module is deployed on node *Node1* and *Receiver* on *Node2*, which are connected to each other via a communication bus. The black blocks indicate the physical execution time of the tasks on a node's CPU. In such a setup the output of *produce* must be transferred via the bus. The communication window for doing so spans from the end of *produce's* physical execution time to the terminate event at the end of its LET. As long as the network communication takes place within this window, this distributed system shows exactly the behavior specified by the LET semantics.

Handling network communication inside the LET leads to the notion of *transparent distribution* [15], as the fact that a system is distributed does not change its observable behavior in comparison to execution on a single node. What might differ is only the physical behavior at runtime, in particular the order and length of task executions and the time when messages are communicated. From the perspective of the developer of a LET component, this means the possibility to focus on the functionality without having the target platform in mind, i.e. without caring whether components will eventually executed on the same node or not. Furthermore, the LET concept lays the basis for automatic communication schedule generation, as the size and timing of network frames can be determined automatically by analyzing the communication requirements between LET components.

## 2.4. Execution of TDL Programs

This section explains how the timing specified in TDL modules is realized on a target platform. For that purpose, a compiler (called the TDL compiler) transforms a TDL program, i.e. a timing definition, into instructions of a virtual machine (called Embedded Code or E-Code) which are executed by an appropriate runtime system (called TDL Machine). On the target platform, the TDL Machine is activated periodically and orchestrates the timing and data flow of TDL modules by interpreting their E-Code. Its execution time is kept to a minimum as it only carries out logical zero time activities such as reading sensors, updating task ports, and setting actuators. The TDL Machine is however not responsible for the execution of task functionality code. This must be handled externally, typically by the operating system's scheduler.

The instruction set of the TDL Machine is small and consists of the nine instructions presented in Table 1. For every module a separate sequence of E-Code instructions is generated. It consists of a number of blocks, each block comprising all instructions that must be executed for a specific module during one invocation of the TDL Machine at a particular point in time. A block of E-Code is terminated with a return instruction. TDL modes may consist of multiple blocks, whereas the last block of a mode is followed by a jump instruction which lets execution continue at the beginning of the first block of this mode.

An E-Code block might be further structured by the use of markers that indicate the last termination driver (end of termination drivers – EOT) and the last actuator update (end of actuator updates – EOA). The former is important for solving cyclic dependencies between modules by first executing all termination drivers of all modules and then continuing execution from there on. The EOA marker is required for the correct simulation of TDL modules in simulation environments such as MATLAB/Simulink. As these markers have no functional purpose, we encode them as a *nop(1)* and *nop(2)* instruction respectively.

| Instruction | Meaning |
|---|---|
| *nop(f)* | A dummy (no operation) instruction. The argument *f* is used as a marker for identifying different sections in the E-Code, such as the end of termination drivers and the end of actuator updates. |
| *call(d)* | Executes the driver *d*. |
| *release(T)* | Marks the task *T* as ready for execution. |
| *future(a, dt)* | Plans the execution of the E-Code block starting at address *a* in *dt* microseconds. |
| *if(g, elsePC)* | Proceeds with the next instruction if guard *g* evaluates to true, else jumps to *elsePC*. |
| *jump(a)* | Jumps to the instruction at address *a*. |
| *return* | Terminates an E-Code block. |
| *repeat(a, n)* | Uses a counter per module for jumping *n* times to instruction *a*. After that it continues with the next instruction. This instruction allows for compacting an E-Code block which repeats itself. |
| *switch(M)* | Performs a mode switch to mode *M*, i.e. the TDL Machine continues at the entry point of *M*. In addition, the module's repeat counter is set to zero. |

**Table 1. TDL Machine instruction set**

The following lists an example E-Code obtained after compilation of the *Receiver* module of the producer-consumer example from above:

```
00 call(1)          // actuator init: setDisplay(display)
01 return()

02 call(0)          // terminate task: consume
03 nop(1)           // EOT - end of task terminations marker
04 call(2)          // actuator update: display := o
05 call(1)          // actuator setter: setDisplay(display)
06 nop(2)           // EOA - end of actuator updates marker
07 call(3)          // prepare task for release: consume
08 release(0)       // release task: consume (uses consumeImpl)
09 future(11,10000)// continue at instruction 11 in 10000 us
10 return()
11 jump(2)          // jump to instruction 2
```

The first block of E-Code (in this example code only instruction 0 and the corresponding return instruction) is only executed once at system startup. It is used for the initialization of actuators and task output ports. The call instruction executes the driver with the index 1. This driver updates the actuator `display` by calling the actuator update function `setDisplay` with the actuator port as argument. Drivers are used to encapsulate functionality such as port copy operations in order to be able to

support different programming languages easily. There are also drivers for evaluating guards, switching modes, and starting and stopping of task functions.

After initialization, execution continues at the entry point of the start mode of the module. In our example there is only one mode and it is therefore the start mode. Its entry point is instruction 7, which prepares task consume for execution by updating its input ports. Afterwards the task is released and then the future instruction sets the time and instruction where the TDL Machine must continue execution for this mode. The future time is relative, i.e. instruction 9 means that after 10000 microseconds execution must continue at instruction 11. Instruction 11 is actually a jump to instruction 2, which terminates task *consume* and subsequently updates the corresponding actuators. This finishes the mode cycle as we again arrived at instruction 7, the entry point of the mode.

The E-Code of the *Sender* module, which contains two modes, looks like this:

```
00 call(1),          // actuator init: setDisplay(display) */
01 return()

02 nop(1)            // EOT - end of task terminations marker
03 nop(2)            // EOA - end of actuator updates marker
04 future(6,10000)   // continue at instruction 6 in 1000000 us
05 return()
06 jump(2)           // jump to instruction 2

07 call(2)           // sensor getter: switch := getSwitch()
08 call(0)           // terminate task: produce
09 nop(1)            // EOT - end of task terminations marker
10 call(3)           // actuator update: display := o
11 call(1)           // actuator setter: setDisplay(display)
12 nop(2)            // EOA - end of actuator updates marker
13 if(0,16)          // if guard 0 is true goto 16; mode switch
                     // guard: exitMain
14 call(4)           // mode switch driver
15 switch(0)         // mode switch -> freeze
16 call(5)           // prepare task for release: produce
17 release(0)        // release task: produce (uses produceImpl)
18 future(20,5000)   // continue at instruction 20 in 5000 us
19 return()
20 call(0)           // terminate task: produce
21 nop(1)            // EOT - end of task terminations marker
22 nop(2)            // EOA - end of actuator updates marker
23 call(5)           // prepare task for release: produce
24 release(0)        // release task. produce (uses produceImpl)
25 future(27,5000)   // continue at instruction 27 in 5000 us
26 return()
27 jump(7)           // jump to instruction 27
```

There is an initialization section comprising the first two instructions, then the E-Code section for mode `freeze` spanning from instruction 2 to 6 and finally a set of instructions for mode `main` from instruction 7 to 27. The E-Code follows the basic pattern as presented for the *Receiver* module above, but additionally includes a conditional mode switch handled by an if instruction (line 13) and the corresponding switch instruction (line 15).

Note that the TDL compiler does not take any platform-specific aspects into account, meaning in specific that the E-Code looks exactly the same no matter if modules are distributed across multiple nodes or not. Platform-specific aspects are entirely handled by the so called glue code, which will be discussed in 2.7.

## 2.5. Extensions for Asynchronous Activities

The most important recent extension to the TDL language is the integration of *event-triggered* (alias *asynchronous*) activities. Before this extension, TDL only supported the platform independent specification of the time-triggered aspects of a real time system by the strictly periodic execution of statically scheduled activities, such as task invocations and actuator updates. A pre-computed schedule guarantees that the timing requirements of the system will be met in any case by taking the worst case execution time (WCET) into account. Such operations are also called *synchronous* (alias *time-triggered*) activities. The timing requirements of such activities are typically in the range of milliseconds or sometimes even below.

While the time-triggered execution of periodic tasks provides the cornerstone of dependable real-time systems, in addition many such systems execute asynchronous activities that are, for example, triggered by the occurrence of an external hardware interrupt or any other kind of trigger. In the context of a dependable real-time system such asynchronous activities are considered to be not as time critical as synchronous tasks are, and can therefore be executed in a background thread while the CPU is idle otherwise.

The main challenge when adding asynchronous activities to TDL was to execute them as timely as possible while not sacrificing the guaranteed execution of synchronous activities. Furthermore, the data flow between the two domains must be properly synchronized. Adding asynchronous activities could be done in a platform-specific way by directly programming at the level of the operating system or task monitor and so to speak "outside" of TDL. However, this approach has two drawbacks: (1) it is platform dependent and (2) it does not support proper synchronization of data exchanged between synchronous and asynchronous activities. Therefore we extended TDL by a notation for asynchronous activities and provided a runtime system for this extended TDL language on a number of target platforms.

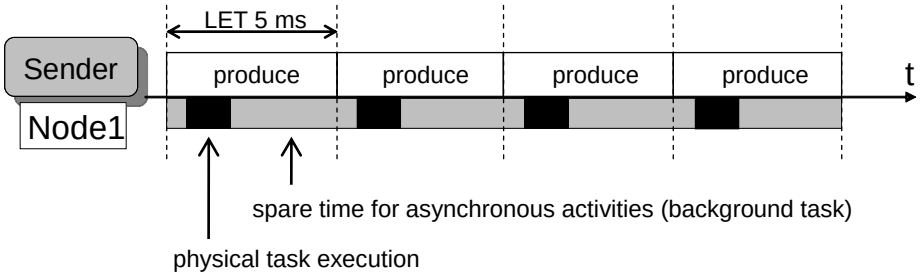


**Figure 5. Integration of asynchronous activities**

An asynchronous activity in TDL is an activity that is carried out in the spare time between the execution of time-triggered (synchronous) activities and thereby does not disturb the real time properties of a system. Figure 5 refers to *Node1* of the producer-consumer example from above and indicates the physical execution of task *produce* and the spare time available for the execution of asynchronous activities. To keep the processing of asynchronous activities simple and as we assume that they are not as time-critical as synchronous tasks, we do not allow asynchronous activities to be preempted by other asynchronous activities but only allow preemption by synchronous activities. The TDL runtime system takes care of the synchronization of the data flow between synchronous and asynchronous activities such that reading input ports, updating output ports, and performing actuator updates are atomic actions.

Asynchronous activities are introduced at the level of the TDL module construct. Every module may optionally declare asynchronous activities as the last section within the module construct using the **asynchronous** keyword. The following version of the producer-consumer example from above uses the same tasks and data flow but both tasks are now triggered asynchronously. The producer task is triggered by an external interrupt and the consumer task by the update of the output port of the producer task.

```
module Sender {

  actuator int display uses setDisplay;

  public task produce {
    output int o := 10;
    uses produceImpl(o);
  }

  asynchronous {
    [interrupt=intLine1, priority=5]
      produce(); display := produce.o;
  }
}


module Receiver {

  import Sender;

  actuator int display uses setDisplay;

  task consume {
    input int i;
    output int o;
    uses consumeImpl(i, o);
  }

  asynchronous {
    [update=Sender.produce.o]
      consume(Sender.produce.o); display := consume.o;
  }
}
```

TDL supports the grouping of asynchronous activities into sequences that are triggered as one unit and executed strictly sequential. Any such sequence has an associated trigger event, an optional guard, and a sequence of asynchronous activities. An asynchronous activity may be a task invocation or an actuator update. In both example modules above, the producer and consumer task invocation is immediately followed by a corresponding actuator update. A task may either be invoked synchronously or asynchronously but not both. Also, an actuator update must either be done synchronously or asynchronously but not both. Note that mode switches are the only TDL activity which cannot be invoked asynchronously, as a mode switch must be synchronized with the corresponding mode period and must not preempt any synchronous task.

Triggering an asynchronous activity sequence means that the sequence is registered for execution at some later time at the discretion of the TDL runtime system. Any additional triggering of a registered activity sequence is ignored until the execution of

this activity sequence starts. Parameter passing takes place as part of the execution not at the time of registration.

The kind of event that triggers the execution of an asynchronous activity sequence is specified by the attribute name `interrupt`, `update`, or `timer`, where the first two can be found in the examples above. In case of an interrupt, the attribute value must be an identifier which needs to be mapped to platform-specific interrupt specifications, e.g. to a specific hardware interrupt pin, outside the TDL source code. This identifier is `intLine1` in the *Sender* module above. In case of a port update, the attribute value must be the name of an output port. For the asynchronous activity in the *Receiver* module, this port is the output port of the producer task, `Sender.produce.o`. Whenever this port receives a value, it triggers the asynchronous activity sequence. In case of a timer, the attribute value must be an integer greater than zero. It describes the period of a timer in microseconds.

The priority of an asynchronous activity sequence is specified by the attribute name `priority` and a value greater or equal to zero, where higher numbers mean higher priority. The default priority is the lowest value. Only a single asynchronous activity is executed in the spare time between synchronous activities until it finishes. The priority attribute determines which activity is executed next by affecting the queuing order of registered asynchronous activity sequences. Therefore it should not be mixed up with a thread priority level.

## 2.6. Visual TDL Tools

For the development and editing of TDL modules and complete TDL systems, two visual tools are available. The TDL:VisualCreator is used for the platform-independent editing of TDL modules, while the TDL:VisualDistributor enables the deployment of TDL modules on a concrete, potentially distributed, target platform. In the following we describe both tools in detail.

The TDL:VisualCreator is a syntax-driven, graphical editor for TDL modules. Consequently, it supports the full feature set of the TDL language and allows the import and export of arbitrary TDL code. Figure 6 shows the user interface of the TDL:VisualCreator, depicting the *Sender* module of the producer-consumer example as presented above. The interface is divided into three main parts. To the left there is a tree representation of all constructs of a module. Specific properties of these elements, e.g. the period of a mode or the frequency of a task, can be edited in property fields below. The large modeling canvas to the right is used to model data flow, e.g. between a task's ports and ports of other tasks, sensor, and actuators with respect to a specific TDL mode.

In addition to running the TDL:VisualCreator tool standalone, it can also be run integrated in MATLAB/Simulink [16]. Simulink is a commercial tool by The Mathworks for modeling, analyzing and simulating dynamic systems. It is widely used for control applications, including automotive system engineering. Simulink is tightly integrated with MATLAB and offers a block diagram style interface. The Simulink integration enables additional features of the TDL:VisualCreator, namely the usage of standard Simulink blocks for the design of the functionality of TDL tasks and the simulation of the behavior of complete TDL systems. Due to the LET abstraction, this simulation is guaranteed to be equal to the observable behavior the system will show when it is finally executed on a concrete hardware platform. This is a unique feature in comparison to other simulation tools for potentially distributed systems, which typically take platform details such as processor speed and communication protocol latencies into account in order to obtain an accurate simulation result. Using TDL, systems can be precisely simulated even before the target platform is even known as

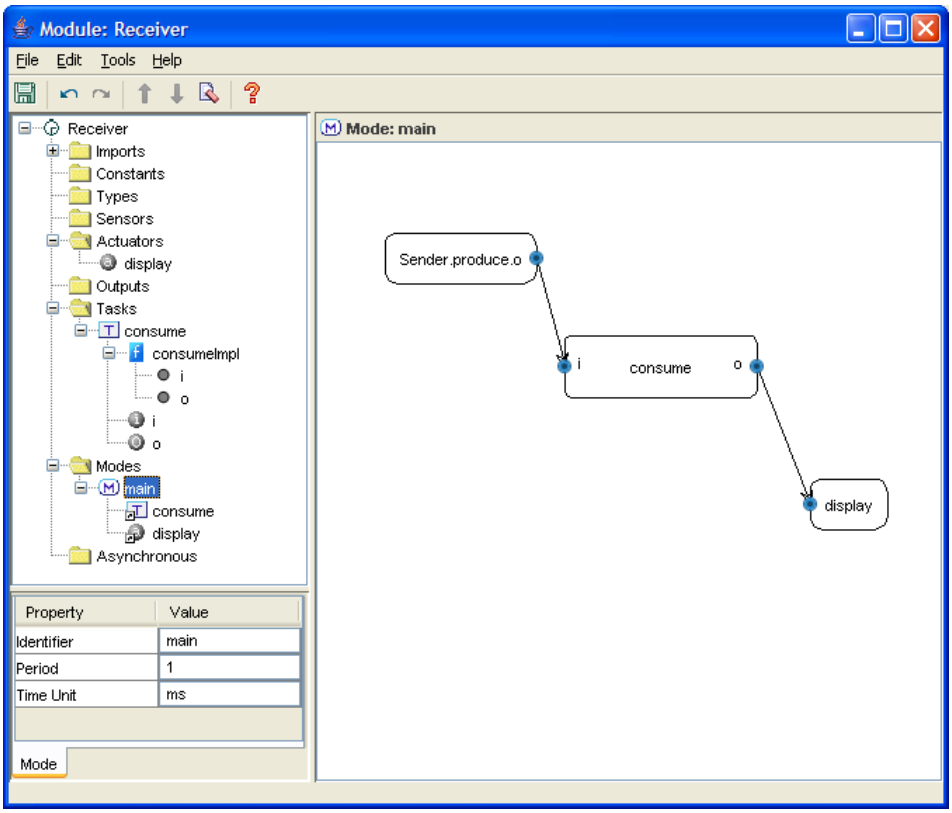


**Figure 6. TDL:VisualCreator user interface**

the LET abstraction exactly specifies the behavior in the time and value domain. This behavior is also guaranteed when it is later executed on any hardware platform by means of the TDL Machine discussed above. The only notable exception, for which accurate simulation independent from the platform is not possible, are TDL systems which incorporate asynchronous activities. Those are simulated as soon as the corresponding trigger event occurs and in logically zero time or within their WCET if one is given. This limitation is inherent to the chosen semantics of asynchronous activities, which are executed as a background task on the platform and are supposed to be used for non-critical tasks. For details on the Simulink integration and simulation refer to [13].

While the TDL:VisualCreator's purpose is to provide platform-independent modeling, the TDL:VisualDistributor is used for mapping TDL programs to specific platforms and eventually to generate code for the complete TDL system. It is a frontend for the deployment of TDL modules on a potentially distributed hardware platform. It allows specifying the platform, i.e. the nodes and communication buses connecting them. Support for an open ended set of communication and node platforms can be added via a plug-in architecture. When mapping a TDL module to a concrete node, a platform-specific Worst Case Execution Time (WCET) must be set for every task of a module running on this node. Furthermore, the sensors and actuators of a TDL module must be assigned to specific hardware devices either by specifying an external function or via a graphical interface in case the corresponding node plug-in supports that. Finally, the complete code for the system can be generated. This also triggers the fully automatic communication schedule generator which determines the communication requirements of TDL modules by their deployment to nodes. When using the MATLAB/Simulink integration feature of the TDL:VisualDistributor, the functionality code can also be generated automatically from the Simulink model by a standard MATLAB tool named Real-Time Workshop Embedded Coder (RTW-EC). The

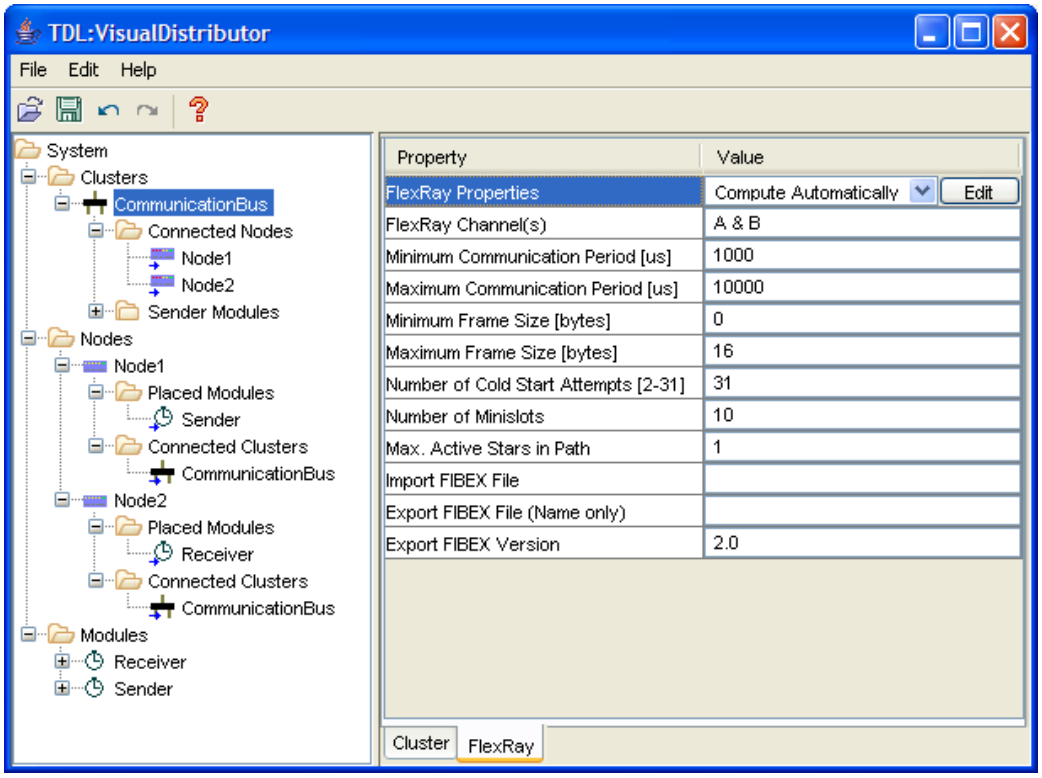


**Figure 7. TDL:VisualDistributor user interface**

next section contains a detailed overview of the code generation process and the complete TDL tool chain which is controlled by the TDL:VisualDistributor.

## 2.7. TDL Tool Chain

The previous sections explained the TDL Compiler and the TDL Machine as well as the two front-end visual tools TDL:VisualCreator and TDL:VisualDistributor. All these are core parts of the TDL tool chain which we will look at in detail in this section and explain how code generation by the click of a button is realized. As pointed out above, the TDL Compiler does not take any platform-specific information into account. This information is contained in the so-called glue code, which is platform-specific and is required in order to obtain a functional TDL system. All tool chain components and tools are implemented in Java 1.5, whereas generated the glue code can be in any language or format which a target platforms requires.

In short, the glue code comprises all code and information which is needed to execute TDL modules on a potentially distributed platform. What it actually contains highly depends on the specific target platform. When for example an operating system which includes a file system is used, it is possible that the TDL Machine directly reads and interprets the E-Code file. On a single node system, all that needs to be done then to get a working TDL system is to ensure the task functionality code is executed by the operating system's scheduler at the proper time instants. Without a file system, E-Code must be represented for example as C code. Distributed systems however require extra glue code for the initialization and utilization of the communication system connecting the nodes.

Apart from the different requirements of individual platforms, Figure 8 sketches the basic tool chain elements which are required for the code generation for every

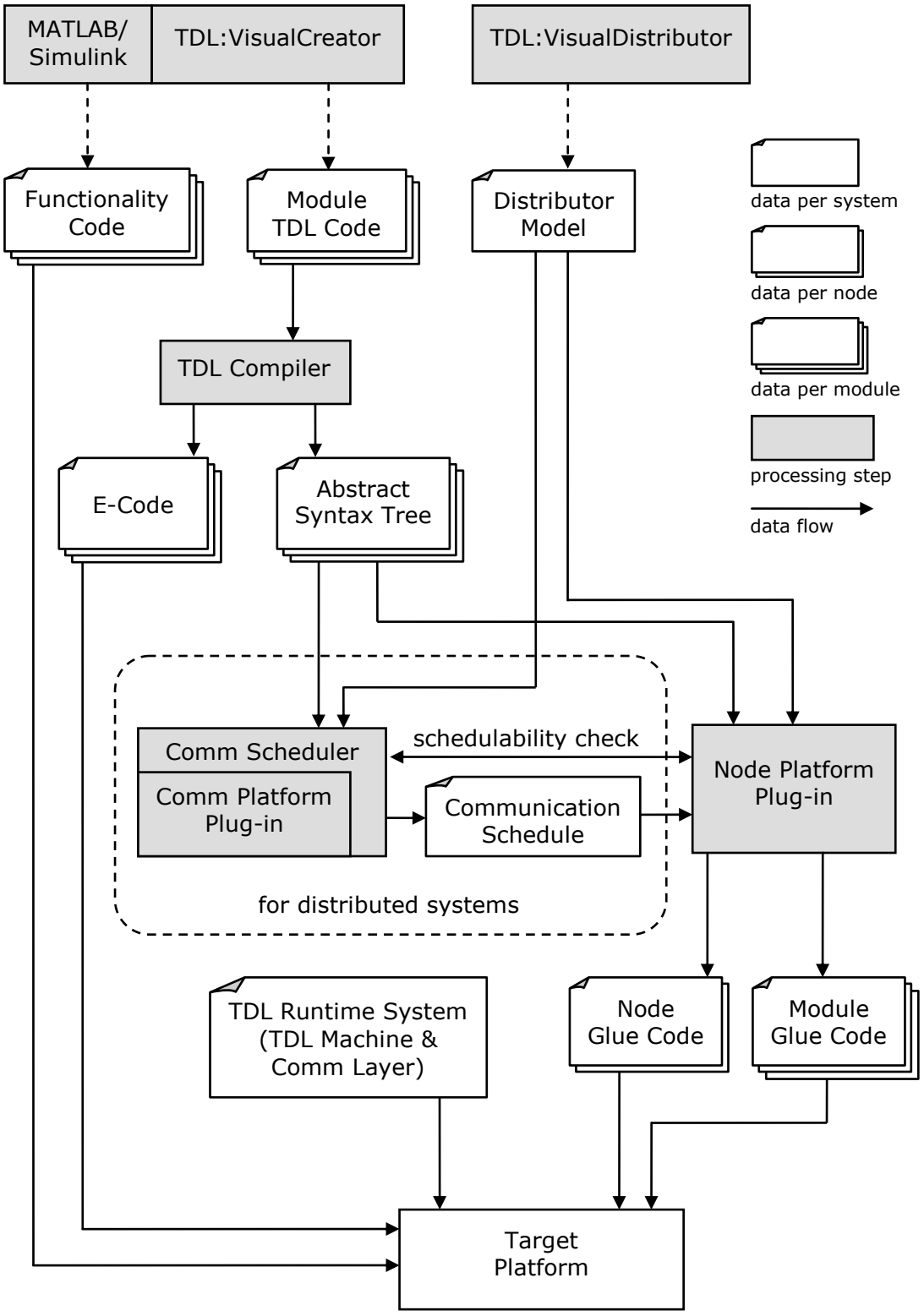


**Figure 8. TDL tool chain overview**

potentially distributed TDL system. On top, we have the TDL:VisualCreator which can be used to create the code of TDL modules and MATLAB/Simulink which optionally generates the corresponding module functionality code. The TDL:VisualDistributor acts as an editor for platform-specific details of modules and nodes, as well as for the mapping of modules to nodes. Furthermore, it controls and coordinates all entities of the TDL tool chain. These three tools are optional (indicated by the dashed arrows) in the sense that it is also possible to provide TDL modules, functionality code, and distributor model in textual form and run the whole code generation process by using the TDL:VisualDistributor in batch mode.

The first processing step is the invocation of the TDL compiler which compiles all modules of the system and creates one E-Code file per module. Note that an E-Code file does not only contain E-Code instructions, but also information on drivers, guards, and asynchronous activities, among others. In addition, the TDL compiler provides the abstract syntax tree (AST) to the communication and node platform plug-ins. Just like binary E-Code files, the AST also contains all information of a compiled TDL module, but in the form of Java objects in order to speed up and simplify the interaction between tool chain entities.

If the TDL system consists of more than one node, those nodes must be connected by some type of communication network. In such a case, a network schedule is generated by the Comm Scheduler. It analyzes the modules and determines the communication requirements between nodes by considering the module to node mapping found in the distributor model. Specific communication protocols are supported via the Comm Scheduler plug-in, which takes properties such as the frame layout and speed of the protocol into account. TDL specific communication information is written to a file called Comm Schedule. This data consists for example of the exact mapping of TDL output ports to communication frames by using so-called *dynamic multiplexing*. This scheduling approach allows the creation of a static schedule for a TDL system whose modules are able to change modes dynamically and independently. It is explained in detail in section 4.3. To ensure the created schedule leads to a schedulable system overall, the Comm Scheduler interfaces with the Node Platform plug-in to check whether it is able to find a corresponding task schedule on basis of the timing of the communication frames. This approach avoids that a communication schedule is generated for which eventually no task schedule can be found.

In contrast to the Comm Scheduler, which is called once per communication bus, the Node Platform plug-in is called once for every node in the system. It is also possible that different types of platform plug-ins are invoked in case the system consists of a set of heterogeneous nodes. The purpose of a Node Platform plug-in is to generate glue code which allows the execution of a TDL system on a target platform. It uses configuration properties from the distributor model, the abstract syntax tree of all modules and the communication schedule provided by the Comm Scheduler in case the system is distributed. The generated glue code consists of module and node glue code. For distributed systems, the glue code also comprises so-called stub modules which act as a remote instance of a module when it is imported by another module on a remote node.

Apart from corresponding platform plug-ins, a TDL Runtime System must be implemented in order to support specific node and communication platforms. It consists of the TDL Machine, which ensures the proper timing of the system according to the LET semantics, and the TDL Comm Layer, which handles communication between TDL modules if they are located on different nodes of a distributed system. In contrast to the glue code, the TDL Runtime System is static code, i.e. it does not depend on concrete TDL modules.

Now we have all building blocks for a complete TDL system, namely the module functionality code, module E-Code, the TDL Runtime System and the module and node glue code. A final processing step might be necessary to integrate all these elements for execution on a platform, e.g. compilation and linking on C-based platforms.

## 2.8. Related Work

This section compares the TDL language and tools to related approaches for the design of distributed real-time systems, which employ different with different models of computation.

### Giotto

As already pointed out above, TDL inherits its basic concepts, most importantly the Logical Execution Time abstraction, from the Giotto language [3]. However, TDL extends Giotto by a number of features. These include a more convenient syntax, more control over the timing of periodic activities by the introduction of slot selection, and the ability to update actuators right after the completion of a task (so-called task sequences). Further notable extensions are (1) the addition of a component model by means of the module construct and (2) the integration of asynchronous activities. The latter is especially significant as Giotto only allows the specification of purely time-triggered activities, while TDL adds support for event-triggered activities. Apart from the listed language related improvements, a full-fledged tool chain exists for TDL, which features graphical modeling tools, simulation support and code generation for distributed systems. The TDL tool chain enables the application of Giotto concepts in real-world industrial projects.

### xGiotto

xGiotto [4] is, as the name already implies, an extended version of Giotto. Most importantly, it adds an implementation language for the body of a task and asynchronous event handling by means of a new syntax for expressing time-triggered and event-triggered activities.

Adding a new language for the functionality code significantly increases the complexity of xGiotto and its tool chain. It is not clear to us what the advantage of this extension for a real-time system is, given that it is supposed to be compiled into so-called F-code, which is an instruction set for a virtual stack machine that needs to be interpreted at run-time.

The new syntax is based on a mechanism called *event scoping*. An event scope (also called a reaction block) defines the actions to be taken in a given time span which will be terminated after a specified time or by the occurrence of a specified event. xGiotto builds on the assumption that asynchronous events reoccur only after a certain waiting time. Event scopes may be nested and, by means of special statements and options, they allow a variety of patterns to be specified for the activities inside an event scope. Besides some exceptions with non-harmonic mode switches, this includes all possibilities of Giotto programs and it adds the execution of LET-based asynchronous task invocations. Event scoping also separates the LET of a task invocation from its execution period, which is similar to TDL's slot selection approach. In fact, many xGiotto examples can be transformed to TDL in a straight-forward way, including the xGiotto asynchronous activities, which can be expressed as guarded synchronous task invocations within selected slots. xGiottos's event scoping syntax looks somewhat verbose and in particular, the timing behavior of an asynchronous task invocation is hard to read because it depends on all reaction

blocks within the same container scope as the asynchronous task invocation. In contrast, TDL sticks more closely to the lean Giotto syntax for specifying synchronous activities and adds additional constructs for specifying asynchronous activities.

The handling of events differs between TDL and xGiotto. There is no guarantee when and if at all an event is handled in TDL whereas in xGiotto the time until an event is processed is bounded according to the specification of the event scope. Also in contrast to xGiotto, in TDL there is no LET assigned to an asynchronous activity as ports are read and written right before and after its execution. TDL's advantage is that it can also express long-running background tasks for which a reasonable worst case execution time is not available.

Further notable differences between xGiotto and TDL are the lack of a component model and that xGiotto is not targeted at distribution. To our knowledge, there is also no simulation support available for xGiotto.

### Hierarchical Timing Language

The Hierarchical Timing Language (HTL) [5] is another language which bases on the LET abstraction introduced by Giotto. Its name is derived from the ability to hierarchically refine abstract task invocations at a later point in time. Although this refinement does not add expressiveness, as refined code can also be expressed by an equivalent non-refined one, it results in a much more compact representation and simplifies program analysis and schedulability tests.

A key concept in HTL is the notion of communicators. Those are typed variables used to arrange time-triggered data flow and are only accessible at specific, periodic time instants. Communicators define a fixed communication matrix used throughout a HTL system. The LET of a task results from the communicator instances it reads from and writes to. This approach allows for decoupling the LET from the execution period of tasks and also provides support for task sequences. In TDL, these goals are achieved by slot selection which provides even more flexibility because TDL allows that a task is invoked several times per mode period and that each invocation specifies its own LET.

HTL uses modules for parallel composition and as units of distribution in a similar way as TDL does. However, HTL modules are neither independent nor self-contained and therefore not truly reusable as they depend on globally defined communicators and their timing. Furthermore, there is no way to specify asynchronous activities in HTL.

### Synchronous Languages

Synchronous languages base on the *synchrony hypothesis*, which states that the output of a system is synchronous with its input. Internal actions are considered to be instantaneous and also communications are performed via instantaneous broadcasting, i.e. every computation is assumed to be executed by an infinitely fast machine and therefore takes zero time. Synchronous languages are designed to program reactive systems, which are systems that maintain a permanent interaction with their environment. Synchronous programs react to some stimulus, i.e. events, by computing some output based on the input and the state of the program, hence also the term *synchronous reactive programming* is used to describe this programming discipline. Prominent examples of synchronous languages are the declarative, data flow language Lustre [17] and Esterel [18], which represents an imperative synchronous language with explicit control flow.

Actual machines for which the ideal synchronous model is realistic do exist, for example strongly synchronized hardware or VLSI architectures, where internal actions and communications occur with on clock tick of the system [19]. Typical implementations however are targeted at asynchronous platforms and only approximate synchrony by computing any reaction to an event as fast as possible and before the next event occurs. Obviously, it highly depends on the power of the execution platform how accurate that approximation is and response time may vary significantly in practice. Implementing the synchronous model has been proven to work for single node systems, but it is no longer feasible when additionally the communication delay of a distributed system is introduced.

In TDL, the synchrony hypothesis is applied only to the TDL Machine, whose execution is assumed to take logical zero time. It is however not applied to computational tasks or network communication, which both are considered to be long running operations with a significant execution time (the LET) that cannot simply be ignored. The LET abstraction of the so-called *timed model* does not only result in value-deterministic systems as the synchronous model does, but also in time-deterministic ones, as the reaction time of the system does not depend on the execution platform in any way [20].

**Timed Multitasking**

Timed Multitasking [21] is another time-centric programming model which aims at the inclusion of timing properties at the programming level so that they are preserved throughout the software lifecycle. Out of criticism of the purely time-triggered Giotto model, which is incapable of handling sporadic events, Timed Multitasking uses events and deadlines instead of time triggers. Tasks, which are called *actors* and communicate between each other via *ports*, are activated when their inputs fulfill certain criteria, i.e. when an associated trigger condition such as the activation of an interrupt is met. However, the results of the task's computation are only available to other tasks at the task's deadline. Although the triggers of actors are unpredictable in general, this results in deterministic timing behavior regarding the reaction time of actors, which is a valuable property for control algorithm design. When no deadline is specified, an actor's outputs are available immediately. An actor's execution can be preempted within the interval between the trigger event and its deadline. The execution time of an actor must be specified, but it does not necessarily have to be its worst-case execution time (WCET) as for Giotto and TDL tasks. Due to this fact, but also as for an event-triggered system it is generally impossible to guarantee that all actors will meet their deadlines, the Timed Multitasking approach must handle missed deadlines. This is accomplished by so-called *overrun handlers*, which are application-dependent and for example can be used to bring the system into a safe state when a specific actor is nor able to finish by its deadline.

After specifying all timing properties at design time, the Timed Multitasking model is compiled to be executed on a specific runtime system. This step is called software synthesis or code generation and transforms the model into executable code, i.e. software tasks and interrupt service routines (ISRs). In combination with the runtime system this code ensures the function and time determinism of the system.

Timed Multitasking has also been extended to support remote communication in distributed real-time systems, an approach which the authors named Distributed Timed Multitasking [22]. Distribution is implemented by transmitting the ports connecting the actors via a communication network using global signals. Although it is transparent to the individual actors if their inputs come from a local or remote actor, the timing behavior might actually be different as communication time

between actors differs depending on whether network or local communication is required and on what type of network is used.

In contrast to Timed Multitasking, TDL overcomes Giotto's lack of event support by the introduction of asynchronous activities as described above, but without sacrificing any of the real-time properties of time-triggered tasks. There is no need for overrun handling in TDL, as time-triggered tasks are guaranteed to finish in time because their WCET must be specified and asynchronous activities simply run until their completion. However, high priority event-triggered tasks must be represented as time-triggered tasks, for which TDL's slot selection provides more flexibility than for example Giotto. In distributed TDL systems, it is not only transparent from which node a port comes from as in Distributed Timed Multitasking, but it is also available at the exact same time instant independent from whether network communication is required or not, thus providing complete transparency.

# 3. TDL Runtime System

The TDL Runtime System enables the execution of TDL modules on a target hardware platform. As illustrated by Figure 9, it represents a middleware layer between TDL modules and the hardware platform, which includes the operating system, hardware drivers, and the communication interface. It thereby abstracts from concrete platforms and ensures TDL modules are executed according to the TDL semantics regardless of the target platform.

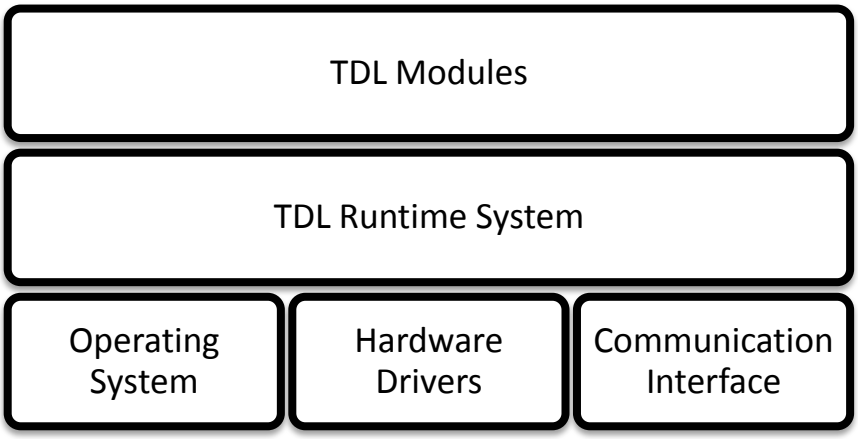


**Figure 9. TDL system layers**

Claudiu Farcas also developed a TDL runtime system including a platform abstraction layer [7], but which lacked event-triggered processing and a modular communication layer. In contrast, our runtime system allows executing asynchronous activities, introduces a plug-in concept for the TDL Comm Layer, and improves support for non-preemptive platforms. Furthermore, we add support for TDL extensions, such as structured data types, global output ports, slot selection, and the cyclic import of TDL modules.

Figure 10 depicts a more detailed view of the TDL system components developed for the C programming language. It shows the TDL Runtime System (indicated by the boxes with thick borders) and how it is connected to the generated glue code (gray boxes) and other entities such as hardware drivers, the operating system, and the functionality code of TDL modules. Note that the file names of generated glue code files are postfixed with an underscore. The figure shows the include relationships between entities which represent C header and body files. An arrow pointing to an entity indicates that it includes the source file or library from which the arrow originates. The elements inside the dashed bounds are only required for distributed systems. The developed runtime system is a framework as the TDL Machine calls

hooks for drivers, guards and module initialization functions which are accessed through structures in the module glue code, which represents compiled TDL modules as C code so that it is not necessary to read E-Code files on embedded platforms.

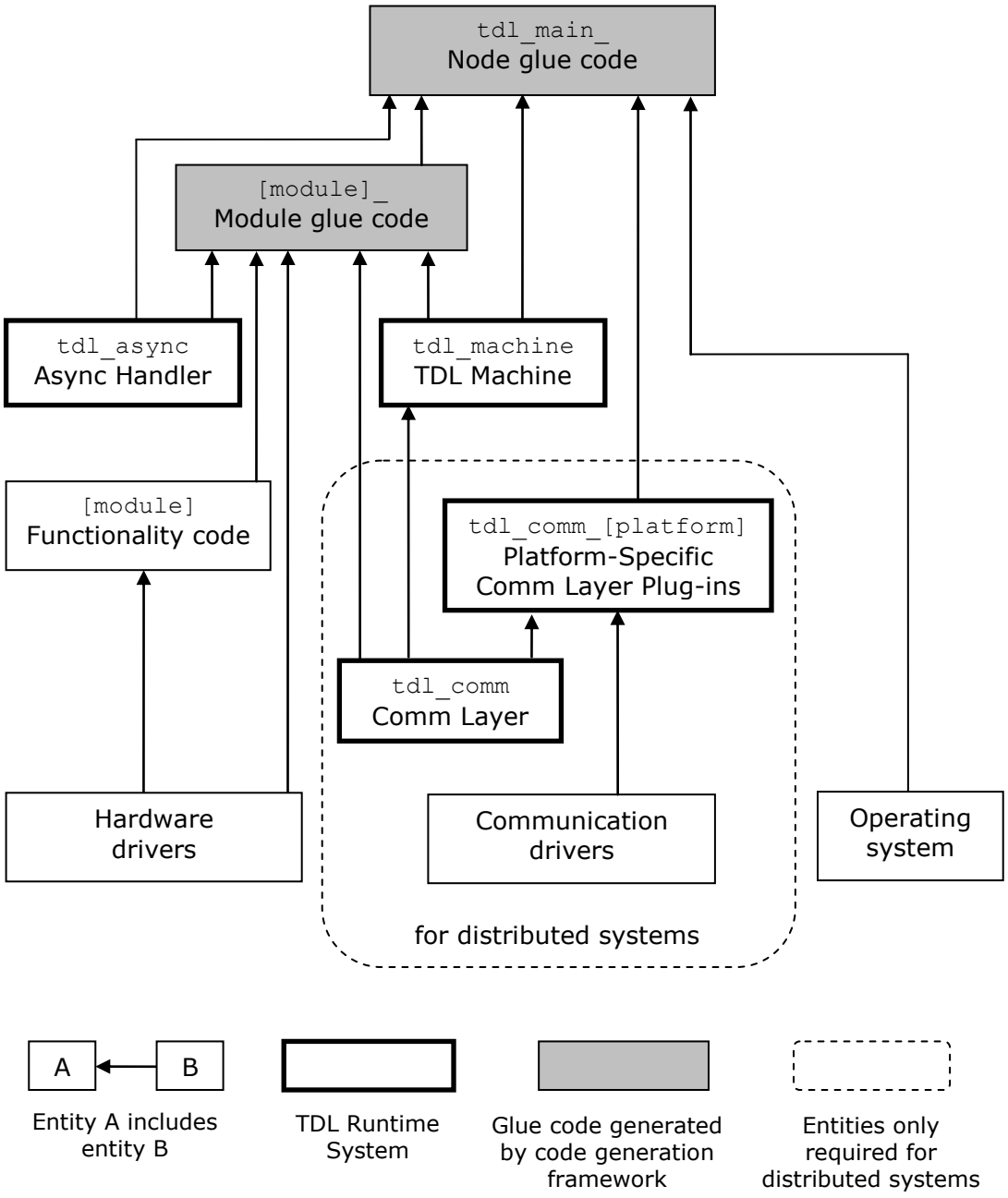


**Figure 10. TDL Runtime System include relationships**

A complete TDL system consists of the following entities:

- TDL Runtime System

  The runtime system is static code which is used in every TDL system in the same way and is therefore application- and platform-independent, with the exception of communication protocol-specific plug-ins to the TDL Comm Layer

framework. It is divided into the following parts which we describe in detail in this chapter:

- TDL Machine (`tdl_machine.c/h`)

  The TDL Machine's main purpose is to interpret E-Code and thereby guarantee the execution of TDL modules according to the timing behavior they specify. See section 3.1.

- TDL Async Handler (`tdl_async.c/h`)

  The Async handler implements a priority queue for the execution of asynchronously triggered activities specified in TDL modules. See section 3.2 for details on the synchronization mechanism for the data flow between synchronous and asynchronous activities and its implementation in the context of the TDL Runtime System.

- TDL Comm Layer (`tdl_comm.c/h`, `tdl_comm_<platform>.c/h`)

  The TDL Comm Layer framework is only required for distributed TDL systems and consists of a communication platform-independent part (`tdl_comm`) and platform-dependent plug-ins (`tdl_comm_<platform>`). It abstracts from the concrete communication protocol used and provides functions which transfer TDL ports via a specific communication bus. It is described in 3.3.

- TDL type mapping (`tdl_types.h`)

  A file named `tdl_types.h` maps TDL types to C language types so that the size of every type corresponds to that defined in the TDL language. A default mapping is provided but it can also be altered for specific compilers and platforms. The type mapping header file is not shown in the figure above but it is used by all TDL Runtime System entities and also by the glue code. The default type mapping is as follows:

```
typedef unsigned char  tdl_boolean; //1 bit flag
typedef signed char    tdl_byte;    //1 byte integer
typedef unsigned char  tdl_char;    //1 byte character
typedef short int      tdl_short;   //2 byte integer
typedef long int       tdl_int;     //4 byte integer
typedef long long int  tdl_long;    //8 byte integer
typedef float          tdl_float;   //4 byte floating point
typedef double         tdl_double;  //8 byte floating point
```

- Glue code

  The so-called glue code comprises all dynamically generated C code required to execute TDL modules on a potentially distributed system. It is specific to the application (i.e. the TDL modules) on one hand and the platform of the TDL system on the other. We call the former part of the glue code the module glue code and the latter the node glue code, which is contained in the TDL main file. Both are generated by the code generation framework which we introduce in chapter 1. In the distributed case, the framework takes the module to node assignment into account and computes a suitable communication schedule. The code generation framework supports concrete communication protocols and hardware platforms via a plug-in mechanism.

- TDL main file (`tdl_main_.c`)

  The so-called main file or node glue code exists *once per node*. It contains initialization code for the runtime system and ensures the periodic invocation of the TDL Machine. In case of a distributed system, it additionally is responsible for the synchronized startup of the system and proper time synchronization between nodes during runtime. The TDL main file must interact tightly with the specific operating system and communication protocol employed on the platform and is therefore highly platform-specific.

- TDL module glue code (`<module>_.c/h`)

  For every TDL module a C file is generated, which contains the modules' drivers, guards, and runtime data structures so that it can be executed by the TDL Machine. The TDL Machine operates mostly on the data structures provided in the module glue code. In case the target operating system does not incorporate a file system and is therefore not capable of handling E-Code files, the module glue code also contains the E-Code represented as C structures. E-Code is stored as an array of structs consisting of an operation code indicating the E-Code instruction and two arguments, which is shown in detail in subsection 3.1.1. For distributed systems, the module glue code also comprises so-called stub modules which act as a remote instance of a module when it is imported by another module on a remote node. Furthermore, the module glue code for distributed systems interfaces with the TDL Comm Layer to transmit port values.

- Other Entities

  - TDL module functionality code (`<module>.c/h`)

    The functionality code exists once per module. It contains the implementation of tasks, guard functions, sensor getters and actuator setters. It can be hand-written C code or also generated code from external tools, e.g. the Real-Time Workshop integrated in MATLAB/Simulink.

  - Hardware drivers

    These are low-level drivers which enable the interaction with the physical environment via sensors and actuators. They are typically mapped to TDL sensors and actuators in the module functionality code or by automatically generated wrapper code located in the module glue code.

  - Communication drivers

    Communication drivers are used to interface with the communication infrastructure which interconnects nodes of a distributed TDL system. Apart from functions to send and receive frames, communication platform-specific plug-ins to the TDL Comm Layer framework also require functions for synchronizing the time base of the protocol to the node time base. In case such functions are not available, which is typically the case with non time-triggered buses, a TDL Comm Layer plug-in must implement time synchronization algorithms itself.

- Operating System

  Operating system functions are typically utilized by the TDL main file in order to ensure the periodic execution of the TDL Machine and to interface with hardware interrupts which can be used as triggers for asynchronous activities.

In section 3.1 we present our implementation of the TDL Machine. Section 3.2 introduces a generic synchronization mechanism for the integration of time-triggered and event-triggered activities in a real-time system. We also describe its application in the TDL Async Handler and the required TDL Machine adaptations. As the last part of the TDL Runtime System, we present the TDL Comm Layer framework in section 3.3. Note that this chapter is solely on platform-independent aspects of the TDL Runtime system. Details about the prototyping hardware and the corresponding platform-specific adaptations can be found in chapter 1.

## 3.1. TDL Machine

The TDL Machine is responsible for interpreting the E-Code of all modules executed on a node. It therefore represents the core of the TDL Runtime System, as it orchestrates the timing and data flow as specified in the TDL modules on a concrete hardware platform. This section describes the implementation of the TDL Machine for our C runtime system, whereas the details of E-Code interpretation were already discussed in section 2.4.

There exists only a single instance of the TDL Machine per node, which handles all modules assigned to this node. For every module the TDL Machine executes drivers, evaluates guards, interprets E-Code, and updates module runtime information. Drivers are used in a TDL runtime system to encapsulate port copying operations and the execution of sensors, actuator and task functionality code.

The TDL Machine is initialized with the TDL modules assigned to a node. For this purpose, the TDL Machine code provides data structures to represent a module's modes, drivers, E-Code and runtime data. These structures are utilized in the module glue code to specify concrete modules and to initialize the TDL Machine. For performance reasons, we only use static data structures. Execution time is critical here, as the TDL Machine is logically executed in zero time.

After initialization, the TDL Machine's so-called *step function* is invoked repeatedly at a fixed interval to execute all module-related actions to be performed at a specific time instant. We call its invocation period the *step period*. It is determined by the code generation framework which also configures the platform's operating system so that the TDL Machine is timely executed. The step period is calculated as the greatest common divisor (GCD) of the periods of all activities which must be performed for all modes of all modules executed on a node. Every such period the TDL Machine advances the individual time for each module and checks whether there is something to do for the currently active mode, i.e. it checks whether the reactivation time set by the last `future` instruction did already pass.

Conceptually, the TDL Machine is not responsible for the actual execution of tasks, but only for ensuring the proper timing of TDL modules by interpreting their E-Code and by executing sensor and actuator code. However, we optionally included a simple dispatcher in our implementation as the time instants when the TDL Machine runs can also be used to execute task functionality code on non-preemptive systems. For that purpose, a dispatch table is generated which contains information on what tasks to execute on these time instants. The dispatcher is activated via the compiler

flag TDL_DISPATCHED, which is also used to alter data types so that the relevant elements are only enabled when required.

The TDL Machine is implemented in the files tdl_machine.c and the corresponding header file tdl_machine.h. Those files include support for distribution and for the execution of asynchronous activities and also the optional non-preemptive task dispatcher. Whether distribution support is activated can be selected with the TDL_DISTRIBUTED compiler flag. If TDL_DISTRIBUTED is not set the TDL Machine is configured to run in stand-alone and therefore single node mode without any communication layer. Otherwise, the TDL Comm Layer is included and used to communicate with other nodes in a distributed system. In the following three sections we will describe the TDL Machine's initialization, its step function and the optional dispatcher in detail.

### 3.1.1. Initialization

The TDL Machine must be initialized at node startup with a list of modules and a step period via the function tdl_machine_init:

```
void tdl_machine_init(tdl_machine_Module** modules,
                      int nofModules,
                      long int stepPeriod);
```

As parameters a pointer to a list of modules, the number of modules and the step period are passed. The step period is the greatest common divisor (GCD) of all periods of all actions the TDL Machine has to perform. It is the time that passes between two invocations of the tdl_machine_step function.

Upon initialization, the module initialization function in the functionality code (<module>_init()) is called for every module. Then the E-Code interpreter function is called for every module in order to execute the initialization section of the E-Code, which is done by executing the code from the first instruction until the first return instruction. After that the program counter is set to the beginning of the start mode.

The list of modules consists of structures of the type tdl_machine_Module which contains all data concerning a module. This includes the E-Code and numerous hook function pointers e.g. for drivers which are called by the TDL Machine:

```
typedef struct
  tdl_machine_ECode *ecodes; //pointer to the module E-Code table
  int nofEcodes; //number of E-codes in the module E-Code table
  tdl_machine_Mode *modes; //pointer to the modes table of the module
  int nofModes; //number of modes in the module
  void (*init)(void); //function pointer to module initialization
  char (*guards)(int); //function pointer to the guards wrapper
  void (*sdrivers)(int); //function pointer to start/stop drivers
                         //wrapper
  void (*drivers)(int);  //function pointer to the drivers wrapper
  tdl_machine_RuntimeData *runtime; //module runtime data
  long int *taskWCETs;  //pointer to list of task WCETs
} tdl_machine_Module;
```

The E-Code of a module is represented by means of the following C structure, containing an operation code and two arguments. Note that we eliminated the need for a third argument in comparison to earlier implementations [7].

```
typedef struct {
  char opcode;
  long int arg1;
  long int arg2;
```

```
} tdl_machine_ECode;
```

TDL modes are stored using another data structure. For the structure of the optional dispatch table see subsection 3.1.3.

```
typedef struct {
  int pcBegin; //E-Code entry point of the mode
  long int period; //mode period
#ifdef TDL_DISPATCHED
  tdl_machine_DispatchEntry *dispatchEntries; //dispatch table entries
#endif
} tdl_machine_Mode;
```

The runtime data of a module (`tdl_machine_RuntimeData`) stores state information during the execution of a module. Upon initialization, all values are set to 0, with the exception of the mode, which is set to the index of the start mode of the module.

```
typedef struct {
  int nextPC; //next program counter
  long int futureTime; //future time relative to mode period
  int repeatCnt; //repeat counter
  int mode; //current mode of the module, also used to set start mode
  long int time; //time relative to beginning of mode
  char eot; //flag to check if emachine encountered a EOT (=NOP(1))
            //instruction (end of termination drivers)
#ifdef TDL_DISPATCHED
  int dispatchTableIndex; //current index of the dispatch table
#endif
  char *tasksActive; //flag for every task; 1 if task is enabled, 0 if
                     //guard evaluates to false
} tdl_machine_RuntimeData;
```

Note that in addition to this runtime data structure, the state of a module also consists of the current values of its ports. Those are declared in the module glue code and are not directly accessed by the TDL Machine but via the corresponding drivers. Ports are initialized by the first E-Code section of a module.

### 3.1.2. Step Function

```
void tdl_machine_step(void);
```

The function `tdl_machine_step` performs all periodic actions of the TDL Machine for all modules on a node. It must be called exactly every step period. The operating system must be configured accordingly, which is done in the TDL main file which is part of the generated glue code. It can for example be implemented via an entry in a dispatch table, a task that is scheduled periodically or directly via programming a timer interrupt.

The step function first checks for every module if its future time is already reached. If that check is positive, it executes a block of E-Code starting at the current position of the program counter until an `eot` or `return` instruction is reached. `eot` stands for end of termination drivers and is represented by a `nop(1)` E-Code instruction. It indicates that all termination drivers for the current TDL Machine step have been executed. It is important that all termination drivers of all modules are called first, as modules might cyclically import output ports from other modules and the termination drivers are responsible for updating these ports.

Subsequently, the modules are iterated again and all modules whose E-Code interpretation had been interrupted by an `eot` instruction are processed until a

`return` instruction is reached. Finally, the individual time of each module is increased by one step period length.

Support for asynchronous activities requires some specific adaptations to the step function, which we describe when introducing the TDL Async Handler in section 3.2.

### 3.1.3. Non-Preemptive Dispatcher

Our C implementation of the TDL Machine includes an optional dispatcher which can be used to execute task functionality code on non-preemptive platforms. As an alternative, tasks dispatching can, for instance, also be handled by the operating system's scheduler. The dispatcher runs as a last step in the TDL Machine step function when it is compiled into the code by setting the `TDL_DISPATCHED` compiler flag. It then sequentially executes the task functionality code of modules without any preemption according to a dispatch table which is generated for every module. An offline scheduler and a given worst-case execution time (WCET) for every task guarantee the time safety of this implementation. A notable limitation is that no task can be executed whose WCET is larger than the step period of the TDL Machine. This limits the ability to have long running time-triggered tasks and tasks with a short period coexisting on a node. Note that as an alternative, long running background tasks can be specified as asynchronous tasks.

The following structure stores dispatch table entries by means of an array in the mode structure (see 3.1.1), meaning there exists one such table for every mode of a module. It contains a task ID which is used to execute the appropriate start driver, a stop driver ID which is executed upon task termination, and a time instant which is relative to the mode period and indicates the time the task is scheduled.

```
#ifdef TDL_DISPATCHED
typedef struct {
  int task; //task id (equals start driver id)
  int stopDriver; //stop driver id or -1 if none
  long int time;  //time when task is scheduled
} tdl_machine_DispatchEntry;
#endif
```

The TDL Machine maintains an individual mode time (relative to the beginning of a mode) per module, which is increased by one step period duration in every TDL Machine step. For every active mode, the dispatcher executes all tasks in the table which are scheduled within the interval starting at the current time and ending at the current time plus the step period.

In case of a distributed system, the dispatcher also takes care of sending task output ports via the communication bus. It does so by calling the TDL Comm Layer function `tdl_comm_sendFramesWithinInterval (long int interval)`, which sends all frames within a given interval starting at the current instant of time. It is called after every task execution with an interval equal to the WCET of the previously executed task, so that frames scheduled to be sent while the dispatcher is executing tasks are processed correctly. After all dispatch tables have been processed, the send function is called again with the interval that is still left in the step period in order to send all frames remaining in this step period. `tdl_comm_sendFramesWithinInterval` is the only TDL Comm Layer function that is called by the TDL Machine. The calls are activated during compilation by the `TDL_DISTRIBUTED` flag using C macros.

## 3.2. Synchronization Mechanism for Asynchronous Activities

This section presents a generic synchronization mechanism for the integration of time-triggered (alias *synchronous*) and event-triggered (alias *asynchronous*) activities. If such activities exchange information among each other, the data flow must be synchronized such that reading unfinished output data is avoided. We present a *lock-free* solution for these synchronization issues that is based exclusively on memory load and store operations and therefore can be implemented efficiently on embedded systems, as these operations are provided by every CPU in hardware. Consequently, our approach does not need any operating system support such as monitors [23] or semaphores [24] and thereby avoids dynamic memory operations and the danger of deadlocks and priority inversions. There is also no need for switching off interrupts and the solution also works in a shared-memory multiprocessor system where the time-triggered and event-triggered activities are performed on separate CPUs. Our approach keeps the impact of event-triggered activities on the timing of time-triggered activities as low as possible. For more information on non-blocking synchronization techniques refer to [25] and [26].

We already motivated and described the integration of asynchronous activities in the TDL language in section 2.5. Throughout this section, we show the application of our synchronization algorithm in the context of the TDL Async Handler, which is part of the TDL Runtime System and is implemented in the files `tdl_async.c` and `tdl_async.h`. Furthermore, we describe required extensions to the TDL Machine. How outputs of asynchronous activities are communicated to other nodes of a distributed system is presented in section 3.3, which is on the TDL Comm Layer framework. Aspects which are specific to the target platform, such as the realization of the background execution of asynchronous activities and the integration of hardware interrupts, are discussed in chapter 1.

It is important to note that our lock-free synchronization approach is not focused entirely on TDL, but rather uses the TDL language as an example for a language supporting the integration of synchronous and asynchronous activities. It can also be applied to other time-triggered systems that need to be extended with asynchronous activities. A generic description of the approach has been published in [27].

### 3.2.1. Asynchronous Activities

We assume that time-triggered activities have the highest priority in a dependable real-time system. The runtime system executes a pre-computed schedule and reads inputs and writes outputs at well-defined time instants, which are synchronized with a global time base such as the clock of a time-triggered bus system. There is always a distinguished time base which drives all time-triggered activities and that is why they are also called synchronous activities.

Asynchronous activities must not interfere with the timing properties of synchronous activities. This is achieved by running asynchronous activities in a thread with lower priority than synchronous activities. However, things get more complicated when synchronization of the data flow is involved, as we describe below.

TDL supports three kinds of synchronous activities. Task invocations and actuator updates also give sense when triggered asynchronously and should therefore be supported. Mode switches however affect the time-triggered operation of a module and are therefore not supported as asynchronous activities.

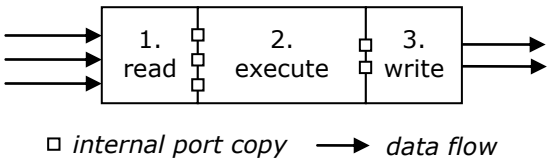


**Figure 11. Assumed task model**

An asynchronous task invocation consists of (1) reading input data (also called input ports), (2) execution of the task's body, and (3) writing of output data (also called output ports). There may be other asynchronous activities as well (e.g. setting of actuator ports) but with respect to synchronization issues, they do not introduce new problems because they can be seen as a special case of a task invocation. Figure 11 shows the task model that we assume.

The execution of a task's body is independent of the environment if input reading and output writing are separated from the implementation. Therefore we assume that internal copies of all input and output ports are maintained by the system. The task's body operates exclusively on these internal port copies.

Reading of input data may involve a sequence of memory copy operations that could be preempted by a hardware interrupt or by a time-triggered operation, which has higher priority. Therefore we need to synchronize input data reading with the rest of the system such that all input ports are read atomically.

Like input data reading, writing of output data is a sequence of memory copy operations that could be preempted by a hardware interrupt or by a time-triggered operation. It needs to be synchronized with the rest of the system such that all output ports are updated atomically.

### Triggers for asynchronous activities

Asynchronous activities may be triggered by different events. We have identified the following three kinds of trigger events, which are consequently supported in our extension of TDL:

- Hardware interrupt

  A (non-maskable) hardware interrupt has the highest priority in the system and may thus even interrupt synchronous activities. We must therefore take care that the impact of hardware interrupts on the timing of synchronous activities is minimized. Hardware interrupts may be used e.g. for connecting the system with asynchronous input devices.

- Asynchronous timer

  A periodic or a single-shot asynchronous timer may be used as a trigger. Such a timer is independent from the timer that drives the synchronous activities because it introduces its own time base. An asynchronous timer may for example be used as a watchdog for monitoring the execution of the time-triggered operations.

- Port update

  Updating an output port may be considered an event that triggers an asynchronous activity. We assume that both a synchronous and an asynchronous port update may be used as a trigger event. In case of a

synchronous port update, i.e. a port update performed in a time-triggered activity, we must take care that the impact on the timing of the synchronous activities is minimized. Port update events may e.g. be used for limit monitoring or for change notifications.

**Semantics of asynchronous activities**

Obviously, the triggering of an asynchronous activity must be decoupled from its execution. In addition, reading input ports for an asynchronous activity must be done at the time of execution, not at the time of triggering. Thereby we move as much work as possible into the asynchronous part and minimize the impact of trigger events on the timing of synchronous activities, which is particularly important for hardware interrupts and synchronous port updates.

If multiple different asynchronous activities are triggered, the question arises whether they should be executed in parallel or sequentially in a single thread. We opted for the sequential case because (1) on some embedded systems there is no support for preemptive task scheduling and (2) because data flow synchronization is simplified when only one asynchronous activity is executed at a time. In practice, we expect this not to be a severe restriction because time critical tasks will be placed in the synchronous part anyway.

We assume that asynchronous activities that are registered for execution may have different priorities assigned. The set of registered events thus forms a priority queue where the next activity to be processed is the one with the highest priority.

If one and the same asynchronous activity is triggered multiple times before its execution, the question arises if it should be executed only once or multiple times, i.e. once per trigger event. We opted for executing it only once because this avoids the danger of creating an arbitrary large backlog of pending activities at runtime if the CPU cannot handle the workload. In addition, this decision also simplifies the mechanism for registering trigger events as will be shown later.

The following list summarizes our design decisions which are key to a simple and efficient synchronization solution:

- Triggering of an asynchronous activity is decoupled from its execution.
- Reading input ports for an asynchronous activity is done at the time of execution, not at the time of triggering.
- Asynchronous activities are executed sequentially.
- The execution order of asynchronous activities is based on priorities.
- If one and the same asynchronous activity is triggered multiple times before its execution, it is executed only once.

### 3.2.2. Threading and Synchronization

Figure 12 outlines the threads involved including their priority and the critical regions. The time-triggered activities are represented by the TDL Machine thread. This thread may need further internal threads but we assume that all synchronization issues are concentrated in a single thread that coordinates the time-triggered activities. It should also be noted that an asynchronous timer thread could also run at a lower priority as long as it is higher than the priority of the asynchronous activities.

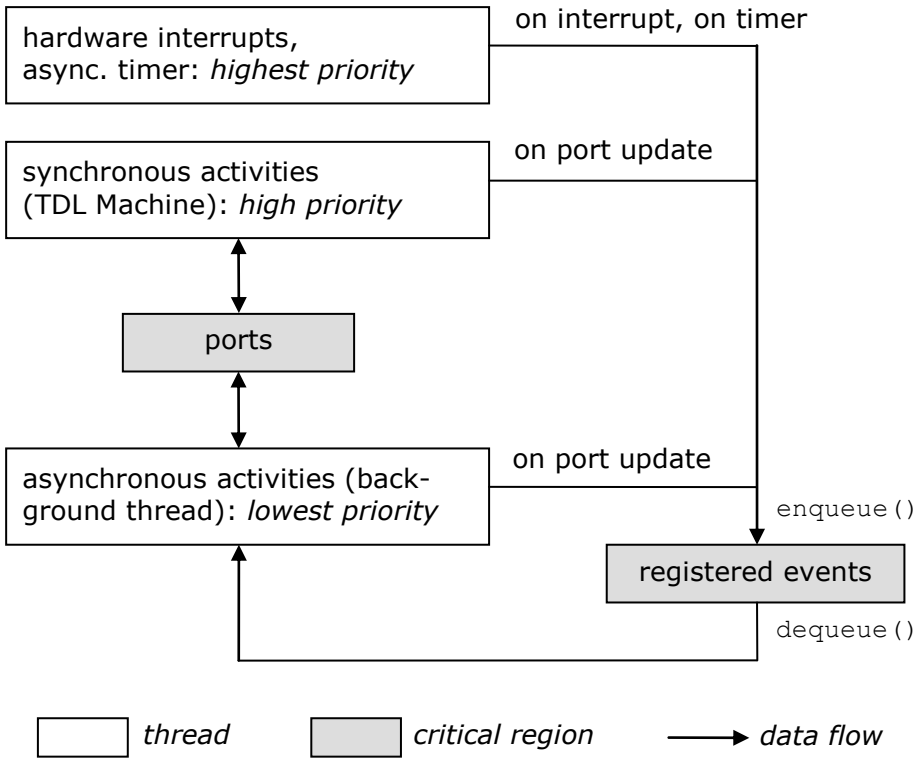


**Figure 12. Threads and critical regions**

The following situations that need synchronization can be identified and will be described below in more details: (1) Access to the priority queue of registered events. (2) Reading the input ports for an asynchronous activity. This must not be interrupted by the TDL Machine. (3) Updating the output ports of an asynchronous activity. This must be finished before the TDL Machine uses the ports.

### The Priority Queue of Registered Events

As mentioned before, asynchronous events are not executed immediately when the associated trigger fires but need to be queued for later execution by the background thread. Since asynchronous events may be associated with a priority, we need a data structure that allows us to register an event and to remove the event with the highest priority. Such a data structure is commonly referred to as a *priority queue*. It provides two operations `enqueue` and `dequeue`, which insert and remove an entry with the property that the element being removed has the highest priority. A number of algorithms exists for implementing priority queues with logarithmic behavior of the `enqueue` and `dequeue` operation. However, in our case it is more important to minimize the run time of `enqueue` in order to minimize its impact on the timing of synchronous activities.

Elements are enqueued when an asynchronous event occurs and the event is not yet in the queue. As mentioned earlier, an event can be a hardware interrupt, an asynchronous timer event, or a port update event. Port updates may origin from an asynchronous task or from a synchronous task that is executed by the TDL Machine. `enqueue` will never be preempted by `dequeue`, however, `enqueue` may be preempted by another `enqueue` operation.

Elements are dequeued by the single background thread that executes asynchronous activities. This thread may be preempted by interrupts and by the TDL Machine. Thus, `dequeue` may be preempted by `enqueue` operations.

| Trigger event | Priority | Pending |
|---|---|---|
| 0 | 0 | true |
| 1 | 2 | false |
| 2 | 2 | false |
| 3 | 1 | true |

**Table 2. Array representation of trigger events**

As shown in the example in Table 2, we chose an array representation of the triggerable events because this is both thread safe and provides for a fast and constant time `enqueue` operation. We use a Boolean flag per event that signals if an event is pending. The flag is cleared when an event is dequeued. From that time on it may be set again when the associated trigger fires. The flag remains set when the same trigger fires again while the flag is already set. The thread-safe `enqueue` operation boils down to a single assignment statement and the `dequeue` operation becomes a linear search for the event with the highest priority over all pending events. Registering an event from a non-maskable interrupt or from a synchronous port update thereby has only a negligible effect on the timing behavior of synchronous activities. The linear search in the background thread is expected to be acceptable for small to medium numbers of asynchronous events (< 100), which should cover all situations that appear in practice. We chose this priority queue implementation to achieve the fastest possible run time of the `enqueue` operation, which is executed inside time-critical code, and because the performance of the `dequeue` operation is secondary as it is executed inside the background thread.

It should be noted that the array representation of the priority queue does not impose any restriction on the number of events the system can handle. There is one array element for every trigger and the number of triggers is known statically. Thus, the array can always be defined with the appropriate size.

The TDL Async Handler implements the priority queue by means of the following C structure:

```
typedef struct {
  char pending; //flag indicating pending async sequence
  int priority; //priority of the async sequence
} tdl_async_AsyncSequence;
```

Upon initialization, a pointer to the array of asynchronous sequences and the number of entries are passed to the TDL Async Handler:

```
void tdl_async_init(tdl_async_AsyncSequence* asyncs, int nofAsyncs);
```

The initialization function is called in the TDL main file, where also the array of asynchronous sequences, i.e. the priority queue, is constructed and the priority for every sequence is set. A sample initialization with the data from Table 2 looks like this:

```
static tdl_async_AsyncSequence asyncs[] = {
```

```
    {0, 0}, //{pending, priority}
    {0, 2}, //{pending, priority}
    {0, 2}, //{pending, priority}
    {0, 1}, //{pending, priority}
  };
  tdl_async_init(asyncs, 4);
```

Apart from the initialization function, the TDL Async Handler provides two functions for `enqueue` and `dequeue`:

```
void tdl_async_enqueue(int index);
int tdl_async_dequeue(void);
```

The `dequeue` operation returns the index of the pending event with the highest priority and removes it from the priority queue. It returns `-1` when no events are pending.

The background thread for executing asynchronous operations could for example be a simple infinite loop that runs with lower priority than the TDL Machine thread and is defined in the TDL main file. For a particular target platform there may be some refinements with respect to the CPU load, which is increased to 100% by permanently polling the event queue.

```
while(1) {
  int next = tdl_async_dequeue();
  if (next >= 0) {
    executeAsyncSequence(next);
  }
}
```

The procedure `executeAsyncSequence` is supposed to execute the asynchronous activity identified by `next`. Within its implementation there will be synchronization issues with respect to reading input ports and writing output ports as described below. In the following, we will show all aspects of our implementation relevant to these synchronization issues.

### Reading the Input Ports for an Asynchronous Task

While performing asynchronous reading of input ports the following situation may arise: An asynchronous input port reading involving multiple input ports (or at least multiple memory load operations) has been started. The first port has been copied. The second port has not yet been copied but the TDL Machine preempts the background thread and updates the source ports. When the background thread continues it would read the next port, which has a newer value than the first port. Moreover, this situation may in principle occur multiple times when the TDL Machine preempts the background thread after the second port has been read, etc. We have to make sure that reading all of the input ports is not preempted by the TDL Machine. Since asynchronous activities don't preempt each other, we know that there can only be one such asynchronous input port reading that is being preempted. Therefore we can introduce a global flag that is set by the TDL Machine in order to indicate to the background thread that it has been preempted. The background thread then has to repeat its reading until all of the ports are read without any preemption. The following code fragments outline our C implementation.

Asynchronous port reading within `executeAsyncSequence` uses a loop in order to wait for a situation where input port reading is not preempted by the TDL Machine. Therefore, our solution does not qualify as a wait-free non-blocking algorithm [25]. It should be noted, however, that (1) starvation cannot occur in the TDL Machine and (2) in practice it does also not occur in the background thread because even in the

unlikely case that the TDL Machine's schedule reserves 100% of the CPU, this refers to the worst case execution time, which typically will not always be required.

```
do {
  tdl_machine_executed = 0;
  //copy input ports
  ...
} while (tdl_machine_executed);
```

The relevant TDL Machine code, which is placed in the central procedure of the TDL Machine (`tdl_machine_step`) looks like this:

```
void tdl_machine_step(void) {
  tdl_machine_executed = 1;
  //perform operations for this time instant
  ...
}
```

Consequently, the flag is added as an external variable in the TDL Machine header file `tdl_machine.h`, so that it is accessible by the background thread:

```
extern char tdl_machine_executed;
```

### Updating the Output Ports of an Asynchronous Task

In the case of asynchronous output port updates the following situation may arise: An asynchronous output port update involving multiple output ports (or at least multiple memory store operations) has been started. The first port has been copied. The second port is not yet copied but the TDL Machine preempts the background thread and reads both output ports. Now one port is updated but the second is not. Since this interruption cannot be avoided, we must find a way for proper synchronization.

Since we assumed earlier that updating the output ports is separated from the implementation of a task, we can encapsulate the output port update operations of a task in a helper procedure that we call the task's *termination driver*. Since asynchronous activities don't preempt each other, we know that there can only be one such termination driver being preempted and it suffices to make that very instance available to the TDL Machine by means of a global variable. Whenever the TDL Machine performs its next step, it checks first if a termination driver has been interrupted. If so, it simply re-executes this driver! This means that the driver may be executed twice, once by the background thread and once by the TDL Machine. This is only possible if the driver is *idempotent* and *reentrant*, i.e. its preemption and repeated execution does not change its result. Fortunately, termination drivers have exactly this property because they do nothing but memory copy operations and the source values are not modified between the repeated driver executions. The source values are the internally available results of the most recent invocation of this asynchronous task and only a new task invocation can change them. Such a task invocation, however, will not happen because the background thread executes all asynchronous activities sequentially.

It should be noted that the property of idempotency does not hold for copying input ports as discussed in the previous subsection because a preemption by the TDL Machine may alter the value of a source port that has already been copied. This means that we need two ways of synchronization for the two cases.

It should also be noted that setting the termination driver identity must be an atomic memory store operation. If storing e.g. a 32 bit integer is not atomic on a 16-bit CPU, an additional Boolean flag can be used for indicating to the TDL Machine that a driver has been assigned. This flag must be set after the assignment of the driver's

identity. If this initial sequence of assignments is preempted, the TDL Machine will not re-execute the driver and that is correct because the driver has not yet started any memory copy operations.

The following C code outlines the implementation of asynchronous task termination drivers and the corresponding code in the TDL Machine. Setting, testing and clearing the driver identity may vary between target platforms. Our implementation uses a function pointer to the drivers of a module (`tdl_machine_asyncDrivers`), an ID (`tdl_machine_asyncDriverID`) to identify a specific termination driver and a flag (`tdl_machine_asyncPending`) indicating a pending termination driver. This requires the following external variables in `tdl_machine.h`:

```
extern char tdl_machine_asyncPending;
extern int tdl_machine_asyncDriverID;
extern void (*tdl_machine_asyncDrivers)(int);
```

An example task termination driver of a specific TDL module with index `T` may look like this:

```
void module_drivers(int id) {
  switch (id) {
    ...
    case T: //termination driver for async task T
      tdl_machine_asyncDriverID = T;
      tdl_machine_asyncDrivers = module_drivers;
      tdl_machine_asyncPending = 1;
      //perform memory copy operations
      ...
      tdl_machine_asyncPending = 0;
      break;
    ...
  }
}
```

The relevant TDL Machine code tests if a driver is pending and executes it if necessary. Including the `tdl_machine_executed` flag introduced previously the code looks like this:

```
void tdl_machine_step(void) {
  tdl_machine_executed = 1;
  if (tdl_machine_asyncPending) {
    tdl_machine_asyncDrivers(tdl_machine_asyncDriverID);
  }
  //perform operations for this time instant
  ...
}
```

It suffices to clear the flag indication a pending termination driver at the end of the termination driver itself. There is no need to do it after `tdl_machine_asyncDrivers()` in `tdl_machine_step` because the driver's re-execution will clear it anyway.

The resulting runtime overhead for supporting asynchronous operations in the TDL Machine is the assignment of the `tdl_machine_executed` flag and the test for the existence of a preempted asynchronous task termination driver, which is acceptable because this happens only once per TDL Machine step. In case of preempting such a driver the time for re-execution must be added. When a port update trigger is used, then the `enqueue` operation is also a small constant time overhead that affects the TDL Machine. There is no other runtime overhead for integration of event-triggered activities in the TDL Machine.

### 3.2.3. Quantitative Analysis of Runtime Behavior

In order to show the feasibility of the proposed synchronization mechanism, we analyzed its runtime behavior on four different platforms. The measurements were conducted using a CPU timer to count clock cycles and by setting a digital output to high during an operation and measuring the duration with a digital oscilloscope. Table 3 shows the results for various operations. The platform named *MicroAutoBox* uses a PowerPC 750FX CPU running at 800 MHz and the Microtec C compiler version 3.2 with optimization level 5. The platform runs the dSPACE Real-Time Kernel as its operating system. The *SHARC* platform uses an Analog Devices SHARC ADSP-21262 CPU running at 200 MHz and the VisualDSP++ C compiler version 5.0 with maximum optimization level. The platform named *ARM* uses an ARM7 TDMI CPU running at 80 MHz and the GNU C compiler with optimization level 2 and runs without an operating system. The platform named *RENESAS* uses a Renesas M32C/85 CPU running at 24 MHz and the GNU C compiler version 4.1 with optimization level 3. The platform runs the Application Execution System (AES) provided by DECOMSYS and executes the programs from read-only memory, which slows down the execution. This system does not support external interrupts for user level programs.

| **Platform (MHz)** | **Interrupt** | **Port Update** | **dequeue N** |
|---|---|---|---|
| MicroAutoBox (800) | 420 | 8 | 11 * N + 60 |
| SHARC (200) | 1030 | 72 | 30 * N + 110 |
| ARM (80) | 700 | 200 | 287 * N + 500 |
| RENESAS (24) | N.A. | 1200 | 790 * N + 2500 |

**Table 3. Measurement results [nanoseconds]**

The column *Interrupt* shows the time needed for an external hardware interrupt trigger, which includes the interrupt handling overhead and the `enqueue` operation. The column *Port Update* shows the time needed for a synchronous port update trigger, which consists only of the `enqueue` operation. The column *dequeue N* shows the time needed for the search for the next event to be processed as a linear function of the array size *N*. All timings are given in nanoseconds.

The values shown in the columns *Interrupt* and *Port Update* are critical for the timely execution of synchronous operations as they impose an overhead that may affect the TDL Machine. Even on the slowest platform the required time is only slightly above one microsecond. In comparison with the *ARM* platform, the *Interrupt* time for *MicroAutoBox* shows that the operating system introduces a significant overhead.

The values in the column *dequeue N* only affect the background thread and are not visible to the TDL Machine. On the slowest platform a time of 81.5 microseconds results for $N = 100$, which means that response times in the range of milliseconds can easily be achieved for asynchronous operations. With regard to the CPU clock speed, the SHARC platform has the best performance for the dequeue operation. This is due to the compiler which efficiently optimizes loops for parallel execution.

### 3.2.4. Related Work

The xGiotto language [4] also aims at the integration of time-triggered and event-triggered activities. Its compiler is supposed to perform a static check for the absence of race conditions, which occur when a port is updated multiple times at the

same logical time instant. Due to the specific design of xGiotto, a precise check is possible, but not in polynomial time. Therefore, only a conservative check is done in the compiler. We do not need such a check at all as we defined appropriate semantics for event-triggered activities and use appropriate synchronization mechanisms for their integration into a time-triggered system. Furthermore, the schedulability analysis is also expensive in xGiotto as it involves solving a two-player safety game. For TDL programs the check is only slightly more complicated (due to slot selection) than for Giotto, for which it can be done by a simple utilization test in polynomial time [28]. Note that asynchronous activities are not taken into account in this test, and need not be taken into account, as TDL provides no guarantees for their execution.

RT-Linux [29] is an extension of the Linux operation system which adds a high priority real-time kernel task and runs a conventional Linux kernel as a low priority task. Its interrupt handling mechanism is similar to what we propose for the event queue as all interrupts are initially handled by the real-time kernel and are passed to a Linux task only when there are no real-time tasks to be run. Our approach is analogous, as the only immediate reaction to an interrupt is its registration in the priority queue so that it can be processed later when no time-triggered activity is executed.

In [30] a non-blocking write (NBW) protocol is presented. The writer is executed by a separate processor and is not blocked. It updates a concurrency control field (CCF) which indicates whether it currently writes data to a shared variable. The reader uses the CCF to loop until no write operation is executed while it reads from the shared data structure. This relates closely to our synchronization strategy for reading input ports for an asynchronous activity. In our case the writer would be the TDL Machine which is not blocked.

A comprehensive overview of the field of non-blocking synchronization can be found in [26]. Among other techniques, it also describes a so-called roll-forward synchronization approach by means of a helper function, which looks similar to the one we used for synchronizing output port writing.

## 3.3. TDL Comm Layer Framework

The TDL Comm Layer framework is responsible for the transparent distribution of port values across the communication system of a distributed TDL system. It provides functions for time synchronization, for reading and writing messages to frames and for sending and receiving frames on a communication bus. The Comm Layer is divided into a generic part, which is implemented in the file `tdl_comm.c` and the corresponding header file `tdl_comm.h`, and communication platform-specific plug-ins, implemented in the files `tdl_comm_<platform>.c` and `tdl_comm_<platform>.h`. Note that the platform-specific functionality is not only specific to a concrete communication bus, e.g. FlexRay, but also to a concrete hardware platform, as typically different communication drivers and controllers are used.

As already laid out in section 2.3 on transparent distribution, the basic idea for communication in LET-based systems such as TDL is to transfer values inside the LET of the sender task. The TDL Runtime System implements this by means of a so-called *stop driver*, which is executed after task termination and interfaces with the TDL Comm Layer. On the receiver node, we use the concept of a *stub module*. It acts as a local representation of a module which is executed on another node. Stub modules can be seen as primitive modules which only consist of ports and a simple E-Code which solely executes termination drivers. The TDL Comm Layer ensures that

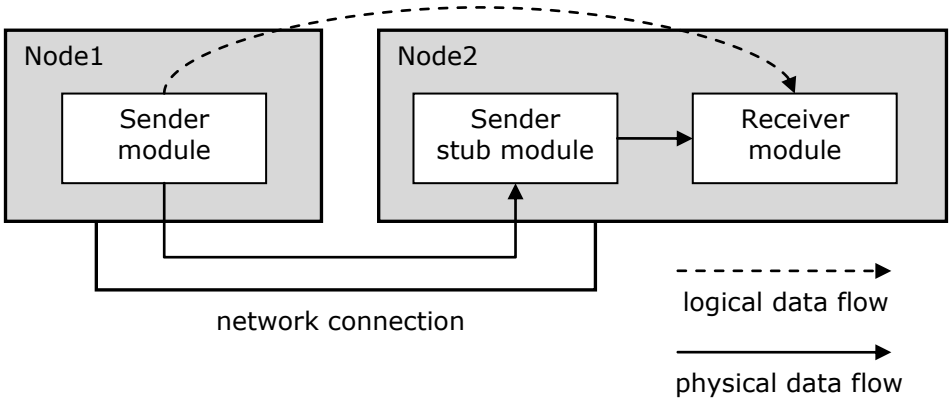


**Figure 13. Stub module data flow**

transmitted port values are copied to internal task output ports of the stub module. Afterwards, the appropriate termination drivers are executed so that the public task output ports are updated and then made available to other modules according to the LET semantics, i.e. at the end of the LET of the task which produced the ports on the remote node. Figure 13 illustrates how a stub module handles the data flow between the *Sender* and *Receiver* modules from the producer-consumer example we introduced in section 2.2.

To illustrate how the TDL Comm Layer framework handles the transmission of port values, we take yet another look at the producer-consumer example. In the following, we describe step by step what happens when the public output port *o* of the *Sender* module is transferred from *Node1* to the *Receiver* module on *Node2*. Figure 14 is a zoomed version of Figure 4, including annotations of when the steps listed below occur. The list presents an overview of the TDL Comm Layer functions involved, which we describe in detail throughout this section.

1) The task *produce* that produces output port *o* is executed by the TDL Runtime System on *Node1*, on which the module *Sender* is executed. Its physical execution time is indicated by the black box in Figure 14.

2) At the end of task execution, the task stores its output in an internal task output port, which we call $o_{internal}$. Upon task termination, the stop driver is executed. It puts all output ports of a task into a TDL frame buffer, together with a tag identifying the module, mode, and task invocation it came from. In our case, the internal port $o_{internal}$ and a corresponding tag is written to the frame buffer via the functions `tdl_comm_putTag` and `tdl_comm_putInt`.

3) The buffered frame is sent via the communication network by using the TDL Comm Layer plug-in function `tdl_comm_sendBuffer`. This is triggered by the TDL Machine dispatcher (see section 3.1.3) on *Node1* by calling `tdl_comm_sendFramesWithinInterval`, which triggers the sending of all frames scheduled within a specific time interval. The function must be is called in time, i.e. before the corresponding frame is scheduled to be sent on the network.

4) The function `tdl_comm_sendBuffer` calls the communication platform-specific function which actually sends the content of the frame buffer via the communication network. As a result, the frame is transferred via the network.

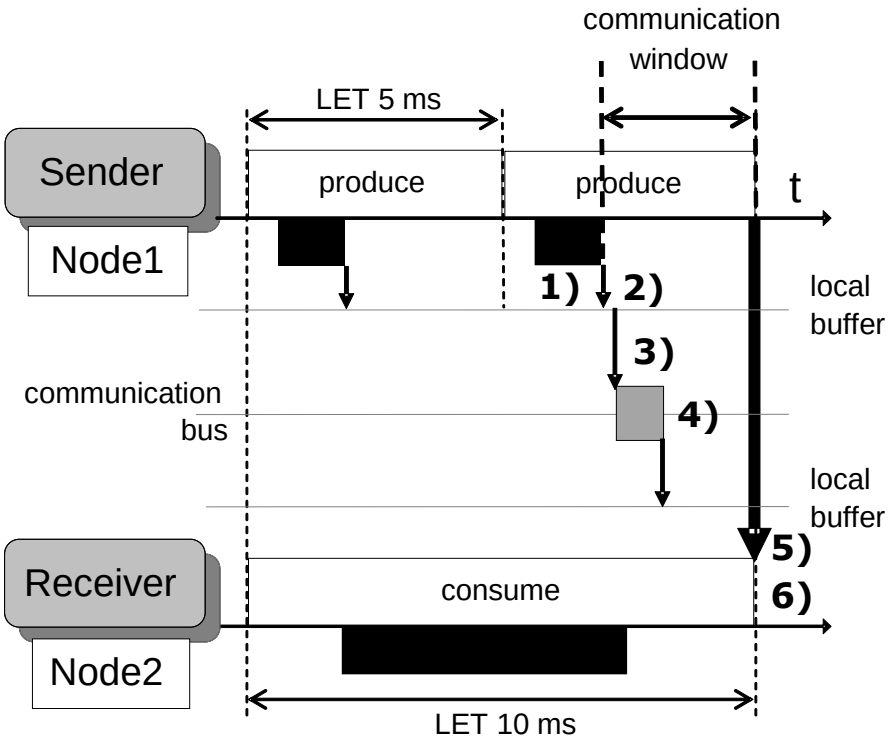


**Figure 14. Transmission of a port value via the network**

5) On *Node2* the frame is received by `tdl_comm_receiveFrames` right before another step of the TDL Machine is executed. This function in turn calls the platform-specific function `tdl_comm_receiveBuffer`, which fills the corresponding buffer with data received from the communication bus. Upon reception, `tdl_comm_receiveFrames` calls the message decoding function (`decodeMessage`, part of the generated glue code in the TDL main file) with the tag and the frame as parameters. The decoding function writes all ports contained in a message to the internal ports of the stub module. In our example, the transmitted port value is written to port $o_{internal}$ of the *Sender* stub module by using `tdl_comm_getInt`.

6) The TDL Machine on *Node2* executes the termination driver in the E-Code of the *Sender* stub module, which copies $o_{internal}$ to the corresponding public output port *o*. This happens at the same point in time as the execution of the termination driver on *Node1*. As a consequence, port *o* is available to other modules at the end of task *produce's* LET, regardless of where the modules are located. This fact exhibits the notion of transparent distribution.

Note that our implementation requires the TDL Machine to execute the stub modules with the same period as the original module. On the node that executes the stub module, this might shorten the step period of the TDL Machine, which is calculated as the greatest common divisor (GCD) of all periods of all actions the TDL Machine has to perform. In case of using the non-preemptive scheduler described in section 3.1.3, this results in a tighter constraint for the maximum worst-case execution time (WCET) of tasks on this node, as the WCET must not exceed the TDL Machine's step period to achieve a schedulable system.

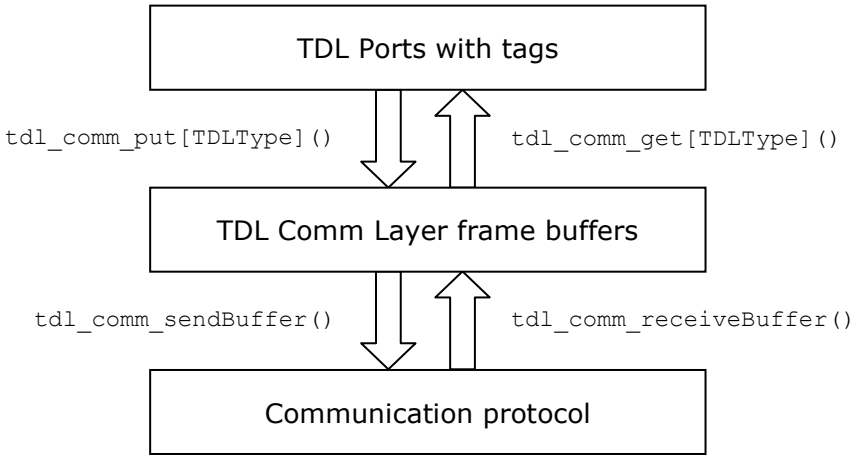


**Figure 15. TDL Comm Layer frame buffers**

### Communication Buffers

To increase the portability of the TDL Comm Layer framework, we introduced an abstraction which uses individual buffers for every communication frame. The generic part of the TDL Comm Layer packs and unpacks TDL ports to those buffers. This is done by adding a message tag to every set of ports originating from a task invocation so that the receiver can identify by which task in which mode and by which module it was produced. The message tag is part of a mechanism we call *dynamic multiplexing*. It allows the creation of a static schedule for TDL systems whose modules are able to change modes dynamically and independently (see section 4.3.1). The transmission and reception of the communication buffers via a concrete communication protocol is handled by platform-specific plug-ins to the framework. Figure 15 illustrates how the communication buffers are used along with the names of the corresponding TDL Comm Layer functions used to access them.

The C implementation of the communication buffers consists of an array of pointers to character arrays of variable lengths, which conserves space by using only the required number of bytes for each buffer. The buffers are initialized in the TDL main file and are therefore declared as external.

```
extern tdl_char* tdl_comm_buffers[];
```

## 3.3.1. Initialization

The TDL Comm Layer framework is initialized by the following function:

```
void tdl_comm_init(tdl_comm_Config* config);
```

The `config` struct contains the following data structures:

```
typedef struct {
  long int stepPeriod; //step period of the node
  long int busPeriod; //bus period of the cluster
  tdl_comm_Frame* frames; //array of frames
  int nofFrames; //number of frames
  char tagSize; //size of the message tag
  tdl_comm_FrameEntry* frameSendEntries; //pointer to array of frame
                                         //send entries
```

```
    tdl_comm_FrameEntry* frameReceiveEntries; //pointer to array of
                                              //frame receive entries
    void (*decodeMessage)(int, tdl_comm_Frame); //function pointer to
                                                //message decoder funct.
} tdl_comm_Config;
```

The initialization function keeps a local copy of the configuration pointer and assigns the correct tag function according to the tag size in the configuration structure, which can either be 1 byte or 2 bytes.

A TDL frame is represented by a struct containing the index of the buffer for the frame data, the size of the frame and the current position required for writing messages to a frame and reading data from a frame:

```
typedef struct {
  int bufferIndex; //index of the buffer of the frame
  int tdlFrameSize; //frame size in bytes
  int position; //current position in the frame buffer
} tdl_comm_FrameStruct;

typedef tdl_comm_FrameStruct* tdl_comm_Frame;
```

A frame entry, used for both the list of sent frames and the list of received frames, contains a frame index and a time when the frame is sent:

```
typedef struct {
  int frame; //frame index
  long int time;  //latest time when frame must be sent
} tdl_comm_FrameEntry;
```

The message decoder function `decodeMessage` has a tag as first argument and a TDL frame as second argument. It handles the content of a frame according to the tag provided, i.e. it updates the internal ports of the corresponding stub module and also sets the mode of the stub module.

### 3.3.2. Frame Handling

This section describes the relevant function for frame access, packing and unpacking of messages to frames and transmission and reception of frames.

#### Frame Access

```
tdl_comm_Frame tdl_comm_getFrame(int index);
```

`tdl_comm_getFrame` is used to obtain a reference to a frame using its index. This function is needed for example because `tdl_comm_FrameEntry` only contains frame indexes but not references directly. Also other data structures in the runtime system or the generated glue code refer to frames using the frame index.

#### Put and Get of TDL types

```
void tdl_comm_put<TDLType>(tdl_comm_Frame frame, tdl_<TDLType> data);
void tdl_comm_get<TDLType>(tdl_comm_Frame frame, tdl_<TDLType>* data);
```

For every TDL type, which are `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, and `double`, there are corresponding put and get methods. The data is read or written to the current position of the frame and subsequently the position is increased by the size of the data type. With the get functions a pointer is passed indicating where to store the obtained data.

The endianness of the system is tested upon initialization at runtime and is then taken into account when packing messages into frames, so that the content of frames sent over the network is always in big endian form. This ensures that systems can communicate regardless of their endianness.

In order to support reading and writing of structured types, the generated module glue code contains code that reads and writes those types by breaking them up into primitive TDL types and calling the appropriate sequence of read and write operations.

#### Tag Handling

```
void (*tdl_comm_putTag)(tdl_comm_Frame frame, int tag);
```

`tdl_comm_putTag` writes a tag to the current position of the frame passed as argument. The function is actually a function pointer which is set to the correct function according to the tag size specified upon initialization.

There also exists a corresponding `getTag` function, but it is not visible outside the TDL Comm Layer as it is only called internally by `tdl_comm_receiveFrames`.

#### Sending and Receiving Frames

```
void tdl_comm_sendFramesWithinInterval(long int interval);
```

`tdl_comm_sendFramesWithinInterval` sends all frames contained in the `frameSendEntries` list that are within the interval passed. The function maintains a current time relative to the start of the bus period. This time is increased by the passed interval and reset when the end of the bus period is reached. The function is called during `tdl_machine_step` (see 3.1.2).

```
void tdl_comm_receiveFrames(void);
```

`tdl_comm_receiveFrames` receives all frames within one step period according to the `frameReceiveEntries` list. It is called just before the invocation of the TDL Machine. This function calls the `decodeMessage` function passed during initialization, which stores all ports contained in the message in the appropriate internal ports of the stub modules.

### 3.3.3. Communication between Asynchronous Activities

When asynchronous tasks provide output ports to other synchronous or asynchronous activities located on another node of a distributed system, these ports must be communicated via a communication network. We call the network frames carrying those ports asynchronous frames. Note that if asynchronous activities use input ports provided by synchronous tasks, no asynchronous frames are necessary. In such a case, communication is done within the LET of the task which updates the corresponding port in the same way as when two synchronous tasks communicate with each other.

In analogy to handling the execution of asynchronous activities in a background thread, asynchronous frames must be sent in a way so that they do not interfere with synchronous frames, i.e. data sent by synchronous activities. Depending on the communication protocol used, this can be done by configuring them as low priority frames (typically done when using event-triggered protocol which often support priorities such as CAN) or by assigning them a designated section in the communication cycle (typically done when using time-triggered protocols such as FlexRay or TTEthernet).

The notion of transparent distribution *does not apply* to the parts of a TDL system involving asynchronous activities. Other than for synchronous activities, it is not guaranteed that asynchronous updates for ports and actuators are performed at the same point in time throughout a distributed system. Asynchronous ports are immediately available to modules mapped to the same node but only after network transmission has finished on remote nodes. Consequently, the fact that distribution might alter parts of the behavior of the system must be taken into consideration at design time.

There are no specific TDL Comm Layer functions for asynchronous frames. They are handled with the same functions as synchronous frames, with the exception that they are not included in the `tdl_comm_FrameEntry` frame lists as their transmission time is only known at runtime. Consequently, the functions which rely on these lists, which are `tdl_comm_sendFramesWithinInterval` and `tdl_comm_receiveFrames`, cannot be used. Instead, this functionality is implemented directly in the generated glue code by using the put and get functions to interface with the frame buffers and the functions to send and receive the buffers. Note that as there are separate frame data structures for each synchronous and asynchronous frame, the sharing of the put and get functions does not create any data synchronization issues.

As an example, here is how an asynchronous frame is sent in the start driver of an asynchronous task invocation. First, the driver executes the task functionality code and then it puts the task's internal output port into a frame buffer which is finally sent.

```
case 0: //start driver for async task Sender.produce
  //execute task functionality code
  Sender_produceImpl(&Sender_produce_o_internal);
  {
    tdl_comm_Frame frame = tdl_comm_getFrame(0);
    frame->position=0;
    //copy internal task port to the frame buffer
    tdl_comm_putInt(frame, Sender_produce_o_internal);
    tdl_comm_sendBuffer(frame->bufferIndex, frame->tdlFrameSize);
  }
break;
```

On the receiving node, the following function is generated in the TDL main file and called after the reception of synchronous frames before the TDL Machine step is executed. After receiving the buffer, the task's internal port is extracted and finally the termination driver copies it to the public output port of the stub module.

```
static void receiveAsyncFrames(void) {
  {
    tdl_comm_Frame frame = tdl_comm_getFrame(0);
    frame->position=0;
    tdl_comm_receiveBuffer(frame->bufferIndex, frame->tdlFrameSize);
    //obtain internal task port from the frame buffer
    tdl_comm_getInt(frame, &Sender_produce_o_internal);
    Sender__drivers(0); //call termination driver
  }
}
```

### 3.3.4. Platform-Specific Plug-Ins

The platform-specific functionality of the TDL Comm Layer framework is implemented by means of plug-ins. They comprise functions which handle initialization, sending and receiving frame buffers and time synchronization on a concrete hardware platform. This section describes the plug-in interface.

A concrete communication platform is initialized with the following function:

```
void tdl_comm_init_platform(void);
```

Sending and receiving a frame buffer is implemented by the following two functions:

```
void tdl_comm_receiveBuffer(int bufferIndex, int size);
void tdl_comm_sendBuffer(int bufferIndex, int size);
```

Both functions have parameters for which buffer to send/receive and how many bytes to send/receive. The latter is used to send/receive the exact number of bytes transmitted on the communication bus. For sending the corresponding `position` of the frame struct is used and when receiving the expected size of the frame associated with the buffer is used.

For synchronization of the time base of the communication bus with the time base of individual nodes, a plug-in must provide additional functions. As the synchronization algorithms vary considerably between different platforms, there are no function prototypes in the generic `tdl_comm.h` header file. Instead, the prototypes must be provided in the plug-in header file `tdl_comm_<platform>.h`, so that they can be utilized in the generated glue code. Chapter 1 describes the implementation of platform-specific TDL Comm Layer plug-ins for prototyping hardware using the FlexRay communication bus.

# 4. Code and Schedule Generation Framework

This chapter presents a code and schedule generation framework for LET-based systems. It uses TDL as an example language and thus generates glue code suitable for the TDL Runtime System implemented in C as described in the previous chapter. It thereby ensures the correct behavior of TDL modules on potentially distributed platforms. LET-based components, i.e. TDL modules, serve as a unit of distribution and run in parallel on one or more nodes. The LET abstraction and the resulting property of transparent distribution lay the basis for efficient automatic code and schedule generation, as the logical timing specification is used as input for the software synthesis process. A manual mapping of LET-based components to target platforms would be error-prone and not effective, as the behavior of the system is already completely specified by the LET. This fact leaves little room for manual optimizations, apart from such concerning CPU and memory utilization. Consequently, the whole LET design flow relies on efficient and reliable automatic platform mapping, i.e. on the generation of glue code, task and communication schedules for all target platforms involved whereby the process enforces the LET-based specification automatically.

For the purpose of code generation we developed a versatile framework which uses a layered architecture making it flexible regarding support for additional platforms. Note that the term *platform* is actually a generic term, ranging from the programming language and operating system a concrete hardware platform uses to its specific communication and input/output controllers. Our code generation framework acknowledges this by the ability of subsequent refinements to the code generation functionality. As the correctness of the platform mapping is essential for applying a LET-based development process, our code generation framework maximizes the reuse of code components to minimize the chance of programming errors. A less detailed description of the framework has been published in [31].

Figure 16 presents an overview of how the different parts of the framework interact with each other. The figure is vertically divided into a platform-independent and a platform-dependent part and horizontally into a cluster and a node level part, whereby the cluster part is only required for distributed systems. The numbers indicate the order in which the depicted steps are performed. Code generation is based on TDL modules and their deployment, specifying which component is executed on which node. This information is used by the node plug-in to generate the appropriate code that executes all modules mapped to a node. Node plug-ins are split into a platform-independent generic part and a platform-dependent subclass implementing a specific hardware target. The platform-independent Comm Scheduler however needs the deployment information to generate a list of frames that must be transferred between nodes. Each frame is assigned a timing window indicating when it must be transferred via the network. The Comm Scheduler Plug-In is the platform dependent part of the cluster level and is tailored to a specific communication protocol. It schedules the frames obtained by the Comm Scheduler and assigns a

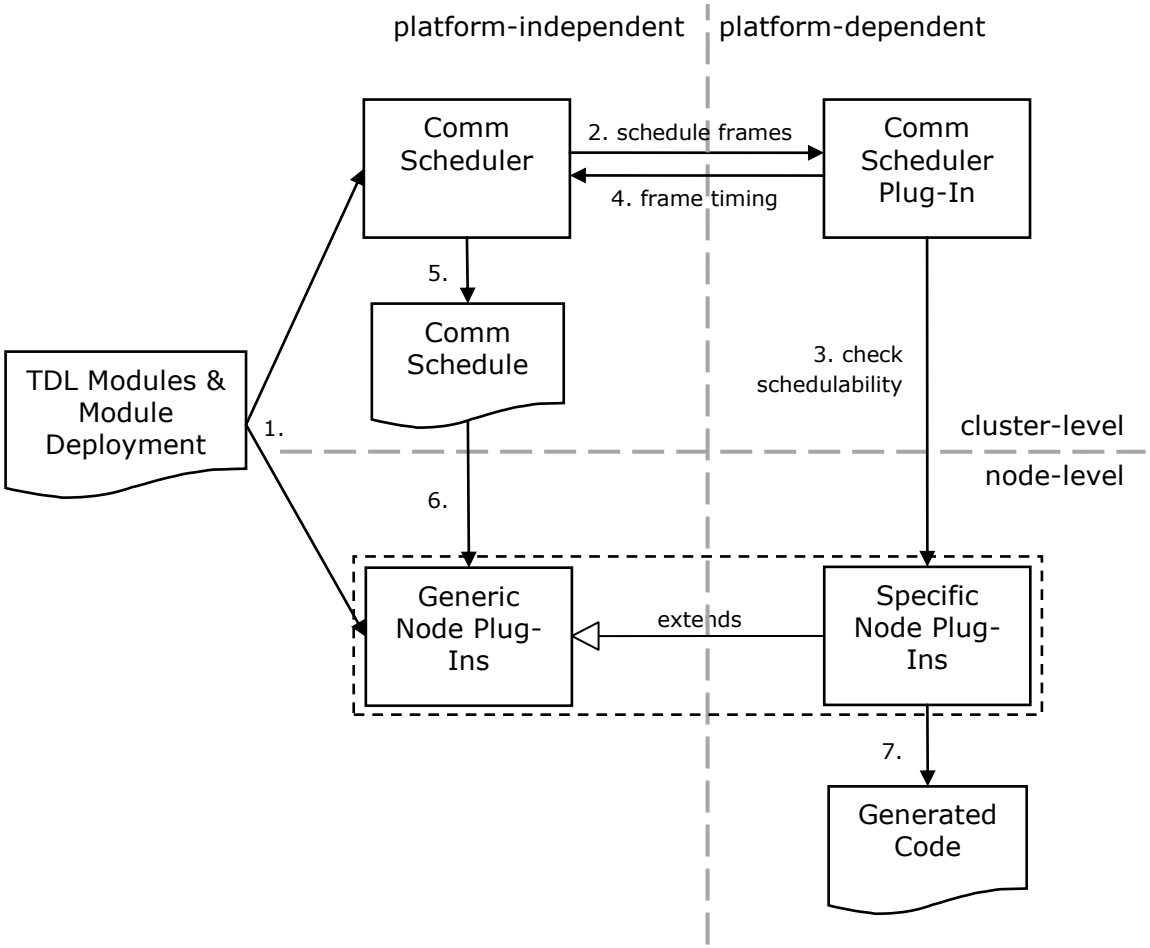


**Figure 16. Framework collaboration diagram**

concrete timing to them which must be within their timing windows. It can query the node plug-in by supplying it with a candidate set of frames with assigned timing to check whether the nodes are schedulable with this set. If it is not schedulable the Comm Scheduler Plug-In can come up with an alternative set leading to a feasible node task schedule. This approach prevents that a communication schedule is produced that eventually is not usable because of restrictions imposed by scheduling constraints on node-level, such as CPU speed limitations. After a feasible schedule is found, the Comm Scheduler stores it in a data structure called Comm Schedule which is subsequently used by the node platform plug-ins to generate code.

The code which is dynamically generated by the framework consists of multiple parts. Those parts are either specific to or depending on (1) the TDL code of modules, (2) the concrete target platform, (3) the communication requirements between TDL modules in a distributed system, or (4) the schedule for a concrete communication protocol between nodes. Table 4 presents a classification of the generated code listing typical examples of what kind of code is generated.

| | Platform-independent | Platform-dependent |
|---|---|---|
| Cluster level | Generic module communication requirements, communication windows for network frames | Communication schedule for a concrete communication protocol obeying frame window constraints |
| Node level | TDL Runtime System configuration: Module glue code, wrappers for drivers and guards, E-Code | Task schedules, platform-specific invocation of the TDL Runtime System, I/O driver assignment to TDL sensors and actuators, make file |

**Table 4. Parts of the generated glue code**

The alternative to pre-runtime code generation would be to do all processing on the nodes itself at runtime. However, this would require numerous dynamic data structures whose handling contradicts the computing power and the dependability requirements of typical embedded systems. When distribution is involved it gets even more complicated as information of the whole system is needed on every node because the communication schedule must be coordinated globally.

A previous implementation of a runtime system for TDL and corresponding code generation functionality uses a Platform Abstraction Layer (PAL) based on C functions [7]. For every platform various functions must be implemented to adapt the runtime system to a specific operating system. However, its design makes implicit assumptions about what functionality an operating system provides and therefore it is not possible to support certain platforms in a straight forward way. In contrast, our approach shifts the platform abstraction to the level of code generation mechanisms and thereby enables the adaptation to a significantly broader range of platforms. For example, it is possible to support multiple programming languages while still reusing parts of the code generation functionality.

Code generation for distributed systems based on Giotto, a predecessor of TDL, has been proposed in [32]. The authors also use the basic idea of handling network communication by scheduling messages within the LET period of its producers. However, they employ a different workflow as they require that a system integrator assigns CPU time and network bandwidth before individual components are designed. The benefit of this strategy is that code can then be generated independently. In contrast, our centralized code generation process allows more flexibility in case components are added or changed during development as then the whole code including the communication schedule is regenerated to accommodate for all changes in the timing requirements.

Concerning the usage of the framework, typically a graphical front-end harnessing our code generation framework supports the deployment of components to a distributed system. Such a tool, as for example the TDL:VisualDistributor presented in 2.6, also provides configuration options concerning the mapping of sensors and actuators to concrete hardware devices and other hardware parameters.

This chapter covers our Java implementation of the framework foundations including basic plug-ins for ANSI-C, whereas the next chapter describes plug-ins for concrete node and communication platforms. In the following sections we first describe the framework foundations and then the code generation mechanisms on the node and cluster level.

## 4.1. Framework Foundations

This section introduces the basic framework elements, consisting of interfaces and (abstract) classes. The framework foundations are not specific to a concrete programming language or hardware platform. They are the root of the platform plug-in class hierarchy which represents a *plug-in architecture* that can be extended for an open set of target platforms on the node and cluster level. Common features for both platform types include access to TDL modules and their deployment, unified handling of platform options and a specified destination directory for storing the generated code.

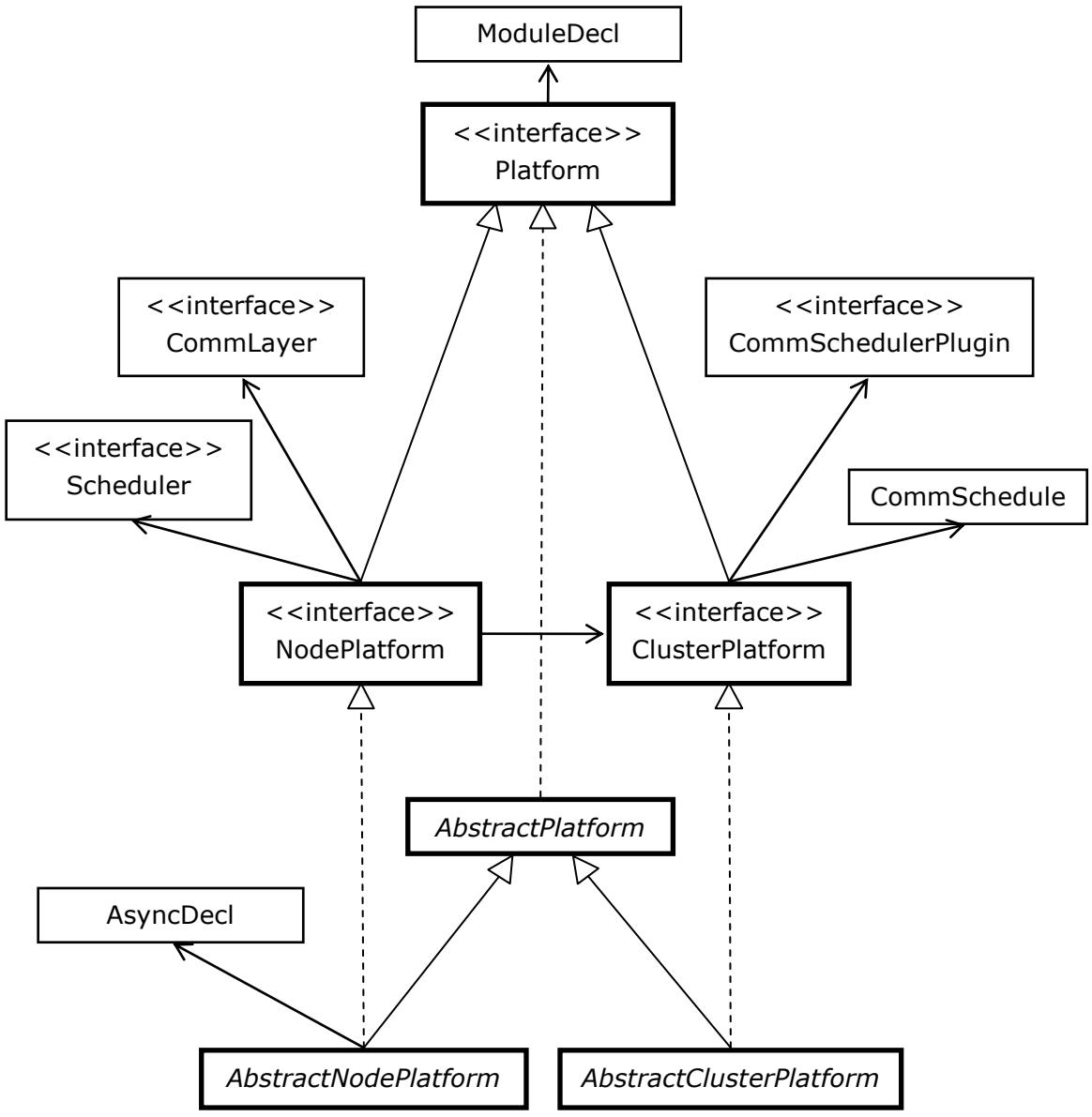


**Figure 17. Framework foundation classes and interfaces**

Figure 17 depicts a UML class diagram of the framework foundations (indicated by the thick borders) including the most important interfaces and classes its elements refer to. On top is the `Platform` interface which contains basic methods all node and cluster plug-ins must implement, such as handling platform options, the destination directory and the passing of TDL module objects. It uses `ModuleDecl` which represents a module's Abstract Syntax Tree (AST) as supplied by the TDL compiler. `AbstractPlatform` is an abstract class that implements the `Platform` interface and provides default implementations which for example store the destination directory and modules in its instance attributes. The `NodePlatform` and `ClusterPlatform` interfaces extend the `Platform` interface by methods which node and cluster platforms must implement. For that purpose, they use a `CommLayer` communication layer, a task scheduler `Scheduler`, a communication scheduling plug-in `CommSchedulerPlugin` and a communication schedule `CommSchedule`. Note that `NodePlatform` also stores a reference to a `ClusterPlatform`. In addition to a number of default implementations, `AbstractNodePlatform` also provides methods and attributes for the processing of asynchronous activities. It uses the class `AsyncDecl` which represents an asynchronous activity. Finally, the abstract class `AbstractClusterPlatform` does not provide any functionality apart from providing a base class for cluster platforms, which combines the `AbstractPlatform` class and the `ClusterPlatform` interface.

In the following we will systematically describe the framework foundation classes and interfaces including their attributes and methods in detail.

### Interface Platform

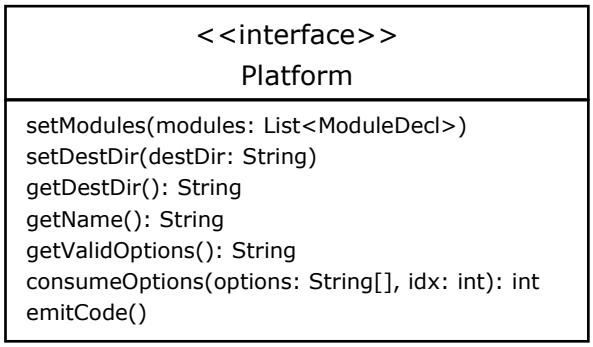


**Figure 18. Interface Platform**

The `Platform` interface (see Figure 18) must be implemented by all classes which generate code, whether it is node or cluster-specific. The output of a platform plug-in class should be written to files as expected by the target platform's implementation of the TDL Runtime System. Platform classes may consume an arbitrary number of custom options.

| | |
|---|---|
| **void** | `setModules(List<ModuleDecl> modules)`<br>This function sets the modules to be processed, where `modules` is a list of abstract syntax trees (AST) of the modules. The AST represents a single, compiled module of the system by an object of the class `ModuleDecl` (see Figure 19), which is created by the TDL compiler. It contains the complete information about a module, including its E-Code, and provides methods to query its data structures. |

| | |
|---|---|
| **void** | `setDestDir(String destDir)`<br>Sets the destination directory to be used for all output files a platform plug-in generates. |
| `String` | `getDestDir()`<br>Gets the destination directory. |
| `String` | `getName()`<br>Returns the human-readable name of this platform class, which typically is a detailed name of the target platform. |
| `String` | `getValidOptions()`<br>This returns the valid options for the platform class with one option per line which may also contain a short comment. If no information about valid options is available, `null` should be returned. Examples for platform options are a `-debug` flag enabling debug output or a `-node <nodeName>` option which passes the name of a node to a plug-in. |
| **int** | `consumeOptions(String[] options, int idx)`<br>This method is called for setting plug-in specific options. A plug-in may consume an arbitrary number of options inside the String array `options`. The index `idx` indicates the first option that might be consumed by the plug-in, whereas the value the function returns is the index of the first option that does not belong to the given plug-in class. |
| **void** | `emitCode()`<br>This method generates the platform specific code for all modules set by `setModules()`. It is called after all setters and `consumeOptions()`. |

**Table 5. Methods of interface Platform**

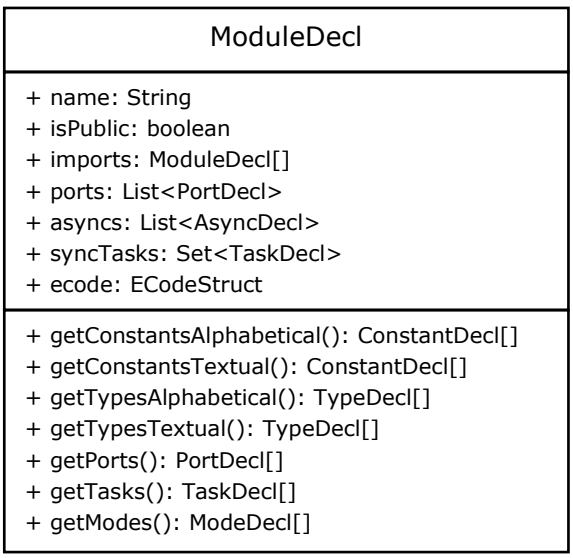


**Figure 19. Class ModuleDecl representing the Abstract Syntax Tree (AST)**

### Class AbstractPlatform

The abstract base class `AbstractPlatform` (see Figure 20) implements the `Platform` interface and provides an empty plug-in, i.e. a plug-in that does not emit any files. It provides straight-forward implementations for handling the destination directory and the abstract syntax trees of modules.

The implemented functions `setDestDir()` and `getDestDir()` set and get the destination directory which the class stores in the protected field `destDir`. The class also implements the function `setModules()` which stores the module ASTs in the

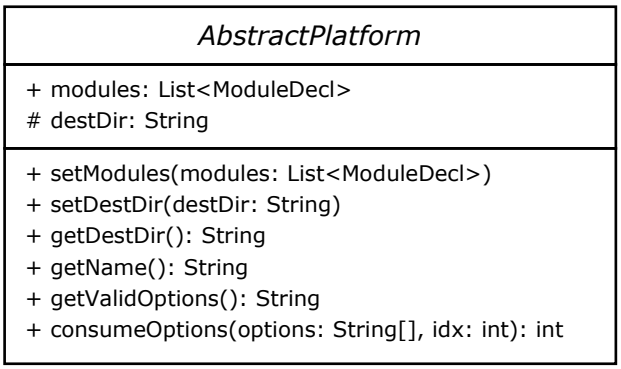


**Figure 20. Abstract class AbstractPlatform**

field `modules`. A default implementation of `getName()` returns the name of the Java class. Furthermore, an implementation of `consumeOptions()` does not consume any arguments and therefore simply returns the supplied index. Consequently, `getValidOptions()` returns `null`.

### Interface NodePlatform

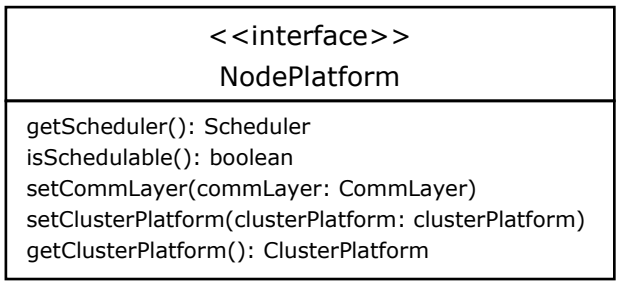


**Figure 21. Interface NodePlatform**

The `NodePlatform` interface (see Figure 21) must be implemented by a class in order to generate platform-specific code on the node level of a potentially distributed system. It serves to provide a task scheduler implementing the `Scheduler` interface. In the distributed case, `NodePlatform` also provides a communication layer and associates a node to a corresponding cluster platform. Single-node systems use a dummy communication layer and `null` as cluster platform.

| | |
|---|---|
| `Scheduler` | `getScheduler()`<br>Returns the task scheduler to be used for this node platform. It must conform to the `Scheduler` interface and is used to schedule all tasks specified by the modules assigned to a node. |
| **`public boolean`** | `isSchedulable()`<br>This function is used to check if the task scheduler is able to find a feasible schedule for the tasks to execute on a node. For distributed systems it takes into account the constraints of the communication system, e.g. the deadline of messages containing a task's ports. |
| **`void`** | `setClusterPlatform(ClusterPlatform clusterPlatform)`<br>Sets the cluster platform plug-in object which represents the communication system a node is connected to in a distributed system. Its main purpose is to create a global communication schedule. If no cluster platform plug-in is set, this node is considered to be a stand-alone node. |
| `ClusterPlatform` | `getClusterPlatform()`<br>Gets the cluster platform plug-in object. |
| **`void`** | `setCommLayer(CommLayer commLayer)`<br>Associates a `CommLayer` object with this platform object. A node platform class is supposed to delegate all code generation functionality concerning network communication to a separate `CommLayer` object. This allows generating code for stand-alone systems and for distributed systems in a similar way. The communication layer generates code for the interaction between a node and the communication system, e.g. code that sends and receives TDL port values via the network. |

**Table 6. Methods of interface NodePlatform**

### Class AbstractNodePlatform

`AbstractNodePlatform` (see Figure 22) provides a default implementation of the `NodePlatform` interface and stores a node name, the associated `CommLayer` and cluster platform as an attribute. All specific node platform plug-ins extend this abstract class. For supporting target specific code generation for asynchronous activities, we provide some base functionality in the class `AbstractNodePlatform`. In particular, it contains a method that prepares auxiliary data structures that are expected to be required by all node plug-in classes. Note that these data structures cannot be provided by the TDL compiler via the AST, as asynchronous activities are handled per node and not per module. Thus, preparing these structures takes into account which modules are placed on a particular node and which are stub modules, i.e. imported from a remote node. The TDL compiler provides the involved data structures (`AsyncDecl`, `QualPortID`, `FunCall`, `TaskDecl`) as part of the abstract syntax tree of a module.

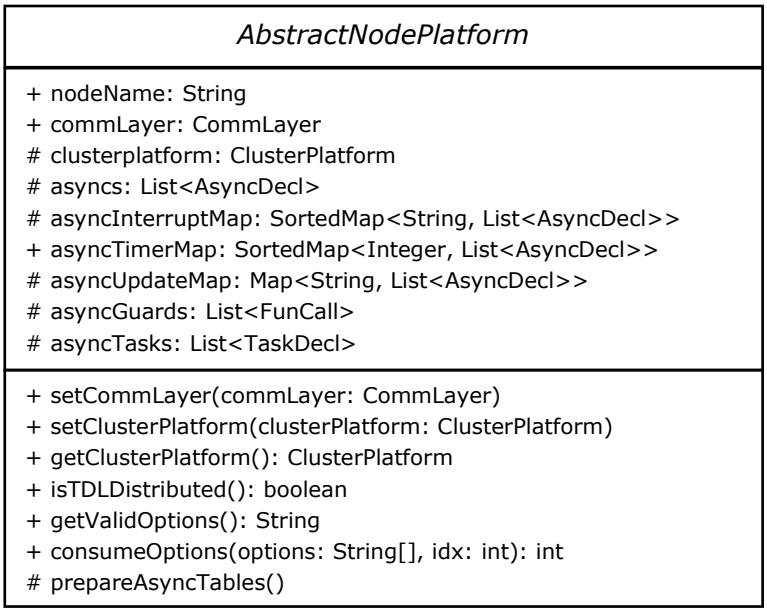


**Figure 22. Abstract class AbstractNodePlatform**

| | |
|---|---|
| **public** `String` | `nodeName`<br>Stores the name of the node, which acts as an identifier of the node in distributed systems. |
| **public** `CommLayer` | `commLayer`<br>This field contains the communication layer being used and is set by an implementation of `setCommLayer()`. |
| **protected** `ClusterPlatform` | `clusterPlatform`<br>Stores the platform plug-in which handles the communication between nodes in a distributed TDL system. Set and get by straight-forward implementations of `setClusterPlatform()` and `getClusterPlatform()`. |
| **protected** `List<AsyncDecl>` | `asyncs`<br>Represents all asynchronous event sequences from non-stub modules on this node. |
| **protected** `SortedMap<Integer, List<AsyncDecl>>` | `asyncInterruptMap`<br>Maps all interrupt numbers to the corresponding asynchronous event sequences of non-stub modules on this node. |
| **public** `SortedMap<Integer, List<AsyncDecl>>` | `asyncTimerMap`<br>Maps all timer periods to the corresponding asynchronous event sequences of non-stub modules on this node. |

| | |
|---|---|
| **protected** Map<QualPortID, List<AsyncDecl>> | asyncUpdateMap<br>Maps all update port triggers to the corresponding asynchronous event sequences of non-stub modules on this node. |
| **protected** List<FunCall> | asyncGuards<br>All async guard calls from non-stub modules on this node. |
| **protected** List<TaskDecl> | asyncTasks<br>All async tasks from stub and non-stub modules on this node. Stub modules are included in this data structure to ensure that terminate drivers of such modules, which might be executed asynchronously to the TDL Machine, are properly synchronized. |
| **public boolean** | isTDLDistributed()<br>This boolean function returns whether this node has an associated cluster platform and is therefore part of a distributed system or not. |
| **public int** | consumeOptions(String[] options, **int** idx)<br>The class AbstractNodePlatform consumes two options: -node <nodeName> which sets the node name whose value is stored in the field nodeName and -debug which activates the debug mode of a plug-in. Correspondingly, the function getValidOptions() informs about those two options. |
| **void** | prepareAsyncTables()<br>This function prepares all async data structures as listed above by iterating over all modules and their asynchronous sequences. This method must be called explicitly by subclasses. |

**Table 7. Methods of abstract class AbstractNodePlatform**

### Interface ClusterPlatform

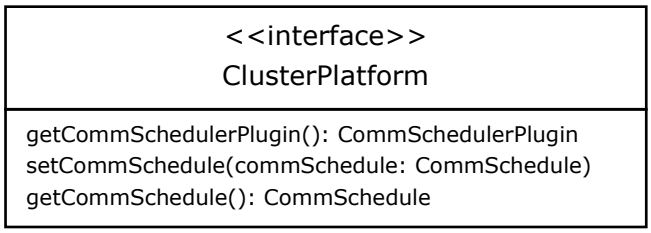


**Figure 23. Interface ClusterPlatform**

This interface (see Figure 23) must be implemented by a class in order to participate in code generation on the level of a cluster. A ClusterPlatform does not itself perform communication scheduling, but provides a Comm Scheduler Plug-In to the

Comm Scheduler for the purpose of communication protocol-specific scheduling. This plug-in maps abstract scheduling data, such as frame windows, to a concrete communication protocol, e.g. the concrete timing of frames on the network. It is represented by an object implementing the `CommSchedulerPlugin` interface and provided by the method `getCommSchedulerPlugin()`. The functionality of this interface and how it is used in the context of communication schedule generation for a TDL system is described in detail below in section 4.3 on cluster-level code generation.

Another important function of the `ClusterPlatform` interface is to provide the Comm Schedule data structure to node platforms which have a reference to it as already described above. The class `CommSchedule` (see Figure 24) contains information on the assignment of modules to nodes and all data that is transferred between nodes,

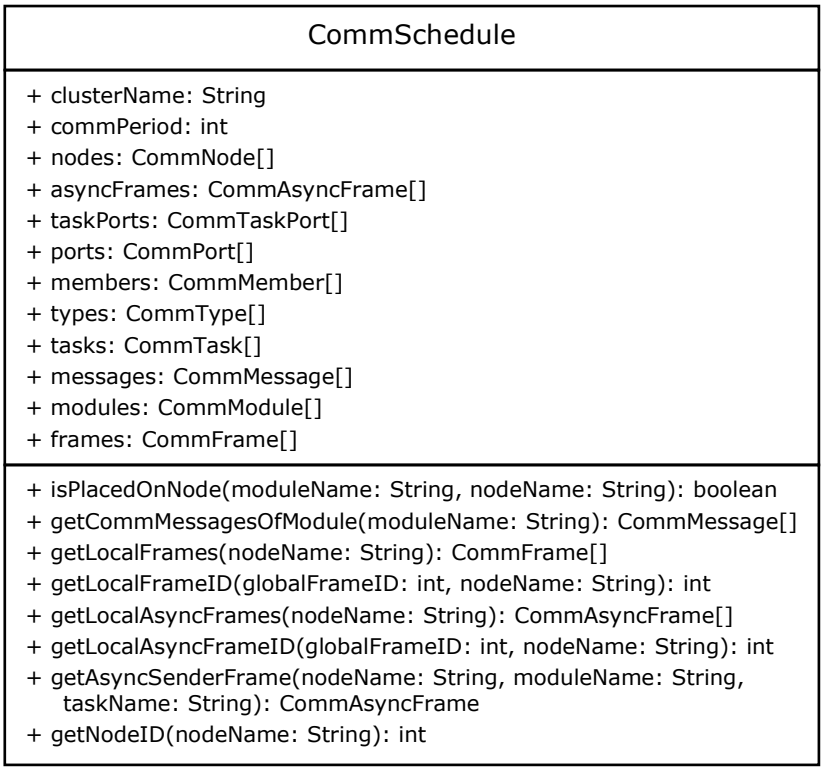


**Figure 24. Class CommSchedule**

i.e. data type and size information of synchronous and asynchronous communication. It includes several methods to conveniently obtain the data stored in it. After the communication scheduling process has finished, it is set and get by `setCommSchedule()` and `getCommSchedule()` respectively.

On the basis of the final communication schedule, a cluster plug-in emits code as required by the protocol it implements, for example a proprietary file specifying the schedule. Note that it is not always necessary to explicitly output code, as it can be the case that the communication schedule is encoded in data that is scattered across the nodes of the distributed system and is therefore part of the generated code on node level.

## 4.2. Node-Level Code Generation

After introducing the framework's foundations, this section is on the generic code generation facilities on node level. Our code generation framework is not specific to a certain target programming language. The TDL language report [14] contains so-called language bindings, which encapsulate programming language specific details in order to minimize the individual adaptations required concerning the tool chain. They contain naming conventions and parameter passing rules concerning the functionality code and specify how TDL types are mapped to programming language data types. The framework plug-ins adhere to these language binding rules, currently specified for Java and ANSI-C. This section focuses on embedded systems based on the C programming language.

The challenge concerning code generation for TDL is to generate code for a variety of different hardware platforms. The design goal is to come up with a flexible architecture that allows maximum code reuse when adding support for additional hardware targets. We decided to use repeated subclassing to achieve this. The degree of platform dependence increases with every additional inheritance level in the class hierarchy. The subclassing approach enables a subclass to reuse code generation functionality code from its ancestor by means of super calls but also to suppress or modify parts of it by selectively overriding methods.

Figure 25 illustrates a sample hierarchy of plug-in classes by means of a UML class diagram. The foundation class `AbstractNodePlatform` as described in the previous section is on top. It contains methods that must be implemented by every class that generates code for a specific platform, most notably a method which consumes specific options, a schedulability check method to test if all modules assigned to a node are schedulable and a method that triggers the actual code generation. Next in the hierarchy are the two classes `JavaPlatform` and `CPlatform`, which implement code generation functionality for all platforms that use the Java or C programming language respectively. `JavaPlatform` is an experimental plug-in for a Java TDL runtime system. Apart from the generic platform there are no further plug-ins for concrete platforms based on Java and also only single-node systems are supported. `CPlatform` generates C glue code for the execution of TDL components by the TDL Runtime System as presented in chapter 3. `EmbeddedCPlatform` aims at embedded systems with simple operating systems which typically lack a file system and therefore the TDL E-Code is represented as C code, which is later compiled and linked to the executable for the particular node. It also adds task scheduling functionality by generating dispatch tables for task execution. As examples for operation system plug-ins we add another two classes. `RTLinuxPlatform` generates code for RT Linux [29] which provides a file system and therefore extends CPlatform directly. One inheritance level below `OSEKPlatform` generates code for the OSEK/VDX [33] operating system and additional classes which implement common code generation features used for multiple hardware platforms. The bottom level, indicated by empty boxes in Figure 25, consists of platform-specific plug-ins that are tailored to concrete hardware platforms and are discussed in chapter 1. Support for any additional programming language, operating system or hardware platform requires adding specific classes that generate the corresponding code at the appropriate level in the class hierarchy.

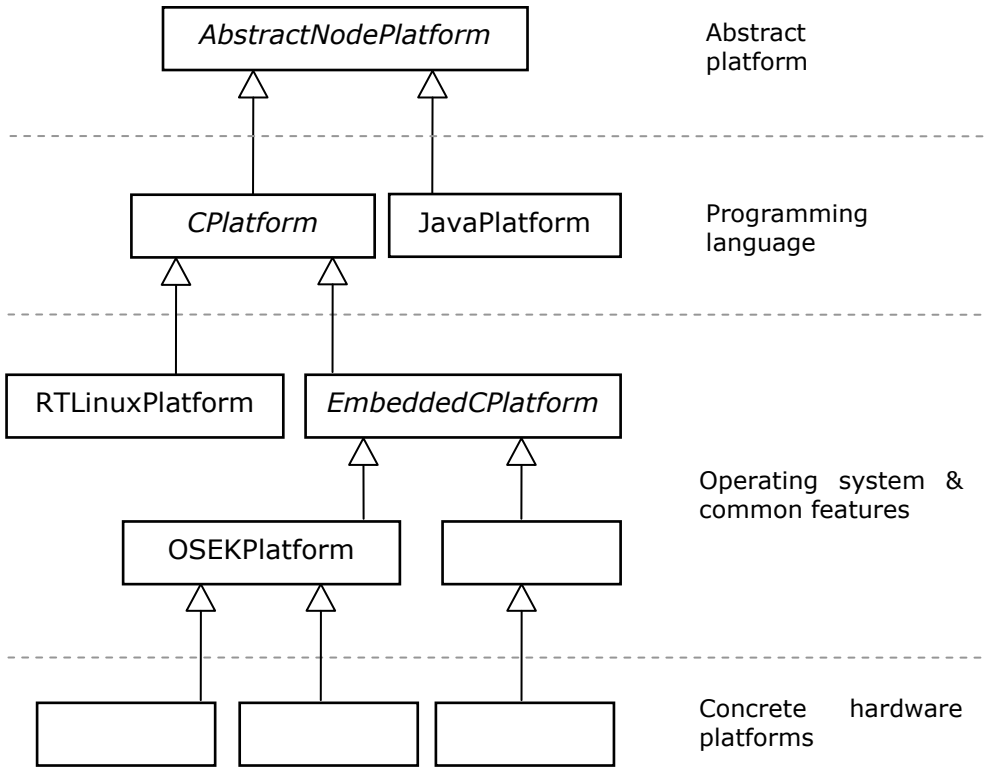


**Figure 25. Node platform abstraction levels**

Our philosophy behind generating code for a target platform is to perform as little computation as possible on the platform itself at runtime. An important goal was to avoid dynamic memory allocations except during startup. Concerning the glue code required for TDL modules, all data structures and their sizes are statically known and space can therefore be allocated statically. This significantly increases determinism and keeps the space and CPU time overhead of executing TDL modules to a minimum, which is especially important for embedded real-time systems with low processing power and high dependability requirements.

In the following we describe the plug-in classes which implement support for embedded systems programmed in C. The next two sections describe the C Platform and the Embedded C Platform plug-ins for single-node systems. Afterwards we point out the differences regarding code generation for stub modules. Finally, we introduce the communication layer, which acts as an interface to the communication system for distributed systems. Plug-ins for specific hardware platforms are subsequently described in chapter 1.

Throughout this section we use the following example TDL application to illustrate how the generated glue code parts from the platform plug-in classes look like. It consists of four modules distributed across two nodes. The modules have already been discussed in chapter 2 and contain the producer-consumer example in the synchronous (modules *Sender* and *Receiver*) and the asynchronous version (modules *AsyncSender* and *AsyncReceiver*). The two sender modules are mapped to *Node1*, whereas the two receiver modules run on *Node2*.

```
module Sender {

  sensor boolean switch uses getSwitch;

  actuator int display uses setDisplay;

  public task produce {
    output int o := 10;
    uses produceImpl(o);
  }

  start mode main [period=10ms] {
    task
      [freq=2] produce(); // LET = 10ms/2 = 5ms
    actuator
      [freq=1] display := produce.o; // updated every 10ms
    mode
      [freq=1] if exitMain(switch) then freeze;
  }

  mode freeze [period=10ms] {}
}
```

```
module Receiver {

  import Sender;

  actuator int display uses setDisplay;

  task consume {
    input int i;
    output int o;
    uses consumeImpl(i, o);
  }

  mode main [period=10ms] {
    task
      [freq=1] consume(Sender.produce.o);
    actuator
      [freq=1] display := consume.o;
  }
}
```

```
module AsyncSender {

  actuator int display uses setDisplay;

  public task produce {
    output int o := 10;
    uses produceImpl(o);
  }

  asynchronous {
    [interrupt=INT0, priority=5]
      produce(); display := produce.o;
  }
}
```

```
module AsyncReceiver {

  import AsyncSender;

  actuator int display uses setDisplay;

  task consume {
    input int i;
    output int o;
    uses consumeImpl(i, o);
  }

  asynchronous {
    [update=AsyncSender.produce.o]
```

```
        consume(AsyncSender.produce.o); display := consume.o;
    }
}
```

### 4.2.1. C Platform Plug-In

The class `CPlatform` is intended to act as a foundation emitting C code that is expected to be needed for every C-based platform. In addition, it provides a set of useful methods for its subclasses. The class writes three files for every module to the plug-in's destination directory: A C header file (`<module>_.h`), a C body file (`<module>_.c`) and an optional functionality code template header file (`<module>_template.h`). Furthermore, it writes a C main file (`tdl_main_.c`) which exists not per module but only once for every node. Note that all these files correspond to the requirements of the C TDL Runtime System as described in chapter 3.

The methods in `CPlatform` and in other framework classes adhere to a naming convention identifying which code fragment they generate. For example, methods generating code for the module header files are prefixed with `emitH_`, those for module body files with `emitC_` and those for the main C file with `emitMainC_`. Following this rule, the three methods for the creation of module header files are named `emitH_Includes()`, `emitH_Ports()`, and `emitH_Prototypes()`. This separation for example enables subclasses of `CPlatform` to add C includes to the header file by overriding the `emitH_Includes()` function with a super call and code that emits additional include statements. Methods can also be overridden without the super call to suppress parts of the code generation functionality.

#### C Module Header File

The C header file of a module contains include statements and glue code elements which must be public, i.e. accessible from outside the module's C code.

- C Includes (`emitH_Includes()`)

  **Node1/Sender_.h**
  ```
  #include "tdl_types.h"
  #include "Sender.h"
  ```

  The included C header files are the TDL type mapping for C (`tdl_types.h`) and the header file of the functionality code of the module (`<module>.h`).

- Ports (`emitH_Ports()`)

  **Node1/Sender_.h**
  ```
  extern tdl_int Sender_produce_o; /* public output port Sender.produce.o */
  ```

  All TDL ports that need to be accessed from outside the module are specified in the header. These are all global and public output ports of the module's tasks.

- Prototypes (`emitH_Prototypes()`)

  **Node1/Sender_.h**
  ```
  void Sender__drivers(int n);
  void Sender__sdrivers(int n);
  char Sender__guards(int n);
  ```

  In order to be accessible from outside, the header file contains function prototypes for drivers, start/stop drivers and guard wrappers.

### C Module Body File

The following glue code elements are written to the file `<module>_.c` in the destination directory:

- C Includes (`emitC_Includes()`)

**Node2/Receiver_.c**
```
#include "tdl_machine.h"
#include "tdl_async.h"
#include "Receiver_.h"
#include "Sender_.h"
```

The included header files are that of the TDL Runtime System (`tdl_machine.h` and `tdl_async.h`), the corresponding `<module>_.h` header just described above, as well as the glue code header files of all imported modules.

- Ports (`emitC_Ports()`)

**Node2/Receiver_.c**
```
static tdl_int Receiver_display; /* private actuator port Receiver.display */
static tdl_int Receiver_consume_i; /* private input port Receiver.consume.i */
static tdl_int Receiver_consume_o; /* private output port Receiver.consume.o */
static tdl_int Receiver_consume_o_phy; /* internal value of private output port
                                          Receiver.consume.o */
```

For every TDL port a C variable is defined. For output ports an additional internal port is generated that holds the result of a task execution until the corresponding logical ports are updated according to LET semantics.

- Drivers (`emitC_Drivers()`)

**Node2/Receiver_.c**
```
void Receiver__drivers(int n) {
  switch (n) {
    case 0: /* terminate task Receiver.consume */
      Receiver_consume_o = Receiver_consume_o_phy;
      break;
    case 1: /* set actuator with private actuator port Receiver.display */
      Receiver_setDisplay(Receiver_display);
      break;
    case 2: /* update private actuator port Receiver.display */
      Receiver_display = Receiver_consume_o;
      break;
    case 3: /* prepare release of task Receiver.consume (= copy input ports) */
      Receiver_consume_i = Sender_produce_o;
      break;
  }
}
```

A function `void <module>_drivers(int n)` is emitted, which executes driver number `n`. The number corresponds to the argument of the E-Code's `call` instruction. Drivers are emitted for mode switch, task release, task termination, actuator update, actuator setting, sensor getting and initialization functions. There are special release, actuator and termination drivers for asynchronous activities. In addition, both asynchronous and synchronous terminate drivers may trigger asynchronous events by calling the TDL Runtime System function `tdl_async_enqueue()`. Note that in contrast to the interrupt and timer triggers, the port update trigger is therefore independent of any specific C platform.

- Start/Stop drivers (`emitC_SDrivers()`)

**Node2/Receiver_.c**
```
void Receiver__sdrivers(int n) {
```

```
  switch (n) {
    case 0: /* start task Receiver.consume */
      Receiver_consumeImpl(Receiver_consume_i, &Receiver_consume_o_phy);
      break;
    default:
      return;
  }
}
```

Start drivers are used to actually call the task implementation function contained in the module functionality code. The purpose of stop drivers is to transfer public ports via a communication network and therefore they are empty in the non-distributed case. The function used to execute start and stop drivers has the signature `void <module>_sdrivers(int n)` and executes driver `n`.

- Guards (`emitC_Guards()`)

**Node1/Sender_.c**
```
char Sender__guards(int n) {
  switch (n) {
    case 0: /* guard exitMain */
      return Sender_exitMain(Sender_switch);
  }
  return 0;
}
```

For the evaluation of guards, the function `char <module>_guards(int n)` is emitted to the module glue code. The guard number `n` corresponds to the first argument of the `if` TDL Machine instruction. The function returns either `1` or `0`, depending on whether the guard evaluates to true or false.

### C Main Body File

`CPlatform` creates a C main file named `tdl_main_.c`. This file is emitted once per node and contains the initial starting point of the node's code, which can be a C `main()` function or also some other operating system specific initialization hook. Because the actually required content varies between different platforms and operating systems, only basic elements are emitted by `CPlatform`.

- Includes (`emitMainC_Includes()`)

**Node1/tdl_main_.c**
```
#include "tdl_machine.h"
#include "tdl_async.h"
#include "AsyncSender_.h"
#include "Sender_.h"
```

The included header files are those of the TDL Runtime System (`tdl_machine.h` and `tdl_async.h`) and the glue code header files of all modules placed on the specific node (`<module>_.h`) as described above.

- Asynchronous Activities (`emitMainC_Asyncs()`)

**Node1/tdl_main_.c**
```
static tdl_async_AsyncSequence asyncs[] = {
  {0, 5}, /* {pending, priority} */
};

/* Asynchronously handle external interrupt 'INT0' */
void handleInterruptINT0(void) {
  tdl_async_enqueue(0);
}
```

```
static void executeAsyncSequence(int n) {
  switch (n) {
    case 0:
      AsyncSender__drivers(2);/* release async task AsyncSender.produce */
      AsyncSender__sdrivers(0); /* start task AsyncSender.produce */
      AsyncSender__drivers(0); /* terminate async task AsyncSender.produce */
      AsyncSender__drivers(3);/* update async act. port AsyncSender.display */
      AsyncSender__drivers(1); /* set actuator AsyncSender.display */
      break;
  }
}
```

`CPlatform` extends the class `AbstractNodePlatform` and thereby inherits the functionality for preparing data structures for handling asynchronous activities. It emits a list to the main file containing all such activities using an array of `tdl_asnyc_AsyncSequence` struct elements, which contain a flag indicating whether the sequence is currently pending and a priority for every asynchronous sequence executed on a node. For the actual execution of asynchronous activities, the function `void executeAsyncSequence(int n)` is emitted. It executes the appropriate drivers for a given asynchronous activity sequence `n`. For interrupt triggers, we emit generic interrupt handlers of type `void handleInterrupt<intName>(void)`, where `intName` is the identifier of an interrupt as specified in a module's TDL code. The body of such a function calls `tdl_async_enqueue()` for all asynchronous activities triggered by interrupt `X`. These functions may be used inside the interrupt service routine (ISR) of a concrete platform. The implementation of the background thread in whose context asynchronous activities are executed, as well as the registration of hardware interrupts and the implementation of timer triggers, are highly platform dependent and are therefore not done in `CPlatform`.

- Initializers (`emitMainC_Init()`)

**Node1/tdl_main_.c**
```
tdl_async_init(asyncs, 1); /* asyncs, nofAsyncs */
```

A function call to `tdl_async_init` is emitted to initialize the priority queue with the asynchronous sequences data structure.

### Functionality Code Template Header File

**Node1/Sender_template.h**
```
#ifndef Sender_H
#define Sender_H

#include "tdl_types.h"

/* Type definition */
/* Module initialization */
void Sender_init(void);

/* Sensor getter function for Sender.switch */
void Sender_getSwitch(tdl_boolean* switch);

/* Actuator setter function for Sender.display */
void Sender_setDisplay(tdl_int display);

/* Task functions */
void Sender_produceImpl(tdl_int* o);

/* Guard functions */
int Sender_exitMain(tdl_boolean Sender_switch);

/* Initializer functions */

#endif /* #ifndef Sender_H */
```

If the optional parameter `-template` is set, `CPlatform` writes an additional file named `<module>_template.h`, which contains all elements that must be contained in the C functionality code for a specific module. The file consists of includes, type definitions for array and struct types, and function prototypes for the module initialization function, actuator setters, sensor getters, task functions, guard functions and initialization functions.

### 4.2.2. Embedded C Platform Plug-In

The abstract class `EmbeddedCPlatform` extends `CPlatform` and produces output for typical embedded systems which often lack a file system and therefore require a single, statically linked executable file that contains the complete executable code. This means that E-Code files cannot be accessed at runtime and all relevant data needs to be expressed as plain C code. In order to achieve this, the glue code is extended by additional elements.

#### C Module Header File

The C header file created by `CPlatform` is extended by a single line of code that defines the module C struct (`tdl_machine_Module`) containing all information specifying a module. It is specified as `extern` so that it can be accessed by the main file and eventually by the TDL Machine.

**Node1/Sender_.h**
```
extern tdl_machine_Module Sender__module;
```

#### C Module Body File

The content of the C body file (`<module>_.c`) is extended by the following elements:

- E-Code (`emitC_ECode()`)

  **Node2/Receiver_.c**
  ```
  static tdl_machine_ECode Receiver__ecodes[] = {
    tdl_machine_CALL(1), /* #0: call 1 -- actuator init: setDisplay(display) */
    tdl_machine_RETURN(), /* #1: return */
    tdl_machine_CALL(0), /* #2: call 0 -- terminate task: consume */
    tdl_machine_NOP(1), /* #3: EOT -- end of task terminations */
    tdl_machine_CALL(2), /* #4: call 2 -- actuator update: display := o */
    tdl_machine_CALL(1), /* #5: call 1 -- actuator setter: setDisplay(display) */
    tdl_machine_NOP(2), /* #6: EOA -- end of actuator updates */
    tdl_machine_CALL(3), /* #7: call 3 -- release task: consume */
    tdl_machine_RELEASE(0), /* #8: release 0 -- uses: consumeImpl */
    tdl_machine_FUTURE(11,10000), /* #9: future 11, 10000 */
    tdl_machine_RETURN(), /* #10: return */
    tdl_machine_JUMP(2), /* #11: jump 2 -- next cycle: main */
  };
  ```

  All E-Code instructions are expressed as C code by means of an array of structs (`tdl_machine_ECode`) that contain the E-Code opcode and parameters as required by the C TDL Runtime System.

- Modes (`emitC_Modes()`)

  **Node1/Sender_.c**
  ```
  /* Mode freeze Dispatch Table {start driver, stop driver, time}*/
  static tdl_machine_DispatchEntry Sender__dispatchtable_freeze[] = {
    {-1, -1, 2147483647}, /* task <sentinel> */
  };

  /* Mode main Dispatch Table  {start driver, stop driver, time}*/
  static tdl_machine_DispatchEntry Sender__dispatchtable_main[] = {
  ```

```
  {0, 1, 0}, /* task produce */
  {0, 2, 5000}, /* task produce */
  {-1, -1, 2147483647}, /* task <sentinel> */
};

/* Modes {pcBegin, period, dispatchtable} */
static tdl_machine_Mode Sender__modes[] = {
  {4, 10000, Sender__dispatchtable_freeze},
  {16, 10000, Sender__dispatchtable_main},
};
```

For every mode, the instruction number where the mode begins in the E-Code, the mode period and a mode dispatch table is emitted by using the `tdl_machine_Mode` struct. The dispatch table contains all tasks that need to be executed during a mode with the corresponding start and stop driver numbers and a time instance in us. The table is obtained by calling an external scheduler that implements the `Scheduler` interface. By default, `EmbeddedCPlatform` uses the `NonPreemptiveScheduler` class which produces a schedule suitable for operating systems that do not support task preemption.

- Module (`emitC_Module()`)

**Node1/Sender_.c**

```
/* Module Runtime */
tdl_machine_RUNTIMEDATA(Sender__runtime, 1, 1)

/* Task WCETs */
static long int Sender__taskWCETs[] = {100, };

/* Module */
tdl_machine_Module Sender__module = {
  Sender__ecodes, /* pointer to the E-Code table of the module */
  28, /* number of E-codes in the E-Code table of the module */
  Sender__modes, /* pointer to the modes table of the module */
  2, /* number of modes in the module */
  Sender_init, /* function pointer to system specific initialization */
  Sender__guards, /* function pointer to the guards wrapper */
  Sender__sdrivers, /* function pointer to the start/stop drivers wrapper */
  Sender__drivers, /* function pointer to the drivers wrapper */
  &Sender__runtime, /* pointer to module runtime data structure */
  Sender__taskWCETs,  /* WCETs of all tasks in the module */
};
```

Emits a C struct (`tdl_machine_Module`) representing a complete module and contains pointers to its E-Code, modes, functionality code init function, guards, start/stop drivers and drivers. Furthermore it stores a runtime data structure and the WCETs of all tasks.

### C Main Body File

The following content is added to the TDL Main file:

- Modules (`emitMainC_Modules()`)

**Node1/tdl_main_.c**

```
static tdl_machine_Module* modules[] = {
  &AsyncSender__module,
  &Sender__module,
};
```

A C array of module structs is defined containing all modules placed on the node.

- TDL Machine initialization (`emitMainC_TDLInitCall()`)

**Node1/tdl_main_.c**
```
static void initTDLMachine(void) {
  tdl_machine_init(&modules[0], 2, 5000); /* modules, nofModules, stepPeriod */
}
```

A function named `initTDLMachine()` is emitted that calls the initialization function `tdl_machine_init()` of the TDL Machine by passing the array of modules and the length of the step period on the node. This function is meant to be called by platform-specific functions emitted by subclasses of `EmbeddedCPlatform`.

### 4.2.3. Stub Module Generation

Instead of the normal glue code for a module, both `CPlatform` and `EmbeddedCPlatform` also generate code for so-called stub modules (see 3.3). A stub represents a module on a remote node on which it is imported by another module but not executed locally. In this case the TDL Comm Layer is responsible for updating the public output ports of the stub module with the corresponding port values from the node where it is actually executed.

`CPlatform` and `EmbeddedCPlatform` use the method `isStub(ModuleDecl module)` provided by the `CommLayer` interface (see next subsection below) to decide whether to emit the regular or the stub glue code. In the main file, stub modules are treated in the same way as regular modules, i.e. their header files are included and they are part of the modules struct emitted by `EmbeddedCPlatform`. Stub modules only execute termination drivers and therefore all other functionality is suppressed. In the following we list the differences when generating glue code for stub modules in detail.

#### C Module Header File

- Ports (`CPlatform.emitH_Ports()`)

**Node2/Sender_.h**
```
extern tdl_int Sender_produce_o; /* public output port Sender.produce.o */
extern tdl_int Sender_produce_o_phy; /* internal value of Sender.produce.o */
```

In addition to public output ports, for stub modules also the corresponding internal task output ports are generated so that they can be updated by the TDL Comm Layer.

#### C Module Body File

- Ports (`CPlatform.emitC_Ports()`)

**Node2/Sender_.c**
```
tdl_int Sender_produce_o = 10; /* public output port Sender.produce.o */
tdl_int Sender_produce_o_phy = 10; /* internal value of Sender.produce.o */
```

Only public output ports and their corresponding internal ports are defined.

- Drivers (`CPlatform.emitC_Drivers()`)

**Node2/Sender_.c**
```
void Sender__drivers(int n) {
  switch (n) {
    case 0: /* terminate task Sender.produce */
      Sender_produce_o = Sender_produce_o_phy;
```

```
      break;
  }
  return;
}
```

Only task termination drivers are generated for stub modules.

- E-Code (`EmbeddedCPlatform.emitC_ECode()`)

**Node2/Sender_.c**

```
static tdl_machine_ECode Sender__ecodes[] = {
  tdl_machine_RETURN(), /* #0: return -- return after empty initialization */
  tdl_machine_FUTURE(3,10000), /* #1: future 3, 10000 */
  tdl_machine_RETURN(), /* #2: return */
  tdl_machine_JUMP(1), /* #3: jump 1 -- next cycle: freeze */
  tdl_machine_FUTURE(6,10000), /* #4: future 6, 10000 */
  tdl_machine_RETURN(), /* #5: return */
  tdl_machine_CALL(0), /* #6: call 0 -- terminate task: produce */
  tdl_machine_FUTURE(9,5000), /* #7: future 9, 5000 */
  tdl_machine_RETURN(), /* #8: return */
  tdl_machine_CALL(0), /* #9: call 0 -- terminate task: produce */
  tdl_machine_JUMP(4), /* #10: jump 4 -- next cycle: main */
};
```

For stub modules a special E-Code is generated. It does not contain any new instructions and essentially only executes termination drivers of public tasks.

- Modes (`EmbeddedCPlatform.emitC_Modes()`)

**Node2/Sender_.c**

```
static tdl_machine_DispatchEntry Sender__dispatchtable_freeze[] = {
  {-1, -1, 2147483647}, /* task <sentinel> */
};

static tdl_machine_DispatchEntry Sender__dispatchtable_main[] = {
  {-1, -1, 2147483647}, /* task <sentinel> */
};

/* Modes {pcBegin, period, dispatchtable} */
static tdl_machine_Mode Sender__modes[] = {
  {1, 10000, Sender__dispatchtable_freeze},
  {4, 10000, Sender__dispatchtable_main},
};
```

For every mode the start of the mode in the stub E-Code and the mode period is emitted. As stub modules do not execute any functionality, only dummy dispatch tables without any tasks are used.

- Module (`EmbeddedCPlatform.emitC_Module()`)

**Node2/Sender_.c**

```
tdl_machine_RUNTIMEDATA(Sender__runtime, 1, 1)
static long int Sender__taskWCETs[] = {100, };

tdl_machine_Module Sender__module = {
  Sender__ecodes, /* pointer to the E-Code table of the module */
  28, /* number of E-codes in the E-Code table of the module */
  Sender__modes, /* pointer to the modes table of the module */
  2, /* number of modes in the module */
  NULL, /* STUB: no initialization */
  NULL, /* STUB: no guards wrapper */
  NULL, /* STUB: no start/stop drivers wrapper */
  Sender__drivers, /* function pointer to the drivers wrapper */
  &Sender__runtime, /* pointer to module runtime data structure */
  Sender__taskWCETs,  /* WCETs of all tasks in the module */
};
```

Certain elements of the module struct, such as the pointers to the module initialization, the start/stop driver and the guard wrappers, are set to `NULL` as for stub modules they are non-existent.

### 4.2.4. Communication Layer

The nodes of a distributed system are interconnected via some sort of communication network, also called bus. To allow different communication protocols to be combined with our node platforms, we introduce the notion of a *communication layer*.

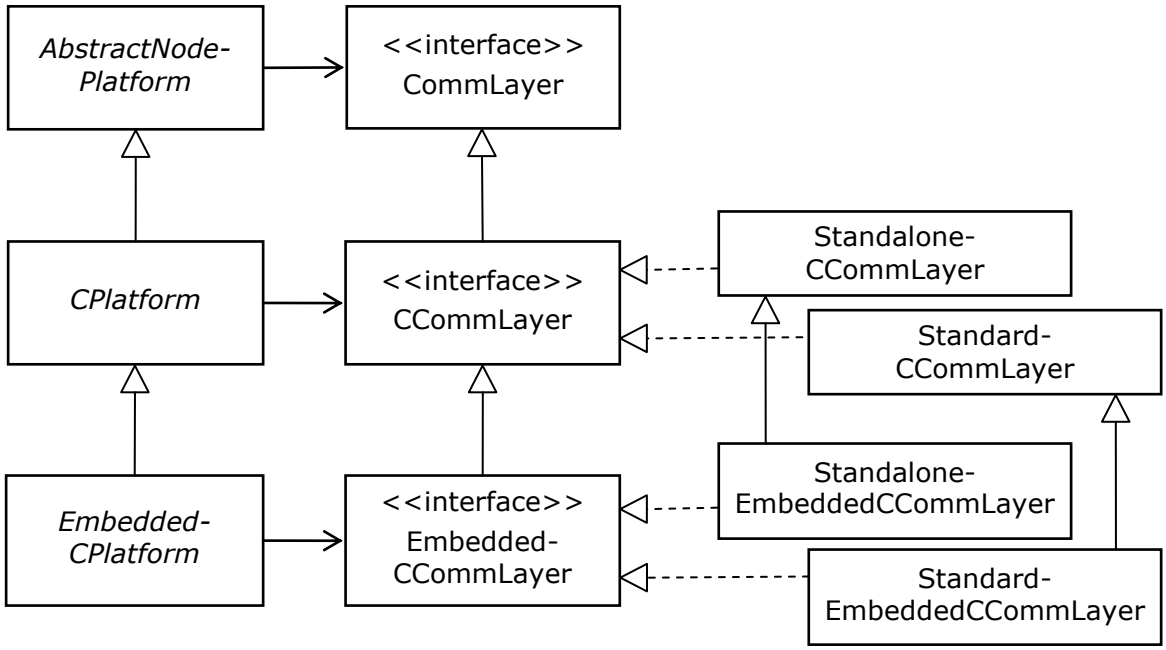


**Figure 26. Communication layer class diagram**

Figure 26 depicts a UML diagram of the platform classes described in the previous sections along with their associated `CommLayer` interfaces. In analogy to the platform classes, repeated subclassing is also performed with these interfaces. Every platform uses a class that implements a `CommLayer` interface which is responsible for the generation of the code needed to interact with other nodes via a specific communication bus. Typically, there are at least two implementations of this interface: A trivial one for standalone nodes which are not connected to a communication system at all and one or more that implement a concrete communication protocol. Implementations of the `CCommLayer` and `EmbeddedCCommLayer` interfaces handle the packing and unpacking of TDL ports independently of the communication system actually used. Additional layers then add communication bus specific functionality and handle code generation for specific communication controllers used by hardware platforms. For single node systems, both `CPlatform` and `EmbeddedCPlatform` use empty `CommLayer` implementations which are named `StandaloneCommLayer` and `StandaloneEmbeddedCCommLayer` respectively.

The hooks provided by the communication layers are called by the generic C platform plug-ins described above and allow the corresponding `CommLayer` interfaces to add specific features to the generated code.

### Generic Communication Layer

The `CommLayer` interface provides basic functionality which is used by `AbstractNodePlatform`. There are no direct implementations of this interface. It provides the following methods:

| | |
|---|---|
| **public void** | `setPlatform(NodePlatform platform)`<br>This method associates the `CommLayer` with a corresponding platform plug-in object. |
| **public boolean** | `isStub(ModuleDecl module)`<br>This function signals if the specified module must be treated as a stub module on this node. Implementing classes determine this by querying the module to node assignment provided in the Comm Schedule. |
| **public void** | `setTiMap(Map<String, Map<String, TaskInvocation[]>> tiMap)`<br>Sets a map containing all task invocations of a node. The inner map contains a String identifying the mode to which a task invocation belongs to and the outer map adds a String to identify the module. The task invocations set here are intended for reuse, i.e. the deadline field may be updated while a number of different communication schedules are evaluated as the deadline of a task invocation depends on the timing of the containing frame. In the distributed case the task invocation map is used to check the schedulability of a node by calling `NodePlatform.isSchedulable()` and for task scheduling on a node. Caching the task invocations avoids recreating the list repeatedly and is therefore solely a performance optimization. |
| **public** `Map<String, Map<String, TaskInvocation[]>>` | `getTiMap()`<br>Gets the map containing all task invocations of a node. |

**Table 8. Methods of interface CommLayer**

### C Communication Layer

The `CCommLayer` interface is associated to `CPlatform` and extends the `CommLayer` interface. It contains multiple hook methods so that code can be added to the include and content sections of the module C header and body file and the C main file. The class `StandardCCommLayer` implements functions needed for generic distributed platforms. It maintains a `CommSchedule` object which contains information on the complete communication schedule produced by the cluster part of the code generation framework. The following list describes its functionality:

- Include TDL Comm (`emitMainC_Includes()`)

  **Node1/tdl_main_.c**
  ```
  #include "tdl_comm.h"
  ```

  An include line for `tdl_comm.h` is added to the main file `tdl_main_.c`.

- Stop drivers (`emitC_StopSDrivers(int firstID)`)

**Node1/Sender_.c**
```
void Sender__sdrivers(int n) {
  switch (n) {
    case 0: /* start task Sender.produce */
      Sender_produceImpl(&Sender_produce_o_phy);
      break;
    case 1: { /* stop driver for task produce , mode 1, taskRelease 0 */
      tdl_comm_Frame frame = tdl_comm_getFrame(1);
      tdl_comm_putTag(frame, 1);
      tdl_comm_putInt(frame, Sender_produce_o_phy);
      break; }
    case 2: { /* stop driver for task produce , mode 1, taskRelease 5000 */
      tdl_comm_Frame frame = tdl_comm_getFrame(2);
      tdl_comm_putTag(frame, 2);
      tdl_comm_putInt(frame, Sender_produce_o_phy);
      break; }
    default:
      return;
  }
  return;
}
```

Emits the stop drivers required in the module body file. The parameter `firstID` is the first driver index number to be used for stop drivers. They facilitate the TDL Runtime System's Comm Layer framework to transmit the values of internal output ports via the network together with the corresponding message tag.

- TDL Comm functions (`emitH_Content()`, `emitC_Content()`)

TDL allows arbitrarily structured types constructed by nested arrays, structs and primitive types. Such types are transmitted via a network by sending a sequence of the primitive types they consist of. The code for transferring values of structured types to and from the buffer associated with a network frame is emitted once so that it can be reused throughout the generated code. For every structured type a putter (`put<customType>`) and a getter (`get<customType>`) is emitted to the module body file. Also, corresponding prototypes are written to the module header file, as the functions might be accessed by other modules importing the custom data structures as well. Note that there is no example code as the producer-consumer application does not contain any structured types.

- Message decoder function (`emitMainC_Content()`)

**Node2/tdl_main_.c**
```
static void decodeMessage(int tag, tdl_comm_Frame frame) {
  switch (tag) {
    case 1:
      tdl_comm_getInt(frame, &Sender_produce_o_phy);
      Sender__module.runtime->mode = 1;
      Sender__module.runtime->futureTime = 5000;
      Sender__module.runtime->nextPC = 6;
      break;
    case 2:
      tdl_comm_getInt(frame, &Sender_produce_o_phy);
      Sender__module.runtime->mode = 1;
      Sender__module.runtime->futureTime = 0;
      Sender__module.runtime->nextPC = 9;
      break;
  }
}
```

In the main C file a function named `decodeMesssage()` is added. This function has a tag number and a TDL frame as parameters. It is later called by the TDL

Comm Layer framework upon frame reception for every tag that is encountered. The function then stores the transmitted ports in the appropriate internal ports of the stub modules and configures the TDL Machine so that the right termination drivers are called by setting the future time and the next program counter (PC).

- Sending asynchronous frames (`emitC_StartDriverAsyncSend(ModuleDecl module, TaskDecl task)`)

**Node1/AsyncSender_.c**
```
void AsyncSender__sdrivers(int n) {
  switch (n) {
    case 0: /* start task AsyncSender.produce */
      AsyncSender_produceImpl(&AsyncSender_produce_o_phy);
      {
        tdl_comm_Frame frame = tdl_comm_getFrame(3);
        frame->position=0;
        tdl_comm_putInt(frame, AsyncSender_produce_o_phy);
        tdl_comm_sendBuffer(frame->bufferIndex, frame->tdlFrameSize);
      }
      break;
    default:
      return;
  }
  return;
}
```

This hook is called during start driver generation so that an asynchronous frame is sent after the execution of an asynchronous task if needed. The emitted code consists of calling the TDL Comm Layer framework functions for accessing the frame, writing the task's ports in the frame's buffer and finally sending the buffer. The asynchronous frame assigned to a task is found by querying the Comm Schedule which contains a list of asynchronous frames.

- Receiving asynchronous frames (`emitMainC_Content()`)

**Node2/tdl_main_.c**
```
static void receiveAsyncFrames(void) {
  tdl_comm_Frame frame = tdl_comm_getFrame(3);
  frame->position=0;
  tdl_comm_receiveBuffer(frame->bufferIndex, frame->tdlFrameSize);
  tdl_comm_getInt(frame, &Sender_produce_o_phy);
  AsyncSender__drivers(0);
}
```

For the reception of asynchronous frames, a function named `receiveAsyncFrames()` is emitted to the main C file. It handles to reception of all asynchronous frames required by the stub modules assigned to a node. The function receives all relevant frames, unpacks the TDL ports and calls the corresponding termination drivers of the stub modules to update their ports.

### Embedded C Communication Layer

The interface `EmbeddedCCommLayer` adds the following function to the `CommLayer` class hierarchy, for which a standard implementation is provided via the class `StandardEmbeddedCCommLayer`.

- Stop driver number (`int getStopDriverOfTaskInvocation(ModeDecl m, int taskID, int taskDispatchTime)`

**Node1/Sender_.c**
```
/* Mode main Dispatch Table  {start driver, stop driver, time}*/
static tdl_machine_DispatchEntry Sender__dispatchtable_main[] = {
  {0, 1, 0}, /* task produce */
```

```
  {0, 2, 5000}, /* task produce */
  {-1, -1, 2147483647}, /* task <sentinel> */
};
```

The dispatch table created by `EmbeddedCPlatform` must contain the stop driver number of a task invocation so that it can be executed. This number is provided by `StandardEmbeddedCCommLayer`. Note that the stop drivers are emitted by `StandardCCommLayer` and therefore consequently both classes need to use the same numbering scheme. If there is no stop driver to execute, e.g. if the given task is non-public or there is no distribution at all, the function returns `-1`. The function identifies a task invocation by its mode, its ID, and a time instant within its logical execution.

## 4.3. Cluster-Level Code Generation

In the last section, we added communication capabilities to individual nodes by assigning them a communication layer. For event-triggered communication protocols, such as Ethernet or CAN, this can already be sufficient to enable network communication between nodes. However, such an approach is not suitable for hard real-time systems as for instance collisions may prevent frames from being transmitted in a predictable way. Time-triggered communication protocols such as TTP (Time-Triggered Protocol) or FlexRay [34] overcome this problem, but require a global communication schedule for their operation. Usually, such a schedule is created manually as it is difficult to automatically extract the scheduling requirements of arbitrary systems. LET-based systems such as TDL however describe the data flow and timing requirements of their components explicitly and consequently it is feasible to perform fully automatic communication scheduling. This section describes the scheduling and code generation process on the cluster-level of our code generation framework.

As preliminaries for communication scheduling, we hold on to a number of assumptions:

- The network infrastructure is based on broadcast semantics, i.e. a frame sent by one node can be received at the same time by all other nodes
- Packets sent by different nodes are not combined into a single packet but are sent as individual network frames
- Collision free access to the shared communication medium via a TDMA (Time Division Multiple Access) approach as used by time-triggered protocols
- Adherence to the producer-consumer model, which means that the nodes that generate information trigger the sending over the network
- A mechanism for distributed clock synchronization

Some of these requirements, such as the TDMA property or the clock synchronization service, can be implemented on top of communication protocols which do not support them natively. This can be done by generating schedules accordingly and by extension of the platform-specific TDL Comm Layer part of the runtime system. For example, it is possible to use a TDMA communication schedule for the event-triggered protocol CAN and also implement a time synchronization service for it [15].

The scheduling mechanisms described below are based on previous work but have been significantly improved and extended. The feasibility of automatic scheduling for LET-based systems has for instance been demonstrated in [15], where the notion of transparent distribution is proposed and a prototype implementation is presented. However, it is tailored to a specific bus protocol, namely CAN, and is not designed in a way so that it allows adaptations to other protocols. Also there is no clear separation between what we call the Comm Scheduler and the Comm Scheduler Plug-In, i.e. between platform independent scheduling tasks that need to be performed for every LET-based system and those which are specific to concrete network architectures. In [35] this separation is improved, but still there is no clean interface specified such as the one we propose and which explicitly states what must

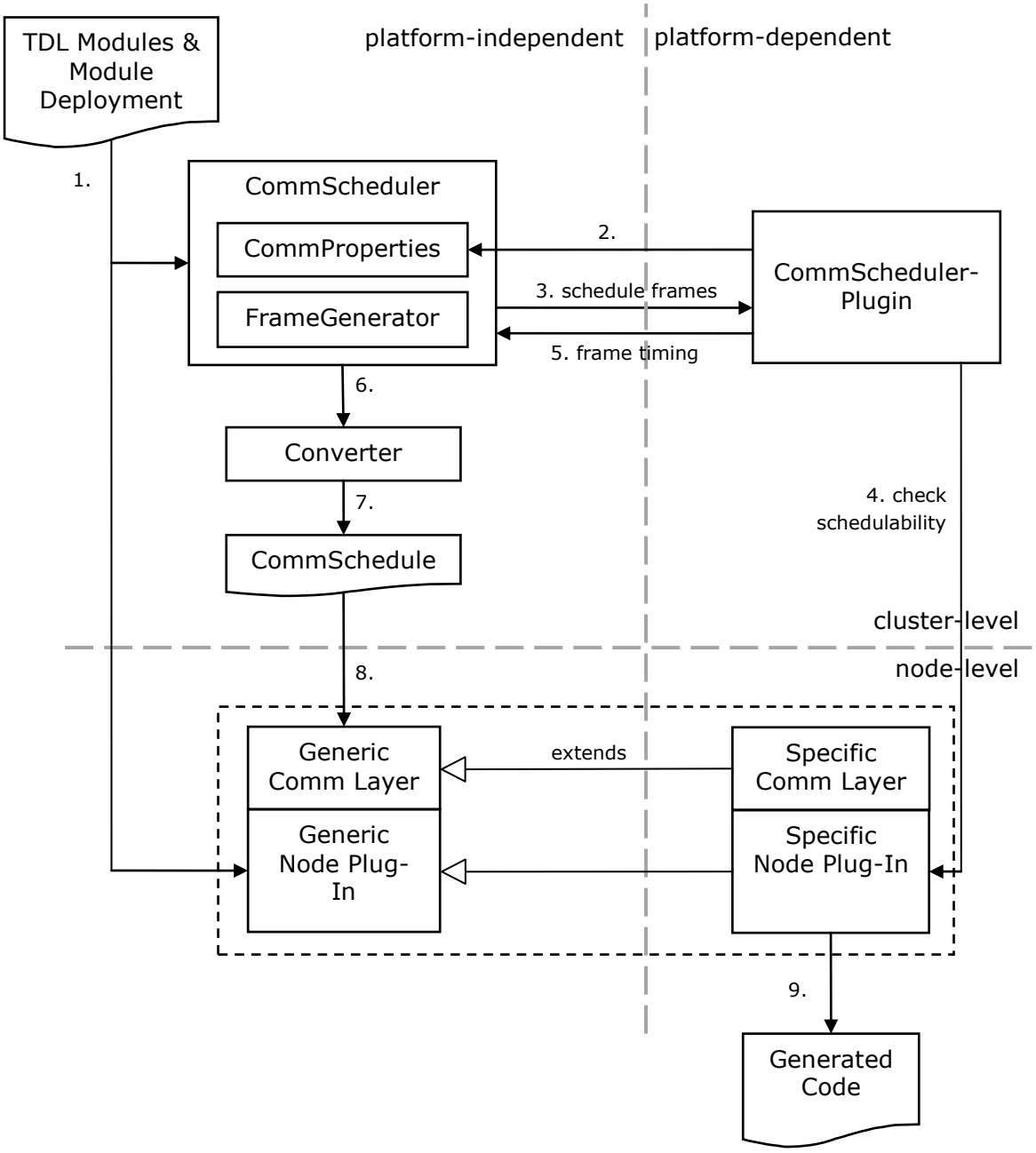


**Figure 27. Detailed framework collaboration diagram**

be implemented to support an additional communication protocol. Apart from this clear separation of concerns, we designed our framework so that the platform-independent strategy by which messages are assigned to frame windows is easily exchangeable. In previous implementations this strategy was fixed and relied on heuristics that assigned messages with the same sender and similar release time and deadlines to the same frame while trying to minimize the number of frames that are generated. We improved this strategy by applying an *iterative approach* which varies the parameters controlling the message-to-frame mapping. As an alternative to the use of heuristics for this mapping, we additionally developed a scheduling strategy which employs a *genetic algorithm*. A further key improvement of our scheduling framework is the ability to check nodes for schedulability during the communication scheduling process. This prevents that a communication schedule is produced that eventually is not usable because of restrictions imposed by node scheduling constraints. Apart from all these extensions, our implementation is also the first to incorporate asynchronous communication frames.

Figure 27 depicts a more detailed overview of the code generation process already shown in Figure 16 above, where the basic steps of the communication scheduling process have already been described. The numbers in the figure indicate the order in which the individual steps are performed. In comparison to the figure above, also the internals of the Comm Scheduler are shown and the communication layer on node level is added. Inside the `CommScheduler`, a class extending the abstract class `CommProperties` is used to store a list of properties of a communication platform. The properties class is supplied by the `CommSchedulerPlugin` interface so that it can be tailored to specific protocols. The interface `FrameGenerator` makes the strategy which is used to map messages to frame windows exchangeable and is supplied in the `CommScheduler`'s constructor. The `CommScheduler` eventually uses the `Converter` class to construct the `CommSchedule` data structure.

The next subsections detail the Comm Scheduler, two strategies for creating frame windows, and finally the Comm Scheduler Plug-in interface.

### 4.3.1. Comm Scheduler

The Comm Scheduler is the platform-independent part of the code generation framework on cluster level. It coordinates the scheduling process which has TDL modules and their mapping to a distributed platform as input. As shown in section 2.3, for every TDL task that needs to transmit a message via the network there is a communication *window* for doing so inside its LET period. It is defined as the time interval between the release and deadline of a message or a frame. The Comm Scheduler computes all those communication windows and generates a list of messages that need to be transferred between nodes. Those messages are then packed into frames, whereby their concrete timing is determined by the platform-dependent Comm Scheduler Plug-In. Finally, the Comm Scheduler produces a `CommSchedule` data structure as depicted in Figure 27 as output, which contains the

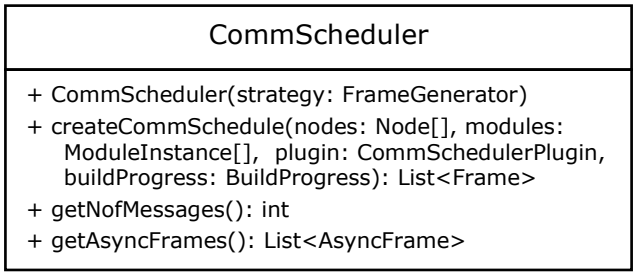


**Figure 28. Class CommScheduler**

complete scheduling information.

As the first step in the communication scheduling process, the `CommScheduler` class (see Figure 28) must be initialized. In its constructor a strategy represented by the `FrameGenerator` interface and which assigns messages to frame windows must be set:

```
public CommScheduler(FrameGenerator strategy)
```

We describe two such strategies, namely one for iterative scheduling and one that employs a genetic algorithm, in the next two subsections.

The actual scheduling process is triggered by calling the following function of the `CommScheduler` class:

```
public List<Frame> createCommSchedule(Node[] nodes, ModuleInstance[]
  modules, CommSchedulerPlugin plugin, BuildProgress buildProgress)
```

It orchestrates the schedule generation and is parameterized with an array of nodes and modules, a plug-in for the Comm Scheduler and an object implementing the `BuildProgress` interface. In comparison to `ModuleDecl`, which is used by the framework foundation classes, `ModuleInstance` represents a TDL module which is placed on a specific node. A plug-in implementing the `CommSchedulerPlugin` interface handles platform-dependent scheduling concerns. This interface most notably contains functions to determine the communication period of the cluster and to set concrete timings for synchronous and asynchronous frames while obeying the constraints of a specific communication protocol. Furthermore, it provides a class which extends the abstract `CommProperties` class containing properties of the communication protocol. `BuildProgress` is used to provide feedback during schedule generation as this might take a considerable amount of time. Finally, the scheduling function returns its results as a list of frames. The individual steps performed by this core function of the Comm Scheduler are described below.

While synchronous frames are returned directly by the `createCommSchedule()` method of the Comm Scheduler, asynchronous frames are obtained by calling the `getAsyncFrames()` method after the function has finished. As a next step, the results of the scheduling process are converted to the `CommSchedule` data structure by using the `Converter` class. The `CommSchedule` contains information about nodes, frames, messages, tasks and task ports. This data structure is important for subsequent code generation for individual nodes. It is also written to disk as a file named `commschedule.properties`. In this way, the scheduling results can be used by external tools, for example by a network analyzer tool to decode the TDL ports contained in the network frames. Finally, the frames and the Comm Schedule are passed to the `ClusterPlatform` using the `setFrames()` and `setCommSchedule()` methods.

In the following, we describe the communication schedule generation process performed by the `createCommSchedule()` method step-by-step.

### 1) Check of Communication Properties

The abstract class `CommProperties` contains a set of properties for a communication protocol, which are typically edited via a graphical user interface such as the TDL:VisualDistributor. The function `checkCommProperties()` checks those properties for correctness and consistency. This is done to prevent creating a schedule that does not conform to the communication protocol's specification. Certain properties such as the minimum and maximum communication cycle length and the minimum

and maximum size of a frame in bytes are mandatory fields already present in the abstract class `CommProperties`.

### 2) Identification of Port Receivers

This step identifies for all output ports of all modules the nodes which receive these ports. The number of such nodes can be from zero to all nodes in the system. The information is stored in the module instance data structure the `CommScheduler` uses. Furthermore, for every node a flag `isSender` is set, indicating whether a module sends anything on the network or not. This flag can be set following two different policies called *port filters* which implement the `PortFilter` interface. While the `PublicPortFilter` does not filter out any ports, the `RequiredPortFilter` checks whether a port is actually received on any other node and filters out the rest. The latter leads to a reduced number of bytes to be transferred via the communication network.

### 3) Compute Communication Period

This step calculates a suitable communication period, which is the time after which the communication schedule of the system repeats itself. The challenge is to find a repeating pattern in the communication requirements of multiple modules with multiple modes with different mode periods. An important constraint of LET-based description languages such as TDL is that they restrict mode switches such that task invocations are never interrupted by a mode switch. Thus, mode switches are said to be *harmonic*, that is, a mode switch must not occur during the LET of any task invocation of the currently active mode. Therefore, the period of a mode switch must be a multiple of the LCM (least common multiple) of the period of tasks invoked in this mode. Furthermore, the mode period is always a multiple of the periods of task invocations and mode switches.

As each mode in every module may have its specific communication requirements, an obvious candidate for the communication period is the longest time span without a mode switch in any module. To calculate this period, we define for a given module $M$ the term $mspGCD_M$ as the GCD (greatest common divisor) of mode periods and mode switch periods of all modes in $M$. We know that within the time span $[N*mspGCD_M \ .. \ (N+1)*mspGCD_M]$ there will not be a mode switch within module $M$. In other words, we can express the mode switch instants as an integer multiple of $mspGCD_M$. Based on this, we calculate the bus period as the GCD of the $mspGCD_M$ of each module $M$ which communicates on the bus, i.e. whose `isSender` flag is set. Consequently, each mode period consists of an integer multiple of communication periods and we introduce the term *phase* in order to distinguish these mutually exclusive parts of a mode.

Note that the resulting period from the calculation just described may be unsuitable for certain communication protocols. Therefore, the Comm Scheduler Plug-In has the possibility to refine it by dividing it by an integer number.

### 4) Generate List of Messages

We generate a global list of messages representing all the information which must be transferred via the communication network of a TDL system. We define the term *message* as the collection of all values produced by a task invocation's public output ports. Note that if a task is invoked $N$ times per mode period, $N$ messages are produced. As an optimization, public output ports that are not used by any client may be ignored according to the port filters described in step 2 above.

A message has a unique *tag* defining its precise origin. The tag defines the node, module, mode, task invocation, and the phase of the mode in which the message has been produced. The size of a message is measured in bytes as the sum of the size of the contained values and the size of the tag.

Each message has individual timing constraints. The *release* constraint is the earliest time instant when message sending can be started. The *deadline* constraint of the message is the latest time instant when the message sending must be finished. A straight-forward approach is to set the release constraint to the release time of the task invocation that produces that message plus its worst case execution time (WCET). Note that this is an optimistic estimate as the actual release time depends on the task schedule used, but which is not known at this point in time in the scheduling process. The deadline constraint results from the end of the LET of the producer task's invocation.

During this scheduling step also the task invocation map `tiMap` is created. It contains all task invocations of all modes in all modules on all nodes. It is passed on to the Comm Scheduler Plug-In which in turn passes it on to the communication layer of the individual nodes. The task invocation map is used for task scheduling on node-level in the distributed case. It is a performance optimization that enables updating the task invocation deadline field while a number of different communication schedules are evaluated. This is necessary as this deadline depends on the timing of the frame containing the ports of the task invocation.

**5) Generation of Frame Windows out of Messages**

A key problem in finding a schedule for a TDL system lies in the fact that every module may switch its mode independently from all other modules. This leads to different communication requirements for every combination of active modes throughout the system. Due to the exponential number of such combinations, it is not feasible to generate all these possible schedules and change them dynamically at runtime. The combinatorial explosion of modes is tackled by the notion of *dynamic multiplexing* as the foundation of communication schedule generation for TDL systems [15]. This approach creates a single schedule by the length of one communication period whose length calculation is described above. Every mode consists of one or more consecutive communication periods, which we call the *phases* of a mode. For every message in every phase, we know its release and deadline constraint and its size. At runtime, the contents of each frame changes dynamically depending on the currently active mode and the phase it is in. The message tag described above identifies the contents of frames, i.e. the origin of its messages, at runtime.

In this scheduling step, we assign every message to a communication frame window in accordance to the dynamic multiplexing approach just described. Frame windows have a release time, a deadline, a size, a sender and a list of receiver nodes. We assign multiple messages to a frame when possible. Messages must have the same sender as the frame they are assigned to. The release constraint of a frame is the maximum of the release constraints of the bound messages and the deadline constraint of a frame is the minimum of the deadline constraints of the bound messages. The schedule generator guarantees that the frame size and constraints are sufficient for the communication requirements of all phases. Note that the mapping of messages to frame windows is not unique, as with every message there is a choice of whether (1) to add it to an existing frame by possibly tightening its constraints or increasing its size or (2) to create a new frame that matches the constraints of the message exactly.

To exemplify a possible mapping of messages to frame windows, we consider a module with a mode of execution that has three phases, and we assume that it produces message 1 of 4 bytes in phase 1, message 2 of 3 bytes in phase 2, and two messages 3 and 4 of 1 byte each in phase 3. Figure 29 shows the individual messages and the frame they are bound to throughout the whole mode period consisting of three distinct phases, i.e. the length of the mode period equals to 3 times the communication period. The left and right bounds of the message and frame boxes represent their release and deadline constraints. Respecting their size and timing constraints, all messages may be bound to the same frame with size 4 bytes in the schedule.

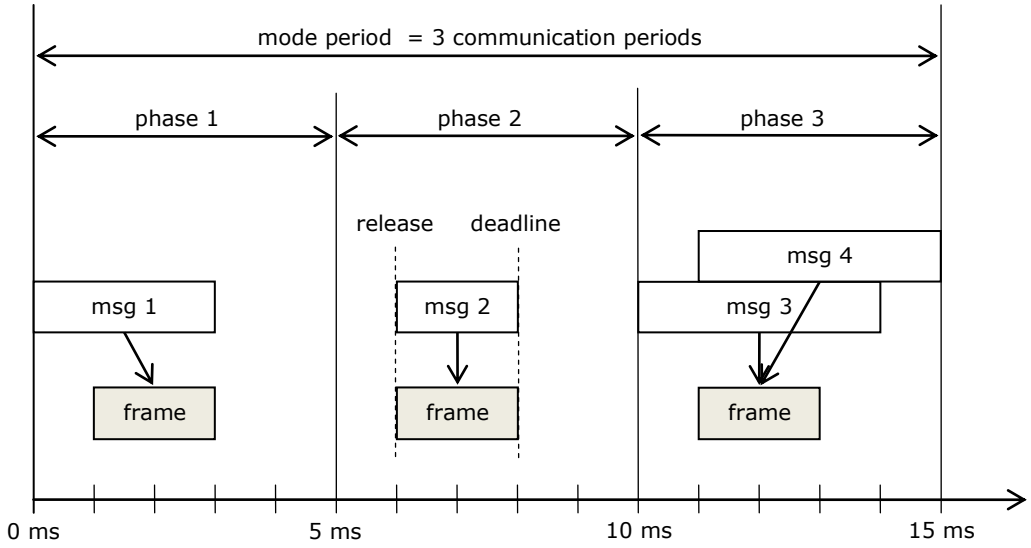


**Figure 29. Sample binding of several messages to the same frame**

Although the concept of dynamic multiplexing is always the same, the actual assignment of messages to frame windows is an optimization problem which we factored out using the `FrameGenerator` interface. In comparison to previous implementations, we had to restructure the scheduling algorithm as the generation of the list of messages and the creation of frame windows were intertwined. In contrast to that, we first generate a complete list of messages as described in the last step and then use the following function to assign them to frame windows, which is the only function in the interface:

```
public List<Frame> generateFrameWindows(List<Message> messages,
  CommSchedulerPlugin plugin, BuildProgress buildProgress)
```

The function `generateFrameWindows` returns a list of frames on the basis of a list of messages. Furthermore, it is supplied with a `CommSchedulerPlugin` which assigns concrete timings to frames, checks whether a set of frames is schedulable at all on a concrete communication protocol and provides a metric in the range from 0 to 1 measuring how good the scheduling solution is. An instance of the class `BuildProgress` is provided to allow the frame generator strategy to output information on its progress, which is especially useful when it can be expected that the algorithm takes a considerable amount of time. We discuss two `FrameGenerator` implementations in the subsections below.

### 6) Frame Scheduling

As a result of the last step, every frame is assigned a window in which it must be scheduled. Now, the Comm Scheduler Plug-In must assign a concrete start and end time for every frame which lies within its timing window. Note that it does make a difference where exactly in this window a frame is scheduled. As already pointed out above, the start of this window is only a best-case estimate and it is not guaranteed that the whole system is schedulable if frames are actually scheduled at the beginning of this window, as this might constrain the task scheduler too much so that no valid task schedule can be found. Therefore we require the Comm Scheduler Plug-In to schedule all frames as late as possible. The transmission time of a frame depends on communication protocol specifics such as the transmission speed and timing properties such as inter-frame gaps. The scheduler might also generate extra

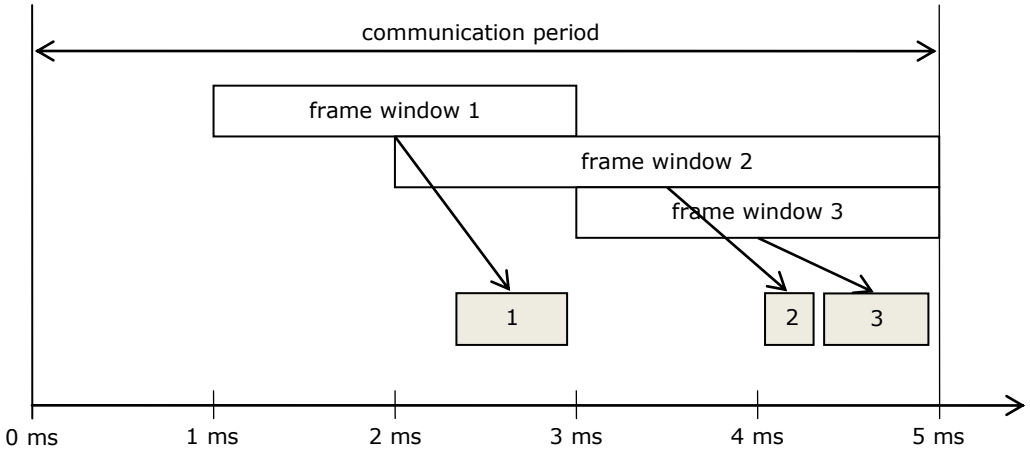


**Figure 30. Sample mapping of frame windows to frames**

frames, for example to implement time synchronization between nodes. All these requirements must be taken into account for calculating correct frame start and end times. The plug-in may also merge frames when their timing requirements are compatible and they have the same sender. Figure 30 shows a sample mapping of 3 frame windows to 3 frames aligned according to the constraints of a communication platform.

Note that while frame scheduling by the Comm Scheduler Plug-in may already be invoked by the frame windows strategy in the previous step, this is only optional and the results are not stored in the frame list which the strategy returns. Therefore, the frame scheduling step must be performed in any case after the list of frame windows is obtained.

The Comm Scheduler Plug-In can check whether the calculated start and end times for frames lead to a feasible schedules on node level by calling the `isSchedulable()` method of the `NodePlatform`. This is a key feature of our framework as it takes into account the interdependence of communication and node task schedules and thus prevents the whole code generation process from running into a dead end by creating a communication schedule for which not all nodes are able to come up with an appropriate task schedule.

The following function implements communication platform-specific frame scheduling inside the `CommSchedulerPlugin` interface:

```
public double scheduleFrameWindows(List<Frame> frameWindows)
```

The list of frames is updated by setting the `startTime` and `endTime` field of a `Frame`. Frames may also be merged, resulting in a change in the number of frames. The return value of type `double` returns a metric indicating how good a solution is on a specific communication protocol, e.g. by reflecting the relative bandwidth usage. It is negative when the set of frames is not schedulable at all. This might for example occur when the bandwidth of the protocol simply is too small to accommodate all frames that must be transferred or also when the resulting frame schedule leads to nodes where no task schedule can be found.

After the list of frames is returned, the Comm Scheduler sorts it so that it is in chronological order.

#### 7) Scheduling of Asynchronous Frames

Until now, we only considered how synchronous frames are scheduled by the Comm Scheduler. Asynchronous frames and their associated tasks are of a lower priority and therefore are scheduled after synchronous frames by using the bandwidth which is still available. This means that that do not participate in finding an optimal schedule for synchronous frames. Note that there is no strict deadline for asynchronous frames.

As a first step of the scheduling part handling asynchronous communication, all asynchronous messages of all modules in the system are identified. As in general all such messages are triggered at a different time at runtime, we map each of them to an individual asynchronous frame. We assign priorities to the frames according to the priorities assigned to the asynchronous activity producing the message. Next, the following function of the Comm Scheduler Plug-in is called to schedule the list of asynchronous frames, which are passed in order of descending priority:

```
public void scheduleAsyncFrames(List<AsyncFrame> asyncFrames,
                                List<Frame> frames)
```

The function performs the mapping of the asynchronous frames to cluster platform specific IDs stored in the field `asyncFrameID` in each `AsyncFrame` object of the list of asynchronous frames. Synchronous frames are passed as well, giving the plug-in the ability to schedule asynchronous frames as synchronous ones.

### 4.3.2. Iterative Frame Generator

In this subsection we describe one implementation of the `FrameGenerator` interface, i.e. a platform-independent strategy for the creation of frame windows out of the list of messages exchanged between modules in a distributed TDL system. It bases on the scheduling algorithm presented in [15] and [36], but is extended (1) by iterating over multiple threshold values used to decide whether to create a new frame window or to bind a message to an already existing frame and (2) by taking into account the metric that is returned by the platform-specific Comm Scheduler Plug-in. The iteration makes it more likely to find a schedule at all or to find a better solution in comparison to using a fixed threshold value as in existing implementations. In the following, we first introduce a metric that measures the compatibility of a message with an already existing frame and then explain the algorithm which consists of an inner loop creating a candidate set of frames and an outer loop which controls how the set is created and evaluates it.

The frame metric consists of two parts called *overlapping metric* and *enlargement metric*. They provide a measure for the compatibility of the timing and size constraints between a frame and a message.

The overlapping metric measures the degree of overlapping between a message and frame window. The window of a message or a frame is the time interval between its release and its deadline. If we allocate the message to a frame, then the new timing constraints for the frame will be the window of the overlapping section. Therefore, we want a high degree of overlapping, as otherwise the timing constrains become too restrictive and we reduce the chance to find a feasible schedule. The overlapping and the overlapping metric as an average percentage are defined by the following formulas. The metric yields 1 if the message and the frame window overlap completely and 0 if there is no overlapping at all.

$$overlapping = Min(frame.d, msg.d) - Max(frame.r, msg.r)$$

$$metric_{overlapping} = \frac{\frac{overlapping}{frame.d - frame.r} + \frac{overlapping}{msg.d - msg.r}}{2}$$

The enlargement metric measures by how much the size of a frame needs to be enlarged so that the message fits in. It yields 1 if the frame does not need to be enlarged at all and a value between 0 and 1 if enlargement is necessary, where smaller values indicate more enlargement. The following formulas are used to calculate the enlargement metric:

$$enlargement = Max(0, msg.size - frame.available)$$

$$metric_{enlargement} = \frac{frame.size}{frame.size + enlargement}$$

In order to get a single metric value the overlapping metric and the enlargement metric are combined using the following formula giving both metrics equal weights.

$$metric = \frac{metric_{overlapping} + metric_{enlargement}}{2}$$

This formula only applies when both metrics yield a positive value. If either metric equals to 0 or less, then the overall metric is 0 as well.

The inner loop of the algorithm creates frame windows based on a specific given threshold. All messages are considered one after another. At the beginning, a new frame is created and its size, release and deadline are simply copied from the message so that the frame window is an exact fit. For all subsequent messages, we check for all frame windows already created how well they fit by computing the heuristic metric described above, measuring the compatibility of a message to an already created frame. If for the best matching frame the metric exceeds the given threshold, the message is bound to the frame and the frame's size, release and deadline constraints are updated accordingly. Otherwise, a new frame is created for the message. This is repeated until all messages are bound to frame windows. Thereby, the threshold value controls whether the algorithm produces a lot of small (in terms of its size in bytes) frame windows or a small number of large frame windows.

The outer loop iterates over a range of thresholds from 0.1 to 1.0 in customizable increments. For every value the inner loop is invoked and then the Comm Scheduler Plug-in is called to evaluate the schedule based on the returned metric of the `scheduleFrameWindows()` method. The strategy eventually returns the set of frame windows which yields the highest metric value.

### 4.3.3. Genetic Frame Generator

The second strategy we developed for mapping messages to frame windows facilitates a genetic algorithm, which is commonly used to solve scheduling problems [37]. Instead of using a heuristic metric and deciding message by message if it should be added to one of the already existing frames, we formulate the optimization problem so that it can be solved by running a genetic algorithm. We found that such algorithms have already been successfully applied in the field of communication scheduling, for example for the FlexRay protocol [38]. The approach described in [38] allocates a set of tasks to nodes which are connected via a FlexRay bus [34] so that various constraints concerning task deadlines and message response and freshness times are met. Our application of the genetic algorithm differs as our task to node mapping is supplied by the user and our all constraints are directly derived from the LETs of the tasks. Also note that as long as those LET-imposed constraints are met the observable behavior of the system does not depend on when exactly a message is sent. Furthermore our approach is not restricted to a specific communication protocol as it is located in the platform independent part of our scheduling framework.

In general, a genetic algorithm requires two things to be defined:

- a *genetic representation* (DNA) of the solution domain
- a *fitness function* to evaluate the solution domain

In our case the solution domain are all possible message to frame assignments where the number of messages is fixed for a given system and the number of frames can vary from 1 up to the number of messages. Consequently, we use an array of integer values with length equal to the number of messages as DNA encoding. We assign each message to a frame by assigning an integer frame number to each message, i.e. DNA[i] = j associates frame j to message i. An advantage of this genetic representation is that every message is assigned to a frame and that the number of frames can vary between 1 up to the total number of messages. One major disadvantage however is that it is possible to create invalid solutions as there are restrictions on what messages can be assigned to the same frame. In order to improve the result when performing crossover of two DNAs, we apply a normalization algorithm every time the DNA is changed, which is after initialization and when applying crossover and mutation. The normalization purges unused frame numbers and sorts frames by how many messages are assigned to it. It does so by giving the number 0 to the frame with the most messages assigned to it and so on. Consider the following DNA example:

```
5 5 5 3 2 3
```

Our normalization step would transform this sequence to

```
0 0 0 1 2 1.
```

The fitness function calculation is actually divided into two steps. As first step we apply a fast check on a solution by checking two basic criteria every valid solution must fulfill. Those are that a frame only is allowed to contain messages from the same sender node and that no messages with conflicting timing requirements are assigned to the same frame. Only when this check passes we apply our more sophisticated and also much more expensive fitness calculation function. We decided that a reasonable fitness function depends on what communication protocol is actually used to take into account its specific properties. Consequently, the fitness calculation is obtained by running the Comm Scheduler Plug-In and using the return value of the schedule function `scheduleFrameWindows()` as described above. This function also takes into account whether the assigned frame timings lead to a

feasible task schedule on all nodes involved. In both steps, negative fitness values indicate solutions that are unschedulable.

A generic genetic algorithm works as follows:

1) Generate initial population
2) Evaluate the fitness of each individual in the population
3) Repeat until termination criteria is met:
    a) Select fittest individuals to reproduce
    b) Breed new generation through crossover and mutation (genetic operations) and give birth to offspring
    c) Evaluate the individual fitness of the offspring
    d) Replace worst ranked part of population with offspring

In step 1 we simply use a random initialization, i.e. we assign each message a random frame with a number from 0 to (number of messages - 1) and apply our DNA normalization as proposed above.

The fitness evaluation in step 2 and step 3c requires that the DNA of each individual is converted to a list of frames with messages assigned according to the DNA frame to message mapping. Only after that conversion the fitness evaluation can be applied.

As termination criteria in step 3 we use a fixed number of generations that can be specified by the user. Other feasible termination criteria would be a timeout or a termination if the mean or maximum fitness does not increase anymore during a certain number generations.

The crossover and mutation step 3c is performed by selecting a random crossover point and creating a new DNA by using two individuals out of the pool of fittest individuals as selected in step 3a. This is done by copying the DNA of one individual up to the crossover point and the DNA of the another individual from there on. Mutation is applied during this recombination process by replacing a DNA element by a random number between 0 and (number of messages - 1) at a certain probability. This probability is called the mutation rate in the context of genetic algorithms. It must be high enough to enable sufficient exploration of the solution space but must not be too high as this would lead to too much destruction of good DNA sequences. We found a mutation rate of about 1 percent to be a feasible compromise.

**Results**

We tested the proposed genetic algorithm based scheduling by generating a random set of TDL modules which exchange data with each other and are distributed across a set of nodes. The algorithm was able to come up with a valid solution in all cases in which the heuristic approach found a valid schedule. With a pool of 100 individuals it typically took only a few generations until at least one valid solution was found. After about 50 generations, the fitness of the best individual reached the level of the heuristic strategy. We observed that when we increased the number of messages in the system the genetic algorithm had problems finding a solution as then the number of generations necessary to obtain at least one valid solution increased considerably. For an example with 50 messages it took almost 100 generations to encounter the first valid solution. However, it is important to note that for the fitness calculation during these 100 generations only the fast fitness check was needed, which is significantly less expensive than the one used to evaluate valid solutions. In such

cases the number of valid solutions that pass the fast scheduling check is small compared to the whole solution domain.

When comparing results of test cases with less than 40 messages with the results of the old, heuristic approach, we observed that we get similar solutions after less than 100 generations using a population size of 100 individuals. We consider this as evidence that the heuristics can indeed be replaced by the use of the proposed genetic algorithm and we are confident that this also holds for cases with more than 40 messages with an improved DNA representation, which restricts the solution domain so that the number of invalid solutions is minimized.

### 4.3.4. Comm Scheduler Plug-In

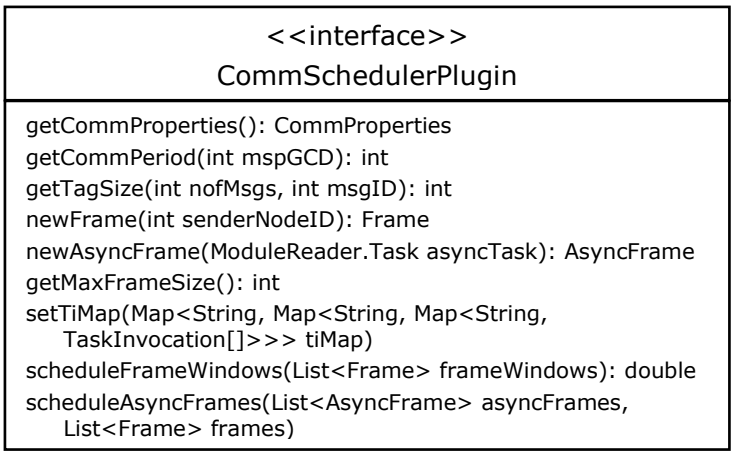


**Figure 31. Interface CommSchedulerPlugin**

This section contains the full specification of the Comm Scheduler Plug-In, i.e. of the `CommSchedulerPlugin` interface as depicted in Figure 31. It lists all interface functions even though some of them have already been discussed in detail above. The interface is used to abstract from a concrete communication protocol. It models aspects relevant to communication scheduling and is most notably used to provide various protocol-specific properties and methods which assign concrete timings to abstract synchronous and asynchronous frames. Example plug-ins implementing the interface are presented as part of chapter 1.

| | |
|---|---|
| **public** | `CommProperties getCommProperties()`<br>Returns an object of type `CommProperties` appropriate for the specific communication protocol the plug-in implements. Basic properties such as the minimum and maximum communication cycle length and the minimum and maximum size of a frame in bytes are mandatory fields already present in the abstract class `CommProperties`. |

| | |
|---|---|
| **public int** | `getCommPeriod(int mspGCD)`<br>This function enables plug-ins to define the communication period so that it obeys platform-specific restrictions. The returned communication period must be an integer divider of mspGCD, which is the maximum possible period calculated as the GCD of all mode periods and mode switch periods of the sending modules of a distributed TDL system. |
| **public int** | `getTagSize(int nofMsgs, int msgID)`<br>Returns the number of bytes required for the tag of a message. The size is calculated based on the total number of messages and a specific message ID. These two parameters enable either a fixed number of tag bytes depending on the total number of messages as well as a variable length encoding depending on the message ID. |
| **public** `Frame` | `newFrame(int senderNodeID)`<br>This is a factory method to create `Frame` objects used for the list of frames in the Comm Scheduler. For this purpose, either the default class `Frame` can be used or a subclass of it that can contain communication platform specific properties or methods. |
| **public** `AsyncFrame` | `newAsyncFrame(ModuleReader.Task asyncTask)`<br>This method is analogous to the `newFrame()` factory method above, but for asynchronous frames and consequently returns an `AsyncFrame` or a subclass of it. |
| **public int** | `getMaxFrameSize()`<br>Returns the maximum frame payload size in bytes, which may for example depend on the length of the communication period or other properties and constraints. |
| **public void** | `setTiMap(Map<String, Map<String, Map<String, TaskInvocation[]>>> tiMap)`<br>This method passes information about task invocations to the plug-in. They are required for a performance optimization as their deadlines are updated according to when the corresponding messages are scheduled during testing various communication schedules. Typically, the `tiMap` is split and passed on to the `CommLayer` of the individual nodes which are responsible for task schedule generation. |

| | |
|---|---|
| **public double** | `scheduleFrameWindows(List<Frame> frameWindows)`<br>This core function of the plug-in interface assigns concrete timings to the frame windows passed by the Comm Scheduler. It sets the start and end time of the `Frame` objects and might also alter the list by merging frames. The calculated timings reflect the requirements of the communication protocol the plug-in represents and for example depends on the data rate, the frame encoding and inter-frame gaps. The return value of type `double` indicates how good a solution is on a specific communication protocol. It is negative when the set of frames is not schedulable at all and between 0 and 1 if it is, where a better schedule returns a higher number. |
| **public void** | `scheduleAsyncFrames(List<AsyncFrame> asyncFrames, List<Frame> frames)`<br>Performs the mapping of the asynchronous frames to communication protocol specific IDs which are set for every `AsyncFrame`. Asynchronous frames are sorted by descending priority of their corresponding events. The list of already scheduled synchronous frames is also passed to enable transferring asynchronous frames as synchronous one by adding them to this list. |

**Table 9. Methods of interface CommSchedulerPlugin**

# 5. Platform-Specific Adaptations for FlexRay

This chapter presents platform-specific implementations of plug-ins to the TDL Comm Layer framework of the TDL Runtime System as well as to the code and schedule generation framework. We describe the prototyping hardware and node platform plug-ins for two networked target platforms, namely the Node Renesas provided by DECOMSYS (now Elektrobit), and the dSPACE MicroAutoBox. Both platforms are widely used for prototyping of embedded systems in the automotive industry. They are connected via a FlexRay communication bus, whose global time base is used to synchronize time-triggered TDL activities and for which we present a communication scheduling plug-in.

By using our framework and the adaptations for the FlexRay platforms and communication bus, we are able to automatically generate a fully functional FlexRay system that runs arbitrary TDL components. The only requirement is that both CPU power and network bandwidth are sufficient to execute all TDL modules – otherwise no code is generated.

In the next two sections we introduce the FlexRay protocol and describe the prototyping hardware. Then we detail all adaptations and plug-ins required to map TDL modules to a distributed FlexRay system. The chapter is concluded with a case study involving three FlexRay nodes.

## 5.1. The FlexRay Protocol

FlexRay [34] is a time-triggered TDMA communication protocol intended for automotive applications. Its development started in 2000 and concluded with specification version 3.0 in 2009. It is designed as successor to the widely used CAN protocol and additionally for safety critical applications such as steer-by-wire systems. FlexRay operates at a speed of 10 MBit/s and thus has a significantly higher bandwidth than other common field bus protocols such as CAN or LIN. Furthermore, it operates collision-free within the time-triggered part and therefore exhibits predictable behavior and includes a distributed clock synchronization service. The first use of FlexRay in automotive series production was in 2006 for an optional adaptive damping system in the BMW X5. Its introduction for vehicle core functions was in 2008 in the BMW 7 Series.

A FlexRay communication cycle as depicted in Figure 32 has a fixed length specified at design time. The cycle constantly repeats itself and consists of a mandatory *static segment*, representing the time-triggered aspect of FlexRay, and an optional *dynamic segment*. Furthermore, a FlexRay cycle may contain a so-called *symbol window* which is used to transmit a single symbol out of a selection of three symbols as pre-defined by the FlexRay specification. A cycle period is concluded with a network idle time (NIT) in which no data is transmitted. This pause is for example required for the implementation of the distributed clock synchronization mechanism.

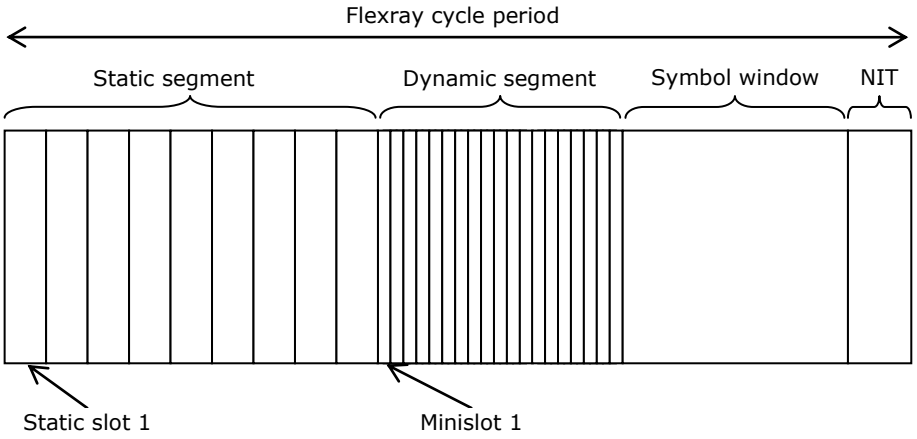


**Figure 32. FlexRay cycle layout**

The static segment is divided into equally sized *static slots* which are statically assigned to specific nodes in the cluster for sending and thus guarantees uninterrupted transmission. The optional dynamic segment also has a static size, but it is dynamically allocated to different nodes upon runtime. This is accomplished via a priority mechanism using so-called *minislots*, which partition the dynamic part and act as small placeholders which are enlarged at runtime when the node assigned to them transmits data. This means that for minislots at the end of the dynamic segment, i.e. those with the lowest priority, there might not always be enough space for their transmission if a lot of higher priority minislots are used. The size of the dynamic segment and the maximum data size allowed to be transferred determine how many minislots are guaranteed to be usable for data transmission in every cycle.

Every FlexRay cluster provides two separate communication channels which share the same basic layout concerning the size of the static and dynamic segment. However, individual slots can either be used simultaneously to increase fault tolerance or independently, also by different nodes, to increase data throughput. Furthermore, FlexRay implements a distributed clock synchronization protocol which ensures that all nodes agree on a global cluster time. To accomplish this, all nodes in the cluster measure the difference of their local clock to the clocks of 2 to 15 designated *sync nodes*. Those nodes transmit a specified sync frame on both FlexRay channels. Out of the measured difference every node calculates offset and rate correction values which are subsequently applied so that all clocks in the system stay in sync.

The FlexRay protocol is typically implemented on *dedicated communication controllers* which autonomously handle the transmission of data via the bus. They are configured with a set of global cluster parameters and local node parameters which completely specify the FlexRay network communication behavior. A FlexRay controller handles all network tasks including startup and time synchronization and provides message buffers as interface to the host CPU, which hold the data of individual slots and minislots to be transmitted and received. The FlexRay specification [39] describes the FlexRay protocol in detail, including all configuration parameters on cluster and node level and their constraints.

For the exchange of data describing a FlexRay cluster, i.e. all cluster and node parameters, slot assignments and signal definitions, the FIBEX data format can be used. FIBEX is an abbreviation for Field Bus Exchange and is an XML format designed

to describe information regarding message-oriented bus communication systems such as CAN, LIN and FlexRay. It is commonly used in the automotive industry as it simplifies the data exchange across tools from multiple manufacturers. The "FIBEX Expert Group" consists of representatives of BMW, Bosch, Daimler-Chrysler, Elektrobit, dSPACE, National Instruments and Vector Informatik, among others.

## 5.2. Hardware Platforms

This section presents the two FlexRay-based hardware platforms we use to demonstrate the application of our code and schedule generation framework.

### 5.2.1. Node Renesas

The NODE<RENESAS> Starter Kit is a FlexRay prototyping package by DeComSys (now Elektrobit) which we use in version R2.0.2. It contains two prototyping boards with a Renesas M32C/85 host CPU featuring a 24 MHz clock, 2MB of RAM, and 2MB of Flash memory, and a dedicated Bosch E-Ray FlexRay controller. Furthermore the package consist of a USB programming interface and a software package including the GNU C compiler, a linker and a make tool supporting the Renesas M32C CPU, a simple operating system, and drivers for the FlexRay controller.

Figure 33 shows a typical Node Renesas hardware setup. Both nodes are connected to the power supply at the front of the node and to the FlexRay bus via a connector at the back panel. Analog I/O is connected via a connector at the back named "AIO". We use analog outputs to visualize actuator output on an oscilloscope. The USB programmer is connected to the blue socket at the front of the boards. The front panel of the board also contains digital I/O in form of 4 LEDs and 4 buttons. Also at

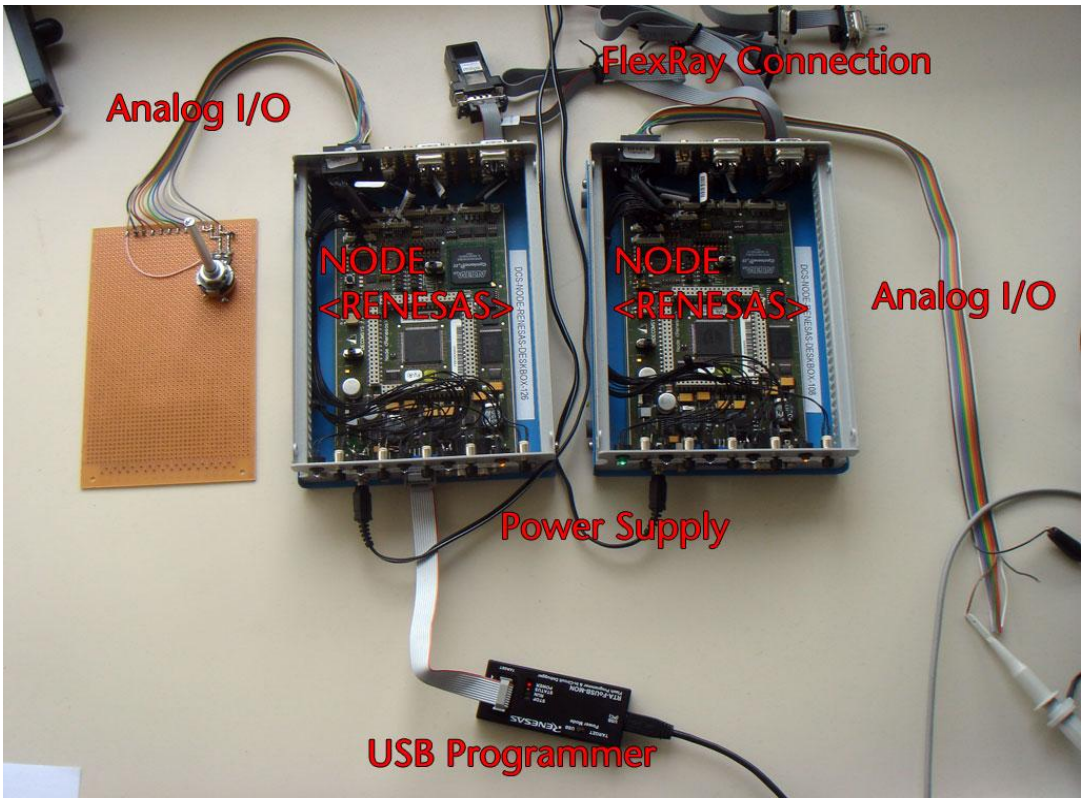


**Figure 33. Node Renesas hardware overview**

the front panel there is a RS232 serial port for debugging purposes.

The Node Renesas is shipped with a custom operating system called AES (Application Execution System). It is an ANSI-C software library which enables executing periodic, time-driven application tasks. AES is pre-runtime configurable and supports synchronization with the FlexRay communication system. Its configuration describes task executions by means of a dispatch table where every entry specifies a task invocation time. Instead of the typical `main()` function which is normally the initial entry point of a C program, AES uses three hooks for initialization, idling and shutdown.

The following shows a complete sample configuration for AES which invokes the function `periodicTask()` every 5ms by executing it twice within an application period of 10ms. Note that all entities are mandatory for both single node and distributed systems. The application cycle length `skAES_ApplCycleLenUs` is the length of the dispatcher round, i.e. within this time the dispatch table is executed exactly once. It can differ between nodes but always needs to be a $2^n$ multiple of the FlexRay cycle period.

```
static void periodicTask(void) {
  [...]
}

/* This task runs when there is no running time-triggered task. */
void skAES_ApplIdleTask (void) {}

/* This function is called at system start up */
void skAES_ApplInitHook (void) {}

/* This function is called when the system (skAES) shuts down. */
void skAES_ApplShutdownHook (skAES_ErrorType skAES_ErrNo) {}

/* AES dispatch table */
const skAES_TaskDescriptionType skAES_TaskDescription[] = {
  /* Offset in us, Run only if synchronized with cluster, Task function */
  {0U, SK_AES_FALSE, periodicTask},
  {5000U, SK_AES_FALSE, periodicTask}
};

/* Number of tasks in the dispatch table */
const uint8 skAES_NumberOfTasks = sizeof(skAES_TaskDescription) /
  sizeof(skAES_TaskDescription[0]);

/* Length of the application cycle */
const skAES_TimeType skAES_ApplCycleLenUs = 10000U;

/* FlexRay synchronization parameters */
const skAES_TimeType skAES_MaxDecreaseUs = 50U;
const skAES_TimeType skAES_MaxIncreaseUs = 50U;
const skAES_SyncModeType skAES_SyncMode = SK_AES_SYNCMODE_HARD;
```

In addition to the AES operating software, further libraries are provided by the Node Renesas Starter Kit. The COMMSTACK FlexRay controller driver provides frame-based access to FlexRay communication controllers. It requires corresponding COMMSTACK configuration files which mainly contain FlexRay cluster and node parameters and configuration of the FlexRay controller buffers. The OS Synchronization Handler (OsSh) is used to synchronize the AES operating system to the time base of a running FlexRay cluster and the Target Platform Infrastructure (skTPI) provides access to basic hardware devices such as the front panel LEDs and buttons.

### 5.2.2. MicroAutoBox

The MicroAutoBox (see Figure 34) is a rapid prototyping platform by dSPACE which is commonly used in the automotive industry. We used the model MicroAutoBox 1401/1505/1507. Its main processing unit is a Power PC 750 FX CPU running at 800


**Figure 34. dSPACE MicroAutoBox**

Mhz and is equipped with 8 MB of RAM. It has a dedicated digital I/O subsystem based on a Motorola 68336 microcontroller which is connected via a dual-port memory with the master CPU. The MicroAutoBox has the following set of external interfaces:

- 4 parallel A/D converters multiplexed to 4 channels each with 12-bit resolution and another 16 A/D channels with 10-bit resolution
- D/A conversion providing 8 D/A channels with 12-bit resolution
- Bit I/O unit providing 16-bit input, 10-bit output, and 16-bit input/output with bitwise selectable direction
- Multiple PWM (pulse width modulation) inputs and outputs suitable for chassis and engine control applications
- Interrupt handling providing 4 external hardware interrupts lines
- 4 CAN controllers
- 2 LIN controllers
- 2 serial interfaces
- FlexRay support via 2 optional IP modules, providing 2 FlexRay channels each. We equipped our MicroAutoBox with one dSPACE DS4330 IP Module containing a PFR4300 FlexRay communication controller

As the MicroAutoBox is designed for rapid prototyping in the automotive context, it is typically programmed by using a MATLAB/Simulink block set called the Real-Time Interface (RTI) and subsequent code generation. However, it can also be

programmed directly in C by using the interface of its real-time operating system called the dSPACE Real-Time Kernel (RTKernel). It comprises functions for task management, task scheduling, and interrupt handling. Furthermore, there is also a documented API for all input/output devices and the FlexRay interface.

The MicroAutoBox is shipped with a build environment consisting of a C compiler by Microtec and a make tool. Compiled binary files are sent to the MicroAutoBox via a proprietary programming interface connected using a PCMCIA card. For that purpose, dSPACE provides the ControlDesk utility.

## 5.3. TDL Comm Layer Framework Plug-Ins

The TDL Comm Layer framework described in section 3.3 requires platform-specific plug-ins that are specific to a communication protocol and hardware platform. The minimum required functionality consists of platform initialization code and the sending and receiving of TDL Comm Layer buffers. Thus, the following three functions, whose prototypes are already present in `tdl_comm.h`, must be implemented in the file `tdl_comm_<platform>.c`:

```
void tdl_comm_init_platform(void);
void tdl_comm_receiveBuffer(int bufferIndex, int size);
void tdl_comm_sendBuffer(int bufferIndex, int size);
```

FlexRay controllers autonomously take care of frame transmission according to the communication schedule specified by the controller configuration. Frame sending and receiving therefore only requires copying data between the TDL Comm Layer buffers and the FlexRay controller, which must to happen before or after the actual transmission of the frame respectively. Another important issue is the synchronization with the FlexRay bus, whose implementation differs among platforms. Prototypes of such additional platform-specific functions must be added to the `tdl_comm_<platform>.h` header file.

### FlexRay Legacy Signal Access

In addition to the get and put methods implemented in the TDL Comm Layer framework for handling TDL data types, we also add functionality to send and receive non-TDL or legacy signals on the FlexRay bus. As those functions should be usable by all FlexRay platforms, we put them in a separate file `tdl_comm_flexray.c` and corresponding header file `tdl_comm_flexray.h`. The signature of the get and put functions are as follows:

```
void tdl_comm_get<TDLType>Signal(int bufferIndex, int bitPosition, int
     size, char isBigEndian, tdl_<TDLType>char* data);
void tdl_comm_put<TDLType>Signal(int bufferIndex, int bitPosition, int
     size, char isBigEndian, tdl_<TDLType>char* data);
```

For every TDL type, which are `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, and `double`, there are corresponding put and get methods to access legacy signals transmitted on the FlexRay bus. The signals are identified by buffer index, bit position, size and endianness. Those parameters can for example be read from a FIBEX file describing an existing FlexRay cluster.

### Node Renesas

The TDL Comm Layer framework plug-in for the Node Renesas is implemented in the files `tdl_comm_noderenesas.c` and `tdl_comm_noderenesas.h`. The platform initialization code configures the FlexRay controller and initializes the FlexRay

synchronization handler. The Comm Layer buffer send and receive functions are implemented by calling the FlexRay driver functions `TDDLL_TxFrameByID` and `TDDLL_RxFrameByID` respectively. The following two specific elements are added to the `tdl_comm_noderenesas.h` header file:

```
void tdl_comm_noderenesas_syncFlexRay(void);
```

The function `tdl_comm_noderenesas_syncFlexRay` must be called periodically at runtime. It synchronizes the local clock of the AES operating system to the FlexRay bus time by using the FlexRay synchronization handler supplied by DeComSys.

```
extern TDDLL_ConfigType* tdl_comm_noderenesas_commstackConfig;
```

This variable holds the FlexRay controller configuration that must be provided for the FlexRay driver delivered with the Node Renesas prototyping kit.

#### MicroAutoBox

The files `tdl_comm_mabx.c` and `tdl_comm_mabx.h` implement the TDL Comm Layer framework plug-in for the MicroAutoBox. For the purpose of synchronization of the TDL Runtime System to the FlexRay bus, the plug-in provides two basic functions, which are facilitated by the synchronization mechanism implemented in the generated glue code for the MicroAutoBox platform:

```
char tdl_comm_mabx_getCommCycle(void)
ts_timestamp_type tdl_comm_mabx_getTS(void)
```

The first function returns the number of the current FlexRay communication cycle in the range from 0 to 63 while the second returns a timestamp indicating the beginning of the next FlexRay cycle. Both values are obtained by using the dSPACE FlexRay API.

### 5.4. TDL:VisualDistributor Interfaces

Code generation for concrete platforms requires information on the deployment of modules to nodes as well as additional platform-specific information, such as the worst-case execution time (WCET) of tasks and the mapping of sensors and actuators to specific hardware devices. In the TDL tool chain this data in managed by the TDL:VisualDistributor tool, which has a graphical user interface but may also run

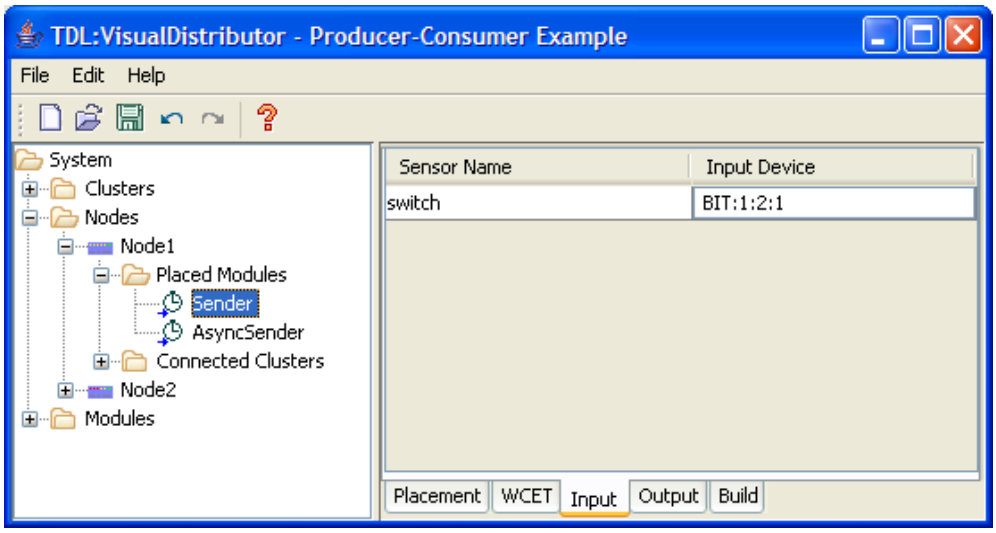


**Figure 35. TDL:VisualDistributor property page example**

in batch mode. The tool acts as front-end to the code and schedule generation framework and provides the Java interfaces described in this section which the node and cluster platform plug-ins must implement so that they can access the data model of the TDL:VisualDistributor. From this data model plug-ins can for example retrieve the deployment of modules to nodes. Furthermore, the interfaces enable plug-ins to provide custom, editable properties. Figure 35 depicts an example for such a custom property page for configuring the input device of a sensor of a module placed on a specific node.

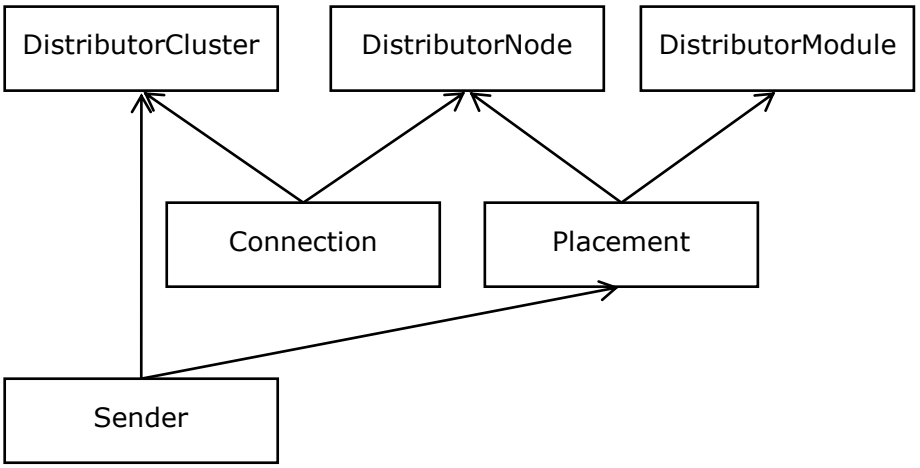


**Figure 36. TDL:VisualDistributor data model classes**

Figure 36 shows a UML diagram of the TDL:VisualDistributor data model which is used to describe the topology of a TDL system. It consists of six classes which all extend the `DistributorObject` class. The classes `DistributorCluster`, `DistributorNode`, and `DistributorModule` represent a communication cluster, a node, and a TDL module respectively. A `Placement` assigns a module to a node, a `Sender` indicates on which cluster a placement sends and a `Connection` links a node to a cluster.

The interface `PropertiesProvider` (see Figure 37) is implemented by node and cluster platform classes in order to provide property pages specific to particular distributor objects. A `PropertyPage` essentially is a table with two columns, containing the name of the property on the left and the value of the property on the right. It extends the Java Swing class `AbstractTableModel` which provides a table model to a `JTable`. The value of the property can be represented by various Java Swing elements, such as a text field, a drop down box or a file chooser. In order to obtain property pages for elements of its data model, the TDL:VisualDistributor calls the function `getPropertyPages()` for every `DistributorObject`:

```
public PropertyPage[] getPropertyPages(DistributorObject dob,
                                       PropertyPage[] base)
```

Individual platforms then either directly return the existing `PropertyPage` array `base` if they do not add any property pages or add or modify `PropertyPage` objects which are then displayed as part of the TDL:VisualDistributor's user interface associated with the specific `DistributorObject dob`.

The interface `DistributorNodePlatform` (see Figure 37) serves to associate a platform plug-in to a node in the distributor model. It extends the interfaces `NodePlatform` and `PropertiesProvider` and adds a set and get a method for a `DistributorNode` object.

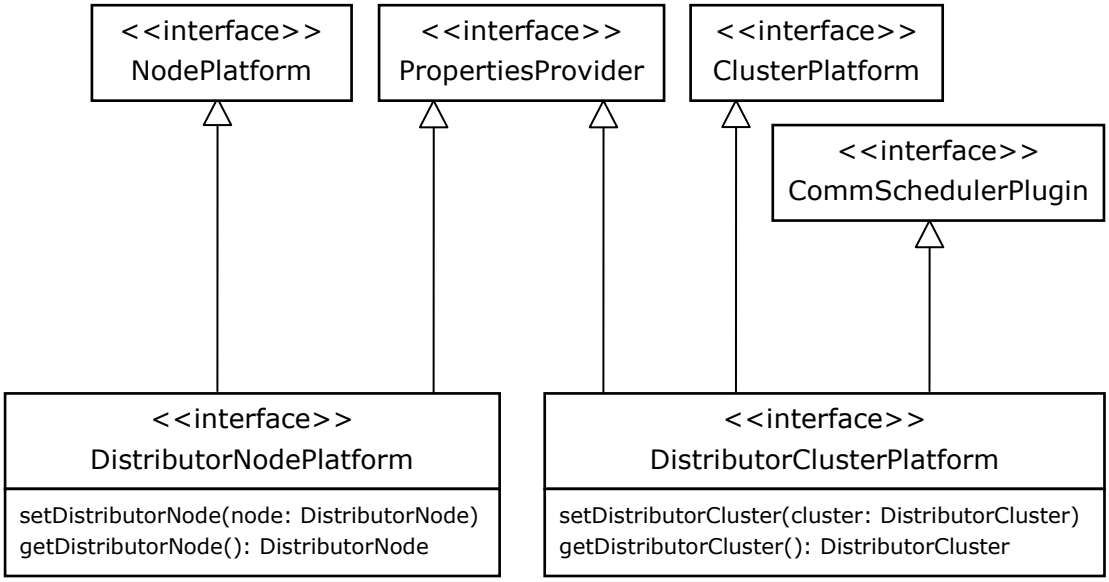


**Figure 37. TDL:VisualDistributor interfaces**

In a similar way, the interface `DistributorClusterPlatform` (see Figure 37) links a cluster platform and Comm Scheduler Plug-in to a cluster in the distributor data model. For that purpose, it extends the interfaces `ClusterPlatform`, `CommSchedulerPlugin` and `PropertiesProvider`. Furthermore, it provides set and get methods for a `DistributorCluster` object.

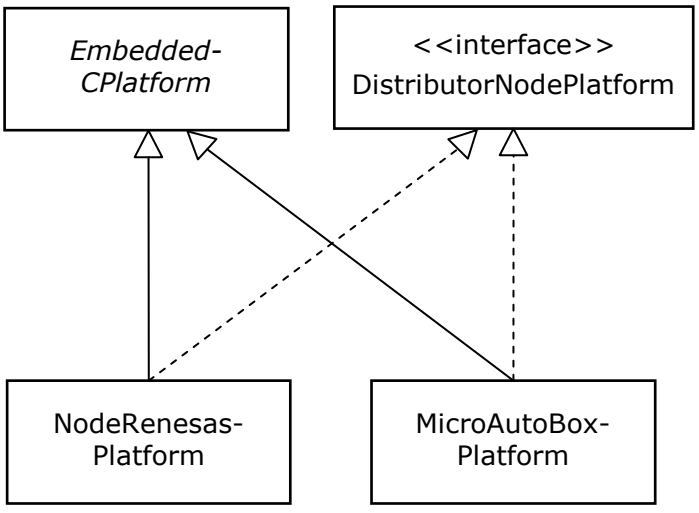


**Figure 38. Prototyping hardware node platforms**

## 5.5. Node Platform Plug-Ins

This section details the platform-specific code generation for the Node Renesas and the MicroAutoBox. It focuses on standalone or single node systems, i.e. on how TDL modules are executed on these platforms without taking distribution into account. Support for distribution is added using communication layer implementations for FlexRay, which we describe in the next section.

Figure 38 shows a UML class diagram depicting the `NodeRenesasPlatform` and `MicroAutoBoxPlatform` plug-ins we developed for our prototyping hardware platforms. As both platforms are programmed in C, the plug-ins are based on the class `EmbeddedCPlatform` which we introduced in chapter 1. The interface `DistributorNodePlatform` links the plug-ins to the TDL:VisualDistributor so that they can query its data model and provide user-editable properties.

The plug-ins are tailored to the concrete hardware platforms including their operating systems, compilation environments and input/output device drivers. The generated code must guarantee that synchronous activities are carried out as specified by TDL modules. Concerning asynchronous activities, it depends on the platform and especially on its operating system how and to what extent asynchronous activities can be implemented. For example, not every platform supports hardware interrupts.

In this chapter we again use the producer-consumer example with synchronous and asynchronous producer and consumer modules as introduced in 4.2. For *Node1* we generate code using the MicroAutoBox plug-in and for *Node2* using the Node Renesas plug-in.

### 5.5.1. Node Renesas Platform

The class `NodeRenesasPlatform` extends `EmbeddedCPlatform` and adds all elements required to run TDL applications on the Node Renesas prototyping platform. It generates only code which is independent of the concrete `CommLayer` associated with it. `NodeRenesasPlatform` implements the `DistributorNodePlatform` interface that enables it to provide user editable properties to the TDL:VisualDistributor and to access its data model. For that purpose, it provides three property pages via the `getPropertyPages()` function of `PropertiesProvider` which are linked to a `Placement`. These are `NodeRenesasInputPropertyPage` for the assignment of input devices to TDL module sensors, `NodeRenesasOutputPropertyPage` for the assignment of output devices to TDL actuators and `NodeRenesasBuildPropertyPage` for build options as the location of the functionality code source directory. Furthermore, `NodeRenesasNodePropertyPage` (see Figure 39) is linked to a `DistributorNode` and provides properties such the install location of the Node Renesas Starter Kit.

In order to meet constraints concerning the application cycle length of the AES operating system, the method `getStepPeriod()`, introduced in `EmbeddedCPlatform`, is overridden. The function is used as input for the task scheduler and calculates the step period of the TDL Machine, i.e. the period in which it must be invoked so that it can fulfill the timing requirements of the modules mapped to a node. For distributed systems involving a Node Renesas, the AES application cycle length must be a $2^n$ multiple of the FlexRay cycle period of the system. Furthermore, the application period must be an integer multiple of the step period so that the TDL Machine can be invoked using one or multiple entries in the AES dispatch table, which specifies all task invocations within one application cycle.

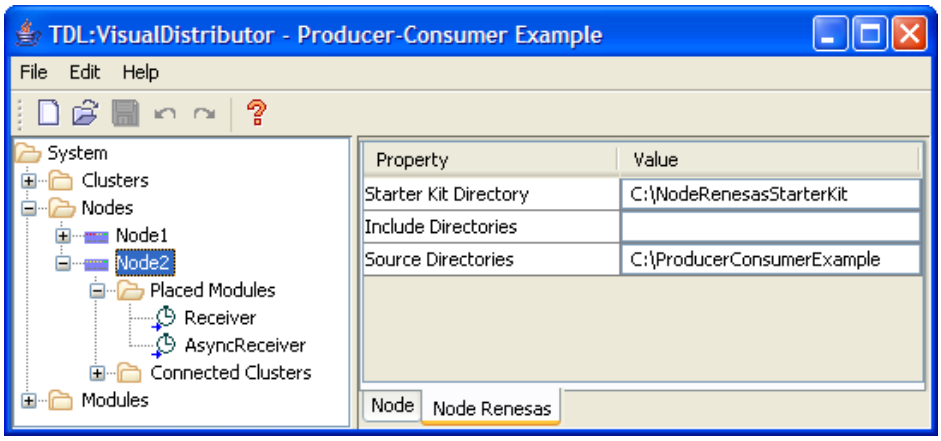


**Figure 39. Node Renesas node property page**

Unfortunately, there is no documented interface in the Node Renesas' AES operating system to access the CPU's interrupt lines. As a result, this trigger mechanism for asynchronous activities is not available. The plug-in checks this limitation and yields an error message when a TDL module contains an interrupt trigger and is deployed on a Node Renesas platform.

### C Module Body File

- Includes (`emitC_Includes()`)

**Node2/Receiver_.c**
```
#include <AnalogIO.h>
#include <skTPI.h>
```

Includes for `skTPI.h` and `AnalogIO.h` are added if they are required by the emitted hardware drivers.

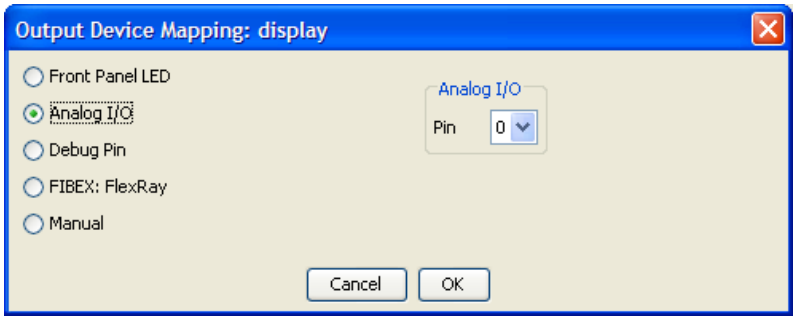


**Figure 40. Node Renesas platform output device mapping dialog**

- Hardware drivers (`emitC_DeviceDrivers()`)

**Node2/Receiver_.c**
```
static void Receiver_setDisplay(tdl_int Receiver_display) {
  AnalogIO_set(0, Receiver_display);
}
```

Via the TDL:VisualDistributor the `NodeRenesasPlatform` provides the ability to assign sensors and actuators to concrete hardware components and I/O pins. Figure 40 shows the dialog used to map a TDL actuator to an output device. The available hardware consists of the 4 LEDs and 4 buttons on the front panel

of the Node Renesas and back panel pins for 8-bit analog in- and output. Depending on the selected mapping the appropriate code is generated in the C module body file from which the emitted functions are also called by the driver code emitted by `CPlatform`.

### C Main Body File

- Includes (`emitMainC_Includes()`)

**Node2/tdl_main_.c**
```
#include <AnalogIO.h>
#include <skTPI.h>
```

The added includes consist of an AES header file (`skAES.h`) and headers required for one-time initializations of hardware drivers (`AnalogIO.h`).

- Timer trigger for asynchronous activities (`emitMainC_AsyncTimerFunctions()`)

Unfortunately, AES provides no access to the CPU timer. As an approximation, we implemented the timer trigger so that it is activated after a certain number of TDL Machine invocations. This corresponds to the semantic of the timer trigger to be activated after *at least* the specified time. We emit a function `handleAsyncTimers(void)` which is called every step period and maintains an individual counter for each timer period. When a timer expires, it is reset and the corresponding `tdl_async_enqueue()` function is called.

- Periodic task (`emitMainC_PeriodicTask()`)

The periodic task (`static void periodicTask()`) is the function that is called periodically once per step period, i.e. the GCD of all periodic actions the TDL Machine needs to perform. The body of the periodic task function is emitted by a `NodeRenesasCommLayer` hook as it depends on whether distribution is required or not. In the single node case, the periodic task consists of calling the TDL Machine step function and of the `handleAsyncTimers()` function just described above.

- AES operating system hook functions (`emitMainC_AESFunctions()`)

**Node2/tdl_main_.c**
```
void skAES_ApplIdleTask (void) {
  for (;;) {
    int index = tdl_async_dequeue();
    if (index >= 0) {
      executeAsyncSequence(index);
    }
  }
}

void skAES_ApplInitHook (void) {
  AnalogIO_init(); /* Initializes sensor/actuator io. */
  tdl_comm_noderenesas_commstackConfig = &TDDLL_Config_Node2_MCU;
  tdl_comm_init(&commConfig);
  tdl_async_init(asyncs, 1); /* asyncs, nofAsyncs */
  tdl_machine_init(&modules[0], 4, 5000); //modules, nofModules, partitionPeriod
}

void skAES_ApplShutdownHook (skAES_ErrorType skAES_ErrNo) {
}
```

The AES operating system provides no support for task preemptions or task priorities. However, it does have a so-called *idle task* which is specified by

implementing the hook function `skAES_ApplIdleTask`. The idle task runs when no time-triggered task is running. It is therefore suitable to execute asynchronous TDL activity sequences as its priority is lower than that of the TDL Machine, which is implemented as a time-triggered task inside the dispatch table. The idle task simply has to poll the priority queue in an endless loop and execute the corresponding activity when it is set active in the queue.

The initialization of the TDL Runtime System and optional hardware drivers is ensured by using the AES initialization hook `skAES_ApplInitHook`.

- AES operating system configuration constants (`emitMainC_AESConstants()`)

**Node2/tdl_main_.c**
```
const skAES_TaskDescriptionType skAES_TaskDescription[] = {
  /* Offset in us, Requires synchronous CS, Task function */
  {0U, SK_AES_FALSE, periodicTask},
  {5000U, SK_AES_FALSE, periodicTask},
};

const uint8 skAES_NumberOfTasks =
  sizeof(skAES_TaskDescription) / sizeof(skAES_TaskDescription[0]);
const skAES_TimeType skAES_ApplCycleLenUs = 10000U;
const skAES_TimeType skAES_MaxDecreaseUs = 50U;
const skAES_TimeType skAES_MaxIncreaseUs = 50U;
const skAES_SyncModeType skAES_SyncMode = SK_AES_SYNCMODE_HARD;
```

The AES operating system runs periodic tasks by use of a sequential dispatch table. The plug-in emits a dispatch table that invokes the `periodicTask` and therefore the TDL Machine exactly every step period.

### Make File

For every node a make file is emitted via the method `emitMake_Content()`. It uses generic make files provided by DeComSys for the Node Renesas and its AES operating system. The make file ensures that the whole collection of C code, which includes the module functionality code, module and stub module glue code, and the TDL Runtime System code, is compiled and linked correctly.

**Node2/Makefile**
```
ROOT ?= C:\RenesasStarterKit\ESW
DESTDIR = C:\Example

TARGET_APP += \
      Node2 \

CUSTOM_CFLAGS = -DPLATFORM_RENESAS -DTDL_DISPATCHED
CUSTOM_CFLAGS += -DTDL_DISTRIBUTED

REQUIRED_COMPONENTS += \
      $(ROOT)/SWP/Types \
      $(ROOT)/SWP/dcsCstFr \
      $(ROOT)/SWP/skTPI \
      $(ROOT)/SWP/skAES \
      $(ROOT)/SWP/OsSh \
      $(ROOT)/SWP/FrSh \

INCLUDE_DIRS += \

SOURCE_DIRS += \
      $(DESTDIR)/Nodes/Node2 \

include $(ROOT)/BuildFiles/arch_common/Makefile_base.mak
```

## 5.5.2. MicroAutoBox Platform

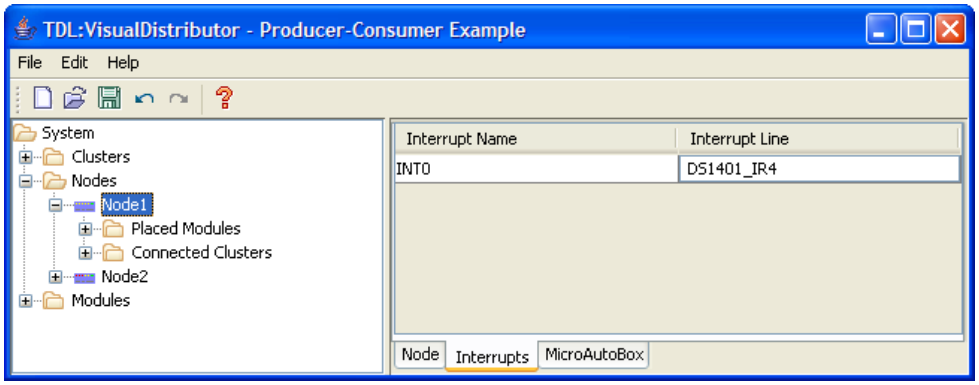


**Figure 41. TDL:VisualDistributor interrupt assignment**

The class `MicroAutoBoxPlatform` implements single-node code generation for the MicroAutoBox 1401/1505/1507 hardware. Through the `DistributorNodePlatform` interface it provides three property pages which are linked to a `Placement`, i.e. a module assigned to a node. These are `MABXInputPropertyPage` for the assignment of input devices to TDL sensors, `MABXOutputPropertyPage` for the assignment of output devices to TDL actuators and `MABXBuildPropertyPage` for build options such as the location of the module's functionality code. Furthermore, two property pages are linked to a `DistributorNode`: `MABXNodePropertyPage` provides options such as include and source directories and `MABXInterruptsPropertyPage` enables the user to assign the logical interrupt names in TDL modules to the hardware interrupt lines of the MicroAutoBox. See Figure 41 for a TDL:VisualDistributor screenshot depicting how to assign an interrupt line to the *INT0* interrupt used in the *AsyncSender* module of the producer-consumer example.

The features of the MicroAutoBox operating system are sufficient to implement asynchronous activities and all corresponding trigger types in a straightforward way. The execution of asynchronous activities is performed by a low priority task. The MicroAutoBox has a number of hardware interrupt lines which can be configured to be used as interrupt triggers for asynchronous activities. For timer triggers, individual tasks are scheduled with the corresponding timer period which call the enqueue function.

Unfortunately, the Microtec C compiler does not support the TDL type `long`. To check if a TDL module uses this type, we override a method of `CPlatform` which returns the C identifier of a TDL type. Before the overridden method returns the identifier via a super call, it aborts the whole code generation process by throwing an exception when it encounters the TDL type `long`. This example illustrates how the repeated subclassing approach employed by the code generation framework enables subclasses to influence the process according to specific requirements.

### C Body File

- Includes (`emitC_Includes()`)

  **Node1/Sender_.c**
  ```
  #include <Brtenv.h>
  ```

  Includes are added that are required for the hardware drivers (see below). The header file `Brtenv.h`, which stands for Base Real Time Environment, includes all further include files related to the MicroAutoBox platform.

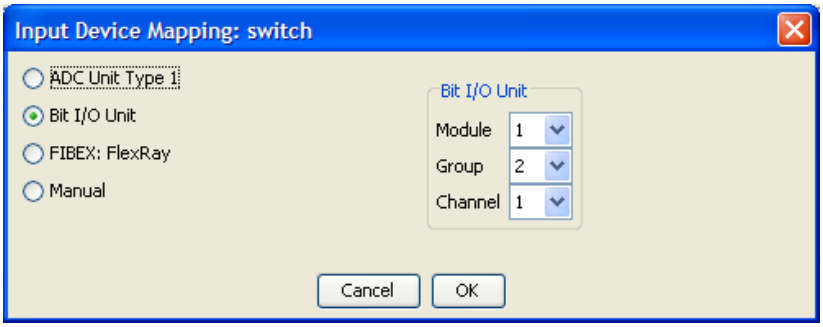


**Figure 42. MicroAutoBox platform input device mapping dialog**

- Hardware drivers (`emitC_DeviceDrivers()`)

**Node1/Sender_.c**

```
/* Device variables */
extern Int32 cTableIdx_1_2_get;
static UInt8 dioValue_1_2;
static unsigned long dioStep_1_2;

/* Getter and setter functions */
static void Sender_setDisplay(tdl_int Sender_display) {
  dac_tp1_write(DAC_TP1_1_MODULE_ADDR, 1, Sender_display);
}

static void Sender_getSwitch(tdl_boolean* Sender_switch) {
  if (dioStep_1_2 != tdl_machine_stepCounter) {
    dioStep_1_2 = tdl_machine_stepCounter;
    dio_tp1_bit_io_get_request(DIO_TP1_1_MODULE_ADDR, 0, cTableIdx_1_2_get);
    dio_tp1_bit_io_get(DIO_TP1_1_MODULE_ADDR, 0, cTableIdx_1_2_get,
                       &dioValue_1_2);
  }
  *Sender_switch = 1-((dioValue_1_2 >> 0) & 1);
}
```

`MicroAutoBoxPlatform` provides property pages to the TDL:VisualDistributor to configure the mapping of sensors and actuators of TDL module to actual hardware devices. Figure 42 depicts the dialog for input device mapping. The specified settings result in the generated code above for `Sender_getSwitch()` in the module glue code file.

### Main file

- Includes (`emitMainC_Includes()`)

**Node1/tdl_main_.c**

```
#include <Brtenv.h>
#include <rtkernel.h>
```

The included headers consist of the base include file of the MicroAutoBox libraries, `Brtenv.h`, and the header file of the RTKernel, `rtkernel.h`, which contains operating system functions for task scheduling and interrupt configuration.

- Timer trigger for asynchronous activities (`emitMainC_AsyncTimerFunctions()`)

  For every distinct period of timer triggers for asynchronous activities, a static function `timer<period>()` is emitted. In the main function every such function is then scheduled separately using the required period.

- Periodic task (`emitMainC_PeriodicTask()`)

  The periodic task (`static void periodicTask(rtk_p_task_control_block pTCB)`) is the function that is called periodically every step period. The step period equals to the GCD of all periodic actions the TDL Machine must perform. In order to match the signature required for task functions by the RTKernel, the periodic task has a task control block as argument. The body of the periodic task function is emitted by a `MicroAutoBoxCommLayer` hook, as it depends on whether distribution is required or not. In the single node case, it only consists of calling the TDL Machine step function.

- Main function (`emitMainC_Main()`)

**Node1/tdl_main_.c**

```
void main(void) {
  init();
  dac_tp1_init(DAC_TP1_1_MODULE_ADDR);
  dio_tp1_com_init(DIO_TP1_1_MODULE_ADDR, DIO_TP1_EC_STD_MODE);
  dio_tp1_bit_io_init(DIO_TP1_1_MODULE_ADDR, 0, 2, DIRMASK_1_2, 0);
  dio_tp1_bit_io_get_register(DIO_TP1_1_MODULE_ADDR, 0, &cTableIdx_1_2_get, 2);
  tdl_comm_init(&commConfig);
  tdl_async_init(asyncs, 1); /* asyncs, nofAsyncs */
  tdl_machine_init(&modules[0], 2, 5000); //modules, nofModules, partitionPeriod
  rtk_initialize();

  {
    rtk_p_task_control_block periodicTaskTCB;
    rtk_p_task_control_block syncTaskTCB;
    lastTSmit = -100;

    periodicTaskTCB = rtk_create_task(periodicTaskWrapper, 2, ovc_count,
                                      NULL, INT_MAX, 0);
    rtk_bind_interrupt(S_INTERVAL_A, 0, periodicTaskTCB, 0.0f, C_LOCAL,0, NULL);
    rtk_set_task_type(S_INTERVAL_A, 0,RTK_NO_SINT,rtk_tt_periodic,NULL,0.0f, 1);

    syncTaskTCB = rtk_create_task(syncTask, 50, ovc_count, NULL, INT_MAX, 0);
    rtk_bind_interrupt(S_INTERVAL_A, 1, syncTaskTCB, 0.0f, C_LOCAL, 0, NULL);
    rtk_set_task_type(S_INTERVAL_A, 1,RTK_NO_SINT,rtk_tt_periodic,NULL,0.0f, 1);
    rtk_it_task_register_rel(S_INTERVAL_A, 1, RTK_NO_SINT,0.0f,0,0.0050f, NULL);
  }

  ds1401_set_interrupt_vector(DS1401_IR4, handleInterruptINT0, SAVE_REGS_ON);
  ds1401_enable_hardware_int(DS1401_IR4);
  DS1401_GLOBAL_INTERRUPT_ENABLE();

  rtk_enable_services();
  while(1) {
    int index = tdl_async_dequeue();
    RTLIB_BACKGROUND_SERVICE();
    if (index >= 0) {
      executeAsyncSequence(index);
    }
  }
}
```

The MicroAutoBox operating system requires a function `void main(void)`, which is executed upon startup. It consists of various initialization calls and concludes with an endless loop handling the execution of asynchronous activities.

The first block of code initializes the RTKernel (`init()` and `rtk_initialize()`), the input/output devices required by the modules of a node and the different parts of the TDL Runtime System.

The next block defines and schedules numerous periodic tasks. To invoke the TDL Runtime System periodically, the `periodicTask` is scheduled every step period. For distributed FlexRay systems, the communication layer (see below)

adds a periodic `syncTask` which handles the synchronization of the local clock to the FlexRay bus clock. Furthermore, every generated asynchronous timer trigger task (`timer<period>()`) is scheduled according to its trigger period.

The block that follows assigns logical TDL interrupts to hardware interrupt lines of the MicroAutoBox according to the corresponding setting in the TDL:VisualDistributor. In the code example above, the logical interrupt *INT0,* for which the `CPlatform` class already created a `handleInterruptINT0()` function, is mapped to the hardware interrupt line *DS1401_IR4*.

As the main function runs at the lowest priority and is preempted by all other tasks, it is a natural choice for executing asynchronous activities. Therefore an endless loop is emitted at the bottom of the main function which constantly polls for pending asynchronous activities and executes them when necessary.

### Make File

**Node1/Node1.mk**

```
APPL = Node1

DESTDIR = C:\Example

SRC_FILES = tdl_machine.c \
            tdl_async.c \
            tdl_main_.c \
            AsyncSender.c \
            AsyncSender_.c \
            Sender.c \
            Sender_.c \

CC_FLAGS = -DPLATFORM_MABX -DTDL_DISPATCHED
SRC_FILES += tdl_comm.c tdl_comm_flexray.c tdl_comm_mabx.c

CC_FLAGS += -DTDL_DISTRIBUTED

.PATH.c = .; \

LIB_FILES = $(DSPACE_ROOT)\ds1401\RTKernel\Rtk1401.lib

C_INC_PATH   = -J$(DSPACE_ROOT)\ds1401\RTKernel \
               -J$(DSPACE_ROOT)\MATLAB\RTIFLEXRAYCONFIG\FlexRayAL \

BOARD_TYPE  = DS1401
BOARD_DIR   = ds1401\RTLib
OBJ_EXT_C   = o03

LIB_FILES := $(LIB_FILES) $(DSPACE_ROOT)\$(BOARD_DIR)\$(BOARD_TYPE).lib
$(PPC_ROOT)\lib\mppcb.lib
C_INC_PATH := -J. -J$(DSPACE_ROOT)\$(BOARD_DIR) -J$(PPC_ROOT)\include $(C_INC_PATH)
CC_FLAGS  := $(CC_FLAGS) -c -p603e -zc -KE $(C_INC_PATH) -D_$(BOARD_TYPE,UC) -QmwC0223
-QmiC0001 -D_INLINE -O5
LD_FLAGS  = -Q i -m>$(APPL).map -o$(APPL).ppc
LK_FILE = $(DSPACE_ROOT)\$(BOARD_DIR)\$(BOARD_TYPE).lk
OBJ_FILES = $(SRC_FILES,S'\.c$$'.$(OBJ_EXT_C)')

build   : startup  $(APPL).ppc cleanup

startup :
        echo building application "$(APPL)" ...
        echo   using local makefile "$(INPUTFILE)" ...
        %if "$(prg_dir)" != ""
          %chdir $(PRG_DIR)
        %endif
        %foreach x in $(OBJ_FILES)
          if exist $(x,R).$(OBJ_EXT_C) del $(x,R).$(OBJ_EXT_C)
        %endfor

cleanup :
        %foreach x in $(OBJ_FILES)
```

```
          if exist $(x,R).$(OBJ_EXT_C) del $(x,R).$(OBJ_EXT_C)
        %endfor
        echo application successfully built

$(APPL).ppc: $(OBJ_FILES)
        echo   linking object modules
        *lnkppc -c $(LK_FILE) $(LD_FLAGS) $(OBJ_FILES) $(LIB_FILES)

%.$(OBJ_EXT_C) : %.c
        echo   compiling $<
        echo *mccppc $(CC_FLAGS) -o $@ $<
        *mccppc $(CC_FLAGS) -o $@ $<
```

The function `emitMake_Content()` emits a make file which is compatible to the make tool shipped with the MicroAutoBox. It ensures that all source code files are compiled and linked correctly.

## 5.6. FlexRay Implementation

This section describes the Node Renesas and MicroAutoBox communication layers and the Comm Scheduler Plug-In for FlexRay. Before going into detail in the subsequent subsections, we introduce a number of base classes (see Figure 43) which are used by all FlexRay-related classes.

The class `FlexrayProperties` extends the abstract class `CommProperties` and is provided by the Comm Scheduler Plug-In for FlexRay (`FlexrayPlatform`) to the Comm Scheduler for cluster scheduling (see 4.3). In addition to the functionality required for scheduling purposes by the Comm Scheduler, it implements all FlexRay constraints according by the FlexRay specification [39]. Furthermore, it is able to calculate all cluster and node properties of a system out of a small set of base properties, which are the cycle length, the static slot size and the size of the dynamic segment. This functionality is intended for rapid prototyping for which it is desirable to quickly produce a working FlexRay system. It is also possible to set all parameters manually. In this case, the `FlexrayProperties` class checks whether all protocol constraints are fulfilled.

Closely related to the `FlexrayProperties` class is the `FlexrayNodeProperties` class, which stores all node-level FlexRay parameters. For each FlexRay node in the system one instance of this class exists. A reference to each of them is stored in a field of `FlexrayProperties`.

The abstract class `FlexrayCC` represents a generic FlexRay communication controller. Concrete controllers differ in regard of the encoding of FlexRay cluster and node properties and in the handling of communication buffers. However, they also share common features implemented in `FlexrayCC` such as the calculation of the CRC (cyclic redundancy check) for frame headers and generic buffer management functions. `FlexrayCC` obtains the FlexRay configuration parameters from the `FlexrayProperties` and `FlexrayNodeProperties` class instances.

For the two different FlexRay prototyping platforms we provide two FlexrayCC implementations, namely `BoschERay` for the Node Renesas and `MFR4300` for the MicroAutoBox. The node platform plug-ins facilitate these classes via their communication layers `NodeRenesasFlexrayCommLayer` and `MicroAutoBoxFlexray-CommLayer` respectively.

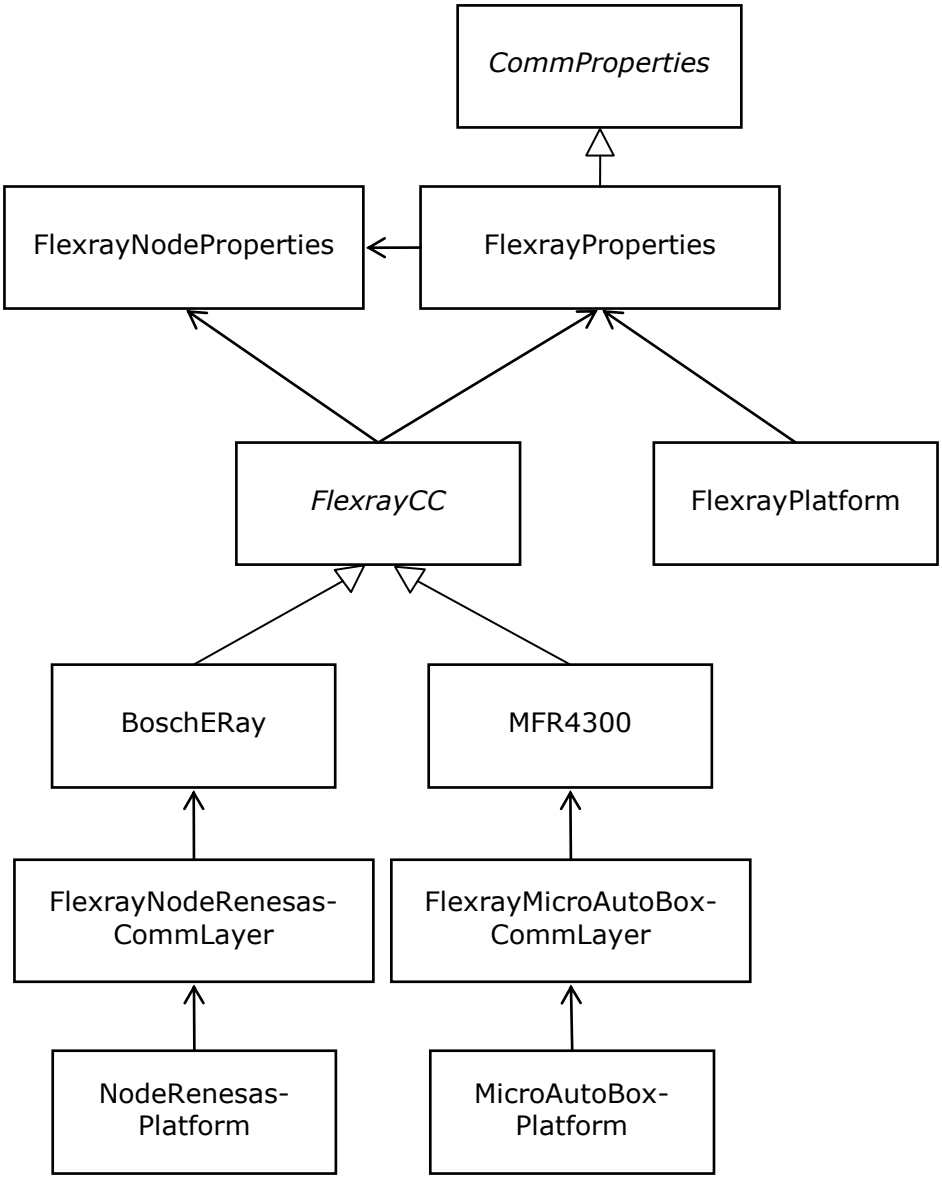


**Figure 43. FlexRay-related classes**

### 5.6.1. FlexRay Communication Layer

`StandardEmbeddedCCommLayer` (see 4.2.4) implements basic communication functionality regarding the transportation of TDL port values across the network and is independent of concrete communication platforms. A class implementing `FlexrayCommLayer` then adds code which is specific to the FlexRay communication protocol but still independent of the concrete FlexRay controller used. The interface has two implementations in analogy to other `CommLayer` implementations: `StandaloneFlexrayCommLayer` and `StandardFlexrayCommLayer`. It has no associated node platform plug-in and only comprises common features needed by all FlexRay-based `CommLayer`s.

Closely related to `FlexrayCommLayer` is the class `FlexrayCC`, which implements the basic functionality of a FlexRay communication controller. It uses a data structure called `CCBuffer` representing a single FlexRay controller buffer. Such buffers are

used to store the contents of static and dynamic slots before sending and after reception. FlexRay parameters on cluster and node level are accessed by using the `FlexrayProperties` and `FlexrayNodeProperties` class respectively. Both property class instances as well as a list of buffers are passed to `FlexrayCC` in its constructor:

```
public FlexrayCC(FlexrayProperties flexProps,
                 FlexrayNodeProperties flexNodeProps,
                 List<CCBuffer> buffers)
```

While buffers for synchronous and asynchronous frames required by a TDL system are already passed with this constructor, additional buffers may be required to exchange data with legacy components connected to the same FlexRay bus. For that purpose, `FlexrayCC` provides the following function to allocate additional communication controller buffers:

```
int requestBuffer(boolean isSendBuffer, int channel, int slot, int
cycleRepetition, int baseCycle, int size);
```

This method is used to request a communication controller buffer with the specified type, FlexRay channel, slot, cycle repetition, base cycle and size parameters. The number of the buffer is returned, which might be a newly allocated buffer or an already allocated one that can be reused in case all its parameters match.

The class `StandardFlexrayCommLayer` implements `FlexrayCommLayer` but is still an abstract class. The following three functions depend on the concrete FlexRay communication controller used and therefore must be implemented by its subclasses:

```
FlexrayCC getFlexrayCC()
String getBufferNumber(CommSchedule.CommFrame frame)
String getAsyncBufferNumber(CommSchedule.CommAsyncFrame asyncFrame)
```

The first method returns a concrete implementation of a FlexRay communication controller. The next two methods return the FlexRay communication controller buffer identifier according to the specified synchronous or asynchronous frame.

`StandardFlexrayCommLayer` implements the following functionality common to all platforms connected to a FlexRay network:

- FlexRay signal call (`flexrayCall()`)

  Generates a function call that sends or receives a FlexRay signal using the `tdl_comm_set<TDLType>Signal()` or `tdl_comm_get<TDLType>Signal()` functions in `tdl_comm_flexray.h`, which is the FlexRay-specific part of the TDL Comm Layer framework (see 5.3). The function is used to generate functionality code for TDL sensors and actuators in the module glue code file which accesses legacy FlexRay signals. For that purpose, a buffer in the FlexRay communication controller is configured using the `requestCCBuffer()` function of `FlexrayCC` so that the a specific signal can be sent or received.

- FlexRay communication controller buffer allocation (`getBuffers()`)

  This function allocates FlexRay controller buffers for all synchronous and asynchronous TDL frames based on the Comm Schedule data structure. It returns a list of buffers of type `FlexrayCC.CCBuffer` and is used to instantiate a concrete `FlexrayCC` instance.

- Synchronized TDL startup (`emitMainC_syncTDL()`)

**Node1/tdl_main_.c (Node1 is startup master)**

```
static char sendInitValues(char readyToGo) {
  if (readyToGo) {
```

```
    buffers[2][0] = 0xEE;
  } else {
    buffers[2][0] = 0x00;
  }
  tdl_comm_sendBuffer(2, 1);
  return 1;
}

static char receiveInitValues(void) {
  return 1;
}

static char receiveReadyToGos(void) {
  tdl_comm_receiveBuffer(0, 1);
  return 1 && (buffers[0][0] == 0xEE);
}

static char syncTDL = 0; // true if all TDL machines are in sync
static char readyToGo = 0;
static char isSyncTDL(void) {
  if (syncTDL==0) {
    if (!readyToGo) {
      readyToGo = receiveInitValues();
      return 0;
    }
    if (readyToGo) {
      syncTDL=receiveReadyToGos();
      sendInitValues(syncTDL);
      return 0;
    }
  }
  return syncTDL;
}
```

**Node2/tdl_main_.c (Node2 is startup slave)**

```
static char sendInitValues(char readyToGo) {
  if (readyToGo) {
    buffers[TDDLL_LookupTxFrame(0, 3, TDDLL_CHA, 1, 0)][0] = 0xEE;
  } else {
    buffers[TDDLL_LookupTxFrame(0, 3, TDDLL_CHA, 1, 0)][0] = 0x00;
  }
  tdl_comm_sendBuffer(TDDLL_LookupTxFrame(0, 3, TDDLL_CHA, 1, 0), 1);
  return 1;
}

static char receiveInitValues(void) {
  return 1;
}

static char receiveStartNow(void) {
  char startNow;
  tdl_comm_receiveBuffer(TDDLL_LookupRxFrame(0, 194, TDDLL_CHA, 1, 0), 1);
  startNow = buffers[TDDLL_LookupRxFrame(0, 194, TDDLL_CHA, 1, 0)][0] == 0xEE;
  return startNow;
}

static char syncTDL = 0; // true if all TDL machines are in sync
static char readyToGo = 0;
static char isSyncTDL(void) {
  if (syncTDL==0) {
    if (!readyToGo) {
      readyToGo = receiveInitValues();
      sendInitValues(readyToGo);
      return 0;
    }
    if (readyToGo) {
      syncTDL=receiveStartNow();
    }
    if (!syncTDL) {
      sendInitValues(1);
    }
  }
```

```
  return syncTDL;
}
```

`StandardFlexrayCommLayer` implements a simple startup protocol which is emitted to the main file and guarantees that all nodes of a TDL system start executing modules at the same time instant. This is necessary as it is possible that not all nodes are powered up at the same time or that they have different boot times. When generating code, an arbitrarily selected node is chosen to be the master of the synchronization algorithm. All other nodes become slaves and send a signal to the master when they are ready to start executing the first TDL Machine step. If a node uses custom functions to initialize TDL ports, these initialization values must be transferred before startup and only after that the node can signal that it is ready to start. The master waits until it receives ready signals from all nodes and then sends a signal to all of them indicating to start module execution in the next FlexRay cycle.

- TDL Comm Layer framework buffers (`emitMainC_FlexrayBuffers()`)

**Node1/tdl_main_.c**
```
tdl_char buffer0[1] = {0};
tdl_char buffer1[5] = {0};
tdl_char buffer2[5] = {0};
tdl_char buffer3[4] = {0};

tdl_char* buffers[] = {
  (tdl_char*) &buffer0,
  (tdl_char*) &buffer1,
  (tdl_char*) &buffer2,
  (tdl_char*) &buffer3,
};
```

As required by the TDL Comm Layer framework, the `StandardFlexray-CommLayer` emits an array of buffers of type `tdl_char`. In order to save space and an additional mapping of those buffers to the FlexRay controller buffers, we map them one-to-one and order them according to the buffers configured inside the FlexRay controller.

- TDL Comm Layer framework frame structures (`emitMainC_FlexrayFrames()`)

**Node1/tdl_main_.c**
```
tdl_comm_FrameStruct frame0 = {0, 1, 0}; //buffer index, size, current position
tdl_comm_FrameStruct frame1 = {1, 5, 0}; //buffer index, size, current position
tdl_comm_FrameStruct frame2 = {2, 5, 0}; //buffer index, size, current position
tdl_comm_FrameStruct frame3 = {3, 4, 0}; //buffer index, size, current position

static tdl_comm_Frame frames[] = {
  &frame0,
  &frame1,
  &frame2,
  &frame3,
};

static tdl_comm_FrameEntry frameSendEntries[] = {
  {1, 4895},
  {2, 9842},
  {-1, -1},
};

static tdl_comm_FrameEntry frameReceiveEntries[] = {
  {0, 154},
  {-1, -1},
};
```

The TDL Comm Layer framework requires data structures describing synchronous TDL frames. These structures are `tdl_comm_FrameStruct` containing the buffer index, the frame size and the current position of the data

pointer of the frame, an array of type `tdl_comm_Frame` containing all frames, and finally two arrays of type `tdl_comm_FrameEntry` indicating the time frames are sent or received respectively.

- TDL Comm Layer framework configuration (`emitMainC_TDLCommConfig()`)

**Node1/tdl_main_.c**
```
tdl_comm_Config commConfig = {
  5000, /* partition period of the node */
  10000, /* bus period of the cluster */
  frames, /* pointer to frame array as defined above */
  3, /* number of synchronous frames */
  1, /* size of tag in bytes */
  frameSendEntries, /* pointer to frame send table as defined above */
  frameReceiveEntries, /* pointer to frame receive table as defined above */
  decodeMessage, /* pointer to message decoding function as defined above */
};
```

Finally, a struct of type `tdl_comm_Config` is generated which contains all values and data structures the TDL Comm Layer framework must be initialized with.

- TDL Comm Layer framework initialization (`emitMainC_TDLCommInitCall()`)

For the initialization of the TDL Comm Layer framework, this hook function emits a call to the TDL Comm layer init function with a reference parameter pointing to the struct containing the TDL Comm layer configuration data.

- Periodic task (`emitMainC_PeriodicTaskBody()`)

**Node1/tdl_main_.c**
```
static void periodicTask(rtk_p_task_control_block pTCB){
  if (isSyncTDL()) {
    tdl_comm_receiveFrames();
    receiveAsyncFrames();
    tdl_machine_step();
  }
}
```

A default body for the periodic task is emitted. In the distributed case, the TDL Comm Layer framework function for the reception of synchronous frames (`tdl_comm_receiveFrames()`) and the function emitted by `StandardCCommLayer` for the reception of asynchronous frames (`receiveAsyncFrames()`) must be called before the TDL Machine step function. The first execution of these three functions is delayed until the startup synchronization algorithm indicates that all nodes in the system are ready to start execution.

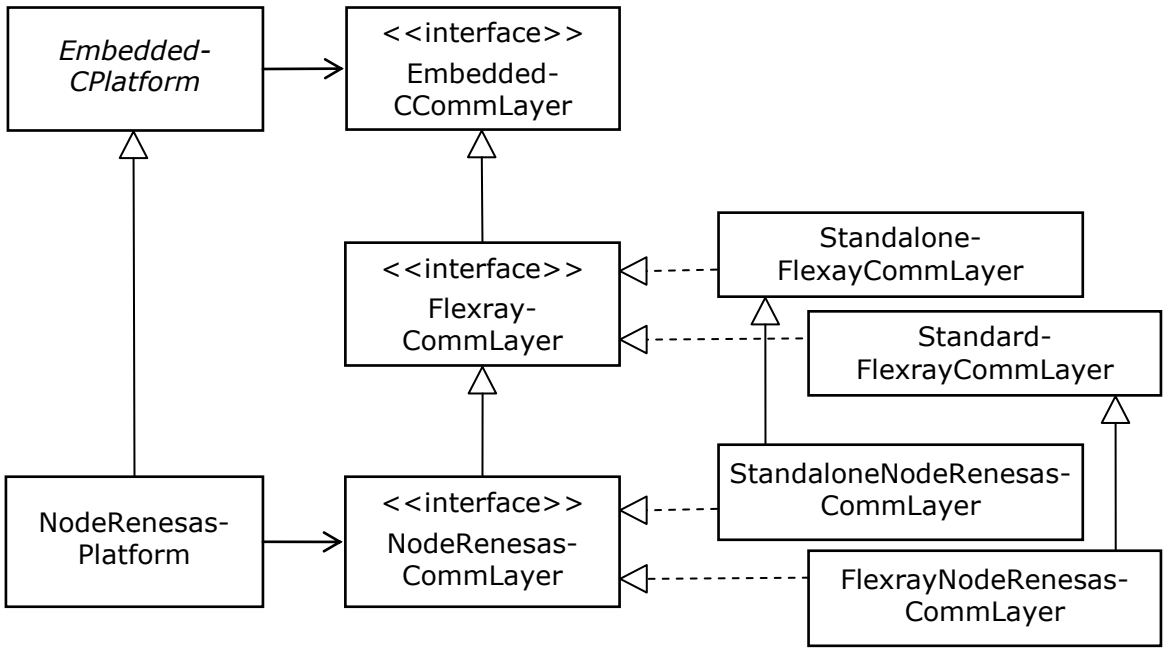


**Figure 44. Node Renesas communication layer class diagram**

### 5.6.2. Node Renesas Communication Layer

The interface `NodeRenesasCommLayer` represents the communication layer interface for the Node Renesas platform. We provide two implementing classes: `StandaloneNodeRenesasCommLayer` and `FlexrayNodeRenesasCommLayer` (see Figure 44). The former only emits a periodic task suitable for a standalone system which solely calls the TDL Machine step function. The latter however generates all the appropriate code to configure and utilize the Bosch E-Ray FlexRay controller used in the Node Renesas prototyping hardware.

`FlexrayNodeRenesasCommLayer` provides the following functionality:

- FlexRay controller buffer assignment (`getBuffers()`)

  This function is already implemented in `FlexrayCommLayer` but is overridden by `FlexrayNodeRenesasCommLayer` as the Bosch E-Ray FlexRay controller requires that the buffer with index 0 is assigned to a FlexRay key slot. This implementation orders the buffers in a way so that this requirement is fulfilled. Note that consequently also the TDL Comm Layer framework buffers must be reordered, as there is a one-to-one mapping between them and the FlexRay controller buffers.

- Application cycle length (`getApplicationCycleLength()`)

  The AES operating system of the Node Renesas requires that its application cycle value is a $2^n$ multiple of the FlexRay communication cycle period. `FlexrayNodeRenesasCommLayer` calculates a suitable value so that this requirement is met.

- Configuration files for COMMSTACK FlexRay controller driver (`emitCCFiles()`)

  The communication layer emits configuration files which adhere to the requirements of the Node Renesas COMMSTACK FlexRay controller driver. They contain all cluster and node parameters and the configuration of the controller buffers in an encoding suitable for the Bosch E-Ray FlexRay controller. The emitted files comprise `COMMSTACK_<nodeName>_Cfg.h`, `COMMSTACK <nodeName>_Cfg.c` and `COMMSTACK_<nodeName>_Memory_Cfg.h`.

- Main file includes (`emitMainC_Includes()`)

**Node2/tdl_main_.c**
```
#include "tdl_comm_noderenesas.h"
#include <dcsCstFr.h>
#include "COMMSTACK_Node2_Cfg.h"
```

  Includes are emitted for the platform-specific TDL Comm Layer plug-in (`tdl_comm_noderenesas.h`), the DeComSys COMMSTACK FlexRay controller driver (`dcsCstFr.h`) and the generated COMMSTACK configuration header file.

- Periodic task (`emitMainC_PeriodicTaskBody()`)

**Node2/tdl_main_.c**
```
static void periodicTask(void) { // This task is called every partition period
  tdl_comm_noderenesas_syncFlexRay();
  if (isSyncTDL()) {
    tdl_comm_receiveFrames();
    receiveAsyncFrames();
    tdl_machine_step();
  }
}
```

  `FlexrayNodeRenesasCommLayer` overrides the `emitMainC_PeriodicTaskBody()` function to add a call to synchronize the FlexRay bus clock to the AES clock (`tdl_comm_noderenesas_syncFlexRay()`). Furthermore, it adds a call to `handleAsyncTimers()` inside the if clause that all timer triggers for asynchronous activities are handled correctly.

- Makefile (`emitMake_Content()`)

  The compiler flag `DISTRIBUTED` is added. This flag configures the TDL Runtime System for distribution.

### 5.6.3. MicroAutoBox Communication Layer

The communication layer interface for the MicroAutoBox platform, `MicroAutoBoxCommLayer`, and its implementations are similar to those for the Node Renesas. Again, two implementations exist: `StandaloneMicroAutoBoxCommLayer` for single node systems and `FlexrayMicroAutoBoxCommLayer` for distributed FlexRay systems. The differences to the Node Renesas version are adaptations to the MicroAutoBox operating system and the use of another type of FlexRay controller. The MFR4300 controller by Freescale requires a different configuration format and buffer handling.

The following functionality is provided by `FlexrayMicroAutoBoxCommLayer`:

- Main file includes (`emitMainC_Includes()`)

**Node1/tdl_main_.c**
```
#include "tdl_comm_mabx.h"
#include <dsfr1401.h>
```

```
#include <dsfral.h>
```

The added include files for distributed systems are the platform-specific TDL Comm Layer framework header `tdl_comm_mabx.h` and the MicroAutoBox FlexRay driver headers `dsfr1401.h` and `dsfral.h`.

- FlexRay synchronization (`emitMainC_TimingControlFunctions()`)

**Node1/tdl_main_.c**

```
static UInt32 lastTSmit;

static void syncTask(rtk_p_task_control_block pTCB){
  ts_timestamp_type ts, syncts;
  syncts = tdl_comm_mabx_getTS();
  if (syncts.mit + 10000 - lastTSmit > 2 * 10000) {
      rtk_it_task_register_rel(S_INTERVAL_A, 0, RTK_NO_SINT, 0.0f, 0,
                               0.0, &syncts);
      rtk_it_task_register_rel(S_INTERVAL_A, 0, RTK_NO_SINT, 0.0050f, 0,
                               0.0, &syncts);
    lastTSmit = syncts.mit;
  }
}
```

In a distributed system connected via a FlexRay bus it is essential that the operating system is synchronized to the communication bus. On the MicroAutoBox, this is done with the help of a sync task which is called twice as often as the TDL Machine on a particular node. This over sampling ensures that there is at least one invocation of the sync task in every FlexRay cycle. The sync task calls a function of the dSPACE FlexRay API via `tdl_comm_mabx_getTS()` to obtain the timestamp of the beginning of the next FlexRay cycle. Subsequently, all TDL Machine steps inside the next cycle are scheduled with the help of this timestamp using the function `rtk_it_task_register_rel()`. The sync task only does this if the difference to the last timestamp is over a certain threshold as otherwise all TDL Machine steps would be scheduled multiple times in the next FlexRay cycle.

- FlexRay controller configuration (`emitMainC_CCConfig()`)

  Configuration for the MicroAutoBox FlexRay controller (MFR4300) needs to be generated. It is stored in the `tdl_main.c` file in a function named `Cfg_CTR0()`. It uses the dSPACE FlexRay API to set the controller registers containing all cluster and node parameters and the configuration for the communication controller buffers.

- Makefile (`emitMake_Content()`)

  The compiler flag `DISTRIBUTED` is added. This flag configures the TDL Runtime System for distribution.

### 5.6.4. Cluster Platform Plug-In

For a TDL FlexRay system to be able to exchange data between nodes, the abstract scheduling data generated by the Comm Scheduler must be transformed to a complete set of FlexRay parameters. For this purpose, the class `FlexrayPlatform` combines the implementation of two interfaces as shown by Figure 45. It represents a cluster platform plug-in in the context of the code generation framework (abstract class `AbstractClusterPlatform`) and additionally acts as a scheduling plug-in to the Comm Scheduler (interface `CommSchedulerPlugin`). Furthermore, it uses the `FlexrayProperties` class for the calculation of FlexRay cluster and node parameters.

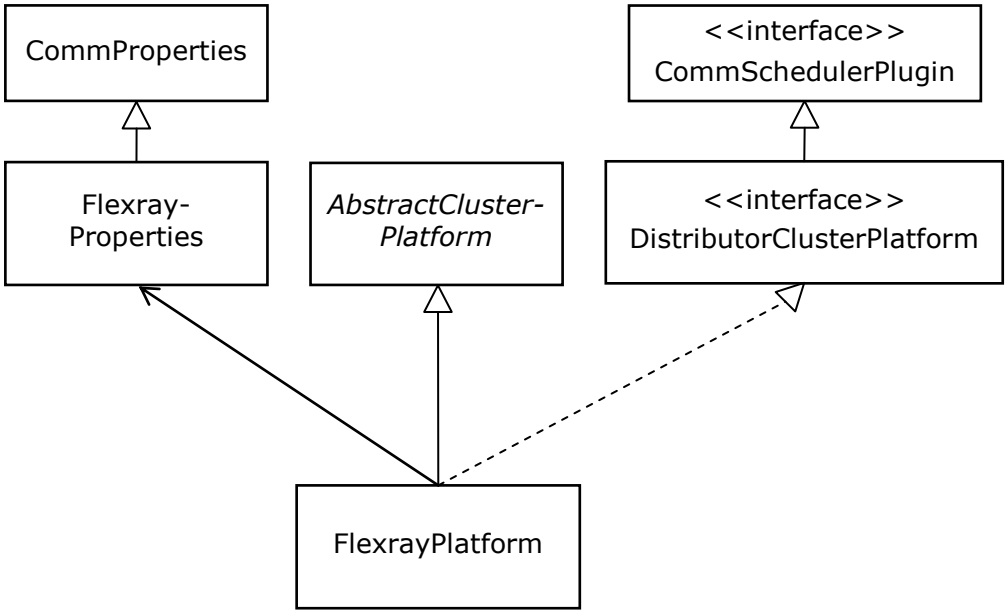


**Figure 45. FlexrayPlatform class diagram**

FlexrayPlatform must implement the following scheduling functions as specified by the `CommSchedulerPlugin` interface described in subsection 4.3.4. Table 10 lists all interface functions and describes their implementation in `FlexrayPlatform`.

| | |
|---|---|
| **public** | `CommProperties getCommProperties()`<br>Returns a `FlexrayProperties` object which extends the `CommProperties` class by FlexRay-specific functionality. |
| **public int** | `getCommPeriod(int mspGCD)`<br>The returned communication period must be an integer divider of mspGCD, which is the maximum possible period calculated as the GCD of all mode periods and mode switch periods of the sending modules of a distributed TDL system. FlexRay has a restriction for the maximum length of the cycle period of 16000 macroticks. A macrotick can be configured to last between 1 and 6 µs. When the passed mspGCD is too large to match the maximum length, it is divided by an integer value so that the resulting cycle period is below the limit. |
| **public int** | `getTagSize(int nofMsgs, int msgID)`<br>Regarding the size of the message tag with respect to the number of messages and the message ID, `FlexrayPlatform` returns a size of 1 if the number of messages is below $2^8$ and a size of 2 if it is below $2^{16}$. For a higher number of messages it throws an exception. Note that this calculation does not depend on the message ID, i.e. the second parameter is ignored. |

| | |
|---|---|
| **public** Frame | `newFrame(int senderNodeID)`<br>This factory method returns a new instance of the class `Frame`, i.e. it uses the default frame class. |
| **public** AsyncFrame | `newAsyncFrame(ModuleReader.Task asyncTask)`<br>This factory method returns a new instance of the class `AsyncFrame`, i.e. it uses the default class for asynchronous frames. |
| **public int** | `getMaxFrameSize()`<br>Returns the maximum frame payload size according to the corresponding property obtained from the `FlexrayProperties` class. |
| **public void** | `setTiMap(Map<String, Map<String, Map<String, TaskInvocation[]>>> tiMap)`<br>This method passes information about task invocations to the plug-in. It is passed on to the `CommLayer` of the individual nodes which are responsible for task schedule generation. |
| **public double** | `scheduleFrameWindows(List<Frame> frameWindows)`<br>This core function of the plug-in interface assigns concrete timings, i.e. FlexRay slot numbers, to the list of frame windows passed by the Comm Scheduler. The assignment algorithm optimizes for minimal bandwidth usage on the FlexRay bus. There is a trade-off between a large static slot size that minimizes overhead but might waste bandwidth by having small TDL frames occupy a complete FlexRay slot and having a small static slot size that introduces a large overhead as every slot introduces additional overhead. The lower limit for the slot size is the size of the largest frame as frames are not split across multiple slots. The maximum slot size is bounded by the maximum of 127 2-byte-words set in the FlexRay specification.<br>The slot assignment algorithm works as follows: First the frame windows obtained from the Comm Scheduler are sorted by decreasing deadline time, i.e. the frame with the latest deadline is first. Then for all slot sizes large enough to hold the largest frame it is searched for a valid frame to FlexRay slot mapping. This is done by mapping every frame from the sorted list to the latest FlexRay slot still available. Of all static slot sizes the one which leads to a valid mapping and produces the FlexRay schedule with minimal bandwidth usage is selected. As a last step all frames in the supplied list are updated with the start and end time of the assigned FlexRay slots.<br>For FlexRay startup, the protocol requires that either two or three nodes are configured as so-called cold start nodes. These nodes follow a special startup procedure using the slots assigned to them to initialize the FlexRay communication cycle. The slot assignment algorithm ensures that these startup requirements are fulfilled. |

| | |
|---|---|
| | Whether a node is a cold start node or not is specified via a property by the user. The plug-in checks the number of cold start nodes in the cluster and automatically assigns them a sending slot to be used for startup frames.<br><br>The return value of the function indicates the quality of the scheduling solution. It is calculated on basis of the bandwidth used by the scheduled slots including overhead, such as gaps between slots and unused parts of slots. This way a schedule with unnecessary large slots yields a lower metric then one with the same data transferred in slots that are mostly filled to their size. |
| **public void** | `scheduleAsyncFrames(List<AsyncFrame> asyncFrames, List<Frame> frames)`<br><br>This function performs the mapping of asynchronous frames to communication protocol specific IDs which are then stored in every `AsyncFrame` object. To ensure their transmission via FlexRay, asynchronous frames must be assigned to minislots of the dynamic segment and must be sent and received at the correct time instants. We require the user to specify the number of minislots and thereby the size of the dynamic segment. The scheduling algorithm then assigns every asynchronous frame to an individual minislots in order of their priority, which is indicated by the order of the passed list. As ID the FlexRay minislot number is used. When no or not enough minislots are available, we output a code generation error and abort the generation process. The sending of asynchronous frames, i.e. writing them to the appropriate buffers of the FlexRay controller, is done as soon as the ports originating from asynchronous tasks are updated. Receiving is done right before the TDL Machine runs, which is when also all synchronous frames are received. |

**Table 10. Implementation of the CommSchedulerPlugin interface by FlexrayPlatform**

`FlexrayPlatform` provides user editable properties to the TDL:VisualDistributor via the `PropertiesProvider` interface, which is part of the `DistributorClusterPlatform` interface (see 5.4). For a `Connection` object of the data model, representing a link between a `DistributorNode` and a `DistributorCluster` node, a `FlexrayNodePropertyPage` is provided. It lets the user determine whether a specific node is used for FlexRay startup and for clock synchronization. Furthermore, for a `DistributorCluster` the class `FlexrayPropertyPage` (see Figure 47) provides a minimal set of FlexRay properties and options to import and export FIBEX files. The properties are required by the automatic property calculation algorithm implemented in the `FlexrayProperties` class.

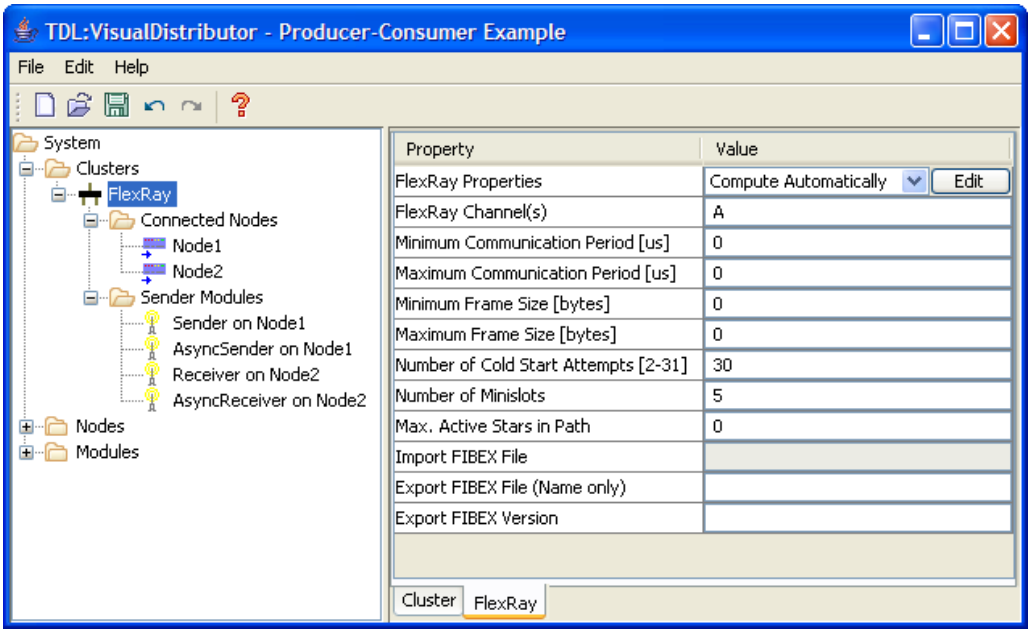


**Figure 47. FlexRay cluster property page**

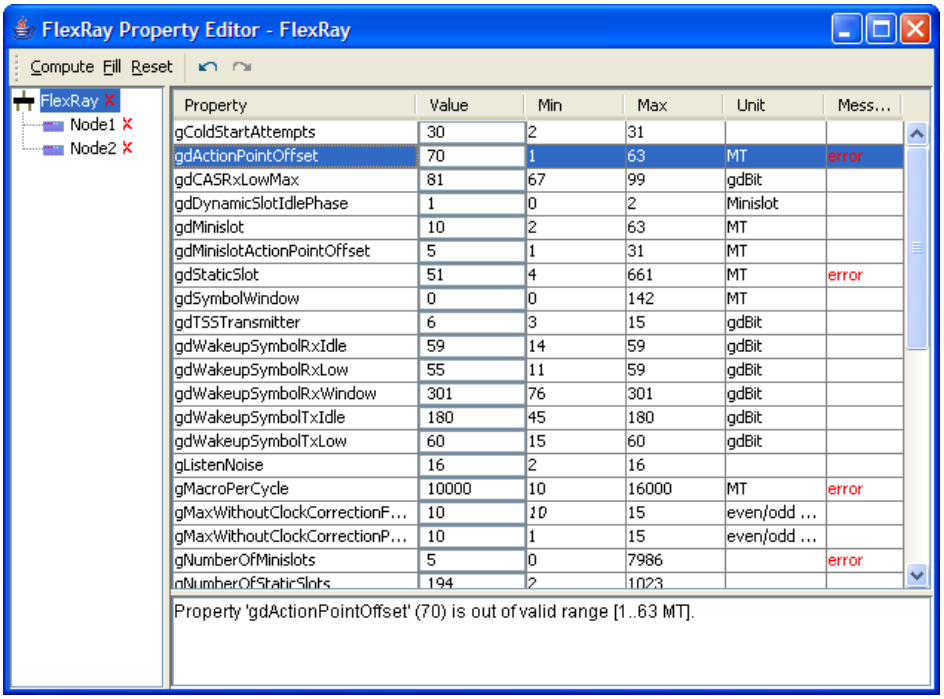


**Figure 46. FlexRay Property Editor**

In addition, both property pages provided access to another dialog which we call the *FlexRay Property Editor* (see Figure 46). It is used to view and edit every single FlexRay node and cluster parameter. While the automatic parameter calculation of the class `FlexrayProperties` is feasible for prototyping as it speeds up the development process, for series production it may be required to alter parameters to tailor them to specific requirements such as maximization of bus throughput or the hardware used. The editor aids the user by checking the ranges and constraints of every parameter and thereby prevents that an invalid set of parameters is generated, which in almost all cases would lead to a non-functioning FlexRay system.

The following depicts the Comm Schedule file which contains the complete communication schedule of the producer-consumer example. It is written by the Comm Scheduler as described in section 4.2. It contains the topology of the network, the mapping of TDL modules to nodes and all details on the timing of synchronous and asynchronous frames and what data is transferred by them.

**FlexRay/commschedule.properties**

```
#
# The name of the cluster:
#
tdl.commschedule.clusterName = FlexRay
#
# The period of the communication cycle in us:
#
tdl.commschedule.commPeriod = 10000
#
# Network configuration:
#
# tdl.commschedule.nodes = nofNodes
# tdl.commschedule.nodes.i = nodeName
#
tdl.commschedule.nodes = 2
tdl.commschedule.nodes.0 = Node1
tdl.commschedule.nodes.1 = Node2
#
# Module assignment:
#
# tdl.commschedule.modules = nofModules
# tdl.commschedule.modules.i = moduleName:key:nodeID
#
tdl.commschedule.modules = 4
tdl.commschedule.modules.0 = Sender:-1984218304:0
tdl.commschedule.modules.1 = AsyncSender:-1450699287:0
tdl.commschedule.modules.2 = Receiver:1793272030:1
tdl.commschedule.modules.3 = AsyncReceiver:616832004:1
#
# The list of frames to be sent on the network:
#
# tdl.commschedule.frames = nofFrames
# tdl.commschedule.frames.i = senderNodeID:startTime:endTime:nofBytes
# tdl.commschedule.frames.i.receivers = nofReceivers
# tdl.commschedule.frames.i.receivers.j = receiverNodeID
#
tdl.commschedule.frames = 3
tdl.commschedule.frames.0 = 1:101:154:1
tdl.commschedule.frames.0.receivers = 1
tdl.commschedule.frames.0.receivers.0 = 0
tdl.commschedule.frames.1 = 0:4895:4948:5
tdl.commschedule.frames.1.receivers = 1
tdl.commschedule.frames.1.receivers.0 = 1
tdl.commschedule.frames.2 = 0:9842:9895:5
tdl.commschedule.frames.2.receivers = 1
tdl.commschedule.frames.2.receivers.0 = 1
#
# The messages to be sent in frames:
#
# tdl.commschedule.messages = nofMessages
# tdl.commschedule.messages.i =
#                   frameID:taskID:modeID:taskRelease:modePhaseNo:nofBytes:nofTagBytes
#
tdl.commschedule.messages = 2
tdl.commschedule.messages.0 = 1:0:1:0:0:4:1
tdl.commschedule.messages.1 = 2:0:1:5000:0:4:1
#
# The tasks which produce output needed on the network:
#
# tdl.commschedule.tasks = nofTasks
# tdl.commschedule.tasks.i = moduleID:name:nofBytes
#
tdl.commschedule.tasks = 2
tdl.commschedule.tasks.0 = 0:produce:4
tdl.commschedule.tasks.1 = 1:produce:4
#
```

```
# The types of ports sent over the network:
#
# tdl.commschedule.types = nofTypes
# tdl.commschedule.types.i = basicType:nofBytes
# tdl.commschedule.types.i = 'struct':nofBytes:module:type
# tdl.commschedule.types.i = 'array':nofBytes:module:type:nofElems:elemTypeID
#
tdl.commschedule.types = 1
tdl.commschedule.types.0 = int:4
#
# The members of structs needed on the network:
#
# tdl.commschedule.members = nofMembers
# tdl.commschedule.members.i = structTypeID:name:memberTypeID
#
tdl.commschedule.members = 0
#
# The output ports which are sent over the network:
#
# tdl.commschedule.ports = nofPorts
# tdl.commschedule.ports.i = moduleID:taskID:name:typeID
#
tdl.commschedule.ports = 2
tdl.commschedule.ports.0 = 0:0:o:0
tdl.commschedule.ports.1 = 1:1:o:0
#
# The association of tasks and output ports:
#
# tdl.commschedule.taskPorts = nofTaskPorts
# tdl.commschedule.taskPorts.i = taskID:portID
#
tdl.commschedule.taskPorts = 2
tdl.commschedule.taskPorts.0 = 0:0
tdl.commschedule.taskPorts.1 = 1:1
#
# The async frames needed on the network:
#
# tdl.commschedule.asyncFrames = nofAsyncFrames
# tdl.commschedule.asyncFrames.i = asyncFrameNr:senderNodeID:taskID:nofBytes
# tdl.commschedule.asyncFrames.i.receivers = nofReceivers
# tdl.commschedule.asyncFrames.i.receivers.j = receiverNodeID
#
tdl.commschedule.asyncFrames = 1
tdl.commschedule.asyncFrames.0 = 195:0:1:4
tdl.commschedule.asyncFrames.0.receivers = 1
tdl.commschedule.asyncFrames.0.receivers.0 = 1
```

### Incremental Scheduling via FIBEX

We have learned that for instance in the automotive industry it is an important requirement to be able to integrate new systems with already existing components and communication buses. Therefore we developed a method we call *incremental scheduling* that enables to combine a possibly hand-written, already existing FlexRay schedule with the TDL approach of automatic schedule generation. We use the FIBEX data format as a means of data exchange as it is already supported by most FlexRay tools. FIBEX is an XML file format describing the complete communication infrastructure of a car, with FlexRay being only one of the bus protocols it supports.

The `FlexrayPlatform` class offers both FIBEX import and export. The export functionality enables that FlexRay systems built from scratch can subsequently be extended using third-party tools. When extending the schedule of an already existing schedule via FIBEX import, all global FlexRay parameters are taken from the FIBEX file. `FlexrayPlatform` must then ensure that these parameters are obeyed as otherwise FlexRay communication is not possible. The parameters include the length of the FlexRay period and the number and size of static and dynamic slots. Furthermore, all already occupied slots are identified and then taken into account

when assigning TDL frames to FlexRay slots. Legacy FlexRay signals are mapped to TDL modules by using sensors and actuators. The TDL:VisualDistributor provides dialogs which let the user select the FIBEX signal a sensor or actuator should read or write respectively. See the next section for a case study using the incremental scheduling functionality.

## 5.7. Case Study

In addition to the producer-consumer example which we used to demonstrate the features of TDL and the TDL tool chain throughout the last chapters, we present an additional case study in this section. It shows how TDL can be integrated with a legacy FlexRay system using incremental scheduling. The setup consists of two Node Renesas and one MicroAutoBox which are connected via FlexRay. One of the Node Renesas nodes acts as a legacy node, i.e. its code is not obtained using the TDL tool chain but created manually so that it sends and receives signals via the bus. In correspondence to the legacy node's functionality, there is a FIBEX file provided which describes the data it sends and receives and the parameters of the FlexRay bus.

Figure 48 presents an overview of the data flow of the case study. All values exchanged between nodes are transferred via a FlexRay bus to which all nodes are connected to. *Node Renesas 1* executes an *Incrementer* and a *Decrementer* module which produce an incremented or decremented value respectively. The *Decrementer* module changes the speed in which it decrements depending on the mode the module is currently in. In one mode the rate in which the value changes is the same as for the *Incrementer* module while in the second mode the rate is doubled. The

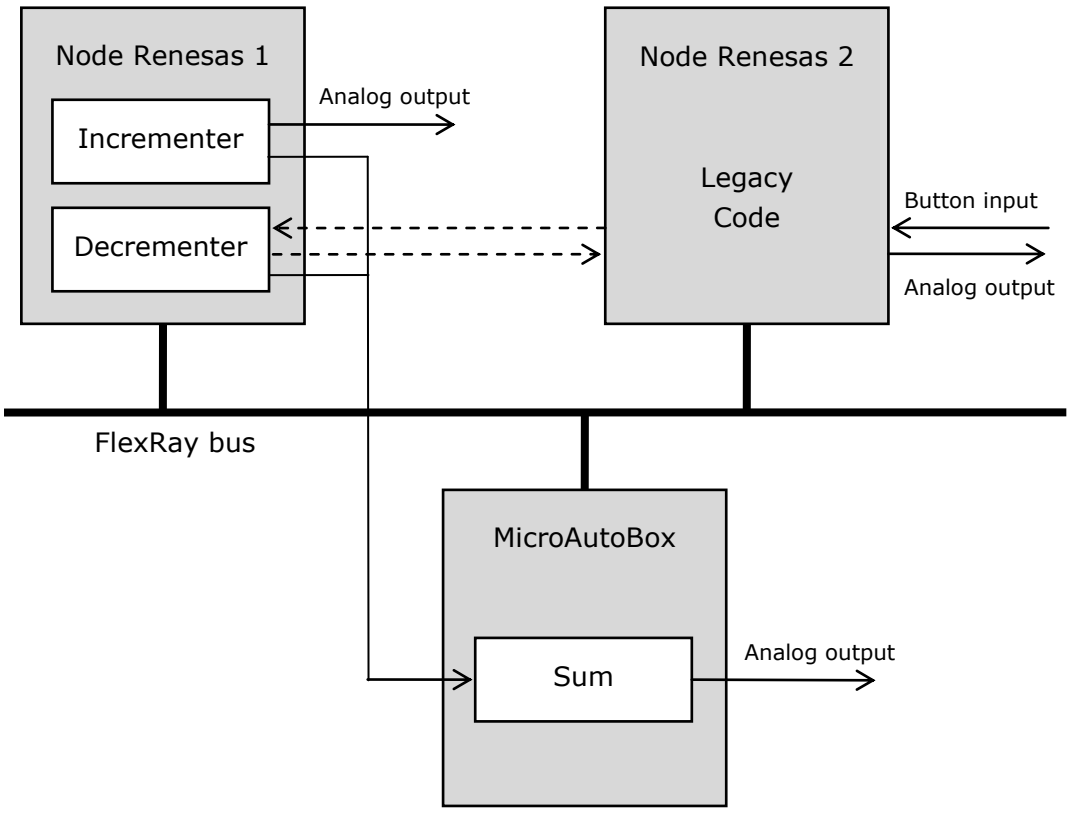


**Figure 48. Legacy case study data flow**

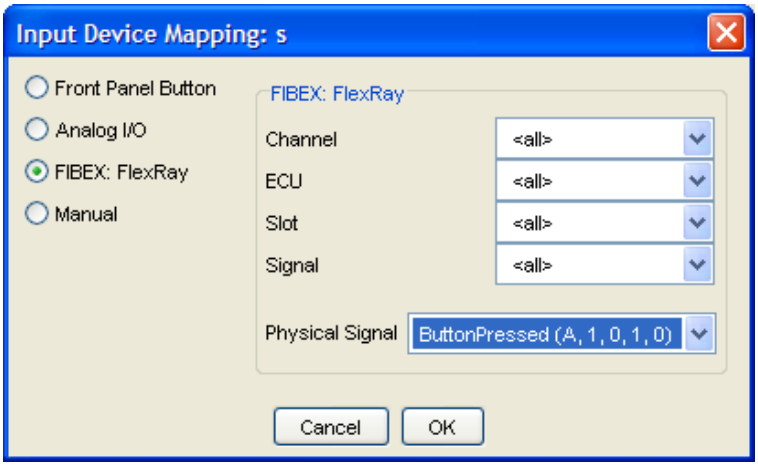


**Figure 49. Mapping of a TDL sensor to a FlexRay signal**

mode switch is triggered by pressing a button during a defined interval. Both modules send their output ports to the third module in the system named *Sum*. It computes the sum of the two values and is executed on the *MicroAutoBox* node. The outputs of *Incrementer* and *Sum* are also output using digital/analog converters on the respective nodes.

The *Node Renesas 2* legacy node runs code which transmits a Boolean value indicating whether the front panel button 1 is pressed via the FlexRay bus and

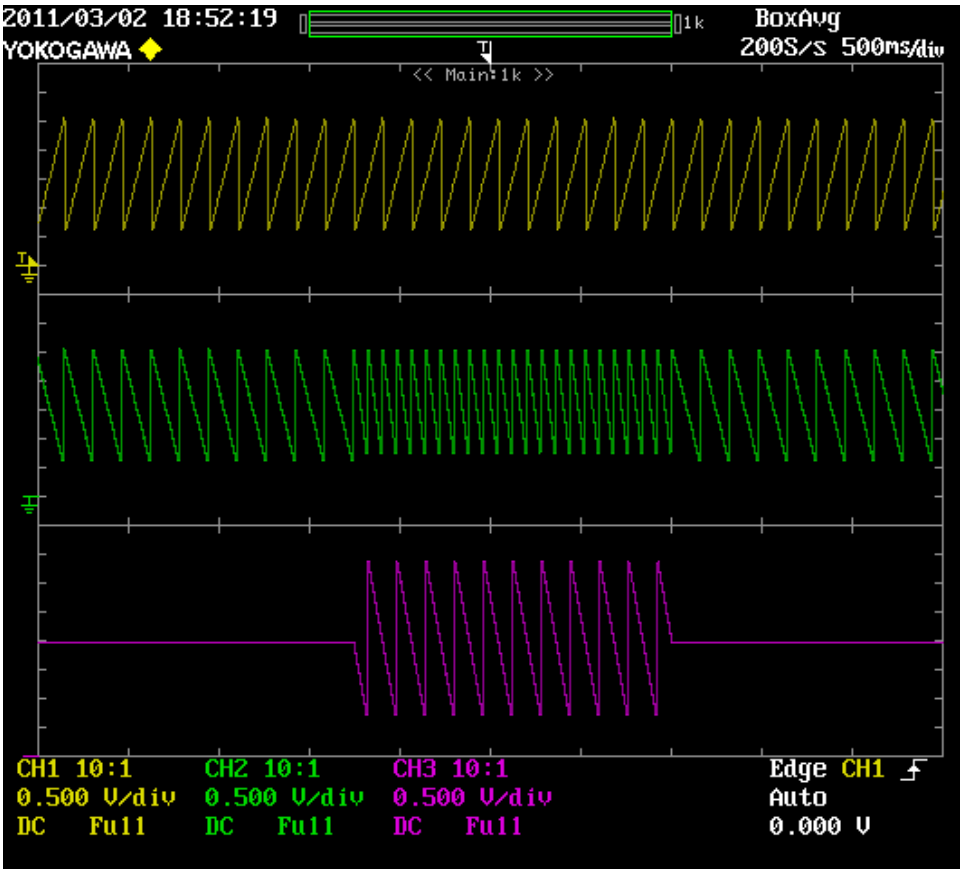


**Figure 50. Case study oscilloscope plot**

receives an 8 bit value which it outputs on its first digital/analog converter port. The FIBEX file contains exact information about these signals and also about all FlexRay cluster parameters. When importing a FIBEX file, the FlexRay scheduling plug-in ensures that the newly added nodes executing TDL modules integrate with these existing parameters and uses the communication slots specified in the FIBEX file. The mapping of FlexRay signals to sensors and actuators of the TDL modules is performed using the sensor and actuator device mapping dialog of the TDL:VisualDistributor. Figure 49 illustrates the mapping of the button signal to the sensor *s* of the *Decrementer* module.

Figure 50 depicts an oscilloscope plot of the running case study for a period of 5 seconds. During this time, the *Decrementer* module changes its mode from the slow to the fast rate mode and back. The first channel in yellow shows the output signal of the *Incrementer* module picked up by a probe connected to the first analog output of *Node Renesas 1*. Channel 2 in green indicates the output of the legacy node *Node Renesas 2* which obtains the decrementer value directly from the FlexRay bus. Finally, the channel at the bottom in purple plots the output of the signal produced by the *Sum* module computed on the *MicroAutoBox*.

# 6. TDL Workflow

This chapter is devoted to an analysis of the workflow typically employed in the industry when developing components of distributed embedded systems. Specifically, we will focus on the automotive industry, which is characterized by a strict separation of concerns between the Original Equipment Manufacturer (OEM) and its suppliers. It is based on hardware components or Electronic Control Units (ECUs), where the OEM specifies the network layout and communication system properties before suppliers develop individual ECUs implementing the required functionality. We argue that platform abstractions such as envisioned by AUTOSAR or the Logical Execution Time (LET) abstraction would allow a fundamental overhaul of the development workflow, eventually leading to a significant gain in productivity and flexibility. We analyze the typical workflow and two standard development tools which are commonly used and compare both to the development workflow employed by tools based on the Timing Definition Language (TDL) which represents a LET-based language. This chapter is an extended version of the work published in [40].

## 6.1. Introduction

So far, the principal means for structuring the growing amount of software in a car is the splitting of functionality into separate Electronic Control Units (ECUs). An ECU corresponds to a software module. This affects the division of work between an Original Equipment Manufacturer (OEM) and its suppliers and thus the overall development workflow. The OEM specifies all signals sent between the ECUs in the overall electronic system and the complete communication infrastructure which carries them. These signals and the topology information, together with a detailed functional specification, are the basis for the development work of the suppliers, which eventually provide one or multiple ECUs to the OEM who is then responsible for the final integration and testing of the overall system.

This approach requires quite a detailed knowledge of the electronic system from the beginning, as the ECUs depend on the communication parameters and signals and vice-versa. Especially when using the FlexRay protocol [34] there are numerous parameters, such as the division into a so-called static (time-triggered) and dynamic (event-triggered) part, the communication cycle length and static slot size, that need to be agreed on in an early phase of the development process as otherwise the ECUs are not able to communicate. Consequently, changes in a later phase are expensive, as they require adaptations in all ECUs of potentially different suppliers.

The original vision of AUTOSAR [11] was to abstract from platform details to allow developing a software component once and then be able to deploy it automatically on any hardware platform, as depicted in Figure 51. This would have held the potential to also change the rigid development process. The Timing Definition Language shares this vision with AUTOSAR. One consequence of an adequate platform abstraction would be that the communication schedule is not a requirement which suppliers need

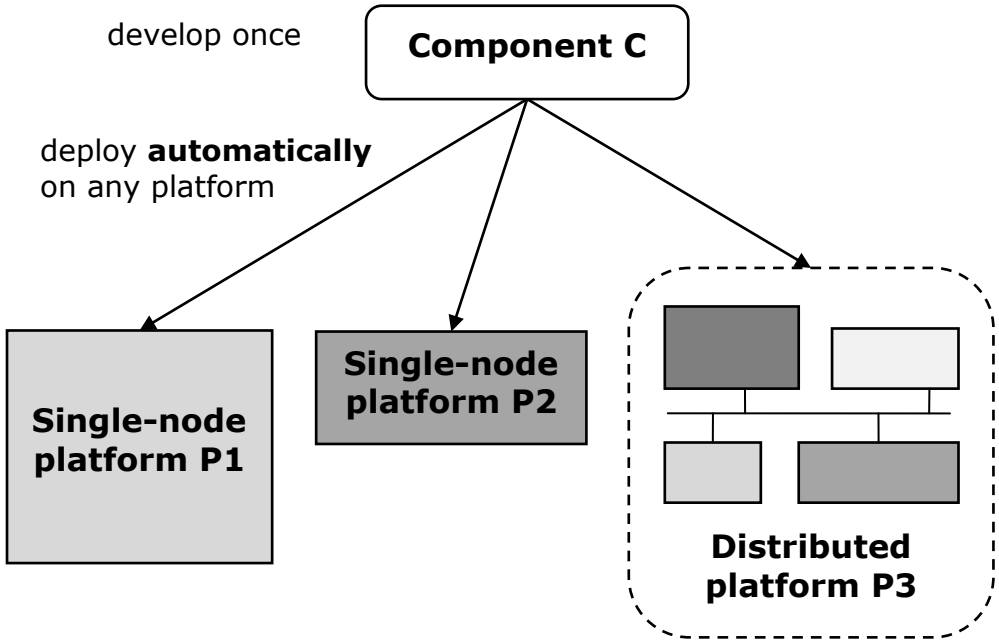


**Figure 51. Automatic platform deployment**

to obey, but which can be generated automatically as a last step when the OEM integrates all components.

In the following we first take a closer look at the AUTOSAR standard. Then we outline and compare a) the non-AUTOSAR workflow based on Elekrobit's EB Designer Pro b) an AUTOSAR-workflow based on Vector's DaVinci Tool Suite and c) a TDL workflow based on the TDL tools integrated in MATLAB/Simulink [16]. We argue that b) is not sufficient to significantly simplify the development workflow in comparison to a) and that only abstractions such as LET that allow the automatic generation of platform-specific code will do so.

## 6.2. AUTOSAR

AUTOSAR stands for AUTomotive Open System ARchitecture and is an international standard developed by major companies of the automotive industry, including Original Equipment Manufacturers (OEMs), suppliers, and tool developers. It aims at an industry-wide standardized automotive software architecture in order to ease software development and the integration of software systems between OEMs and suppliers. One of the main motivations for AUTOSAR was the increasing complexity of automotive software and systems, induced by the growing number of networked ECUs. To tackle this challenge, AUTOSAR introduces an architectural level of system design and fosters the modularization of systems and the portability and reuse of the resulting components, especially targeted at distributed automotive systems.

These goals are only reachable after a paradigm shift from traditional ECU-oriented software development to a function-oriented development process, which is exactly what AUTOSAR tries to accomplish. A specific use case would be the process of combining software of multiple vendors on a single ECU. Previously, this was difficult, as specific functionality typically was provided by one vendor which delivered one complete ECU to be integrated in the system by the OEM. AUTOSAR now provides the means so that OEMs can split up the system on a software component level and

allocate those components to ECUs. This process of ECU consolidation is vital, as the growing number of functions required in future cars would otherwise lead to an equally growing number of ECUs, being expensive, difficult to maintain, and error-prone. Furthermore, a lot of functions in today's cars involve multiple sensors values and actuators shared by other functions scattered across the whole vehicle, making the one-ECU-per-function approach unfeasible.

In order to reach its goals, AUTOSAR focuses on the following three areas [41]:

- Architecture

  A layered architecture provides independence of application software from specific hardware platforms. It consists of three main layers: The application software, the Run-Time Environment (RTE) and the Basic Software (BSW). The BSW is the bottom layer which abstracts from ECU-specific hardware and can be seen as a standard operating system for the automotive industry. It is utilized by the RTE middleware layer, which consists of generated code according to the connections modeled between components.

- Methodology

  The AUTOSAR methodology facilitates XML exchange formats to configure the Basic Software and to enable the exchange of components across suppliers and OEMs and their deployment to ECUs. The ECU development process is divided into a System View, an ECU View, and a Component View. Although AUTOSAR prescribes no timeline and no roles and responsibilities, the typical work-split is that the system configuration is performed as a first workflow step by the OEM and ECU configuration and component implementation is subsequently done by the suppliers. System configuration mainly consists of the specification of the Virtual Functional Bus (VFB), which describes the communication relationships between components in a way which abstracts from whether components are eventually executed on the same ECU or not. In the next development step, components are assigned to ECUs, ECU-specific RTEs are generated, and ECU and component templates are extracted, which form the basis for the subsequent development of those ECUs and components.

- Application interfaces

  These are interfaces of typical automotive applications, which are specified in order to ease their development and integration. According to the AUTOSAR motto "compete on standards, cooperate on implementation," the concrete implementation of these applications is not covered by the standard.

AUTOSAR also leads to better and standardized documentation, especially as it includes the explicit description of networks, which previously was only available in prose form and is now structured in the form of interface definitions. Furthermore, it enables to automate certain development steps, e.g. by generating template code for components out of the system level description. This speeds up development and ensures consistency throughout the whole process.

Release R4.0 of the AUTOSAR standard includes timing extensions enabling the specification of timing properties for the different development phases. Those extensions can be used at VFB, System, and ECU level to describe the timing behavior of an AUTOSAR system. The basic entity in the timing specification is an *event*. Events are chained together to form so-called *timing chains* which include all events occurring between a defined stimulus and a response in chronological order.

This allows describing end-to-end timing constraints which span across multiple views and may also include physical sensors and actuators.

## 6.3. Current Workflow and Tools in the Automotive Industry

According to [42], the automotive systems engineering process consists of the following phases: First, an analysis of the requirements is performed. The system level requirements are then decomposed into sub-functions. In a step called *partitioning*, those sub-functions are mapped to ECUs, sensors and actuators. In addition, appropriate bus systems are selected. Next the workflow continues with the actual component development and finally concludes with system integration and validation.

The tools available for developing distributed automotive systems reflect the described workflow which is commonly employed in the industry. Typically one has to specify the communication properties as one of the first steps in development as all further steps depend on it. We take a closer look on two established tools, namely Elekrobit's EB Designer Pro and the Vector's AUTOSAR-based DaVinci tool suite.

### 6.3.1. EB Designer Pro

EB Designer Pro by Elekrobit [10] (formerly DECOMSYS::DESIGNER_PRO) is a tool for the design of distributed real-time systems using the FlexRay communication protocol. Figure 52 illustrates its main user interface. The tool aids the user to set up all FlexRay parameters and produces configuration files for FlexRay controllers and the operating system running on the ECUs of the system. Task functions must be provided separately. The tool is available in a full version and also as two separate units, the EB Designer Pro <SYSTEM>, which is limited to OEM design tasks and the EB Designer Pro <ECU>, limited to design tasks performed by ECU suppliers. The

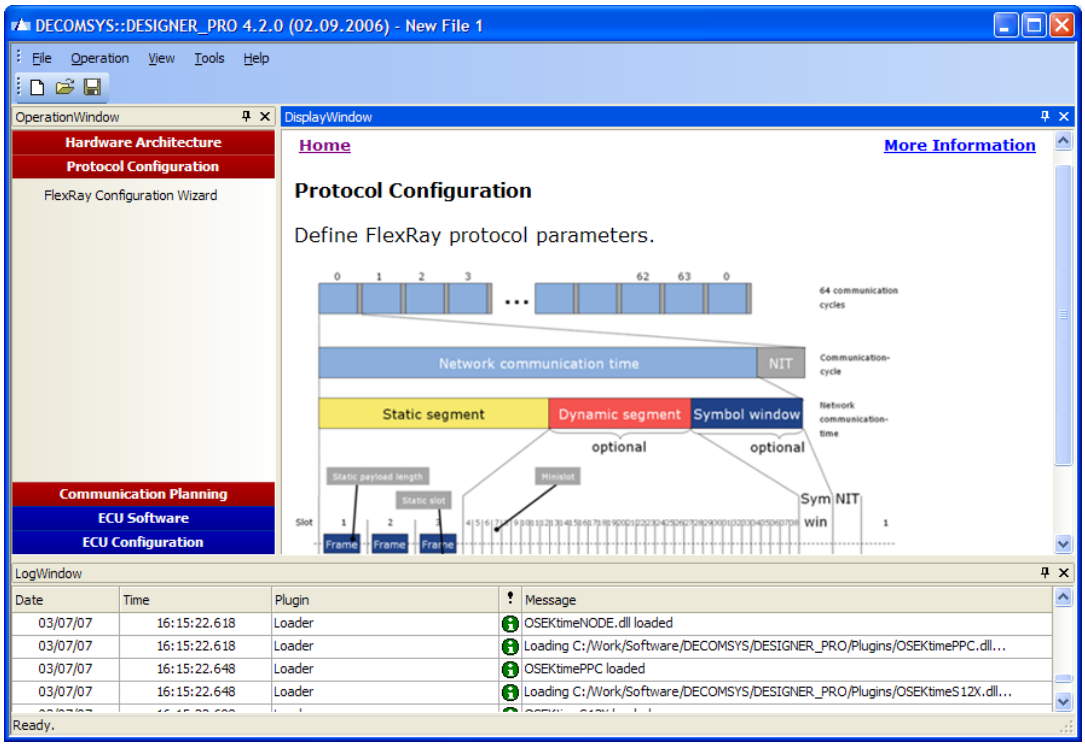


**Figure 52. EB Designer Pro Main User Interface**

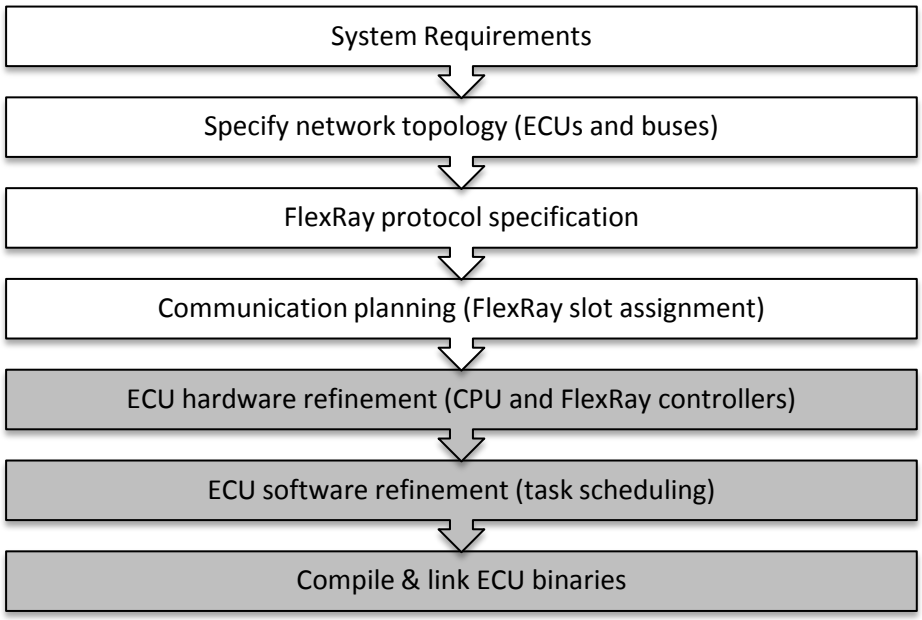


**Figure 53. EB Designer Pro workflow overview (white: OEM, gray: supplier)**

developer is guided step-by-step through all required settings to obtain a working system. The steps are divided into a system part and an ECU part which corresponds to the two versions of EB Designer Pro as mentioned above.

Figure 53 outlines the complete development workflow of EB Designer Pro. The first step in the system part is the architecture definition, where the network topology including the number of ECUs and communication controllers in the system and the bandwidth of the FlexRay bus is specified. Next, the detailed settings of the FlexRay protocol must be entered using an optional wizard. The wizard and further parameter entry forms support the user by checking the supplied properties against the constraints of the FlexRay specification. The system part is then concluded with a step called communication planning, which involves the assignment of FlexRay communication slots to ECUs in the system. To perform this step, at this point in development it must be already known about the exact communication requirements between nodes, i.e. which functions each node executes.

The next development phase is the ECU part which is typically done by one or more suppliers, who are able to import all the settings the OEM has already specified in the system part. The ECU workflow starts with an ECU hardware refinement step, where the type of Microcontroller Units (MCUs) and FlexRay controllers are selected and operating system parameters are specified. Next, the ECU software is refined by defining application and system tasks and assigning them to MCUs. Finally, automatic code generation for every ECU is triggered after the detailed configuration of the communication layer. The code generated by EB Designer Pro consists of operating system configuration files based on the tasks an ECU must execute and FlexRay controller configuration files containing all FlexRay cluster and node parameters.

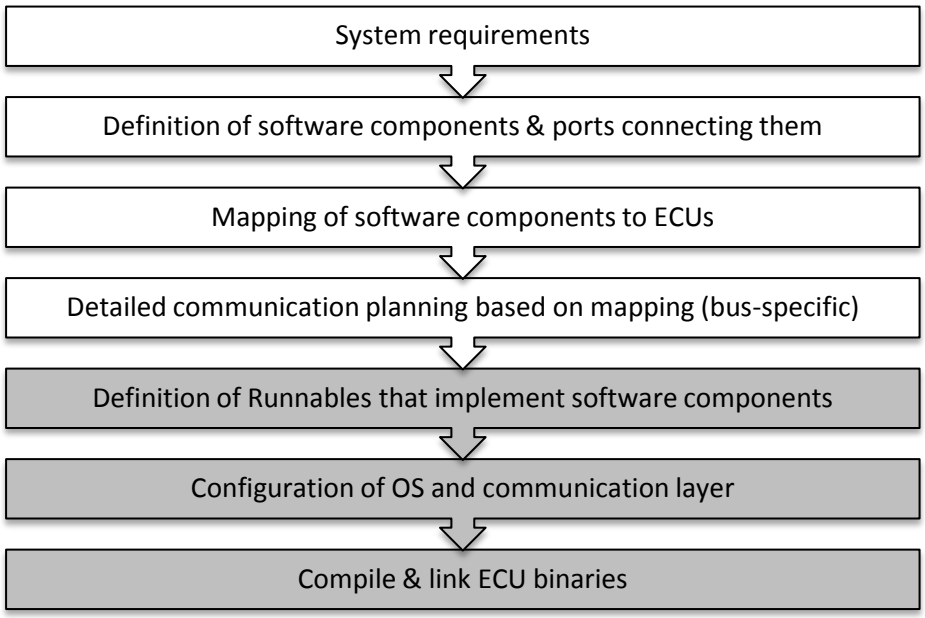


**Figure 54. DaVinci Tools workflow overview (white: OEM, gray: supplier)**

### 6.3.2. DaVinci Tool Suite

The DaVinci tool suite by Vector Informatik [43] consists of three parts. The System Architect and the Network Designer are typically used by OEMs, whereas the DaVinci Developer is targeted at ECU suppliers. Every tool is used to perform distinct design tasks according to the AUTOSAR methodology, as described in 6.2. See Figure 54 for an overview of the workflow.

DaVinci System Architect is used to define AUTOSAR software components on an abstract level, using AUTOSAR's Virtual Functional Bus abstraction. This means that no functionality is specified, but only the interface and connections of components, i.e. so-called ports that have a type and a data size. Figure 55 shows the System Architect's user interface, depicting three interconnected software components. In addition, a network of ECUs is defined and subsequently every software component is mapped to an ECU where it is later executed. After this step, ports can be distinguished by whether the associated software components are mapped to the same ECU and therefore are ECU-local (so-called internal ports) or require network communication as they are located on different ECUs (so-called external ports).

DaVinci Network Designer is available for different communication buses such as CAN and FlexRay. It is used to set up all properties of the specific protocol, including bandwidth, communication layout, frames and messages. The most important workflow step is the assignment of external ports to messages so that the required values for exchanging data between software components are transferred via the bus.

On basis of the former specification of the system, an ECU supplier can then use DaVinci Developer to create the complete ECU software. So-called Runnables must be defined which are used as a container for user code and finally implement the functionality of software components. Runnables then need to be mapped to operating system tasks, which requires also a priority to be assigned to them. Finally, the operating system and the communication layer must be configured before the complete ECU software can be compiled and linked.

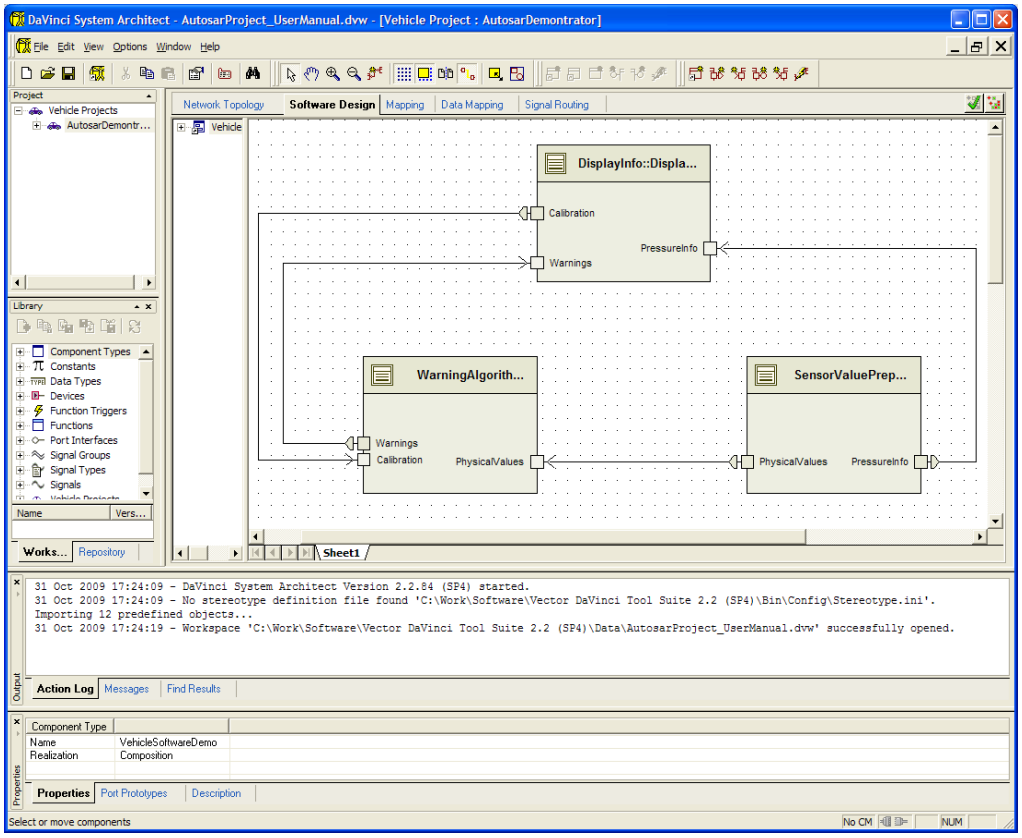


**Figure 55. DaVinci System Architect user interface**

### 6.3.3. Evaluation

In order to evaluate the flexibility of the workflow of the two tools, let us consider the following example use case: For reasons such as ECU consolidation, a software component of a previously completely specified system needs to be moved from one ECU to another. This typically leads to a change in the communication requirements for the involved ECUs and therefore also to a change required in the communication schedule. For both tools this means that adaptations are required early in the workflow, and as all subsequent steps depend on it, they all need to be reevaluated and in many cases a redesign is necessary.

When using the EB Designer Pro, it depends on the concrete change that is required to determine to which workflow step one has to go back. If it is sufficient to add or change the contents of individual FlexRay slots, changes in the communication planning workflow step are required. If this is the case, subsequent changes in the ECUs are local to the ECUs involved in the relocation of the software component. If however moving the component requires changes in either the slot size or the communication cycle length, this leads to a change in the FlexRay protocol configuration and thereby invalidates the design of all ECUs in the cluster. In this case all FlexRay controllers must be reconfigured which potentially leads to a change in their timing and consequently also a change to the behavior of every single task on every ECU of the system.

Unfortunately, also the AUTOSAR-based DaVinci Tools provide only little support for the described ECU consolidation use case. As the mapping of software components to ECUs is done by the OEM early in the workflow, a change again invalidates all subsequent steps to a certain degree. Most importantly, the communication planning

step, which is done manually with DaVinci Network Designer, is critical as it later is the basis for ECU development with DaVinci Developer.

The AUTOSAR methodology is meant to promote a less ECU-centric workflow by supporting the reuse of components and the freedom of moving them between ECUs. Indeed, these tasks are simplified by the introduction of the standardized AUTOSAR Basic Software and the introduction of the software component abstraction. Unfortunately, it is questionable whether AUTOSAR can actually deliver its promises, as core aspects of compositionality are not taken thoroughly into account. In specific, the timing behavior of AUTOSAR components is still subject to the concrete deployment of the components. It depends on complex timing issues regarding factors such as the layout of the communication schedule, the CPU power of the ECUs, the task priorities of AUTOSAR Runnables, and the timing of sensors and actuators, among others. While the introduction of timing chains simplifies the analysis of the timing of an AUTOSAR system, it does not lead to predictable timing behavior as it relies on assumptions such as the frequency of event occurrence. As a result, (a) moving a software component from one ECU to another requires significant manual design and development efforts and (b) it is not guaranteed that the component will behave equally as before. Consequently, the consolidated system must again be rigorously tested.

One approach to tackle the lack of timing information in AUTOSAR is the TIMMO (TIMing MOdel) methodology [44], which also influenced the design of the AUTOSAR timing extensions. Its main purpose is to support the enrichment of design models such as AUTOSAR with timing information, including timing requirements, timing constraints, and timing properties, that specify the required and existing dynamic behavior of systems. Although TIMMO aids developers in handling the timings aspects of a system, it does not go as far as the possibility to automatically deploy a component on any platform including a guarantee that its timing is preserved, while this is a key feature of TDL using the LET abstraction.

## 6.4. The TDL Approach and its Impact on the Workflow

This section outlines the TDL approach and its corresponding tool chain and shows its possible impact on the automotive industry's development workflow. As already presented in detail in chapter 2, TDL is based on the concept of Logical Execution Time (LET). It abstracts from the physical execution time of tasks and, in the distributed case, from network communication. As long as both physical task execution and potential network communication at runtime take place within the LET of a task, the software will exhibit exactly the same observable behavior on any (distributed) platform. It the following we will present how the TDL tools can be applied to automotive software development, the advantages of a TDL-based workflow and finally how the transition from today's workflow could be accomplished.

### 6.4.1. TDL Tools

The main TDL tools are the TDL:VisualCreator and the TDL:VisualDistributor, where the former is used for platform-independent modeling and the latter for platform mapping. The TDL:VisualCreator is used to create TDL modules, which are software components that act as a unit of composition and distribution. Using the MATLAB/Simulink integration feature of the TDL:VisualCreator allows the simulation of the TDL system, which due to the LET abstraction is guaranteed to be equal to the observable behavior on the platform. The TDL:VisualDistributor lets the user deploy TDL modules on a potentially distributed hardware platform. It allows specifying the platform, i.e. the ECUs and communication buses connecting them. After setting a number of hardware-specific properties, the complete code for the system can be

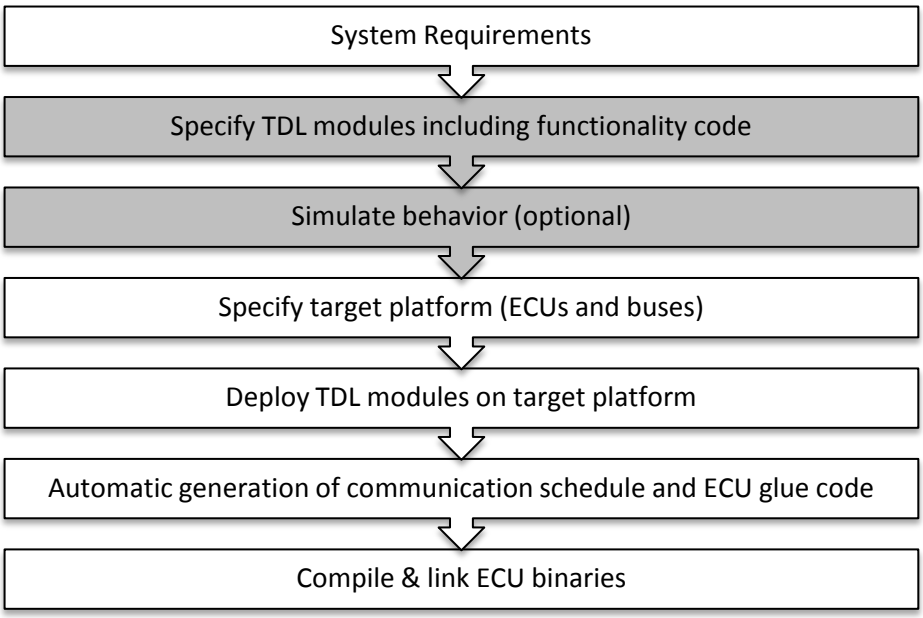


**Figure 56. TDL tools workflow overview (white: OEM, gray: supplier)**

generated. This also triggers the fully automatic bus schedule generator which determines the communication requirements of TDL modules by their deployment to ECUs. Details on both tools can be found in section 2.6 and 2.7, where the latter presents an overview of the complete tool chain.

Regarding the automotive workflow, the TDL tools can be used as shown in the workflow overview in Figure 56. Suppliers may use the TDL:VisualCreator to model software components according to requirements provided by the OEM. The OEM then uses the TDL:VisualDistributor to map these TDL modules to the target platform and finally, ECU code is generated automatically. Concerning intellectual property (IP) protection, it should be noted that the TDL code of a component does not reveal any details on its implementation apart from the timing requirements of (arbitrarily named) individual functions. The functionality code itself does not have to be provided in source code, but can also be delivered to the OEM as object code for integration.

### 6.4.2. Evaluation

Considering the ECU consolidation use case as described in 6.3.3, it can be performed with much less effort using the TDL tools. As no TDL modules need to be changed in such a case, only the mapping of modules to the hardware platform must be adapted in the TDL:VisualDistributor. This is done by assigning the module to another ECU and setting the sensor, actuator and WCET properties accordingly. After that, the code of the whole system–including the network schedule–is simply regenerated. Note that if the schedulability check passes and code is generated the observable behavior is exactly the same as before ECU consolidation, without requiring additional testing.

### 6.4.3. Workflow Advantages

The TDL workflow offers a new level of flexibility and productivity for OEMs and suppliers that range from testing to the optimization of hardware platforms.

In contrast to conventional tools and also the generic AUTOSAR methodology, the specification of the communication network is not done manually and early in the development workflow, but instead it is generated automatically as a last step. The design of TDL modules is completely platform-independent and lets the supplier focus on the functionality to implement without having the target platform in mind. When using the Simulink-integrated TDL:VisualCreator, the behavior of the modeled functionality can be accurately simulated. The supplier can also utilize the fact that TDL modules behave exactly the same on any (distributed) platform by testing the functionality in a real car by deploying it to any platform for which a TDL runtime system exists. The fact that it is sufficient to test functionality only in the Simulink simulation or on one hardware platform also greatly reduces the testing efforts.

For the OEM, the TDL methodology provides the flexibility of choosing the hardware platform, i.e. the ECUs and all connecting communication infrastructure, after all functionality is implemented and not beforehand. Suppliers do not provide complete ECUs but instead TDL modules and corresponding functionality code. The mapping of TDL modules to ECUs is then up to the OEM, who can then for example select numerous less powerful nodes or a small number of powerful nodes in an effort to reduce costs, to increase reliability or to improve electrical stability late in the development process. Another example is the selection of the communication bus: On basis of the actual bandwidth requirements, the OEM can choose for example between CAN, FlexRay [34] and TTEthernet [45] without redesigning or retesting the software, as it is guaranteed that it behaves the same as long as TDL is able to generate code for the specific hardware platform.

### 6.4.4. Transition from Today's Workflow

As the TDL methodology introduces fundamental changes to the current workflow, we are aware that the transition will be a difficult task. However, we think the advantages outlined above are strong arguments and that this transition will quickly pay off. This will be especially true if an OEM does not want to commit to a specific communication protocol and wants to be able to change it easily. The TDL tools provide a single development environment that can be adapted to existing target platforms by developing a plug-in and runtime system for it. Choosing the hardware late in the development process avoids pessimistic hardware choices or complex analysis on what platforms might be adequate to perform the required functionality.

Suppliers can reuse their functionality code or Simulink models and construct TDL modules out of them. However they need to make sure that the functionality still lies within the specification after adding LETs to all functions. The main benefit for suppliers is that they can focus on the functionality and develop in a platform-independent way and therefore are released from the burden of testing the same software repeatedly on different platforms.

Legacy systems can be integrated with TDL by so-called *incremental scheduling*. This approach enables to import a legacy communication schedule and to extend it by adding the communication frames required by the TDL system. It is also possible to exchange values between the two domains by mapping legacy signals to sensors and actuators of TDL modules. This integration allows OEMs and suppliers to phase-in the TDL methodology without starting completely from scratch by replacing parts of an existing system with TDL modules step-by-step.

# 7. Conclusion and Future Work

In the chapters above we demonstrated the feasibility of automatic code and communication schedule generation for LET-based systems by using FlexRay systems as examples. The proposed TDL runtime system and code and schedule generation framework are a further contribution towards a comprehensive, flexible, and platform independent modeling tool chain for time-triggered real-time systems. While even the AUTOSAR methodology fails to fulfill its vision of proper platform abstraction, the TDL tools deliver this vision. We outlined how employing the LET concept finally enables the industry to move away from the traditional ECU-centric workflow to a truly software component-centric workflow. In our view, the newly proposed workflow would have a beneficial impact on the OEM-supplier relationship, leading to increased efficacy, productivity and flexibility.

**Future Work**

While the thesis proves the feasibility of using TDL and the TDL tool chain for industrial, distributed real-time systems, there are still numerous challenges to further improve its functionality and the range of applications.

One such challenge is the integration of *multiple timing domains* or time sources. Currently a TDL system is assumed to have a single clock shared by all modules in the system and that they all start synchronously. All nodes in the system must be in sync so that the LET start and end instances of tasks occur at the same time on each node of the system. However, it is often not feasible to adhere to this strict requirement. Clocks on a computing node are often an order of magnitude higher that the clock on the bus connecting nodes and therefore are expensive to synchronize. In larger systems there may be multiple time triggered busses which cannot be synchronized to each other at all. And in motor control applications for example, the software is often synchronized to the crank shaft of the engine and therefore changes during operation. All these examples demonstrate the need to relay the strict synchronization requirement. A solution might introduce groups of TDL module or nodes which share a common clock and a way to integrate those groups without losing all real-time guarantees and simulation accuracy.

Concerning the code and schedule generation framework and its frontend, the TDL:VisualDistributor, a possible additional functionality is the *automatic assignment of a set of modules to a set of nodes*. Apart from testing the schedulability of all possible combinations, is also possible search for a mapping which optimizes for minimal data transfer between nodes by placing modules which communicate frequently and with large messages between each other on the same node. The fact that some modules require sensors and actuator hardware which might not be present on every node could be tackled by introducing constraints specifying which hardware a module requires. The automatic mapping functionality could also be used

to recommend an alternative mapping to a user in case a manual mapping turns out to be unschedulable. Further possible improvements include support for systems with multiple and heterogeneous communication networks, e.g. for combining FlexRay with CAN and other busses as often done in automotive systems. These would require generating gateway nodes translating between these networks automatically.

As *multiprocessor and multicore systems* become increasingly important also in real-time systems, it is a logical step to support such systems by the TDL Runtime System. A straight-forward approach would be to keep a single instance of the TDL Machine and distribute the execution of task functionality code across all processors or cores available. A multi-threaded version of the TDL Machine is also possible, but depending on the concrete E Code there might not be many instructions that can be executed in parallel. However, the parallel execution of complex sensor and actuator code would help to keep the execution time of the TDL Machine as short as possible as required by the assumption that this code is executed in logically zero time.

The dependability standards of embedded real-time applications often require *fault tolerance* and therefore the *redundancy* of components. While TDL guarantees predictable behavior of the modules on a computing node, it cannot prevent hardware failures. As TDL modules are already used as units of distribution, they are also a natural choice for the unit of replication. Major challenges however are the management of multiple values of the public output ports of a module, the reintegration of modules into a running system and the propagation of information about redundant modules into the system. An example for the latter is the triggering of a mode switch when specific modules or nodes are missing. The handling of only partially complete TDL systems could also be used to support systems with unreliable communication links, such as wireless sensor networks.

# References


[1] **Charette, R. N.** *This Car Runs on Code.* 2009. IEEE Spectrum, http://www.spectrum.ieee.org/feb09/7649.

[2] **Henzinger, T. A. and Sifakis, J.** *The Discipline of Embedded Systems Design.* 2007. Computer, pp. 32-40.

[3] **Henzinger, T. A., Horowitz, B. and Kirsch, C. M.** *Giotto: A Time-triggered Language for Embedded Programming.* 2003. Proceedings of the IEEE 91, pp. 84–99.

[4] **Ghosal, A., et al.** *Event-driven Programming with Logical Execution Times.* 2004. Hybrid Systems Computation and Control, Lecture Notes in Computer Science 2993, Springer.

[5] **Ghosal, A., et al.** *A Hierarchical Coordination Language for Interacting Real-Time Tasks.* 2006. Proc. ACM International Conference on Embedded Software (EMSOFT).

[6] **Project, MoDECS.** *Model-Based development of Distributed Embedded Control Systems.* 2003-2005. http://modecs.cc.

[7] **Farcas, C.** *Towards Portable Real-Time Software Components.* 2006. PhD Thesis, Department of Computer Science, University of Salzburg.

[8] **Farcas, E.** *Scheduling Multi-Mode Real-Time Distributed Components.* 2006. PhD Thesis, Department of Computer Science, University of Salzburg.

[9] **NXP Semiconductors.** *NXP drives active safety with world's first FlexRay transceiver.* 2006. Press Release, http://www.nxp.com/news/content/file_1279.html.

[10] **Elekrobit.** *EB Designer Pro.* http://www.elektrobit.com.

[11] **AUTOSAR.** *AUTOSAR standard.* http://www.autosar.org.

[12] **Kopetz, H.** *Real-Time Systems - Design Principles for Distributed Embedded Applications.* 2007. Springer. ISBN 0792398947.

[13] **Naderlinger, A.** *Modeling of Real-Time Software Systems based on Logical Execution Time.* 2009. Dissertation, University of Salzburg.

[14] **Templ, J.** *Timing Definition Language (TDL) Specification 1.5.* Salzburg : University of Salzburg, 2008. Technical Report.

[15] **Farcas, E., et al.** *Transparent Distribution of Real-Time Components Based on Logical Execution Time.* 2005. Proceedings of the 2005 ACM SIGPLAN/SIGBED

conference on Languages, compilers, and tools for embedded systems (LCTES).

[16] **The Mathworks.** *MATLAB/Simulink.* http://www.mathworks.com.

[17] **Halbwachs, N., et al.** *The synchronous data-flow programming language LUSTRE.* 1991. Proceedings of the IEEE, 79(9):1305–1320.

[18] **Berry, G. and Gonthier, G.** *The ESTEREL synchronous programming language, design, semantics, implementation.* 1992. Science Of Computer Programming, 19(2):87–152.

[19] **Benveniste, A. and G., Berry.** *The Synchronous Approach to Reactive and Real-Time Systems.* 1991. pp. 1270-1282, Proceedings of the IEEE.

[20] **Kirsch, C. M.** *Principles of Real-Time Programming.* 2002. pp. 61-75, Proceedings of the 2nd international Workshop on Embedded Software (EMSOFT), LNCS 2491.

[21] **Liu, J. and Lee, E. A.** *Timed Multitasking for Real-Time Embedded Software.* 2003. IEEE Control Systems Magazine: Advances in Software Enabled Control, pp. 65-75.

[22] **Angelov, C. and Berthing, J.** Distributed Timed Multitasking - A Model of Computation for Hard Real-Time Distributed Systems. *From Model-Driven Design to Resource Management for Distributed Embedded Systems.* 2006, pp. 145-154.

[23] **Hoare, C. A. R.** *Monitors: An Operating System Structuring Concept.* 1974. Communications of the ACM Vol. 17 (10), pp. 549-557.

[24] **Dijkstra, E. W.** *Cooperating sequential processes.* 1968. In Programming Languages, Academic Press, New York.

[25] **Herlihy, M. P.** *A Methodology For Implementing Highly Concurrent Data Structures.* 1990. Proceedings of the Second ACM Symposium on Principles and Practice of Parallel Programming, ACM, New York.

[26] **Greenwald, M. B.** *Non-Blocking Synchronization and System Design.* 1999. PhD Thesis, CS-TR-99-1624, Stanford University.

[27] **Templ, J., Pletzer, J. and Pree, W.** *Lock-Free Synchronization of Data Flow Between Time-Triggered and Event-Triggered Activities in a Dependable Real-Time System.* 2009. Proceedings of the 2nd International Conference on Dependability (DEPEND 2009), Athens, Greece.

[28] **Henzinger, T. A., et al.** *Time-safety checking for embedded programs.* 2002. Embedded Software. Lecture Notes in Computer Science 2491. Springer.

[29] **Yodaiken, V. and Barabanov, M.** *A Real-Time Linux.* 1997. Proceedings of the Linux Applications Development and Deployment Conference (USELINUX), Anaheim, CA.

[30] **Kopetz, H. and Reisinger, J.** *The Non-Blocking Write Protocol NBW: A Solution to a Real-Time Synchronization Problem.* 1993. Proceedings of the 14th IEEE Symposium on Real-Time Systems, 131-137, IEEE, New York.

[31] **Pletzer, J., Templ, J. and Pree, W.** *A Code Generation Framework for Time-Triggered Real-Time Systems.* 2009. Int. Symposium on Software/Hardware Optimizations for Embedded Systems (SHOES09) in conjunction with 2009 IEEE

Int. Conference on Embedded Software & Systems (ICESS), Hangzhou, P.R.China.

[32] **Henzinger, T. A., Kirsch, C. M. and Matic, S.** *Composable Code Generation for Distributed Giotto.* 2005. Proc. of the ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems (LCTES).

[33] **OSEK Group.** *OSEK/VDX Operating System Specification.* 2005. Version 2.2.3, available from http://www.osek-vdx.org.

[34] **Makowitz, R. and Temple, C.** *FlexRay - A Communication Network for Automotive Control Systems.* 2006. Proceedings of 2006 IEEE International Workshop on Factory Communication Systems, pp. 207–212.

[35] **Farcas, E. and Pree, W.** *Hyperperiod Bus Scheduling and Optimizations for TDL Components.* 2007. Proceedings of the 12th IEEE Conference on Emerging Technologies and Factory Automation (ETFA), Patras, Greece.

[36] **Farcas, E., Pree, W and Templ, J.** *Bus Scheduling for TDL Components.* 2006. Dagstuhl Conference on Architecting Systems with Trustworthy Components.

[37] **Nakano, R and Yamada, T.** *Conventional genetic algorithm for job shop problems.* 1991. pp. 474-479, Proc. of the 4th International Conference on Genetic Algorithms.

[38] **Ding, S., et al.** *A GA-based scheduling method for FlexRay systems.* 2005. Proceedings of the 5th ACM International Conference on Embedded Software.

[39] **Flexray Consortium.** *FlexRay Communications System Protocol Specification Version 2.1 Revision A.* 2005.

[40] **Pletzer, J. and Pree, W.** *Impact of Platform Abstractions on the Development Workflow.* 2009. Symposium on Automotive/Avionics Systems Engineering (SAASE), San Diego, CA, USA.

[41] **Kindel, O. and Friedrich, M.** *Softwareentwicklung mit AUTOSAR.* s.l. : dpunkt.verlag GmbH, 2009. ISBN 978-3-89864-563-8.

[42] **Weber, J.** *Automotive Development Processes.* s.l. : Springer, 2009. ISBN 978-3-642-01252-5.

[43] **Vector Informatik.** *DaVinci Tool Suite.* http://www.vector-worldwide.com.

[44] **The TIMMO Consortium.** *TIMMO Timing Model, Methodology Version 2.* 2009. TIMMO Deliverable D7.

[45] **TTA Group.** *TTEthernet Specification.* http://www.ttagroup.org/ttethernet/overview.htm.