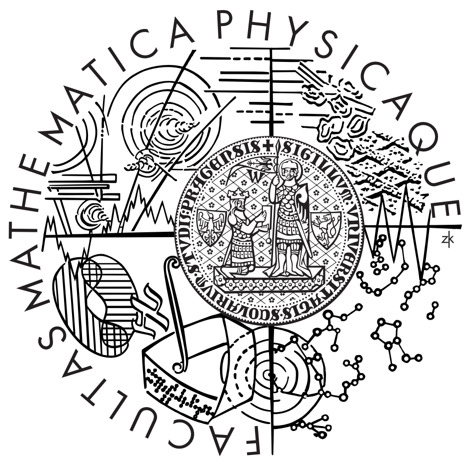
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Jiří Kunčar

ProCom middleware

Department of Distributed and Dependable Systems

Supervisor of the master thesis:	RNDr. Tomáš Bureš, Ph.D.
Study Programme:	Informatics
Specialization:	Software Systems

Prague 2012

This thesis would not be possible without the help of my mentor Etienne Borde. I would like to thank a lot to Jan Carlson for his benefical advices and invaluable input to my research at Mälardalen University. I also thank Rafia Inam for her kind support and introduction to the embedded systems development. And finally to Tomáš Bureš for giving me the opportunity to write this thesis.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague .... Jiří Kunčar

Název práce: ProCom Middleware
Autor: Jiří Kunčar
Katedra (ústav): Katedra distribuovaných a spolehlivých systémů
Vedoucí diplomové práce: RNDr. Tomáš Bureš, Ph.D.

Abstrakt: Cílem práce je navhnout a implementovat části midlewaru, který poskytuje nutnou podporu pro běh ProCom komponent nad real-time operačním systémem FreeRTOS. ProCom je název komponentového modelu pro vestavěné systémy vyvinutý na Mälardalen University.

Primární úlohou je nalezení vhodného kompomisu mezi úrovní abstrakce a ohleduplného využívání systémových zdrojů ve vestavěných systémech. Definovaná cílová platforma má mnohé limitující faktory v porovnání s běžným počítačem. K těmto omezením patří zejména omezená paměť, procesor nebo přenosová kapacita komunikačních kanálů a zároveň striktní požadavky na spolehlivost a odezvu systému. Při řešení jsme čelili problému s limitujícími nebo chybějícími technickými prostředky pro odstraňování chyb programu.

V práci jsou řešeny problémy s rozdílností operačních systémů bez a s real-time podporou. Zaměřili jsme se na nalezení společné podmnožiny funkcí systému nezbytné pro zajištění adekvátní podpory běhu navržených komponent. Rovněž jsme nalezli a otestovali vhodné knihovny pro různé druhy síťové komunikace zejména TCP/IP, i když jsme si plně vědomi jejich limitů při použití v real-time systémech a analýze jejich chování.

Klíčová slova: ProCom, middleware, FreeRTOS

Title: ProCom Middleware
Author: Jiří Kunčar
Department: Department of Distributed and Dependable Systems
Supervisor: RNDr. Tomáš Bureš, Ph.D.
Abstract: The goal of this thesis is to develop and implement parts of a middleware that provides necessary support for the execution of ProCom components on top of the real-time operating system FreeRTOS. The ProCom is a component model for embedded systems developed at Mälardalen University.

The primary problem is finding an appropriate balance between the level of abstraction and thoughtful utilization of system resources in embedded devices. The defined target platform has limitations in comparison to general purpose computer. These include constraints in available resources such as memory, CPU or bandwidth together with strict requirements in terms of worst-case response time and reliability. We have to also face the problem of limited debugging facilities or their complete absence.

In this project, we have examined differences between several real-time and non real-time operating systems. We focus on finding a common subset of core functions that the system must support in order to ensure adequate support for running designed components. We have also identified and tested the suitable libraries to support different types of communication especially TCP/IP. However, we are keenly aware of the limitations of used communication types for analysis of the behavior of real-time systems.

Keywords: ProCom, middleware, FreeRTOS

Contents

1 Introduction **8**
1.1 Problems Definition 9
1.2 Benefits of the Implementation 10
1.3 Outline of the Thesis 10

2 Background **11**
2.1 Real-time Embedded Systems 11
2.2 Middleware 12
2.3 Component Based Development 13
2.4 PROGRESS and the ProCom Component Model 14
2.5 Development process 15
2.6 Technological Background 16

3 Modelling and Comunication Design **21**
3.1 Physical Nodes 22
3.2 Virtual Nodes 23
3.3 Channels 23

4 The ProCom Middleware **26**
4.1 Platform Abstraction Layer 26
4.2 API proposal 28
4.3 Data structures 31

5 Message Communication **36**
5.1 Message Sending 36
5.2 Message Receiving 39
5.3 Connection Reliability and Message Delivery Confirmation 41

6 Application Example **42**
6.1 Components 42
6.2 Virtual Nodes 43
6.3 Physical Nodes 44
6.4 Code Structure 47

7 Related Work **49**
7.1 AUTOSAR 49
7.2 SOFA HI 50

7.3 MyCCM-HI 50
7.4 RubusCMv3 51
7.5 Similar Models 51

8 Conclusion and Future Work 52

A API Documentation 57
A.1 Functions 57
A.2 Macros 58

B How to build own application 60
B.1 Prerequisites 60
B.2 Obtaining Source Code 61
B.3 Compilation 61

C AVR Studio 64

List of Figures

2.1 Overview of deployment modeling and synthesis [3] 16
2.2 AVR Dragon and EVK1100 Evaluation Board 18

3.1 Single channel design 23
3.2 Local and network connection 23
3.3 Two channels 24
3.4 Two way connection 24
3.5 Two channels (another layout) 25
3.6 Channel collocation 25

4.1 Message port structure initialization 32

5.1 Atomic message send 37
5.2 Independent trigger and data writing 37
5.3 Multiple writers 38
5.4 Two readers of same message 40
5.5 Periodic and sporadic tasks read the same message 40
5.6 Only periodic tasks 41

6.1 Sender 42
6.2 Forwarder 42
6.3 Receiver 42
6.4 Virtual node connection design 43
6.5 Local communication 45
6.6 Ethernet communication 46
6.7 Serial communication 46

C.1 Creating new example project 64
C.2 Select example project 65
C.3 Choosing name and location 65
C.4 Project Explorer 66

Chapter 1

Introduction

There is a great deal of interest in communication nowadays. It is here to help us to solve the problems faster and more efficiently. But to do so we need to break the language barriers, understand each other and deliver the messages to the right listeners.

Communication is widespread and takes many forms. Not only people communicate, but also (even thought we do not realize) in our common life computers in offices, machines and their parts — everything communicates with each other and meets with similar problems. Progression in this area is evident every day, especially in the field of industry and it puts emphasis on communication time requirements and efficiency. This general trend is seen for example in automotive industry, where the complexity of electronic components (such as engine control, anti-lock braking system, electronic stability control, collision avoidance system, parking assistants, etc.) is growing exponentially, according to Bureš et al. [1].

Besides, new functionalities are developed and needed to be integrated into current systems. Component Based Software Engineering (CBSE) brings many improvements to developing software for distributed real-time embedded systems by reducing the complexity, organizing the division of the functionality into independent subsystems often developed by different suppliers and enabling easier reuse of once developed and tested components.

The component model used in this thesis is developed within a large research vision called Progress which *aims at providing theories, methods and tools for the development of real-time embedded systems* [2]. In order to allow the automatic code generation, we implement and give precise enough description of the functional blocks needed for components code integration in order to produce executable binaries. The specific code must be included to provide *execution support* and enable *inter-component communication* independently on the underlying platform.

This thesis starts with description of the problems originated in the development of a middleware that provides necessary support for the execution of ProCom[1] components on heterogeneous hardware and operating systems of embedded devices. The main focus is dedicated to support of the transparent communication between these devices.

[1] Progress component model

1.1 Problems Definition

Creation of a wrapper code for the integration of the software components used in Distributed Real-Time Embedded System (DRTES) faces several conflicting requirements. In addition, the semantics of the ProCom model and the Progress deployment process designate a set of requirements on the runtime environment. There are also several demands on the resulting code structure and runtime library. All requirements are closely described in [11], but only some of them are related to the thesis.

To create as much as possible accurate design of the middleware it is necessary to take into account the significant differences between operating systems and hardware on which the middleware is supposed to work. Moreover, there is a serious concern and need for its integration into the development environment and support for as easy as possible code generation components using this middleware. We also need to consider difficulties of reusing existing code in different usage contexts.

Particularly with regard to the reusability of the components, the knowledge of the inner structure of the entire system should be put outside the component code scope. Referring to the communication the knowledge of receivers should not be included in generated component code.

The developed middleware should allow transparent reallocation of components (or tested set of components) to take advantage of CBSE approach. It means that we should be able reuse the code on different platform with any or minimal modifications. The structure of the components should be easily generated from the model of components. However, the runtime library code can be more complex. The complexity of the runtime library code is not a problem in this case, because there is time to test and validate it. It must be clearly defined for every individual section of the final code if it is automatically generated according to component model, or whether it is part of the runtime library.

Futuremore, the middleware should also provide access to the shared system resources to avoid conflicts during accessing them. However, there is a contradiction between an effective utilization of resources and re-usability that inhibits to fulfill the requirement. The most thoroughly discussed set of requirement concerns communication.

Support for channels with more than one sender and reader is a preliminary step to build model of communication between components. The real-time analysis entails the need to ensure that the channel topology (or its change during design process) can not affect analysis results by changing behavior and characteristic of a component within performing operation adherent to the channel communication.

From a modeling point of view there are various possibilities how the components can be associated to message ports and use asynchronous message communication. The middleware has to fulfill rigorously the defined semantics of message passing. We should also not overlook to ensure binary compatibility of sending data between different platforms. Typical example of such a problem is a big and little endian byte order (description is available in [21]).

Finally, we would like to remember the problem with limited resources and real-time requirements that needs to be considered to make predictable system with time and event triggered tasks.

1.2 Benefits of the Implementation

With our developed middleware we are able to compose previously created components into monolithic firmware for the defined platforms. Since composition is made in design time under the ProCom component model, therefore we can perform optimization of components and connections between components, which can be replaced by direct function call.

The middleware provides functionality covering most of the features, that the developer of the synthesis mechanism may require. Several additional improvements for saving resources were implemented. For example we made transparent sharing of the physical connection among several components on the same device possible. However, developer of the system does not have to think about these problems at the beginning of development process and he can focus on right design of his application, instead of developing underlying supporting functionality because of separation of modeling and deployment phases.

Building and analyzing of the system become easier, because middleware can be more thoroughly tested and analyzed based on well know or measured properties at different platforms. It also helps with separation of software and hardware design, which we hope to expand to the independent use and allocation of not only software but also hardware component in future.

The middleware is easily extensible to support new hardware for different communication media or other operating system and as an extra benefit it is possible to use the library outside context of the ProCom model. Developers can focus only on creating functional code and the library solves the communication and task portability for them. Finally, the middleware can be distributed either with full source codes or even in pre-compiled form for supported operation systems.

1.3 Outline of the Thesis

The remaining part of this thesis comprises these chapters: Chapter 2 gives definitions and an overview of the technologies used when developing software for the distributed real-time embedded systems. The design and structure of middleware layers are discussed in Chapter 3. Chapter 4 presents architecture of the ProCom runtime environment and introduce the API[2]. The description of communication process is concisely introduced in Chapter 5. Chapter 6 gives an example of how our middleware can be used in developing component systems that use different types of connection for message communication. Related projects are presented in Chapter 7, and Chapter 8 concludes the report.

[2] Application Programming Interface

Chapter 2

Background

In this chapter, we start with several definitions of the real-time and embedded systems, and the middleware. Following part describes component based development and component framework for system modeling according to the ProCom model. Essential parts of the PROGRESS project and the ProCom component model are described in section 2.4 followed by technological background about operating systems, libraries and hardware supported by current version of the middleware. Last parts of the technical section consist of a short introduction to the development environment (AVR32 Studio[1]) and presentation of another tested experimental hardware.

2.1 Real-time Embedded Systems

In this section, the definitions of real-time and embedded systems will be briefly introduced. The various demands and the most important properties of these systems are presented and understanding them is important for comprehension of decisions made during the work.

2.1.1 Embedded Systems

Electronic devices containing microprocessors are almost everywhere around us. In fact, only about one percent of them is in personal-computers (PC) including laptops [15]. The rest are included in many common devices and helps people with simple daily tasks such as preparation of lunches in microwaves or doing laundry in the washing machines. Users may not even notice the existence of the processor and software in the device that they are using. The microprocessors are also part of some larger and complex systems in automotive industry.

Embedded systems are computer systems that are part of larger systems and they perform some of the requirements of these systems [5]. They are designed to perform their task in very efficient way, mostly (partially) independent of human intervention. The device may be extended or connected with additional mechanical parts or sensors, e.g. detection of closed-door in a car, wheel rotation or distance sensors, to be able interact with the environment where is located. In larger systems that are composed of many smaller units, effective and reliable communication is the integral part of the system.

[1]Available at http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2725

Embedded systems are closely related to real-time systems (discussed below) often in safety-critical applications, e.g., transportation vehicles, where they take care of all different kinds of tasks under difficult environmental conditions such as dust, vibration, electromagnetic interference, etc.

2.1.2 Real-time Systems

A real-time system is a *computing system in which correctness depends not only on the logical results of the computations, but also on physical instant at which the system produces the correct results* [13]. Another possible definition found in [12] describes a real-time system as a set of concurrent programs called *tasks* that have to respond to an internal or external event in a determined time period. The time when the period ends is called *deadline*.

A real-time system can be classified as a *hard* real-time system if the result must be produced before deadline otherwise there is no value to the computation. The extreme of missed deadline may have a serious consequences to the system or its user. We can describe the situation using examples with the collision avoidance system or engine control of a car. In the first example, the system could be incapable to stop the car and avoid a collision, because a computation of distance and speed can take more time than is expected. When the ignition is delayed due to missed deadline of engine control task, irreversible engine damage can occur. However, in *soft* real-time systems it is acceptable that a task deadline is missed with no serious threat or damage. For instance while an audio or video stream is being listened, missing some deadlines causes a quality degradation of the sound.

An important requirement for many real-time systems is to achieve predictability. In hard-real time systems, it is also necessary to predict the peak-load performance and avoid missing predefined deadline. The easiest way how the system can meet deadlines of all concurrent time-critical tasks is to use a static scheduling. The run-time overhead is very small, but all scheduling decisions have to be made at compile time. However, if the system has to adaptively react to non-regular external events, than the static scheduler is not suitable. Dynamic scheduler is more flexible, although it can incur nontrivial system overhead and requires detailed system analysis to ensure desired behavior.

2.2 Middleware

A middleware is, in our case, the software layer that lies between operating systems and software components that can be located on different devices connected by network. It consists of a set of services that allows transparent inter-process communication based on messages within one or between several nodes. Middleware can help to manage complexity and heterogeneity of the underlying operating systems and hardware and also it facilitates using multiple network technologies. It tends to provide consistent and integrated distributed programming environment.

Through indisputable advantages, there are also disadvantages in terms of performance degradation, that must be taken into account on systems with limited resources. A higher level of abstraction usually increases code re-usability, but it can decrease efficiency in using resources.

2.3 Component Based Development

While demands and expectations on functionality included in machines around us (such as refrigerators, televisions, cell phones and cars) are growing, developers have to deal with higher complexity of the software driving the devices. A possible solution lies in dividing software into smaller independent units — *components*. This type of development is called *component based development* (CBD) and it is described more precisely in [4].

The advantage of such a division is in the possibility to develop the units independently and reuse or compose them, thanks to well defined interfaces of each unit. Developed, analyzed and tested components can be stored in a software repository prepared for a future use. Furthermore, these components can be pre-compiled or shared among developer groups. Therefore increasing reliability and reduced development time can potentially be achieved.

However, the CBD approach has also several problems. Most of them come from desired component properties described in [20]*: isolation* (a component is an atomic deployable unit, that can be run independently), *composability* (a new component can be constructed from existing hierarchically structured and interconnected components) and *opaqueness* (a knowledge of component implementation is not necessary for using component – only the interface).

The first problem is connected with a need of isolated and reusable code in components. This can lead to suboptimal utilization of resources. A strict decomposition into small components and re-composition causes an overhead but on the other hand using too big components does not solve the complexity problem mentioned at the beginning of this section. Opaqueness of component implementation can bring a problem when changing the implementation. Therefore the specification of the component model should be complete and accurate. Development of a component system can be more time-consuming than creating the application in the usual way. Especially the first time, since it requires to adopt the component semantics. Nevertheless, the appropriate framework and tools should decrease the needed time.

2.3.1 Component Framework

With CBD, developers can build systems by assembling existing components. To take more advantage from CBD, a software component framework automatically generates the non-functional code, i.e., the "glue" that plugs together individual components. Such a framework has been developed within the PROGRESS project. *The purpose of ProCom Integrated Development Environment (*PRIDE[2]*) is to support design, analysis, and realization of components and component-based systems using different tools integrated in a common environment* [10]. The implementation of the ProCom middleware created as a part of the thesis should help when generating "glue" code connecting components and runtime support for different execution platforms with the code generated by PRIDE.

[2]web page: www.idt.mdh.se/pride

2.4 PROGRESS and the ProCom Component Model

This thesis is a part of the larger research vision called PROGRESS, which is the Swedish national research center for predictable development of embedded systems. In this section a brief overview of the vision is introduced as it is described in [2].

The overall goal of the Progress project is to cover the whole development process starting from a vague specification at the beginning to the final specification and implementation including reliability predictions, analysis of functional compliance, timing analysis and resource usage analysis [11]. The ProCom component model has been introduced as one part of the PROGRESS project to be used mostly on real-time embedded systems. It consists of two different (but related) layers which help to solve the problem with the right level of component granularity. The larger units are using message passing. At more fine-grained level, the information about timing and synchronization requirements are known and the communication within one subsystem can be much more simple and optimized.

The lower level, called ProSave, aims to design functional components, which could be hierarchically structured and used as composite units. These components are passive and the communication on the ProSave level is based on *pipe-and-filters paradigm* [2] in contrast of ProSys layer, where the components compose a collection of concurrent, communicating subsystems. The communication is mediated by sending messages through channels connecting message ports.

ProCom has also particular elements that enable communication between the ProSave and ProSys layers. The clocks generate periodic triggers and message ports create mappings between message passing and trigger or data communication on ProSave level.

More detailed information about ProSys and ProSave layers are included in 2.4.1 and 2.4.2, respectively. The software development and deployment according to the PROGRESS vision are described in 2.5.

2.4.1 ProSys

As mentioned in the beginning of this section, ProSys models a system as a collection of subsystems. The subsystems can be composite — contain one or more interconnected ProSys subsystems, or primitive — usually build from a set of ProSave components. In the final system a ProSys subsystem is represented by set of tasks, message ports (input and output) and parameters required for execution. If the composite ProSys subsystem contains any local communication between its subsystems then it is statically resolved during the synthesis and it is not visible outside of the subsystem. This simplify middleware concept of the communication part.

Communication on this level is possible only through the asynchronous message channels connecting message ports. Each channel is typed and only a message port that sends or receives that type can be connected to the channel. There are no limitations to channel complexity (it is possible to use channels with more that one writer and reader), but it is very important to say, that no dynamic changes of the channel structure or subsystems are allowed after deployment. The support of transparent network communication is also needed, because subsystems are reusable and can be mapped to different nodes.

2.4.2 ProSave

ProSave is the lowest layer in ProCom using several modeling constructs. Services consist of one *input port group* and one or more *output port groups*. Each group consists of one *trigger port* and zero or more *data ports*.

For full understanding of the thesis it is not necessary to know all parts from which the ProSave component can be composed. The bindings to the developed library is very simple. ProSave component routines are represented by C functions and corresponding data structure. The activation connection with ProSys layer can be created in two different ways. The first one is done by a clock event with defined period and the other trigger can be an incoming message on an associated input message port.

2.5 Development process

The usage of components provides substantial benefits in development of real-time embedded systems. The development process based on the ProCom model adds the ability to design distributed systems using further modeling layers of virtual and physical nodes.

ProCom development process can be partitioned into *deployment modeling* and *synthesis* parts which are described in following paragraphs.

2.5.1 Deployment Modeling

The modeling part is divided into four related formalisms having distinct objectives. The first two — ProSave and ProSys — were described in previous sections. The third part is a *virtual node* model. The aim of this modeling concept is mainly to allow detailed timing analysis of virtual node independently from other virtual nodes and tasks allocated on the physical node in the final deployment. A virtual node is a collection of ProSys subsystem instances from which the information about interfaces, interconnections, dependencies on libraries and specific hardware are derived. CPU utilization and network bandwidth allocation should also be counted based on components demands for each virtual node.

The last part of the deployment model defines the *physical nodes* of the system and the way of mutual interconnection. A physical node is primarily intended as a container of virtual nodes and other entities needed of communication and hardware support. The definition of physical node includes information about processor type, available memory, hardware components, possible network types and used operating system. During the allocating process of virtual nodes to physical node, demands on resources are compared with possible utilization of the physical node. The whole process of system modeling is shown on top of Figure 2.1.

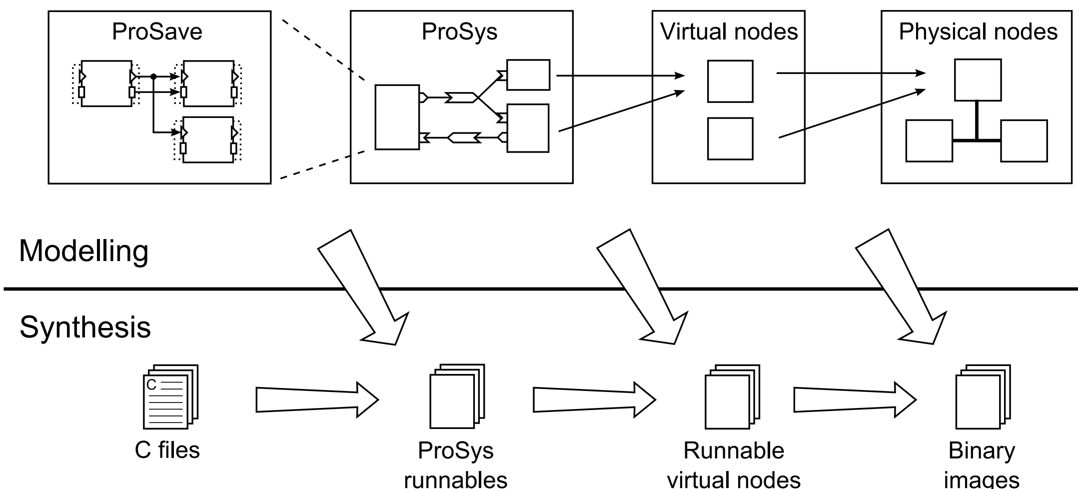


Figure 2.1: Overview of deployment modeling and synthesis [3]

2.5.2 Synthesis Overview

Synthesis is the process that constructs runnable representations of ProCom model entities (see bottom part of Figure 2.1). Composition levels correspond to deployment models and for each composition level should be possible to build a binary library. The build process incrementally composes binary files from lower level components and necessary glue code. A ProSave component can be represented by one C-function for each entry point of the component, statically allocated data and set of implementing functions. Synthesis of composite ProSave components requires more effort in optimization to produce an efficient runnable representation, because it adds locks and synchronization of data transferred between internal components. One of the ProSys subsystem synthesis results is a number of needed task for ProSave component execution. For future system analysis, information about period, offset and deadline are included in the result.

The runnable representation of a virtual node is more or less set of a ProSys subsystems combined with information about resource allocation for guaranteed execution behavior. The last step of the synthesis process produces executable binary file based on knowledge of the targeted system from virtual nodes. It also adds an implementation of message channels used to send messages between virtual nodes.

The process design, allocation to virtual nodes, allocation to physical nodes and generating final executable files mostly intended for embedded real-time systems, but there is no limitation to use it in some other supported system.

2.6 Technological Background

Before we begin designing a library, it is a good idea to get acquainted with potential target operating systems and hardware platforms. Finding the intersection of similar key system attributes can help us create the middleware. During our work it has been shown, that adding support for new systems after commencement of development can be difficult if any of the key system properties is not fully supported.

2.6.1 Execution Support

In real-time operating systems (but not only there), we can find different approaches how to manage multiple routine execution support — tasks and co-routines. A good comparison of tasks and co-routines can be found in [8] in section "Getting Started — Tasks and Co-routines". The brief overview follows.

Tasks are independent real-time processes executed within their own context. The scheduler is responsible for choosing which task to execute, for example based on priority information associated with each task. Anytime the running task is swapped out, the scheduler has to ensure that the processor context is saved and correctly restored when the task is swapped back in. In order to achieve this each task has to have its own stack to which the information will be stored.

Co-routines are intended for use in small systems with limited memory. They are similar to tasks, but they do not require individual stack for each routine. All the co-routines share a single application stack, that reduces the amount of required memory. The role of the scheduler is limited there, because co-routines use cooperative scheduling.

As we mentioned earlier (in section 2.6.5) some operating systems are missing support for dynamic task creation. This situation requires a different style how to write a program. There is not a user defined main function from which other tasks are started, but they are defined in special configuration files. The final C implementation with all definitions and statically allocated space is generated from these files.

There is problem how to generate the configuration files and keep platform independency of generated code as much as possible. A feasible solution is to use macros for task creation in the main function or kernel configuration file.

2.6.2 POSIX

POSIX (stands for Portable Operation System Interface [for Unix]) consists of a set of specifications to enable the portability of applications software across Unix environments. However, POSIX standard is not limited to use only in the Unix environment. The great strength of this standard is in significant reduction in effort and time for porting application to another conforming platform.

Pthreads is a POSIX standard defining an API for creating and manipulating threads [18]. The library iRTncludes support for mutexes, condition variables, read/write locks and barriers. However, there is a problem with POSIX semaphore API compatibility on Mac OS 10.6, that was solved by using platform dependent API.

Socket interface (Berkeley) is an API dedicated to communication between hosts using TCP/IP. It is used to send or receive packets through sockets of different types (stream, datagram) using several supported protocol families. It is also part POSIX standard described in section "2.10 Sockets" in [17].

Using almost any POSIX compatible platform gives us the advantage of the wide range of free programming and especially debugging tools that make the development process easier.

2.6.3 FreeRTOS

FreeRTOS is a real-time operating system (RTOS) for embedded devices. Based on information available in [8], it can be run on several supported architectures (for example: ARM, Atmel AVR, AVR32, x86, PIC) and for each officially supported architecture a pre-configured example application demonstrating the kernel features is included. The kernel is very simple and actually really small. The core of the kernel is contained in only three C files.

Despite its simplicity, FreeRTOS supports a wide range of inter-task communication primitives. A primary form of inter-task communication is provided by queues usually used to send messages between tasks. Binary semaphores and mutexes are very similar primitives, however only mutexes include priority inheritance mechanism. Semaphores with defined maximal value — counting semaphores — can be seen as queues of defined size same as maximal value of the semaphore.

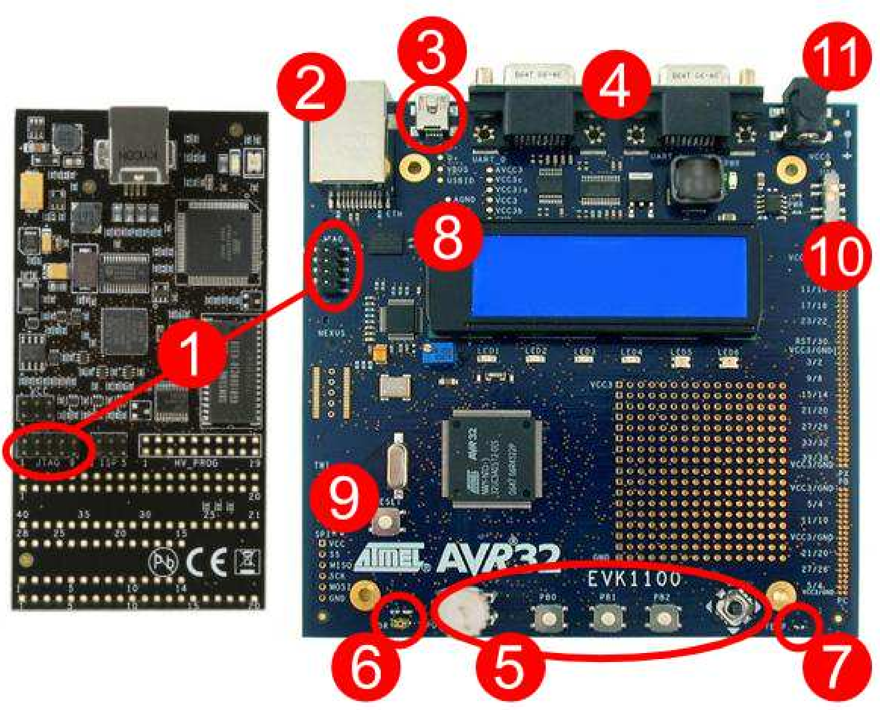


Figure 2.2: AVR Dragon and EVK1100 Evaluation Board

FreeRTOS provides good support for all necessary functionality of middleware for the testing platform introduced later on. Further, we describe a library used for network communication support.

LwIP is an implementation of the TCP/IP protocol stack with the focus on small memory usage and code size, which makes it suitable for systems with limited resources. It provides three levels of API in order to reduce processing and memory demands at lower level and provide compatible sockets API at highest level.

The EVK1100 evaluation kit (see right board in figure 2.2) with the AVR32 AT32UC3A micro-controller was chosen as a testing platform for the middleware port. Uploading developed program and debugging is realized by AVR Dragon programmer (see left board in figure 2.2) connected on JTAG (no. 1). The description how to the setup development environment is included in next section.

The EVK1100 kit is equipped with a rich set of peripherals displayed on figure 2.2 and the most important ones are marked with numbers 2-10. Other devices can be connected on Ethernet port (no. 2), USB (no. 3) or 2 serial ports (no. 4). The board can be powered via either DC input (no. 11) or USB 2.0 (no. 3) port. The potentiometer, set of buttons and joystick (no. 5) are placed at bottom part of the board. A light sensor (no. 6) and temperature sensor (no. 7) are located at bottom corners. A blue LCD (no. 8) can display 4 lines of 20 characters. Remaining not described controls are restart button (no. 9) and power switch (no. 10). MMC card reader is not shown on figure 2.2, because it is situated on the other side of the EVK1100 board.

2.6.4 AVR Studio

The AVR Studio is based on the Eclipse platform with extensible plug-in architecture. The main reason to choose AVR Studio was to have integrated developing and debugging environment together with possibility to setup target architecture and upload the runnable binary file into a device.

The usage of this environment has the main advantage in the integration of building tools for our hardware, FreeRTOS, LwIP and drivers including examples. It also allows to upload final binary file to the device directly from main window. The process how to setup the environment is described in Appendix C.

2.6.5 LEGO Mindstorm NXT 2.0

The LEGO Mindstorm NXT 2.0 is an advanced toy containing several types of sensors (contact, ultrasonic, color), servo motors, large matrix display, programmable control unit and of course versatile brick building system. Communication with a computer or with other NXT Intelligent Bricks is possible using a USB cable or bluetooth wireless technology.

Nowadays, numerous operating systems, drivers and applications are available to use with the NXT brick. We have chosen *nxtOSEK* platform based on C language support and API for sensors, motor and other devices. According to information from the nxtOSEK web page [19], it consists of device driver code of *leJOS NXT*[3] and two possible real-time operating systems:

- TOPPERS/ATK provides real-time multi tasking features proven in automotive industry
- TOPPERS/JSP provides real-time multi tasking features compiling with Japan original open RTOS specification μITRON 4.0

[3]LeJOS web page http://lejos.sourceforge.net/

A huge limitation of both included real-time operating systems is in providing only static API for creating tasks and synchronization primitives. Two ways of dealing with the problem were experimented, however none of them has been completely finished due to limited time.

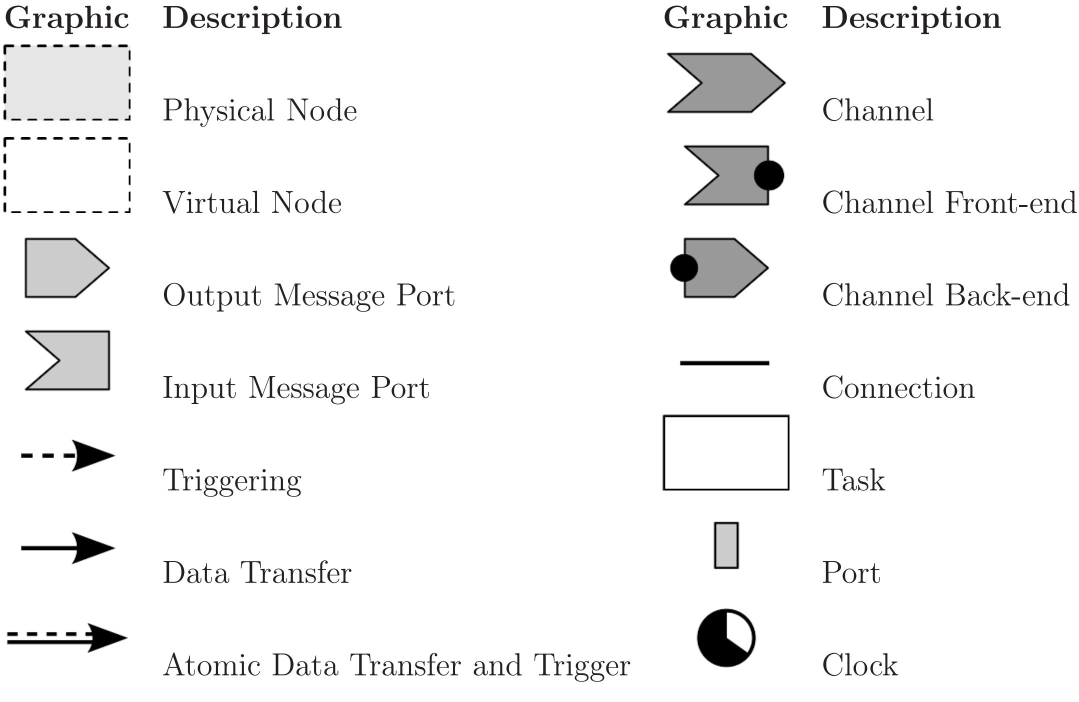
Because the hardware was brought after initial design and implementation had been done, overwriting existing library would require a major effort. Fortunately, adding basic support to limited dynamic tasks and semaphores creation was possible and additional work on this problem is planned as a future work.

Chapter 3

Modelling and Comunication Design

The significant advantage of component based development is the possibility to build system by assembling prepared components. Such components could be composed to larger units of defined functionality which are later deployed to specified hardware. The goal of the designed middleware implementation is to allow simple composition of the components and provide transparent communication of these component regardless of where they are deployed.

This chapter describes and explains the important design decisions made during the development of the ProCom middleware with main focus on communication. We start from the top abstraction layer describing situations that led to key decisions in the design and implementation of channel communication.



Graphic	Description	Graphic	Description
	Physical Node		Channel
	Virtual Node		Channel Front-end
	Output Message Port		Channel Back-end
	Input Message Port		Connection
	Triggering		Task
	Data Transfer		Port
	Atomic Data Transfer and Trigger		Clock

Table 3.1: Graphical elements

Before we present the design description, we provide information about a graphical notation of elements in Table 3.1. The graphical notation is similar to ProCom elements

notation and there is a clear corresponding coupling between them. We have created the different notation to express that the diagrams are showing synthetized elements of the ProCom model.

3.1 Physical Nodes

When the system grows in number of connected components (e.g. car systems) we have to deal with its complexity and possible resource over-utilization. To avoid this problem some extra-functional properties of the components allocated to defined hardware are kept in the system model and compared with the hardware capabilities during the design time. Whenever reallocation of some components is required we would like to achieve a maximal reusability of the previously generated code.

The deployed system is decomposed to one or many physical nodes that can be connected. When we talk about a physical node, we mean a hardware device containing a CPU capable of task execution. Several properties such as hardware platform, operating system and communication ports have to be defined for each node. The middleware participates in the initialization of hardware components and establishing connections based on information gathered from the overall system model.

There is a complicated communication part of the initial connection check. But first we elaborate why the check is (or is not) necessary. Because system can consist of more than one physical node and it is needed to ensure that all nodes are fully functional before a message is sent. For example in the situation when one node sends message to another one, but the task located on the second node has not been activated yet. Hence it can not handle and react to incoming message.

This situation implies that the sender should check if the reader is ready or not, and only when the reader is ready the sender can activate its tasks. However, this can lead to a cyclic dependency between physical nodes. We can also see the fact that even if the connection is successfully established, the system may not be able to run correctly. From this point of view we have the system where every message (including the first one) has to be delivered and we should be also able to detect that. However our channel communication has only one-direction, hence we would need to extend its functionality. The current solution uses unreliable TCP connection, hence we decided to ignore the possibility of message loss and we assume that the node is ready when all its connections are established.

Furthermore, let us shortly discuss the possibility of accessing specific hardware parts (sensors, buttons, displays, etc.) required by tasks included in a virtual node. The first option is to create an universal drivers library, however it would be hard to cover differences in the hardware capabilities. Another way is to develop specific components with defined interfaces, which are usually tightly bound to the used hardware. Even if we consider using some abstract ProCom drivers or hard-coded functionality inside components there is still a tradeoff between reusability and effective utilization of resources.

3.2 Virtual Nodes

As we mentioned in section 2.4, a virtual node is a set of ProSys subsystems. From a deployment point of view, we can look at virtual nodes architecture as an abstraction of target hardware devices which allows flexible allocation of the subsystems to different platforms. Information about these subsystems (periodic and event driven functionalities and appropriate data structures) is collected during the synthesis process and the virtual node definition is formed by combining the information. Therefore, the virtual node structure describes a set of functionalities (realized by *tasks*) and data structures.

The internal structure of task (shown in section 4.2.1) unifies the way how to store the information about functionalities of different type. This virtual node implementation uses the advantage of simplified access to the information about tasks structured as an array. The start-up procedure easily calls initialization functions (`init_routine`) of each task and than it creates threads from methods `periodic_task` or `sporadic_task` depending on type of the task. Besides information about the tasks in the definition of the VN there are included lists of incoming and outgoing message ports.

However, we would like to mention that this structure was created for testing purposes and for further use it is necessary to consider the problems of multiple allocation of the same VN to one physical node. Consequently the support of multiple instances on the same physical node is missing for now and to use the same functionality of a virtual node multiple times it is necessary to create the copy of its code and change variables names.

3.3 Channels

The objective of this section is to show steps of the designing system communication. We focus on top two layers — physical and virtual nodes. Let us remind that channels are used for virtual nodes composition. Virtual nodes are not connected to each other directly, but they are using channels instead. This section also discuss general issues of the message channels from the runtime point of view.

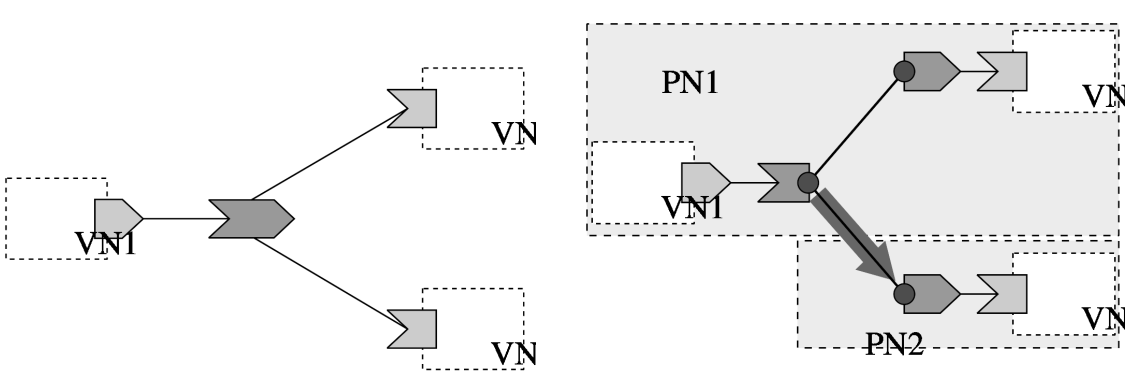


Figure 3.1: Single channel design

Figure 3.2: Local and network connection

Example 1. To simplify the architecture example we will focus only on the physical layer and virtual nodes communication. Let us start with three virtual nodes (see Figure 3.1) communicating by passing messages through one message channel. More specifically the first VN sends messages to the other two. Now we have designed communication and the next step is to allocate VNs to physical nodes. For illustration, we deploy *VN1*, *VN2* on *PN1* and *VN3* on *PN2*. A challenging phase of the realization is the right division of channel implementation to the right physical nodes and their connections. One possible approach, how this channel structure could be deployed, is presented in Figure 3.2 where the gray arrow illustrates direction of inter-node network communication.

In general, we create a channel front-end/back-end for each physical node if there is at least one writer/reader associated with the channel. Let N be a number of front-ends and M be a number of back-ends of the same message channel. Then we need to define $M * N$ connections. In the previous example we have shown a simple one way communication where a shipped message was distributed to different nodes. However, consider the following scenario.

Example 2. Assume that we need to send replies back to the first physical node as it is shown in Figure 3.3. Therefore we add a new channel to the system model and connection between its back-end and front-end. We could straightforwardly define another connection, but if the type of physical connection provides two-way communication we take this advantage into account and use it for both channels (see the double-headed gray arrow grouping connections to the single network connection in Figure 3.4). An important motivation is the reduction of allocated resources on both physical nodes.

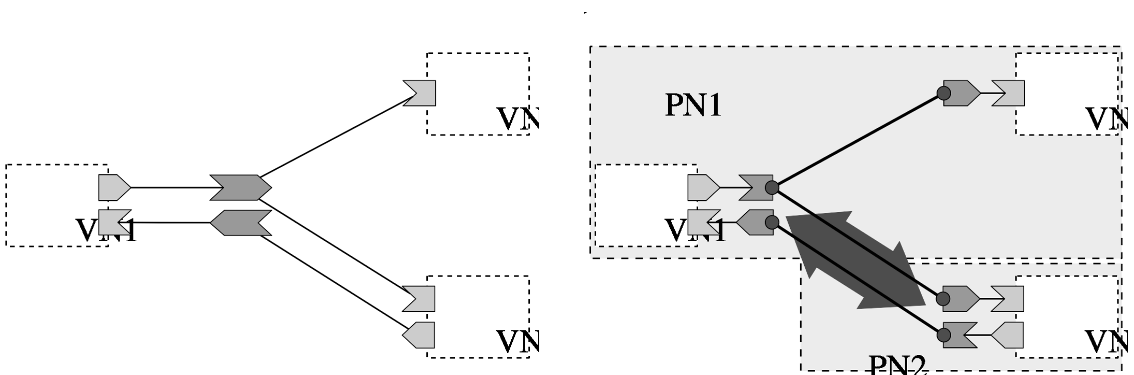


Figure 3.3: Two channels

Figure 3.4: Two way connection

This solution was chosen because embedded systems typically have resource limitation in terms of maximum open connections. For example LwIP library uses a single task for each open connection. On the other hand, sharing a connection between multiple channels leads to the problem of message delivery to the right channel on the receiver side as we present in following example.

Example 3. Let us slightly arrange previous example as it is displayed in Figure 3.5. We keep the deployment layout same then we get the collocation of multiple channels

to one physical connection (the gray arrow in Figure 3.6). The problem arises when the received message should be moved to the right channel on physical node PH2. Our solution consists of an additional channel identifier in the message structure. A small disadvantage is the larger size of transmitted data.

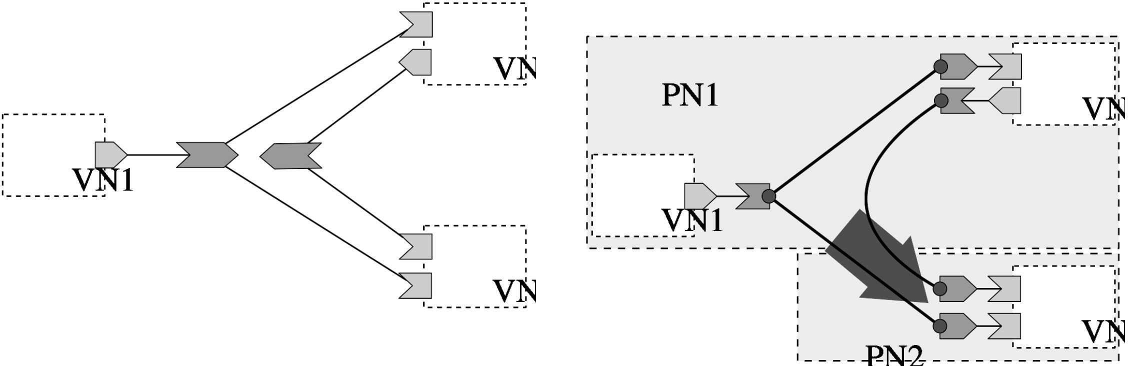


Figure 3.5: Two channels (another layout)

Figure 3.6: Channel collocation

Chapter 4

The ProCom Middleware

The term *runtime environment*, as it is used in the context of this thesis, describes a set of services providing an API intended to allow running of the same component's code allocated at virtual nodes (VN) on different platforms. To design and implement an API we have to look closer at the used operating systems (OS) features and communication semantics. In the following sections the platform abstraction layer and middleware API are described.

Because there are no dynamic changes allowed after the deployment process is finished, all the information about nodes and physical connections among them can be included directly in the generated code. This solution has many advantages in embedded systems, where it would be complicated to implement the reading of configuration files. Because of the fixed configuration storage there is no need for change mechanism implementation.

According to the requirements it would be beneficial to design the library architecture as two-layered to expandable parts. The top layer provides the API defined in section 4.2 that supports tasks creation, inter-component communication and virtual node creation. The underlying part, (described in section 4.1) provides an unified access to shared resources, operating system, and some common hardware.

4.1 Platform Abstraction Layer

To create an easily maintainable runtime library implementation it is necessary to define a certain level of platform abstraction. It helps not only with maintaining but also with porting the library to other hardware architectures or operating systems and in an ideal case without any changes in the top layer. The abstraction layer provides simplified access to system primitives such as threads, semaphores, mutexes and timers. Most of these primitives are same in the major part of used real-time operating systems. However, several problems have appeared.

The operating systems running under the middleware provide varying hardware abstraction. We have focused only on hardware needed for communication such as Ethernet and serial ports. The hardware initialization is normally provided by the operating systems, but the function `hardware_init` is implemented to initialize the hardware of embedded systems, where in some situations the initial values need to be written to the hardware registers. However, this part is hard-coded in the middleware which is not the optimal solution. The function `connection_init` is intended to initialize communication ports

and library for network communication (e.g. Ethernet listener thread in the LwIP library, interrupt handlers and peripheral devices).

4.1.1 Threads

The basic purpose of the runtime environment is to enable independent execution of multiple components. In embedded or real-time systems threads are mostly called *tasks*. They are performing an activity in their own context.

Functions `progress_thread_create` and `progress_thread_exit` wrap system specific calls for creating and exiting system thread. These functions are available only on systems that support dynamic task creation.

4.1.2 Synchronization

Process synchronization support is needed to ensure that certain parts of code do not execute concurrently while they access to shared resource. If one thread attempts to get access to a resource that is already in use by another thread, the thread will be blocked until the resource is released.

However, blocking the thread is undesirable for many reasons mainly while high-priority task is running. Blocking in real-time systems brings more problems with their analysis and improper usage of locks or interactions between them can lead to undesirable state of system (deadlock, live-lock or priority inversion). In any case system has to provide at least one type of synchronization primitive.

We have implemented support for basic operations with semaphores (`progress_sem_init`, `progress_sem_signal` and `progress_sem_wait`) and also mutexes (`progress_mutex_init`, `progress_mutex_lock` and `progress_mutex_unlock`).

4.1.3 Sleeping

In some situations it is useful to resume task execution. A sleep system call places the running process into an inactive state for a defined period of time. The time parameter usually specifies only minimum amount of time that the process is inactive. The passive waiting is provided by `progress_usleep` function. More accurate timed sleeping that takes into account priority of the sleeping task (if it is supported by OS), is implemented in function `wait_for_next_period` and introduced in section 4.2.1.

4.1.4 Memory Allocation

There is a problem with dynamic memory allocation, because of lacking or restricted support of functions like `malloc` in many RTOS. For example the dynamic allocation is not allowed after a scheduler is started in some systems. In case that dynamic allocation is unavoidable, it is possible to use a statically allocated memory pool.

4.2 API proposal

The API is supposed to help with producing reusable code for components by the developers as well as easily generated code for virtual and physical nodes. To achieve these goals we use the abstraction of the system primitives for synchronization (e.g. semaphores, mutexes), threads creation and sleeping for periodic actions. The set of auxiliary functions is also a part of the library to facilitate initialization of the hardware components and ProCom runtime environment in cooperation with underlying abstraction layer.

Before describing individual functions from the API, we provide an overall information about major functions. The middleware aims to provide easily usable data structures and functions for modeling, initialization and creation of elements at different ProCom modeling levels. Note that since the components are hierarchically composed, it is not necessary to use some of initialize functions explicitly, but they are called from the upper level initialization function automatically.

Let us provide a simplified list of the most important functions divided into categories according to model layers (full list can be found in appendix A.1):

1. Task
   (a) `Get_message`
   (b) `Send_message`
   (c) `Write_data`
   (d) `Trigger_port`
2. Virtual node
   (a) `Init_task`
   (b) `Create_task`
3. Physical node
   (a) `Init_virtual_node`
   (b) `Create_virtual_node`

In addition to the above functions we provide a wide range of extra macros (full list with description is provided in appendix A.2) for generating static data structures required for system modeling. These macros simplify the code generation templates and serve as interlayer allowing small changes in library functions and structures without requiring a change in a previously created template.

4.2.1 Task Creation

Model transformation for the system synthesis generates two types of tasks that differ in the way they are activated. The first type of task, called *periodic*, is formed by the combination of a clock element and the entry function of a connected ProSave component. We need to ensure further periodical activations at defined time, regardless of possible

varying execution time of the component entry function. The second task type is called *sporadic* and the name comes from the nature of the task to be executed in response to sporadically incoming messages to a connected message port.

Looking for solutions we focused on the question of providing easily maintainable code, that is open to slight modifications whenever new requirements on the middleware are discovered. The final implementation came from solution dealing with the problem of creating a common function for virtual node initialization, which is described later section.

The initial idea was to have one initialization function for each task type as it is described subsequently: `create_periodic_task` (`create_sporadic_task`) initializes the data structure of a periodic (sporadic task) activated by timer (activated by incoming message) and is used as the entry function of corresponding thread.

However, during the implementation it became clear that it would be better to use only one function for creating tasks no matter of its type. The proposed solution has advantage in easier code generation, because all information are stored in a single data structure (shown in listing 4.1) and can be also referenced from virtual node to which they are allocated.

`Init_task` executes initialization routine.

`Create_task` creates new thread from functions `sporadic_task` and `periodic_task` that include infinite loop with different internal implementation (described below).

Listing 4.1 Task data structure

```
typedef struct __progress_task {
    enum { TASK_PERIODIC, TASK_SPORADIC } type;
    void (*init_routine)();
    void *(*start_routine)(void *);
    void *arg;
    union {
        const progress_time_t   period;
        progress_sem_t *        trigger;
    } info;
    int                         priority;
    /* private: */
    progress_thread_t           thread_handler;
    struct __progress_node *    virtual_node;
} progress_task_t;
```

The process of creating tasks is split to two phases. In the first phase `init_routine` is called and internal task data are set up. This task memory initialization can usually be executed in any order, however there is problem with hardware initialization (in general with any shared resource) that does not have any abstraction layer handling multiple accesses.

During next phase the corresponding thread is initialized and then suspended until all tasks of all virtual nodes have been initialized and all physical nodes are ready to

communicate. Detailed description of system startup problem is introduced in section 3.2.

Internal Implementation

The internal implementation of tasks uses the following functions to achieve desired behaviour while preserving platform independence. They are not directly part of the API, but they are closely connected to task definition.

`Wait_for_other_tasks_init` performs synchronization step. It waits for all the tasks of a virtual node to be initialized.

The next two functions are used in the periodic task loop. Their implementation is platform dependent with a view to gain benefits from operating systems that directly support periodic actions.

`Compute_next_period` enables to know when the task needs to be waken up based on time from previous task execution. It does not seem useful using FreeRTOS or RESCH[1].

`Wait_for_next_period` waits until next period when the tasks can be executed again. It is called at the end of the working loop of the periodic task.

The last function is intended to wait for an external trigger at the beginning of the task loop. The activation can currently come only from an input message port that is associated with the task. When the task is woken up, the incoming message must be read from the message port.

`Wait_for_activation` waits for a trigger to execute the routine of a sporadic tasks.

4.2.2 Virtual Node Creation

A virtual node is a set of ProSys subsystems, however for its successful initialization we need access to the tasks of each subsystem. Moreover, every virtual node will have a generated list of the message ports which have to be initialized before task activation.

`Init_virtual_node` initializes the tasks and message ports allocated to a virtual node.

`Create_virtual_node` creates and initializes the data structure of a virtual node.

Before tasks can be activated, all virtual nodes (not only on same physical node) have to be ready.

`Wait_for_other_nodes_init` is a synchronization function that waits for the other VNs initialization before to launch the VN tasks.

[1] A Loadable Real-Time Scheduler Suite for Linux (RESCH) is available on web page http://www.ece.cmu.edu/~shinpei/resch/.

4.2.3 Inter-node Communication

The only allowed way how a task can communicate with the other tasks located at a different virtual node is realized by message passing.

`Send_message` function packs data into a message and sends the message to an associated message channel. The action of packing and sending is atomic.

`Get_message` function reads data from an incoming message.

Because the semantic of message sending allows some special cases, we decided to split the functionality of message sending into two following independent functions. Together they do the same as `send_message`, but a message in the message port can be overwritten before it is triggered.

`Write_data` only packs data into a message and stores it in the associated message port.

`Trigger_port` sends last stored data from the message port to the associated message channels.

If data are sent over network, it is necessary to convert them into the expected format for the communication protocol. Functions should be implemented for converting simple data types before they are sent. A typical examples of functions converting integers *to* and *from* the network format are `htonl` and `ntohl`.

4.3 Data structures

This section describes data types used on different levels of a system composition. We have experimented with dividing the information into two sets. The first one includes the information needed for establishing a connection and the other one contains the run-time information about the connection (e.g. socket identifier). This approach has shown to be really helpful, because it simplifies code generation and it also allows to hide the implementation of physical connections.

Based on the communication design introduced in previous sections following structures were implemented to achieve desired behaviors.

4.3.1 Task Port

As the name suggests, this structure is used by tasks (services) to read data from an input message port or store data into associated output message ports. There are three supported ways how the task port can be connected to an output message port — only write data, only trigger port and atomic write and trigger. Similarly, the task can read the latest data from an input message port. The term – *latest data* – indicates content of message that has not been read by all associated sporadic tasks (triggered by event of the incoming message). In case that there are only periodic tasks reading from the message port they will read data from the newest received message.

Output task port is connected to possibly multiple message ports with attributes defining if the *data connector* between task port and message port is present and if the message port should be *triggered* when the data is written. We use the attributes rather than different methods in the API for the different types of message port associations.

Input task port can be connected only to a single input message port. In addition to the connector it includes only one other attribute that is *true* if the task is activated by the message port.

4.3.2 Message Port

Message ports provide a *mapping between message passing and trigger/data communication* [2]. Taking into account various demands to reliability of message delivery we have decided to add buffers to the message port instead of channel. Moreover, it saves coping of data in case that the port is assigned to multiple channels. Another advantage of this solution is in simplifying the analysis of necessary buffer size to avoid message overwriting.

During the implementation work we faced a problem of minimizing data coping while using only statically allocated memory. Moreover, it is desirable to reduce the length of critical sections (buffers are possibly shared by multiple writers). After a closer exami-

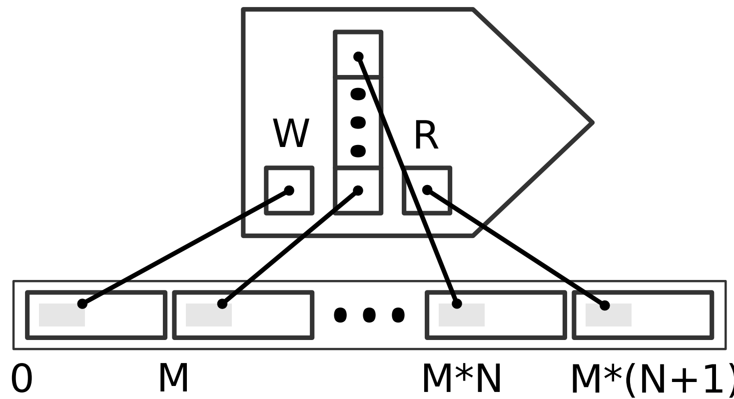


Figure 4.1: Message port structure initialization

nation of several possibilities (e.g. data stored in the task port, cyclic buffer for whole messages in the message port), it has been concluded that the message port should contain a cyclic buffer of the pointers to space for whole message and extra pointers to free memory for writing (W) and reading (R) message. The extra space for writing message was chosen for keeping the message outside the buffer before the message port is triggered. Hence, the messages from the buffer can be sent by other task meantime. This also applies symmetrically for reading and sending messages from the message port.

During initialization phase all pointers are initialized as it is displayed in Figure 4.1.The main advantage is in the minimization of unnecessary data copying because we swap only pointers W and R with first unused and used item from buffer respectively. Let N be size of buffer and let M be size of the message. Then the memory of size $(M + sizeof(void*)) * (N + 2)$ is used.

Input message port contains the defined space for at least one whole message. Whenever the message is received, the message data are written to the message port buffer. If all *tasks* triggered by the message port have read the message data from R, then the pointer to the oldest message data in the buffer is swapped with R pointer and the associated tasks are activated.

Output message port is an inverse to the input message port. When the trigger is activated, the port stores a message with the data currently available on the input data port to its buffer and wakes up the sender task. Whenever the sender task is waken up, it sends messages from buffer belonging to the message port that woke him up to all associated *channels*.

Experiments have shown that the manual generation of structures of message ports with various options brings a lot of mistakes and, in addition, for minor modifications of the library it was needed to rewrite a large amount of already created code. To solve that problem we have defined a set of macros for the code generation which also makes possible to do some minor changes in the library without affecting existing code, and most importantly without changing the code-generating templates in the PRIDE tool. Equally important facts shall be revealed that this can solve most problems with generating different code for various platforms.

4.3.3 Channel

The channel is a kind of abstraction that stores information about connections used to send messages to the other virtual nodes possibly located on different physical nodes. It participates in ensuring compliance of transmitted data.

A channel is logically composed of a set of *front-ends* and *back-ends* as was already described in Section 3.3. This approach brings several advantages. First, when a virtual node is deployed to different physical node only the information about connection between physical nodes is changed in channel front-end structure of the sender. Similarly, information about new channel back-end is saved to the connection structure on receiver side. Using two independent structures can reduce the memory requirements on nodes where none of associated input or output message port exits, and it also simplifies the code generation.

Front-end part holds information about message channel *connections* and the *size* of allowed message payload. Another possible solution would be the runtime type checking of data during saving process. Because no dynamic changes are allowed after system is deployed, the size check seems sufficient.

Back-end is supposed to check *size* of incoming message and distribute it to the *message ports* associated with this channel and placed on this physical node. The whole process was described in Section 5.2.

4.3.4 Inter-Channel Connection

Connections define how the message is transferred between each front-end to all back-ends. During the development three types of connection were tested — local, TCP/IP (Ethernet) and serial line.

We focus on issues related to the implementation of interactions between channels and physical communication media. First we tried to minimize the number of necessary physical connections in a way that if possible, they are used for communication in both directions by multiple channels. Then we separated the information needed to initialize the connection from the handlers required for sending and receiving data. Here it was discovered that the connection handlers are similar (or same) on all tested platforms, but data structures needed for creating connections of same type are different. In this case, we will benefit from code generation using predefined macros.

Local connection is the simplest type of inter-channel communication. This connection is usually used for the communication within a single physical node — more accurately — within a single system process. The implementation itself is very simple and consists of a simple distribution of the sent message from the channel front-end to all input message ports associated with the channel back-end.

Network connection (TCP/IP) is more complicated, since the TCP/IP communication is established by three-way handshake between client and server. At first all defined server *ports* are opened for listening and they are waiting for clients whose IP address match up with any IP address from the *list of allowed clients*. This list also includes references to inter-channel connections, that is set up every time when allowed client connects to server. On the client side it is defined only *IP address* and *port* of the server and the inter-channel connection is set up right after the TCP connection is established.

Serial line connection has the most differences of implementation across operating systems. While on UNIX systems working with a serial port is as simple as working with regular files, on the evaluation board the hardware has to be initialized first and we need an extra task to handle every connection. The information needed for initialization mainly consist of a port identifier, speed, data bits, stop bits and parity. This implementation is highly experimental.

Because sometimes it is necessary to create system tasks for handling physical connection, system events or library call-backs, we decided to store all information for connection initialization of a physical node in a specialized structure. This structure is used by the connection manager during system startup and the decision about creating system tasks is done there based on the number of connections of certain types.

Once we begin to allow the interconnection of different node architectures, it is necessary to consider possibly different interpretations of the transmitted binary data and their compatibility. The most common problem are endianness and serialization of data structures. We have focused only on the solution that converts simple data type to universal network type to avoid endianness problem during transmission of a message from one physical node to another one. Currently it is necessary to do the data transformation before they are stored in the message structure.

Finally, we would like to point out on problem that has arisen during testing of different variants of interconnection of physical nodes. Imagine a situation with three physical nodes P_1, P_2 and P_3, where P_1 has only one serial port connected to P_2, but it needs to communicate with the both remaining nodes. In case the connection was TCP/IP and P_2 supported routing, everything would be fine. However, we need to solve the routing

on application level. We see these two possible solutions. First one uses an extra task associated with an input and output message port which forwards the message to desired destination (a possible implementation can be simillar to Listing 6.1 and the overall design of the task including the message ports is shown in Figure 6.2). The other approach is to add “routing tables” to connections with information like this: “messages from channel X forward/copy to connection Y”.

Chapter 5

Message Communication

We have described communication based on messages, where the channels are engaged as distributors of messages to the other virtual nodes using a defined set of connections. In this chapter we look closer at sending and receiving data passed through the channels and we also describe precisely and in more detail the concepts of binding components to message ports. The chapter concludes with a description of the structural elements of communication.

5.1 Message Sending

The semantics of sending message from an output message port says: *"Whenever the trigger is activated the output message port sends a message with the data currently present on the input data port"* [11]. The main requirements for middleware functionality, that we have focused on, are:

- non blocking behavior of store function;
- simple trigger function; and
- atomic combination of store and trigger functions (save and send data encapsulated in one message).

An output message port can be triggered by one or many tasks, and similarly one or many tasks can write data to the same message port. Each task can also be associated with zero or many message ports. Three different ways how to associate one task with message port exist and they will be described below.

Example 4. Probably the simplest and mostly used situation is sending data right after it is copied to message port. Demonstration of such a task T_1 with associated message port is shown in Figure 5.1. When the trigger and data come from the same task port, the middleware can ensure that the stored data in the message port will not be overwritten before it is sent.

In our solution we lock the message port for writing when a task starts copying data to the message port and we release the lock after it has pushed them to the message

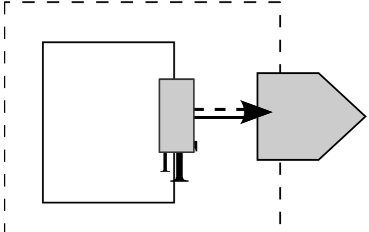


Figure 5.1: Atomic message send

port buffer (a detailed description of message port structure is given in the section 4.3.2). Though the critical section is very short, in situation when using locks for writing is prohibited, it is possible to use a single message port for each task (and remove the locks). The second proposal consumes more memory, because every task port has its own space where the whole message can be stored. When the message is prepared, the pointer to the message from the task port is swapped with a pointer to memory space for writting message from the message port buffer.

Example 5. Let T_1 be a task triggering message port M_1 without providing the data. There is only one problem, namely to prevent sending non-initialized data in a message, if the message is not supposed to have empty payload. This can happen at the beginning when task T_2 had not written any data to message port but T_1 triggered the message port (see Figure 5.2). Hence during the initialization period a default data should be written to the message port.

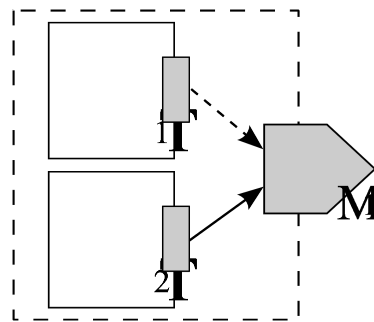


Figure 5.2: Independent trigger and data writing

When the data is stored in the message port and the message port is triggered, then it is pushed to queue of message ports waiting for send by sender thread.

Example 6. Let T_1 and T_2 be tasks writing only data to message port M_1. These two tasks do not care about sending the data. They only store the latest data from some sensors or computation into the message port. When some other task triggers that port, the message will be sent with the latest stored data.

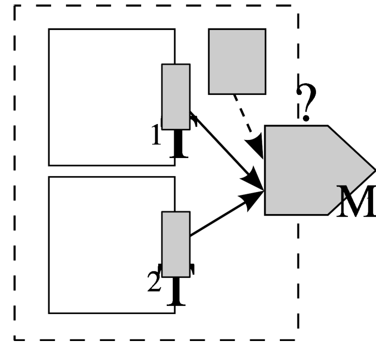


Figure 5.3: Multiple writers

To support this example it is necessary to add a pointer on the latest data to send. But it is also necessary to add write lock for that pointer to prevent incomplete writing and moving the pointer to message port buffer on architectures where writing pointer is not atomic.

In the next subsection we describe the work of the sender thread and the advantages and disadvantages of its use.

5.1.1 Sender Task

This section will clarify the situation, which resulted in the use of solutions with an extra high priority task for processing outbound messages at every single physical node.

Motivation: Consider the situation where two tasks T_1 and T_2 are sending messages through one shared connection C_1 and T_2 is also sending messages through connection C_2. If something gets wrong with C_1 during sending message from T_1 then the tasks are blocked and even T_2 can not send its data. Message passing over physical media is usually performed by a blocking function. It is necessary to make sure that the message was sent whole at once.

Using buffers on different levels can help to solve this problem. We can look at the task message sending as the Producers-Consumer, where T_i are producing messages and sender is consuming (sending) them. A question is, what to do in the situation when any buffer or queue is full. If it happens in a message port buffer, then the oldest record is overwritten. It may be seen as a too old message which is not worth to send or it was a problem with definition of buffer size.

A normal buffer with data needs to copy them every time they are stored in buffer or read from buffer. The other approach is to store only pointers to the data. Because it is not good idea to use dynamic allocation (`malloc()` and `free()`) in real-time systems (it is hard to predict how much time the allocation will take), the space for the data is allocated statically.

Example 7. Let T_1, T_2 and T_3 be periodic tasks with periods 500ms, 300ms and 200ms and all of them are sending message to same message port M_1. Let M_1 have buffer size equal to 2. At the beginning the sender has no work to do and has the highest priority. So he can send the messages immediately. However every 3000ms[1] three new messages are

[1]LCM — Least Common Multiple of numbers 500, 300 and 200.

created and queued for sending. If the sender is busy at that moment, the oldest message will be overwritten. The solution is to extend the buffer size.

But if the enlargement to size greater than number of tasks writing to message port does not help either, then it is probably necessary to upgrade hardware or split functionality to different physical nodes. However in some cases it does not matter that one or more messages will not be dispatched.

If we look at overwriting of the oldest message in the buffer in more details we find out that it is not always allowed to overwrite the oldest record in the buffer, because it can be used by the sender by that time. The solution consists of using an extra space outside the buffer that sender swaps with the oldest record.

When the sender is woken up, it reads the message port from the queue of ready message ports and does the following operations until the queue is empty: first it gets an array of associated channels and then for each channel from the array it sends the message through every connection used by the channel.

5.2 Message Receiving

The way in which individual messages are retrieved from the physical link is dependent on the system implementation and used libraries. This section introduces the process by which incoming messages are processed as soon as they are completely received from a connection.

When a message is received, the corresponding information about channels associated with the connection are compared with the channel identifier included in the message header, and the data are distributed to the matching one. The comparison is needed because of the possibility to send messages belonging to different channels through one physical connection between nodes as was described in section 3.3.

Whenever a message is passed to the channel, the data stored in that message is saved in the message port buffer and the message port updater is waken up. The purpose of having an extra thread responsible for this is explained in the next part of this section.

5.2.1 Message Port Updater

The message port updater is a high priority task which is used for doing work that is not connected with the main purpose of component tasks. The solution with an extra thread is not the only possible one. During the development we considered a possibility of storing data directly in task data ports, however it may require some unnecessary data copying. The reason why the message port updater has its own thread is illustrated in the following examples.

Example 8. Let T_1 and T_2 be sporadic tasks associated to the same incoming message port (see Figure 5.4). According to the semantics, the incoming message has to be read by either all or none of the associated sporadic tasks. The problem occurs when multiple messages arrive within a very short time interval. This can happen due to some networking problems or perhaps because several messages from different nodes were sent at once.

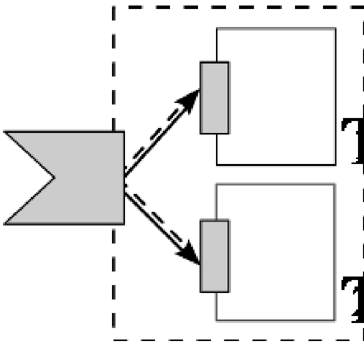


Figure 5.4: Two readers of same message

The additional work with updating message port structure has been done by tasks in the initial implementation without message port updater task. Hence every time the task read the message from message port, it had to check if all other tasks had already read that message and it also had to update message port properties in conformity with the situation. However, this had disadvantagous in point of consumed CPU time for the last reader, because it has less time that remains for useful computation and most probably this action would be executed by same one most of the time. In terms of more uniform distribution of work done in receiving messages by tasks, it is better to separate the work from the library function into the system thread, leaving only the minimum of necessary actions performed by the tasks.

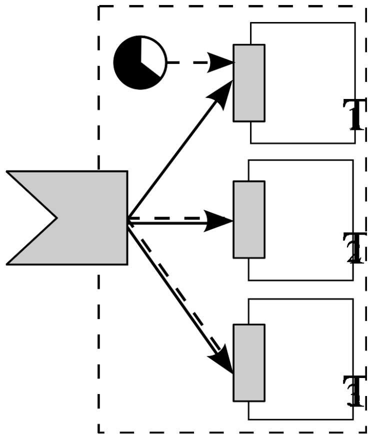


Figure 5.5: Periodic and sporadic tasks read the same message

Example 9. Let T_1 be a periodic task and let T_2, T_3 be sporadic tasks associated to the same incoming message port (see Figure 5.5). Task T_1 will possibly read the same message several times until both sporadic tasks read the message. This is the desired behavior that does not require any additional changes in previously introduced solution.

If it is necessary to make sure that the delivery of messages to T_1 is not delayed by aperiodic tasks, then separate message port only for task T_1 should be created during the design. Message port updater will keep only latest message in that message port.

Example 10. Let T_1 and T_2 are periodic tasks associated to the same incoming message port. In this situation when two or more periodic tasks read from one message port we do not guarantee that they will read same message. The situation is illustrated in Figure 5.6, where even if both tasks are activated at the same point of time, another message can arrive after only one of them has read the previous message.

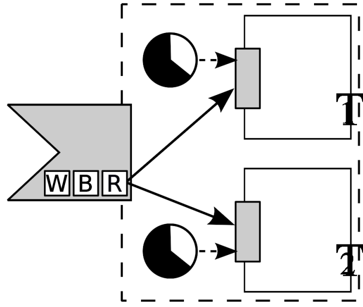


Figure 5.6: Only periodic tasks

In last example it is pointless to have the message port buffer (B) bigger than one. Even if more than one message arrive, only the latest one is used for reading (R) by all tasks.

5.3 Connection Reliability and Message Delivery Confirmation

The question of the connection reliability and the message delivery confirmation has been mentioned in section 3.1, but we would like add a few more suggestions for possible future ways of solving it.

The connection reliability is dependent on the type of physical connection. The real-time requirements and re-sending overhead need to be considered if we decide to supply the reliability feature of the physical connection in the middleware.

On the other hand, we can ask the question: “Is it really necessary to check the delivery status of every single message?” A negative answer brings new options. System start-up check can be simplified and we can check the message delivery status by adding special component for sending and receiving messages. The user (ProCom programmer) will only use the component whenever message delivery confirmation is required. This can be automated in the tool analogous to the semantic check of type of connected message ports and channels. However, there is a problem if the message has more recipients. In this case we would have to generate the checking component in conformity with deployment of nodes but it reduces re-usability a lot.

Chapter 6

Application Example

The objective of this trivial application is to present virtual node allocation and simple switch of physical connection in channel communication. Firstly, we introduce a complete overview how to create the applications using our middleware implementations and we also show its advantages in implementing changes in the final application. We would like to emphasize that it is certainly possible to use the library outside of the Pride, however it provides many other useful features.

Remember that the channel is logically separated into front-ends and back-ends, which can be connected with multiple connections as the layout of physical nodes requires. For every physical node, where at least one output message port is associated with a channel, the channel front-end of defined channel is created. Similarly, if there is an input message port located on a physical node, appropriate channel back-end is created.

6.1 Components

At the beginning we create two components - sender (see Figure 6.1) and receiver (see Figure 6.3). Sender has own internal status stored as integer and its only job is to increment own status and sends it to the output port when it is triggered. Receiver stores received data into its private variable and prints them on a screen (display).

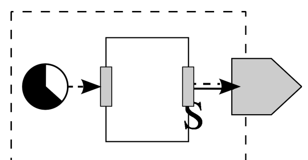


Figure 6.1: Sender

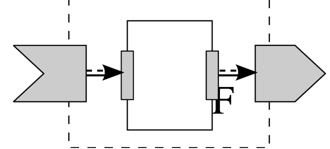


Figure 6.2: Forwarder

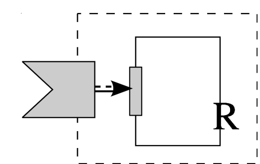


Figure 6.3: Receiver

To show component that sends and receives data, we have created a third component - forwarder (see Figure 6.2 and code on listing 6.1). Its data-structure comprises pointers to task, input and output data port and a supplemental variable. In the listing it is also shown the usage of functions `get_message` and `send_message`.

Listing 6.1 Forwarder

```
// part of forwarder.h
struct sys_Forwarder_t {
  progress_task_t * pg_task_forwarder;
  task_port_in_t *  in_port;
  task_port_out_t * out_port;
  int number;
};

// part of forwarder.c
void * forwarder_job(void * task) {
    sys_Forwarder_t * forwarder = (sys_Forwarder_t *) task;
    int message_data = 0;
    if (get_message(forwarder->in_port, &message_data) > 0)
    {
        forwarder->number = ntohl(message_data);
        send_message(forwarder->out_port, &message_data);
    }
    return NULL;
}
```

6.2 Virtual Nodes

In the next step we create the instances of previously defined components, tasks and ports. We recommend to use macros prefixed with letters `GEN_` and defined in the file *progress_macros.h*, however it is not required. The definition of component contain information about tasks, theirs ports and message ports instances used in the allocated instance. For a clear idea of the component connection design, the scheme is given on Figure 6.4, where `P1` and `P2` are message ports and `C1` is channel.

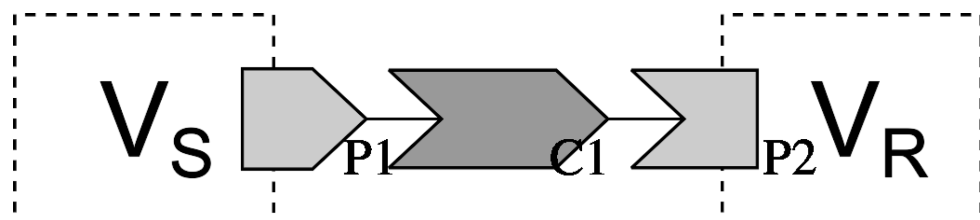


Figure 6.4: Virtual node connection design

Let components `S` and `R` are allocated to individual virtual nodes as can be seen also in Figure 6.4. The transformation creates a periodic task from the clock and component `S`, and a sporadic task triggered by message port `P2` from the component `R`.

In next step, we create task and message port instances to enable communication. The sender has one port that atomically triggers and writes data to message port `P1`. And the receiver port is associated with message port `P2`.

Listing 6.2 Task instances

```
/* Part of virtual_node_sender.c */
/* Periodic Task Instance */
progress_task_t sys_Sender_instance_task = {
        TASK_PERIODIC, sender_init, sender_job,
        (void *) &sys_Sender_instance,
        { .period = 500 }, TASK_PRIORITY_MEDIUM };

/* Part of virtual_node_receiver.c */
/* Trigger */
progress_sem_t sys_Receiver_instance_trigger;
/* Sporadic Task Instance */
progress_task_t sys_Receiver_instance_task = {
        TASK_SPORADIC, NULL, receiver_job,
        (void *) &sys_Receiver_instance,
        { .trigger = &sys_Receiver_instance_sr1_t1_sem },
        TASK_PRIORITY_HIGH };
```

Listing 6.3 virtual_node_sender.c

```
/* Message Port */
GEN_MessagePortOut(P1,sizeof(int),1)
/* Task Port */
#define Sender_out_port_AssocPorts(_) _(SS1_OUT, true, true)
GEN_TaskPortOut(sys_Sender_instance_out_port,
   Sender_out_port_AssocPorts)

/* Sender Component instance */
sys_Sender_t sys_Sender_instance = {
    .pg_task_sender = & sys_Sender_instance_task,
    .out_port = & sys_Sender_instance_out_port,
    .number = 0 };
```

The last step to complete virtual node is to define unique variables of type `progress_node_t` with all tasks and message ports.

6.3 Physical Nodes

To synthesize physical node finally, we select virtual nodes definitions and then we create channel parts instances with correct types of connections according to the real location of the other channel part and also add binding of the message ports to channels.

The easiest way to test our application lies in the allocation of both VNs on the same physical node `PHY1` (see Figure 6.5) and using local connection. Let us present the definitions of the channel parts and connection between them.

- Define the association of channel front-ends to connections:

Listing 6.4 virtual_node_receiver.c

```
/* Message Port */
#define P2_assoc_tasks(_) _(sys_Receiver_instance_task)
GEN_MessagePortIn(P2,sizeof(int),10,P2_assoc_tasks)
/* Task Port */
GEN_TaskPortIn(sys_Receiver_instance_in_port, P2, true)

/* Receiver Component Instance  */
sys_Receiver_t sys_Receiver_instance = {
    .pg_task_receiver = & sys_Receiver_instance_task,
    .in_port = & sys_Receiver_instance_in_port,
    .number = 0 };
```

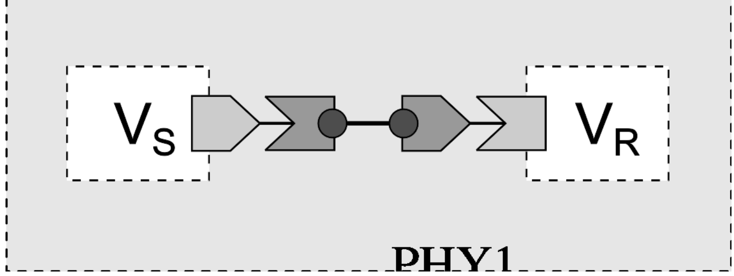


Figure 6.5: Local communication

```
#define C1_connections(_) _(connection1)
```

- Channel front-end has defined name, size of payload and previously defined associations:
  `GEN_ChannelFrontend(C1,sizeof(int),C1_connections)`
- Define the local connection writing to the channel back-end:
  `GEN_Connection(connection1,local,C1)`
- Define the association of message ports to channel back-end:
  `#define C1_ports(_) _(P2)`
- Similarly to channel front-end define channel back-end:
  `GEN_ChannelBackend(C1,sizeof(int),C1_ports)`

The last missing associations are between the output message ports of all allocated virtual nodes. The generation of needed structures is done by macros `GEN_VNPortAssoc` and `GEN_VNSetPortAssoc`.

This virtual nodes were also used to compose two physical nodes in order to test different types of communication (see Figure 6.6 and 6.7). With only the small change in connection type definition two different transport media can be used - Ethernet (TCP/IP) or serial line.

Listing 6.5 Testing Physical Node

```
#define VN_SENDER_P1(_) _(C1)
#define eP1 1 // port ID
GEN_VNPortAssoc(VN_SENDER_P1)
void physical_node_init() {
    GEN_VNSetPortAssoc(VN_SENDER,0,eP1,VN_SIMPLE_SENDER_P1)
    init_virtual_node(&VN_SENDER);
    init_virtual_node(&VN_RECEIVER);
    create_virtual_node(&VN_SENDER);
    create_virtual_node(&VN_RECEIVER);
    /* omitted part */
}
```

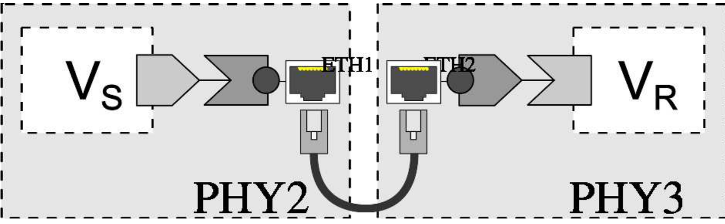


Figure 6.6: Ethernet communication

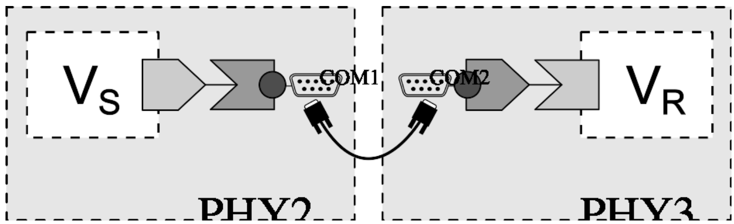


Figure 6.7: Serial communication

We propose the template generating physical connections (listing 6.6) and the structure for managing connections. The template also helps with keeping right order of the definitions and avoiding mistakes when redefining or adding new connections.

The macros from the listing 6.6 prefixed with string `FOREACH_` can be defined in the following way:

- Let us define the port `ETH1` for listening on port 2222, and accepting client with IP address 192.168.0.2:

```
#define ETH1_clients(_) _(ETH1_clients_1, "192.168.0.2", CONN2)
#define FOREACH_TcpPortListen(_) _(ETH1, "2222", ETH1_clients)
#define CONN2_AssocChannels(_)
#define FOREACH_Connections(_) _(CONN2, network, CONN2_AssocChannels(_)
```

- The port `ETH2` on the other physical node will connect the previously defined port with information defined subsequently:

Listing 6.6 Template for Physical Connections

```
FOREACH_Connections(GEN_Connection)
#ifdef FOREACH_TcpPortListen
FOREACH_TcpPortListen(GEN_TcpPortListen)
#endif
#ifdef FOREACH_TcpPortConnect
FOREACH_TcpPortConnect(GEN_TcpPortConnect)
#endif
#ifdef FOREACH_SerialPort
FOREACH_SerialPort(GEN_SerialPort)
#endif
```

```
#define FOREACH_TcpPortConnect(_) _(ETH2,"192.168.0.1", "2222", CONN3)
#define CONN3_AssocChannels(_) _(C1)
#define FOREACH_Connections(_) _(CONN3, network, CONN3_AssocChannels(_)
```

To take use of previously defined macros we have also created the template for generating structure used by the connection manager (see listing 6.7).

Listing 6.7 Template for Connection Manager

```
connection_manager_t CM = {
    .in_ports = { FOREACH_TcpPortListen(GEN_RefValue) },
    SUM_FOREACH(FOREACH_TcpPortListen),
    .out_ports = { FOREACH_TcpPortConnect(GEN_RefValue) },
    SUM_FOREACH(FOREACH_TcpPortConnect),
    .serial_ports= { FOREACH_SerialPort(GEN_RefValue),
    SUM_FOREACH(FOREACH_SerialPort),,
    .channel_frontends = {FOREACH_ChannelFrontends(
       GEN_ChannelFrontendRef) },
    SUM_FOREACH(FOREACH_ChannelFrontends),
    .channel_backends = {FOREACH_ChannelBackends(
       GEN_ChannelBackendRef) },
    SUM_FOREACH(FOREACH_ChannelBackends)
};
```

6.4 Code Structure

The code is divided into four directories. The *progress* directory contains middleware implementation including header files providing the runtime and system abstraction APIs. For easier orientation we use prefixes for contained files:

- `connection_*` — implementation of physical connections and necessary library callback functions

- `message_*` — data-structures and functions for modeling communication channels and ports
- `progress_*` — functions providing the system abstraction and middleware API

The other three directories — *components*, *virtual_ nodes* and *physical_ nodes* — contain subfolders with name of included components or nodes respectively. Each subfolder should contain directory *src* with handwritten or generated source code and makefile (*subdir.mk*).

Chapter 7

Related Work

Increasing requirements for progressive software productivity and quality resulted in the use of component-based development. Currently, there exists many component models for a wide range of applications used for example in automotive industry, consumer electronics, telecommunications and other business domains. Each specific domain places emphasis on different properties of individual components, the underlying platform and the binding mechanisms. A number of middleware have been created in order to create runtime and communication support of application components. However, these mostly do not take into account various requirements for the domain of real-time systems. The requirements and constraints to embedded systems, as they are presented in [6], do not allow to use existing technologies for enterprise component system, such as CCM, J2EE, CORBA and .NET/COM+.

Design-time analysis according to PROGRESS is essential for achieving system compliance with desired non-functional requirements and predictable behavior of the interaction between software components and real hardware. In addition to the requirements of development of applications for real-time embedded systems, a middleware has to satisfy the specific requirements for component design (tasks, message ports), communication (virtual channels), scheduling (virtual nodes) and hardware platform (EVK1100 board running FreeRTOS). The approach we have proposed in this report is based on ProCom component model, therefore the middleware can be easily incorporated into the existing development tool in order to complete the whole development life-cycle.

Several component models has been already developed in either academia and industry. In this chapter, we provide a brief overview of some component models for real-time embedded systems.

7.1 AUTOSAR

A relatively long tradition of component-based development is within automotive industry. The **AUTomotive Open System ARchitecture** (AUTOSAR) [22] was founded as a development partnership between several manufacturers and suppliers from the automotive field. The main focus of AUTOSAR is to establish an open standard in automotive industry to master the growing complexity of automotive electronics. The standardization of architecture, architectural components and their interoperability allows a separation of development of component-based applications from the underlying hardware platform.

The AUTOSAR standard introduces the *Virtual Function Bus* (VFB) and the *Runtime Environment* (RTE) architectural concepts and different layers of abstraction to capture several aspects of the physical system on which the application will later be deployed on. An AUTOSAR software component is wrapped into a so-called *Runnable* which are implemented in C.

7.2 SOFA HI

The component model SOFA 2 [23] has been developed at Charles University in Prague. **SOFA HI** [24, 25] is an extension of this component model supporting high-integrity real-time embedded systems. SOFA HI keeps control over all hardware resources through a service mediating access to the real-time operating system and hardware. The component model allows hierarchical component composition and definition of extra-functional properties. Components can be *active* – contain own thread, or *passive* – executed in the context of an active component.

In many situations, a component can provide access to a hardware device, or its part, and therefore it is useful to mark this component as singleton during system design phase. The specification of SOFA components and system is described by an ADL-like language which is extended to keep the information about additional architectural attributes. The implementation of SOFA HI components is mainly based on usage of C macro definitions with various restrictions in order to keep them predictable and lightweight. The applications can use only provided system API providing platform abstraction. The SOFA HI system API is the same for each supported operating system and its implementation is selected at compile time.

7.3 MyCCM-HI

MyCCM[1] High Integrity is *a component framework for critical, distributed, real-time and embedded software* [26]. It is based on LwCCM with specific extensions, addons, and limitations. MyCCM-HI application model is described with its own input architecture description language called COAL (Component-Oriented Architecture Language). MyCCM-HI allows to develop composite components in one of the following languages: C, C++, Java, and Ada.

The MyCCM generator produces also an AADL [27] model describing the thread definitions with associated priorities and periods, and communication code. MyCCM-HI uses Telecom ParisTech *Ocarina*[28] for generating *PolyORB*-based application-specific underlying middleware layer from an input AADL model. PolyORB [29] is a versatile middleware that aims at providing distribution and concurrent capabilities for High Integrity systems. The final generated code is very compact and thus easy to verify, and it is adapted to several execution platforms (Linux, VxWorks, OSE-ck, OSEK).

[1]MyCCM stands for “Make Your Component Container Model”

7.4 RubusCMv3

The Rubus component model was designed for development of distributed real-time systems especially to support development of embedded control systems with a mix of *hard*, *soft* and *non real-time* requirements. The component model aims to express the interaction between software components in terms of *data and control-flow* as well as *non-functional attributes* e.g. timings requirements and resource utilization.

Rubus uses hybrid scheduling – static scheduling for critical core functions (clock-triggered tasks run at highest priority), and event-triggered tasks use the remaining processing time. Two types of real-time timing requirements are supported: (1) *deadlines* and (2) *jitter bounds*.

The architectural elements are represented by graphical entities, which can be connected by data and triggering flows. The basic units of composition in Rubus are software circuits which can be hierarchically composed. Component behavior is represented by a specific entry function. Each component is also associated with *profile describing the execution-time and memory consumption on different platforms* [14].

7.5 Similar Models

Furthermore, we could state other component models for example **COMDES II**[2][30], **Palladio**[3][31], **Robocop**[4], and others according to the comparison found in [7]. Since one of the main task of middleware is to provide code portability we would like to mention a specialized real-time embedded middleware framework creating Component Portability Infrastructure (CPI). **OpenCPI** is a middleware developed to reduce complexity and improve code portability of real-time embedded systems. OpenCPI creates a hardware abstraction layer for component-based applications on heterogeneous architectures and communication technologies.

The presented component models are similar in spirit to our work, however we have focused more on the support for modeling elements and semantics of message communication through the virtual channels. Moreover we added support for using the two-level Hierarchical Scheduling Framework (HSF) for FreeRTOS presented in [16]. The goal of HSF integration is to allow predictable integration of virtual nodes on one physical node, and profit from reusability of timing analysis of previously developed virtual nodes.

[2] COMponent-based design of software for Distributed Embedded Systems, version II, developed at University of Southern Denmark.

[3] It was started at University of Oldenburg, actively developed by Karlsruhe Institute of Technology (KIT), FZI Research Center for Information Technology, and University of Paderborn.

[4] Robust Open Component Based Software Architecture for Configurable Devices Project.

Chapter 8

Conclusion and Future Work

The goal of this thesis was to develop and implement parts of a middleware that provides necessary support for the execution of the ProCom components not only on the top of the real-time operating system FreeRTOS. The major interest concerns the transparent communication over the connections of different types while the connected physical nodes can be of the various execution platform.

To summarize, the main contribution of the thesis consists in analysis of various possibilities of the channel communication and implementation of the basic execution and communication support for the virtual nodes based on analysis of the ProCom model semantic and requirements. The substantial part of the thesis is dedicated to examine the semantic of sending and receiving messages by tasks, allowing the transparent reallocation of virtual nodes and sharing physical connections by multiple channels.

The key issue tackled in the design of the communication interface is to ensure fair (mostly equal) consumption of resources when a task communicates regardless of genuine structure of channels and type of physical connections. Proposed solution consists of extra tasks — *sender* and *updater*, which do not only facilitate the handling of messages, moreover they are engaged in the implementation of the semantic of task triggering.

We would like to draw the attention to a problem with message delivery delay caused by buffering in combination with the defined communication semantic. The investigation of the system starting process has raised questions related to the detection of physical node state and managing cyclic dependencies during establishing of the physical connections. The answer is connected to the future definition of reliable channels.

The middleware consists of two layers: system abstraction layer and modeling API. The system abstraction layer enables us to develop platform independent modeling API covering most of the features needed for the integration of the synthesis mechanism into the Pride tool. Our approach increases the reusability of the components by moving the information about system and communication outside the component code without negative impact to the effective resource utilization.

As it turned out at the end of our implementation, we need to move our effort from simple hardware support for communication to the general hardware model promoting unified input/output access to devices both for transparent inter node communication, and the possible need to communicate with attached peripherals.

The future work may consist of improving and adding support for more platforms (especially the systems without dynamic task creation) and connection types, optimization

of message sending (joining multiple messages into one packet), and finally add ability to use multiple instances of the same virtual node on one physical node. The concept of virtual nodes waits for an early integration of the support for hierarchical scheduling to achieve the temporal isolation between the components and predictability of the system behavior.

Finally, in order to complete development chain for model driven engineering of the ProCom component based system, the implementations of the physical platform modeling and deployment in the PRIDE tool need to be finished.

Bibliography

[1] Tomáš Bureš, Jan Carlson, Séverine Sentilles and Aneta Vulgarakis, *A Component Model Family for Vehicular Embedded Systems*, Mälardalen Real-Time Research Centre, Västerås, Sweden.

[2] Tomáš Bureš, Jan Carlson, Ivica Crnković, Séverine Sentilles, and Aneta Vulgarakis, *ProCom — the Progress component model reference manual*, version 1.0. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-230/2008-1-SE, Mälardalen University, June 2008.

[3] Jan Carlson, Juraj Feljan, Jukka Mäki-Turja and Mikael Sjödin, *Deployment Modelling and Synthesis in a Component Model for Distributed Embedded Systems*, 36th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Lille, France, 2010.

[4] Ivica Crnković, Magnus Peter Henrik Larsson, *Building reliable component-based software systems*, ISBN 9781580533270, Artech House, 2002.

[5] Ivica Crnković, *Component-based Software Engineering for Embedded Systems*, ICSE'05, 2005.

[6] Ivica Crnković, *Component-based approach for embedded systems*, Ninth International Workshop on Component-Oriented Programming, 2004.

[7] Ivica Crnković, Séverine Sentilles, Aneta Vulgarakis, and Michel Chaudron, *A Classification Framework for Software Component Models*, IEEE Transaction of Software Engineering, 2010.
http://www.mrtc.mdh.se/index.php?choice=publications&id=2139

[8] *FreeRTOS homepage*, http://www.freertos.org, December 2010.

[9] Tomáš Pop, *The Progress Run-time Architecture*, Master of Science Thesis, Mälardalen University, February 2009.

[10] *ProgressIDE*, http://www.idt.mdh.se/pride/?id=features, November 2010.

[11] Jagadish Suryadevara, Aneta Vulgarakis, Jan Carlson, Cristina Seceleanu and Paul Pettersson, *ProCom: Formal Semantics*, version 1.1, Mälardalen University, July 2010.

[12] Dan Ionescu, Aurel Cornell, *Real-time systems: modeling, design, and applications*, ISBN 9789810244248, World Scientific, 2007.

[13] Hermann Kopetz, *Real-time systems: design principles for distributed embedded applications*, ISBN 9780792398943, Springer, 1997.

[14] Kaj Hänninen, Jukka Mäki-Turja, Mikael Nolin, Mats Lindberg, John Lundbäck and Kurt-Lennart Lundbäck, *The Rubus Component Model for Resource Constrained Real-Time Systems*, 3rd IEEE International Symposium on Industrial Embedded Systems, June 2008.

[15] Jim Turley, *The Two Percent Solution*, http://www.eetimes.com/discussion/other/4024488/The-Two-Percent-Solution , 2002.

[16] Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, S. M. H. Ashjaei, and Sara Afshar, *Hierarchical Scheduling Framework Implementation in FreeRTOS*, Technical Report, Mälardalen University, 2011.

[17] *POSIX.1-2008 specification*, http://pubs.opengroup.org/onlinepubs/9699919799/functions/contents.html

[18] *The Open Group Base Specifications Issue 6*, IEEE Std 1003.1, http://www.opengroup.org/onlinepubs/007904975/basedefs/pthread.h.html

[19] *NxtOSEK homepage*, http://lejos-osek.sourceforge.net/whatislejososek.htm, 2010.

[20] Shengquan Wang, Sangig Rho, Zhibin Mai, Riccardo Bettati, and Wei Zhao, *Real-Time Component-based Systems*, 11th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'05) 1080-1812/05 $ 20.00 IEEE.

[21] Steve Zucker, *Endianness in the Solaris Operating Environment*, http://developers.sun.com/solaris/developer/support/driver/wps/endianness/SOLENDIAN.pdf, 1999.

[22] *AUTOSAR homepage*, http://www.autosar.org/, 2011.

[23] *SOFA 2 homepage*, http://sofa.ow2.org/, 2011.

[24] *SOFA-HI homepage,* http://sofa.ow2.org/sofahi/, 2011

[25] Petr Hošek, Tomáš Pop, Tomáš Bureš, Petr Hňetynka and Michal Malohlava, *Supporting real-time features in a hierarchical component system*, Technical report No. 2010/5, December 2010.

[26] *MyCCM-HI homepage*, http://sourceforge.net/apps/trac/myccm-hi/wiki, 2011.

[27] *AADL homepage*, http://www.aadl.info/aadl/currentsite/, 2011.

[28] *Ocarina homepage*, http://libre.adacore.com/libre/tools/ocarina/, 2011.

[29] *PolyORB middleware homepage,* http://www.adacore.com/home/products/gnatpro/add-on_technologies/distributed_systems, 2010.

[30] X. Ke, K. Sierszecki, and C. Angelov, *COMDES-II: A ComponentBased Framework for Generative Development of Distributed Real-Time Control Systems*, in Proc. of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications. IEEE, 2007.

[31] *Palladio homepage*, http://www.palladio-simulator.com/, 2010.

Appendix A

API Documentation

A.1 Functions

`void init_task (progress_task_t *task)`
: Function initializes task datastructure and it calls task initialization routine.

`void create_task (progress_task_t *task)`
: Function creates new thread from the correct routine according to task type.

`void init_virtual_node (progress_node_t *node)`
: Function calls the init routines of tasks allocated on the virtual node.

`void create_virtual_node (progress_node_t *node)`
: Function creates the tasks from components allocated on the virtual node.

`int get_message (task_port_in_t *mpin, void *read_data)`
: Function reads lastest message from the associated message port.

`void send_message (task_port_out_t *out_port, void *data)`
: Function writes and sends data in message through the associated message ports.

`void write_data (task_port_out_t *out_port, void *data)`
: Function only writes data to associated message ports without triggering the message ports.

`void wait_for_other_nodes_init ()`
: Function should wait until connected nodes are ready.

`void wait_for_other_tasks_init ()`
: Function waits until function `activate_tasks()` is called.

`void activate_tasks ()`
: Function activates all created tasks.

`void wait_for_activation (progress_sem_t *task_port_in)`
: Function waits for incomming message on specified trigger.

`void wait_for_next_period (save_period_t *period_info)`
Function waits for the next period to be activated.

`void compute_next_period (save_period_t *period_info)`
Function calculates when the task needs to be waken up.

`void periodic_task (progress_task_t *task)`
Periodic task routine.

`void sporadic_task (progress_task_t *task)`
Sporadic task routine.

A.2 Macros

`#define GEN_MessagePortOutInit(PORT,...)`
Macro generates initialization function of output message port.

`#define GEN_MessagePortInInit(PORT,...)`
Macro generates initialization function of input message port.

`#define GEN_MessagePortOut(NAME, SIZE, BUFFER_COUNT)`
Macro generates message port variable for outgoing messages.

`#define GEN_VNPortAssoc(CHANNELS)`
Macro generates channels association list.

`#define GEN_VNSetPortAssoc(VN,I,ID,CHANNELS)`
Macro generates association of information about channels to a message port.

`#define GEN_MessagePortIn(NAME, SIZE, BUFFER_COUNT, ACTIVATED_TASKS)`
Macro generates message port variable for incoming messages.

`#define GEN_MessagePortInExtern(NAME,...)`
Macro generates task port variables.

`#define GEN_AssocPorts(PORT, IS_TRIGGERED, DATA_CONNECTOR)`
Macro generates output task port.

`#define GEN_TaskPortIn(NAME, MESSAGE_PORT, IS_ACTIVATED)`
Macro generates input task port.

`#define GEN_ChannelFrontend(NAME, SIZE, CONNECTIONS)`
Macro generates channel frontend from the main list of channels.

`#define GEN_ChannelBackend(NAME, SIZE, PORTS)`
Macro generates channel backend from the main list of channels.

`#define GEN_local_info(CHANNEL)`
Macro generates information about local connection between channel parts.

`#define GEN_network_info(CHANNELS)`
Macro generates information about network connection.

`#define GEN_serial_info(CHANNELS)`
Macro generates information about serial connection.

`#define GEN_TcpPortAccept(NAME, IP, CONNECTION)`
Macro generates list of accepted clients.

`#define GEN_TcpPortListen(NAME, IP, CLIENTS)`
Macro generates information to open TCP port for listening.

`#define GEN_TcpPortConnect(NAME, IP, PORT, CONNECTION)`
Macro generates information to connect to TCP port.

`#define GEN_SerialPort(NAME, DEVICE, BAUD, CONNECTION)`
Macro generates connection variable.

Appendix B

How to build own application

The building process can be divided into two potentially independent parts. First part consists of building ProCom middleware library and the second one builds the runnable representation of physical node. Although the process of building library can be independent, it can useful to rebuild the library based on physical node requirements and reduce the size of resulting executables if some parts of library are not needed (for example TCP/IP communication).

B.1 Prerequisites

In order to build ProCom middleware and sample application from source, several libraries and programs are required to be present on the system depending on target physical node. Generally we use GNU/Make tool and GCC compiler, however there are some specific programs needed for each supported platform. Further we provide basic information and links to

AVR32 EVK1100/FreeRTOS

- AVR32 GNU Toolchain 2.4.2
  `http://www.atmel.com/dyn/products/tools_card.asp?tool_id=4118`
  - cross compiler, assembler and linker
  - debugger
  - flash programming tools
- AVR32 Studio 4
  `http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2725`
  - preconfigured examples
  - FreeRTOS, LwIP library and other drivers

Apple Mac OS X

- XCode Tools 2.2.1
  `http://developer.apple.com/technologies/xcode.html`
  - tested with GCC 4.2.1 (build 5664)
  - POSIX Threads - IEEE 1003.1c

Microsoft Windows

- Cygwin
  `http://cygwin.com/install.html`
- Install GCC 4 and Make packages using the Cygwin installation wizard.

GNU/Linux

- All major distributions already includes GCC and make packages in latest versions.

B.2 Obtaining Source Code

Source core is available from the CD-ROM attached to printed version of this thesis or the latest version is accessible online from subversion repository (username: `mdh`, password: `mdh2010`) `http://server.openagency.cz/svn/jirka/netdemo`. Process of merging downloaded source code with the environment for EVK1100 including FreeRTOS, LwIP library and necessary drivers is described in Chapter C.

B.3 Compilation

To ensure maximal compatibility across different platforms we decided to use a *Makefiles* for driving build process. First we present sub-makefiles for components (tasks), virtual

Listing B.1 Component *subdir.mk*

```
include Makefile.inc
C_SRCS += ./components/sender/src/sender.c
OBJS += ./components/sender/src/sender.o
COMPONENTS_SENDER_INCLUDE = -I./components/sender/src/ \
    -I./progress/ -I. -I./CONFIG/

./components/sender/src/sender.o: ./components/sender/src/sender.c
        $(CC) $(CFLAGS) $(SYMBOLS) $(COMPONENTS_SENDER_INCLUDE) -o"
           $@" "$<"
```

nodes and physical nodes. In each makefile we add used source and object files to variables `C_SRCS` and `OBJS`.

The virtual node includes the sender task component *subdir.mk* and the rule for creating object file from the source code.

Listing B.2 Virtual node *subdir.mk*

```
include ./components/sender/src/subdir.mk
C_SRCS +=  ./virtual_nodes/vn_sender/src/vn_sender.c
OBJS += ./virtual_nodes/vn_sender/src/vn_sender.o

VN_SENDER_INCLUDE= -I./virtual_nodes/vn_sender/src \
    -I./components/sender/src/ -I./progress/ -I. -I./CONFIG/

./virtual_nodes/vn_sender/src/vn_sender.o: ./virtual_nodes/
   vn_sender/src/vn_sender.c
    $(CC) $(CFLAGS) $(SYMBOLS) $(VN_SENDER_INCLUDE) -o"$@" "$<"
```

The physical node includes the sender virtual node *subdir.mk* and also the rule for creating object file from the source code. If there are more virtual nodes allocated to single physical node, we only include its *subdir.mk*:

- `include ./virtual_nodes/vn_receiver/src/subdir.mk`

Listing B.3 Physical node *subdir.mk*

```
include ./virtual_nodes/vn_sender/src/subdir.mk
C_SRCS += ./physical_nodes/pn_sender/src/pn_sender.c
OBJS += ./physical_nodes/pn_sender/src/pn_sender.o
PN_SENDER_INCLUDE= -I./physical_nodes/pn_sender/src \
    -I./virtual_nodes/vn_sender/src -I./progress/ -I. -I./CONFIG/

./physical_nodes/pn_sender/src/pn_sender.o: ./physical_nodes/
   pn_sender/src/pn_sender.c
    $(CC) $(CFLAGS) $(SYMBOLS) $(PN_SENDER_INCLUDE) -o"$@" "$<"
```

The listing B.4 shows the shorten version of main makefile that includes the physical node *subdir.mk* file according to `TARGET` variable. Hence to compile `pn_sender` following command should be executed:

- `make TARGET=pn_sender ARCH={WINDOWS|LINUX|MACOSX|FREERTOS}`

If the `TARGET` variable is not specified it compiles only the ProCom middleware library.

Listing B.4 Shorter version of system *Makefile*

```
include Makefile.inc
PROGRESS_SRCS = $(wildcard $(PROGRESS_DIR)/*.c)
C_SRCS =
OBJS =
AS_FILES =
# Specific parts for different platforms
ifeq ($(ARCH), FREERTOS)
# omitted
endif
DEPEND = $(GCC) -MM -MT '$*.o $*.d'
TARGET=lib
VPATH = $(SRC_DIR)
OBJECT_FILES = $(sort $($(basename AS_FILES):.x=.o) $(OBJS))
LST_FILES = $($(basename AS_FILES):.x=.lst) $(sort $($(basename
   C_SRCS):.c=.lst))
DEPENDENCY_FILES = $(OBJECT_FILES:.o=.d)

ifneq ($(TARGET),lib)
include ./physical_nodes/$(TARGET)/src/subdir.mk
-include $(DEPENDENCY_FILES)
endif
.PHONE all
all: lib $(TARGET)
build: $(TARGET)
lib: libprocom.a
libprocom.a : $($(basename PROGRESS_SRCS):.c=.o)
    $(AR) -rs $@ $($(basename PROGRESS_SRCS):.c=.o)
ifneq ($(TARGET),lib)
$(TARGET) : $(OBJECT_FILES)
    $(LD) -L. -lprocom $(LDFLAGS) -o $@ $(OBJECT_FILES)
endif

%.d: %.c
    $(DEPEND) $(CFLAGS) $(INCLUDES) $(SYMBOLS) $< > $@
%.o: %.x
    $(AS) $(ASFLAGS) $< -o $@
%.o: %.c
    $(CC) $(CFLAGS) $(INCLUDES) $(SYMBOLS) $< -o $@
clean:
    rm -f $(OBJECT_FILES) $(DEPENDENCY_FILES) $(TARGET)
```

Appendix C

AVR Studio

This part describes how to start development easily using software and hardware introduced in previous three sections. At the beginning, we present creating new example project using FreeRTOS and LwIP library. With this integrated solution, it is not necessary to download and setup operating system and library separately. Creation of new example project is done by simple wizard dialog. Desired action is achievable by following these steps:

- Open menu: *File > New > AVR32 Example Projects* (see figure C.1)

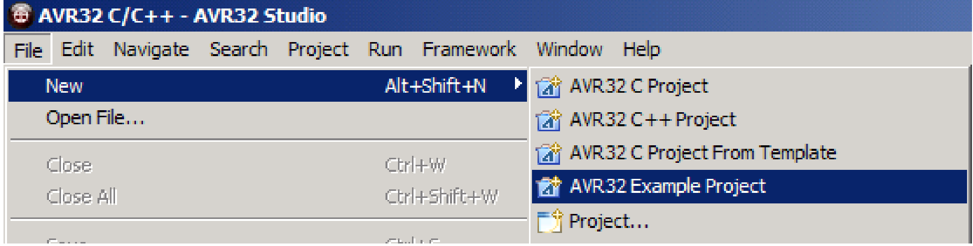


Figure C.1: Creating new example project

- Filter only "lwip" projects and choose example for EVK1100 (see figure C.2).

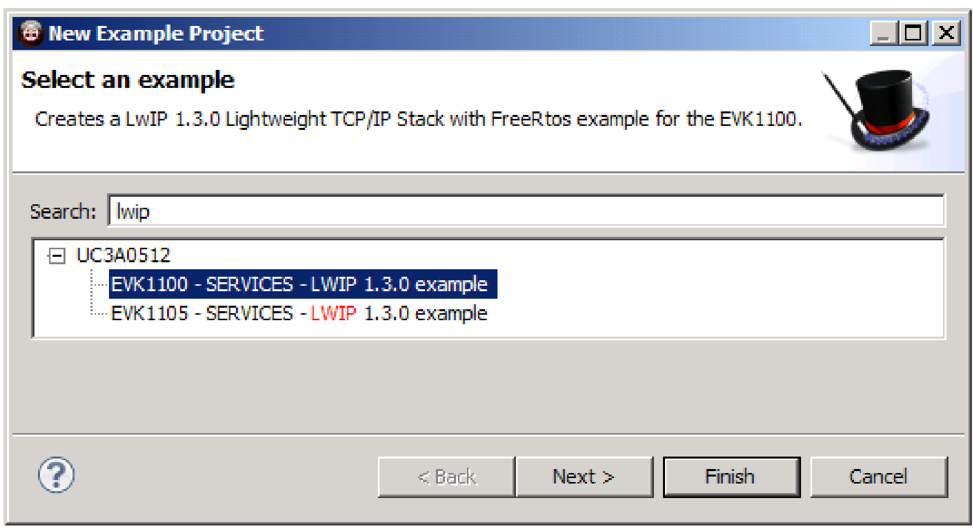


Figure C.2: Select example project

- Setup project name and location if the default location is not checked (see figure C.3).

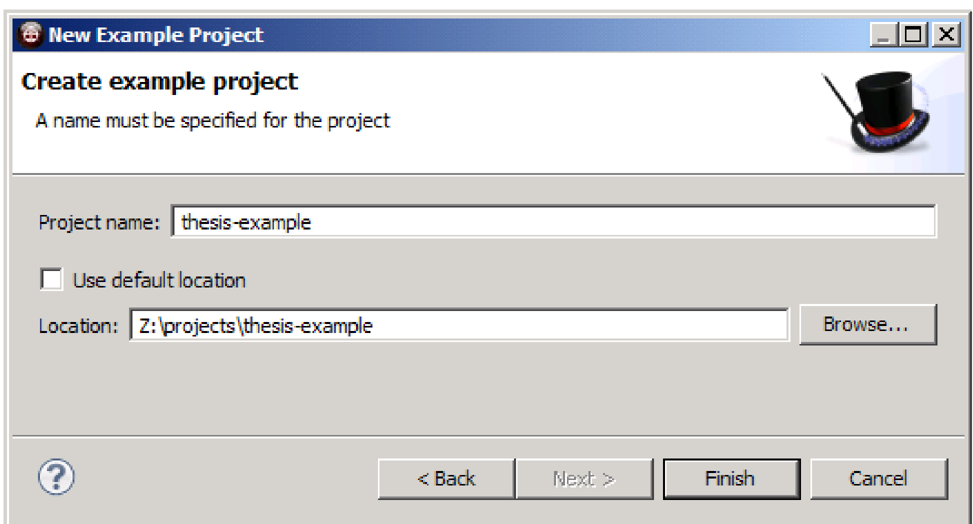


Figure C.3: Choosing name and location

- AVR Studio generates the project and copies FreeRTOS and LwIP files into created project (see figure C.4).

Now we have a working example application, that can be uploaded and run on the board or modified in order to use the middleware for our programs. However, if you want to use other parts of the board is beneficial to use feature of the AVR Studio, which adds drivers and header files to our folder using a simple wizard.

For using the middleware and some example application import the directory with source codes into your project. The compilation of the firmware is facilitate by the make utility that needs to be executed on command line.

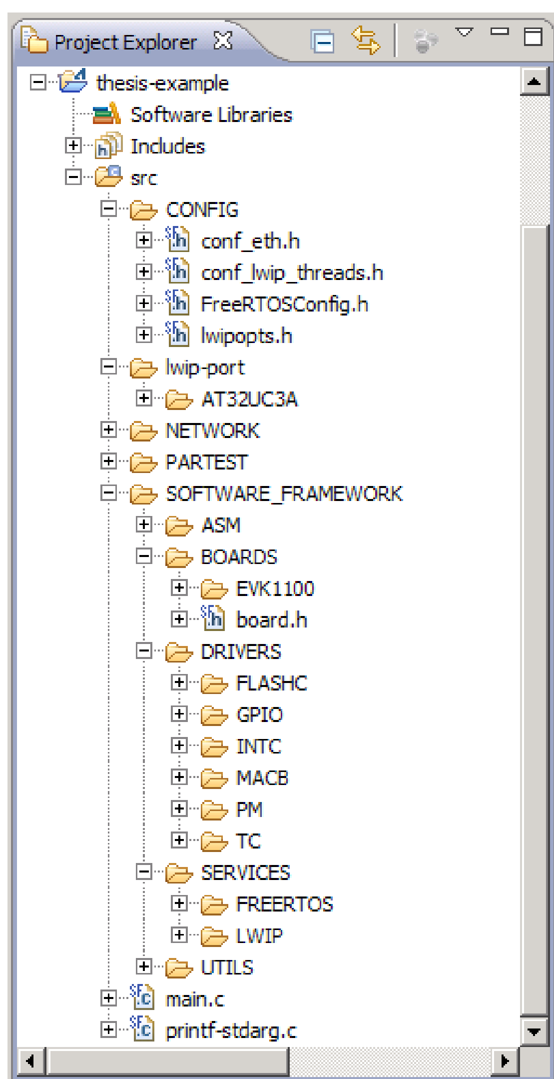


Figure C.4: Project Explorer