Tero Vallius

AN EMBEDDED OBJECT APPROACH TO EMBEDDED SYSTEM DEVELOPMENT

FACULTY OF TECHNOLOGY, DEPARTMENT OF ELECTRICAL AND INFORMATION ENGINEERING, UNIVERSITY OF OULU, INFOTECH OULU, UNIVERSITY OF OULU
AN EMBEDDED OBJECT APPROACH TO EMBEDDED SYSTEM DEVELOPMENT

Academic dissertation to be presented with the assent of the Faculty of Technology of the University of Oulu for public defence in Auditorium IT 116, Linnanmaa, on 6 November 2009, at 12 noon

OULUN YLIOPISTO, OULU 2009
Abstract

Building an embedded system from an idea to a product is a slow and expensive process requiring a lot of expertise. Depending on the developer’s expertise, the required quantity and price level of the final product, and the time and money available for development, the developer can build a device from different granularity of components, ranging from ready-made platforms, kits, and modules to individual components. Generally, solutions requiring less expertise, time and money produce products with higher production costs.

The main contribution of this thesis is the EOC (Embedded Object Concept) and Atomi II Framework. EOC utilizes common object-oriented methods used in software by applying them to small electronic modules, which create complete functional entities. The conceptual idea of the embedded objects is implemented with the Atomi II framework, which contains several techniques for making the EOC a commercially feasible implementation.

The EOC and the Atomi II Framework decreases the difficulty level of making embedded systems by enabling a use of ready-made modules to build systems. It enables automatic conversion of a device made from such modules into an integrated PCB, lowering production costs compared to other modular approaches. Furthermore, it also enables an automatic production tester generation due to its modularity. These properties lower the number of skills required for building an embedded system and quicken the path from an idea to a commercially applicable device. A developer can also build custom modules of his own if he possesses the required expertise.

The test cases demonstrate the Atomi II Framework techniques in real world applications, and demonstrate the capabilities of Atomi objects. According to our test cases and estimations, an Atomi based device becomes approximately 10% more expensive than a device built from individual components, but saves up to 50% time, making it feasible to manufacture up to 10-50k quantities with this approach.

Keywords: embedded systems, microprocessor applications, modularity, object-oriented design
Acknowledgements

The work reported in this thesis was carried out in the Intelligent Systems Group of the Department of Electrical and Information Engineering at the University of Oulu.

I would like to express my gratitude to my supervisor Professor Juha Röning for giving me the opportunity to work in his research group with a subject that I found very interesting along with the valuable support and guidance that I have received during the research. I would also like to thank Mr. Jari Pakarinen for making a lot of PCB milling, component sourcing, and assembling during the years of Atomi development, as well as for the countless coffee breaks that we have had among all the hard work.

I am also grateful to professor Timo D. Hämäläinen and professor Johan Liljus for reviewing the thesis and for the valuable comments and constructive criticism that I received from them. Several people have contributed to this thesis with ideas, testing, and also developing Atomi objects. These include M.Sc. Antti Tikkanmäki, Dr. Janne Haverinen, M.Sc. Risto Sipola, Mr. Ville Tuhkanen, M.Sc. Janne Kivijakola, M.Sc. Antti Gärding, M.Sc. Juha Angeria, prof. Jukka Riekki, Mr. Markus Piippola, and Dr. Marko Pudas. For the financial support for this thesis, I would like to thank the Infotech Oulu Graduate School, the Finnish Academy and the University of Oulu Robotics Group.

Finally, I would like to give thanks to my wife Taru for providing a support to lean on.
### Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>Analog to Digital Conversion</td>
</tr>
<tr>
<td>ASCIS</td>
<td>Architectural Synthesis of Complex Integrated Systems</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuit</td>
</tr>
<tr>
<td>ATA</td>
<td>Advanced Technology Attachment</td>
</tr>
<tr>
<td>CPLD</td>
<td>Complex Programmable Logic Device</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DC</td>
<td>Direct Current</td>
</tr>
<tr>
<td>DLL</td>
<td>Dynamically Linked Library</td>
</tr>
<tr>
<td>DRL</td>
<td>Dynamically Reconfigurable Logic</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>EGA</td>
<td>Extended Genetic Algorithm</td>
</tr>
<tr>
<td>EMC</td>
<td>Electromagnetic Compatibility</td>
</tr>
<tr>
<td>EOA</td>
<td>Embedded Object Architecture</td>
</tr>
<tr>
<td>EOC</td>
<td>Embedded Object Concept</td>
</tr>
<tr>
<td>GACSYS</td>
<td>GAC’s Co-design System</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Description Language</td>
</tr>
<tr>
<td>HW</td>
<td>Hardware</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>IPR</td>
<td>Intellectual Property Rights</td>
</tr>
<tr>
<td>LED</td>
<td>Light Emitting Diode</td>
</tr>
<tr>
<td>LRM</td>
<td>Language Reference Manual</td>
</tr>
<tr>
<td>MCU</td>
<td>Microcontroller Unit</td>
</tr>
<tr>
<td>NRT</td>
<td>NanoResolution Tools project</td>
</tr>
<tr>
<td>OOP</td>
<td>Object-Oriented Programming</td>
</tr>
<tr>
<td>PC</td>
<td>Personal Computer</td>
</tr>
<tr>
<td>PCI</td>
<td>Peripheral Component Interconnect</td>
</tr>
<tr>
<td>PCB</td>
<td>Printed Circuit Board</td>
</tr>
<tr>
<td>PHP</td>
<td>PHP: Hypertext Preprocessor</td>
</tr>
<tr>
<td>R&amp;D</td>
<td>Research &amp; Development</td>
</tr>
<tr>
<td>RF</td>
<td>Radio Frequency</td>
</tr>
<tr>
<td>SATA</td>
<td>Serial ATA</td>
</tr>
<tr>
<td>SoC</td>
<td>System on Chip</td>
</tr>
<tr>
<td>Sps</td>
<td>Samples per second</td>
</tr>
<tr>
<td>SW</td>
<td>Software</td>
</tr>
<tr>
<td>UART</td>
<td>Universal Asynchronous Receiver Transmitter</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>USB</td>
<td>Universal Serial Bus, a commonly used serial bus in computers</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHSIC hardware description language</td>
</tr>
<tr>
<td>VHSIC</td>
<td>Very-High-Speed Integrated Circuits</td>
</tr>
<tr>
<td>VLSI</td>
<td>Very Large Scale Integration</td>
</tr>
<tr>
<td>VSS</td>
<td>VHDL-based System Specification</td>
</tr>
<tr>
<td>WLAN</td>
<td>Wireless Local Area Network</td>
</tr>
</tbody>
</table>
List of original articles


## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>5</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>7</td>
</tr>
<tr>
<td>Abbreviations</td>
<td>9</td>
</tr>
<tr>
<td>List of original articles</td>
<td>11</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>13</td>
</tr>
<tr>
<td>1.1 Terms</td>
<td>14</td>
</tr>
<tr>
<td>1.2 Motivation</td>
<td>16</td>
</tr>
<tr>
<td>1.3 The goal and scope of this research</td>
<td>26</td>
</tr>
<tr>
<td>1.4 The contribution of this thesis</td>
<td>29</td>
</tr>
<tr>
<td>1.5 Summary of original papers</td>
<td>31</td>
</tr>
<tr>
<td>1.6 Author’s contribution to published work</td>
<td>33</td>
</tr>
<tr>
<td>2 Object-oriented principles and software development</td>
<td>35</td>
</tr>
<tr>
<td>2.1 Introduction to object-oriented system development</td>
<td>35</td>
</tr>
<tr>
<td>2.1.1 Fundamental properties</td>
<td>36</td>
</tr>
<tr>
<td>2.1.2 Other properties</td>
<td>38</td>
</tr>
<tr>
<td>2.1.3 Development Process</td>
<td>39</td>
</tr>
<tr>
<td>2.2 Object-Oriented Programming (OOP)</td>
<td>42</td>
</tr>
<tr>
<td>2.3 UML</td>
<td>43</td>
</tr>
<tr>
<td>2.4 Discussion</td>
<td>44</td>
</tr>
<tr>
<td>3 Embedded system development</td>
<td>49</td>
</tr>
<tr>
<td>3.1 Traditional embedded system development</td>
<td>49</td>
</tr>
<tr>
<td>3.2 HW/SW co-design</td>
<td>52</td>
</tr>
<tr>
<td>3.2.1 Introduction to co-design</td>
<td>53</td>
</tr>
<tr>
<td>3.2.2 Co-design process</td>
<td>55</td>
</tr>
<tr>
<td>3.3 Object oriented enhancements to embedded system design</td>
<td>56</td>
</tr>
<tr>
<td>3.4 Platform-based design</td>
<td>58</td>
</tr>
<tr>
<td>3.4.1 Introduction to platform-based design</td>
<td>58</td>
</tr>
<tr>
<td>3.4.2 Design process</td>
<td>61</td>
</tr>
<tr>
<td>3.5 Discussion</td>
<td>63</td>
</tr>
<tr>
<td>4 Modular embedded systems</td>
<td>65</td>
</tr>
<tr>
<td>4.1 Electronic modules</td>
<td>65</td>
</tr>
<tr>
<td>4.2 Motherboard with extension cards</td>
<td>66</td>
</tr>
<tr>
<td>4.3 Enclosed interconnectable blocks</td>
<td>68</td>
</tr>
<tr>
<td>4.4 Chip and software</td>
<td>69</td>
</tr>
</tbody>
</table>
1 Introduction

Embedded systems are everywhere: in a washing machine, TV, car, cash register, microwave oven, remote control, mobile phone, and almost every other device that uses electricity. The embedded systems started in the 1970s from the arrival of microprocessors such as Intel 4004 (Intel 2008) and 8008, and Motorola 6800 (Peckol 2008). In the early years, the designing of an embedded system was simple when the scale of the systems was small and applications simple. The design of such a system consisted of giving some thought to the problem, putting a microcontroller to the device along with some simple input and output peripherals, writing a few lines of code with assembly language and debugging it. Components were relatively large sized through whole components, being easy to handle by hand, and prototypes of devices were easy to be made.

Contemporary embedded applications can be very large and sophisticated, controlling for example a nuclear plant or jet aircraft. Applications like that are orders of magnitude more complex than the old simple applications. Design by hand is no longer feasible, and utilizing an ad hoc approach tailored to each different application is too expensive and too fraught with error. Nowadays, the design of an embedded system requires models, architectures, simulations, synthesizing, design methods, sophisticated software languages, and software tools that enable all these. Furthermore, the component size has decreased so much that creating the concrete hardware requires high resolution PCB manufacturing processes and very accurate component assembly machines. Assembling electronic devices by hand is very difficult, and often cannot be done without special tools. Because of that, creating a real world prototype is expensive and slow, since the prototype must usually go through a manufacturing process that has considerable initial work to be done, costing time and money.

As a result of this, embedded systems are cheap to manufacture but expensive to develop. The mixture of electronics and software that forms an embedded system becomes very easily complicated, and a number of tools and methods are required to make the designing possible. The designing of an embedded system requires a lot of expertise in both electrical and software engineering, is prone to errors, and is difficult to manage.
1.1 Terms

This thesis presents an approach to object-oriented embedded systems, and discusses design and development related issues. Since the terms are not unambiguous, it is relevant to examine their definitions and meaning in detail.

Term object-oriented refers to an object model, which is a means of helping manage the complexity of software (or other) design (Booch et al. 2007). The term is usually known from software design, even though the object-oriented principles have also been used for electronic design.

The object model consists of objects. An object is an element of a system that combines both data structure and behaviour in a single entity (Rumbaugh et al. 1991). Objects are the building blocks of object-oriented systems. Originally, the concept of objects was derived from real world items that are used as building blocks for concrete systems. From there, the idea has been taken to software development, where the complexity of a system easily increases and becomes difficult to grasp. In software development, the system under design is decomposed into objects (and classes) in order to get the complexity of the implementation organized, understandable, and manageable by a human. The object-oriented decomposition is an alternative to the algorithmic decomposition, which was the earlier commonly accepted way of decomposing software. (Booch et al. 2007)

Object-oriented system is a system that is organized as a collection of discrete objects. The objects of a system interact with each other in order to achieve the desired functionality. The objects interact with each other through an interface. The interface of an object usually consists of properties, methods, and events. Properties represent information that an object contains. Properties are like variables in that they can be read or set using property Get and property Set procedures. Methods represent actions that an object can perform. For example, a "Car" object could have "StartEngine," "Drive," and "Stop" methods. Events are notifications an object receives from, or transmits to, other objects. Events allow objects to perform actions whenever a specific occurrence takes place. An example of an event for the "Car" class could be a "Check_Engine" event. (Microsoft 2008)

Object-based system is similar to an object-oriented system, but it does not fulfil all the requirements of an object-oriented system: abstraction, encapsulation, modularity and hierarchy (Booch 2007). Hierarchy includes the inheritance property, which often is the property missing from object-based systems.
Object-oriented analysis is a method for analyzing the functional requirements for a system with object-modelling techniques. It produces a conceptual object-based model of the system, which does not consider the implementation constraints.

Object-oriented design is a method for design encompassing the process of object-oriented decomposition and a notation for depicting logical and physical as well as static and dynamic models of the system under design (Booch et al. 2007). In other words, based on the analysis, the implementation of the system is constructed through object-oriented design. The design phase takes into account the constraints imposed by the implementation technology or environment, and produces a detailed description of how the system is to be built.

Embedded system is defined as a combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a dedicated function (Barr 2008). Usually, an embedded system is a part of bigger device, such as a washing machine or a microwave oven. Often, the embedded systems are highly integrated entities that are built around some main unit, which could be, for example, a central processing unit (CPU), programmable logic (for example, FPGA or CPLD), System-on-Chip (SoC), or a microcontroller unit (MCU). The main unit is expanded with some additional circuitry that, for example, drives some external devices or creates operating voltages for the electronics, and other components such as connectors. The devices consist of discrete components that are soldered to a PCB and software that gives the intelligence to the system.

Object-oriented embedded system is a combination of the two terms. It usually means embedded systems that are designed with object-oriented design methods. In other words, it means that the object-oriented fundamentals are applied when:

1. designing the hardware of an embedded system with a hardware description language or a system description language, for example VHDL, Verilog, or SystemC (Djafri & Benzakki 1997, Grötker et al. 2002),
2. designing the software of an embedded system (GentleWare 2008, Mattos et al. 2005), or
3. designing both the hardware and the software of the system with particular object-oriented design methods (Kumar et al. 1996, Edwards & Green 2000).

In our approach, the term, however, has been further extended towards a more concrete form than the object-oriented co-design methods. An object-oriented embedded system is a system that consists of embedded objects. An embedded object is a physical electronic entity (a PCB with components) that consists of hardware and, if needed by the functionality, software. Each object is physically encapsulated, has a
standardized physical and functional interface, and performs some simple function. In this thesis, we call our approach an object-oriented embedded system, but we acknowledge that it is debatable whether it is object-oriented, object-based, or just modular.

Development process is the process of developing a system. It focuses on the sequence of operations or events over time that produces desired outcomes.

Design process is a sub-process of the development process, which accomplishes the design of the system.

Design method describes how the system is designed, involving a systematic use of specific techniques.

Design flow describes how design moves from one tool to the next. It describes the explicit combination of tools to accomplish the design.

1.2 Motivation

An embedded system development process is slow and expensive. Survey (CMP media 2006) states that, in year 2006, 70% of the embedded system development projects lasted longer than six months, 45% lasted over a year, and 19% over 2 years, respectively. Based on the average number of people on the project team (15.8) and the amount of their simultaneous projects (1.4), a half year project takes approximately 67.7 man-months and costs approximately 400,000 euros for the employer as salaries (calculated with average salary of M.Sc. from TEK 2008), excluding material, tool, and subcontractor costs.

An embedded system development process also requires a lot of expertise. To create an embedded system up from an idea to a product, the developers must have expertise on system design, digital and analog electronic circuits, PCB layout design, software design and languages, testing of electronics, and manufacturing. Usually, a team of people is involved in a project, and everyone does not need to have all the aforementioned expertise.

A normal development process starts from creating and inspecting a requirement specification, and then partitioning the design into software and hardware components (Berger 2002). The main components are selected in the partitioning phase, most importantly the MCU (for example, ¹ and ²), FPGA (for example, ³ and ⁴) or other

¹ http://www.atmel.com/
² http://www.microchip.com/
³ http://www.xilinx.com/
⁴ http://www.altera.com/
processor or processors that the system is based on. Often, the development phase may involve buying a development kit for the MCU and/or other devices (for example, 5), and creating some test codes for them, in order to become familiar with the devices. Similarly, different electronic parts may be separately created and tested on a strip board or prototyping boards before being actually implemented on a PCB. In some cases, the whole system may be simulated with a computer and emulated on an FPGA-based system before actually implementing anything (Staunstrup & Wolf 1997: 75–112, Edwards & Fozard 2002).

During or after the partitioning and component selection phases, depending on the development process, the development proceeds to creating hardware (FPGA configuration and PCB) and software for the product, then testing them separately and as an integrated device. Sometimes, the process involves some iteration of the first phases, and more than one revision of the hardware needs to be made to correct flaws found in the first versions. Finally, after strict product tests, the product is released for production. The production phase requires a testing plan and a tester device, which means yet another design process. (Berger 2002)

After the product release, maintenance and upgrading must be managed until the end of the product’s life cycle. Upgradability and maintainability are significant properties for an embedded system, as upgrading a device usually means a new project. According to the CMP media 2006 survey, 55% of embedded projects were upgrades or improvements to an earlier or existing project in 2006.

Trial and error or incremental development are not feasible approaches in electronics design, since it is very expensive to modify the hardware once it has been made. Usually, robust simulation and prototyping are needed in the partitioning phases to confirm the partitioning and component selection decisions. Still, there is room for errors as the prototyping hardware is often just an emulation of the actual implementation hardware, and the electrical characteristics of the design are often based on experience with previous projects. The low-level electronics, in particular, are prone to errors that may take some time to resolve. As a result, estimating the duration and costs of a specific project is difficult. Schedule and cost expectations are usually based on experience with similar earlier projects, but they often seem to be too optimistic. According to the survey (CMP media 2006), 55% of embedded projects were late of schedule in 2006.

5 http://www.olimex.com/dev/lpc-2378stk.html
Sometimes, an existing, well-established electronics platform (meaning a hardware platform such as the one in 6), electronic modules or a PC-based system is used in order to avoid problems in creating a new platform from individual components. This is common in low quantity products, because the manufacturing costs of a device tend to become more expensive in this way. There is generally a relationship between the development costs, and time and a cost of the resulting product: the quicker method is used to create the device, the more money is saved in the shortened development time, but the resulting product becomes more expensive to manufacture. Also, using well established, ready-made platforms is often a less risky approach compared to a project that starts from scratch. On the other hand, if the production quantities amount to millions, even a slight difference in production costs may become worth of the risks and a lengthy development time. Therefore, the best way to develop an embedded system for an application depends on production quantities, production costs, development time, and also an ability and will to take risks. It also depends on the developer’s expertise: the platform, module, and PC based development methods do not require as much expertise as developing from individual components.

If a “customer” (for example, a person, a company, or a research group) needs an embedded system, there are different ways on developing it, depending on what expertise the customer possesses. We have categorized the customers into three different levels of expertise, and listed the options for developing an embedded system on each level. These options are shown in Fig. 1.

---

6 http://www.espotel.fi/ratkaisut_ed9200.htm
If the customer has programming skills and knows basics of electronics, then the customer can make a device with existing modular electronic building blocks, PC104-based or other embedded computer based systems, or use an existing development kit or a platform to develop the device. Customers like that could be companies that have a main product that requires an embedded system to support it, but have some technical background, for example a computer game company requiring a game controller to support their game software. Another such customer would be universities or other research institutes, where new algorithms or device
concepts are created and simulated, and they need to be tested also on a hardware that could be suitable for making the research result usable in real life.

A development process involving modular electronic building blocks or PC104-based or other embedded computer based modular systems is shown in Fig. 2. The project starts by specifying the initial requirements for the device and finding suitable modules for making it. The end result does not need to be accurately specified yet, because the setup can easily be changed during the development due to a modular system. The customer then builds the device according to the initial requirements and tests the outcome. By gaining some experience on the result, the customer can now alter or further specify the requirement specification and modify the device accordingly. This iteration is quite fast and makes it easier for people that are not experts on embedded system development to form the end result to match the actual requirements set by the target application. In contrast to subcontracting, this approach does not require legal knowledge or much paperwork.

When the development of the device is completed, the device manufacturing and selling of the device can start. However, devices made in this manner cannot very easily be automatically assembled, making mass producing expensive. Furthermore, the modules tend to be relatively expensive for a mass product. Thus, the only viable option is to make products with small production quantities in this manner. If a mass product were required, the device should be redesigned and assembled from individual components in order to gain the lowest costs for manufacturing the device.

The benefits of this approach are the easy and quick development, and the low level of expertise required in both electronics and low level software development as the low level drivers usually come with the product. The downsides are that the end product is expensive compared to a device made from individual components, and the difficulty of automatic assembly for such devices, making this method suitable only for products that are made in low quantities. An exception to this is a device, such as a PC104 computer, if it suits the application as such, without any additional modules. Then, its development process is similar to the development kit based approach (introduced next). Still, a product made with this method has significant restrictions on its size and shape. Custom shapes are not possible. Furthermore, the resulting device may not be physically very robust due to the connectors between modules. This may make these devices unusable for vibrating or otherwise rough conditions, unless the device is strengthened or shielded with special structures.
Customer needs a device. Customer knows programming and basics of electronics.

Requirement specification is made.

Customer buys suitable easy-to-use electronic modules, makes his own device setup, and programs the device according to functional requirements.

After first experiences on using the device, some modifications are made. Then, the device is tested again. This iteration can take place many times, but it is quick and easy. Several different setups can be tested in a matter of days, or even hours.

Modules that are to be updated are changed to new ones. Extension is done by adding new modules. Remaining functionality is minimally affected due to modular design.

Not applicable, because
- Parts too expensive for mass product
- Automatic assembly is very challenging in production.

If mass product is required, it must be designed from individual components again.

A device made of individual modules can be manufactured and sold in small quantities.

Fig. 2 Development process using modular, PC104 or other embedded computer based systems.
A development process using an existing development kit or a platform is shown in Fig. 3. The process starts again by creating a requirement specification. The specification needs to be quite accurate, especially on behalf of the hardware, because hardware cannot be modified later. If hardware proves to be inadequate for the resulting product, then the whole hardware needs to be changed. To avoid this, a thorough specification is essential in the beginning for making an educated decision for selecting the kit or platform that the product will be based on.

Next the customer builds the device according to the specification and tests the device. After testing, the requirements or specification can be changed on the software and the system can be modified accordingly.

When the device is completed, the device manufacturing and selling can start. If the product is a mass product, the high quantity prices are negotiated with the platform manufacturer and in some cases a functional tester is developed. If the device need to be upgraded later, only software upgrades are possible. If HW upgrade is needed a new kit/platform is required, which means a new project. However, existing higher level software can often be ported in most part to a new platform, but new low level drivers are needed.

The benefits of this approach are the easy and quick development, and the low level of expertise required in electronics. The result is a product that is both low quantity and mass producible with little organizing, as the manufacturer can most likely provide the device with a reasonable price as a mass product.

A downside is that one may have only one manufacturer for one’s product, as one may not get the rights to manufacture the hardware in any other subcontractor than the one that owns the platform. Thus, there is no competition on manufacturing costs and the manufacturer may raise the price of the device. Furthermore, if the manufacturer has problems in manufacturing or delivery, there is no way to get obtain the product from elsewhere. These situations can sometimes be handled with agreements before the mass product is developed.

Another downside is that depending on the development kit or platform, the low level drivers may also need to be developed, which requires some expertise on low level software. Further downsides are that using a development kit restricts the shape of a device, the hardware upgrade or expansion is not possible, the desired end result must be well known from the beginning, and it is hard to find an optimal kit or platform for your product.
Fig. 3 Development process using an existing development kit or a platform.
If the customer is an expert on electronics design, testing and manufacturing, the device can be created from individual components. This kind of customer is for example embedded system research and development companies and institutes.

A development process where a device is developed from individual components is shown in Fig. 4. The presented process is a traditional development process (see section 3.1) and other processes exist, as well (see chapter 3), but from the point of view of the motivation of this thesis, they have similar properties in terms of expertise, and the overall development process. The process starts with initial specifications and creating a proof of concept with existing kits, bread board based electronics, or other quick prototyping method. After the concept has been established viable, the actual design process is conducted. The development process contains many phases, which are explained in detail in chapter 3. The development process includes creating the actual design of the hardware and the software, possibly simulating both, designing the layout of the hardware, acquiring components, physically assembling the board, and testing both hardware and software separately and together in the final board, and debugging problems. After the first version of the device is made, the device may require modifications, and the development process is done again. Finally, when the actual device is ready, a production tester is designed and developed for the mass production of the device. If the device requires upgrading or expansion, the design itself may be quite easy to upgrade if upgrading was anticipated in the design phase of the original design, but all the physical work, including layout design, must be made partially again.

The downside is that it requires a lot of expertise. However, some parts of the development process can be ordered from subcontractors, making the development easier but involving the subcontractor related issues, such as agreements and paperwork to make the order in an organized fashion and to avoid legal issues. Further downsides are that the desired end result must be well known from the beginning, and when more details are involved in the design, the more prone it is for errors, requiring also more testing and verifying. Overall, the development process is slow and expensive, often not feasible for low quantity products.
Customer needs a device. Customer is expert on electronics design, testing and manufacturing.

Initial specifications and proof of concept with existing kits, modules, bread board based electronics, or other test methods.

- Specifications
- Sw/Hw partitioning

Evaluation / simulation: Initial experiences on the device are received simulated versions or early prototypes, i.e. not the actual hardware.

- Software and hardware development
- PCB design and manufacturing, component orders and assembly
- System integration

Testing: After first experiences on using the actual device, some modifications may need to be made. This process aims to do the device right with minimal iterations as iterations are slow and expensive to make at this point.

Production tester development for the mass produced device.

Customer can start mass producing and selling the device.

Similar process is done again for the device upgrade or extension. If the original design anticipated upgrades, this iteration should be shorter than the first time.

Device needs upgrading or expansion.

Fig. 4 Traditional development process when creating a device from individual components.
1.3 The goal and scope of this research

The goal of this research is to combine the good qualities of module and development kit based development processes, and to conserve an option for individual component based design as well. The proposed Embedded Object Concept (EOC)/Atomi objects presented in this thesis aim:

- to make the difficulty level of making embedded system low, enabling customers to make embedded systems themselves. The customer needs to know programming and basics of electronics, but does not need comprehensive knowledge of electronics design, PCB design, testing, or manufacturing. In addition to Atomi objects themselves, this goal involves also system design and simulation tools that are yet to be made.
- to preserve the production cost of the resulting device competitive.
- to get companies that currently provide kits from their chips or have a module as their product, to make their kits and modules as Atomi objects. As a result, there would be a vast amount of different objects available for Atomi-based embedded systems.
- to quicken the path of an embedded system from an idea to a commercially applicable device.

To achieve these goals, the Embedded Object Concept (EOC) utilizes common object-oriented methods used in software by applying them to small electronic modules, which create complete functional entities. The modules are concrete entities consisting of both hardware, software, and the printed circuit board (PCB). These modular entities represent objects in object-oriented design methods, and they function as building blocks of embedded systems. These entities are called embedded objects.

The Atomi objects are the implementation of the concept, i.e. the conceptual idea of the embedded objects is implemented with the Atomi II framework. It is defined by a technical specification developed particularly for implementing the EOC and ensuring universal physical, electrical, and functional compatibility between embedded objects. The Atomi II framework contains several techniques for making the EOC a commercially feasible implementation. The most important techniques include the inexpensive bus system and the single board conversion.

The development process using Atomi objects is shown in Fig. 5. It is similar to the process with modular, PC104, or other embedded computer based systems, except for that the Atomi process enables mass products. The Atomi objects are
physically shaped in such a way that they form a layout of a single PCB when they are compiled together as a device. This enables a direct automatic conversion of a set of Atomi objects into an integrated PCB for mass production (singleboarding). Furthermore, the circuitry that enables modularity is inexpensive, making the integrated device also cost effective. As the product is compiled from standard modules, a production tester can also be automatically generated by integrating the testers of each individual module that are present on the product. On the other hand, the resulting device will not be as highly optimized for the application as when built from individual components. There are also restrictions on the shape of the device: each device will be rectangular and not utterly minimized in size.

When the device requires upgrading or expansion, the prototype consisting of individual modules can be modified and then converted again to an integrated PCB. The development process with Atomi objects is shown in more detail in section 5.4.

The thesis focuses on the concept of embedded objects, its implementation and its applicability for making a device from an idea to a product. The main application area is robotics, but some other areas are also studied.

The current trend in embedded systems design is to integrate as much as possible of the functionality of the device on a single chip, and much research is currently involved in System on Chip (SoC) design. The scope of this thesis excludes the SoC approach because the main idea of the EOC involves physical modules that each performs one function, which is not possible with SoC systems. However, SoC systems can be used in building a module instead of the whole system, but since the EOC or Atomi II Framework does not limit the internal implementation of a module, using SoC is similar to individual component based embedded systems in the EOC perspective.
Fig. 5 Development process using Atomi objects.

Customer needs a device. Customer knows programming and basics of electronics.

Requirement specification is made.

Customer buys easy-to-use electronic modules, makes his own device setup, and programs the device according to functional requirements.

After first experiences on using the device, some modifications are made. Then the device is tested again.
This iteration can take place many times, but it is quick and easy, and does not require any contracts. Several different setups can be tested in a matter of days, or even hours.

A completed device is automatically converted to a mass producible form (singleboarding).

A production tester is automatically generated for the mass produced device.

The customer can start mass producing and selling the device.

Objects that are to be updated are changed to new ones. Interfaces remain (at least mostly) the same.
Extension is done by adding new objects. Existing functionality is minimally affected due to modular design and encapsulation of the objects.

Mass product from Atomi objects

A device made of individual modules can be manufactured and sold in small quantities.

Device needs upgrading or expansion.

A completed device is automatically converted to a mass producible form (singleboarding).

No mass product

Requirement specification is made.
1.4 The contribution of this thesis

The contribution of this thesis is the embedded object concept (EOC) and the Atomi II Framework. The EOC shows how several object-oriented fundamentals can be mapped into the embedded objects, and how to use UML notation in design process. The mappings of object-oriented fundamentals include the following:

- Abstraction
- Encapsulation
- Modularity
- Hierarchy through
  - inheritance (hardware, software, interfaces) and
  - aggregation (on Atomi level)
- Object-oriented architecture (complexity growing)

The difference to existing approaches to object-oriented embedded systems is that existing approaches consider the objects to be non-physical software or hardware description language objects. In the EOC the objects are concrete physical entities consisting of both hardware, software, and the printed circuit board (PCB), assembled, programmed with default software, and ready to be used. In other words, the utilization of object-orientation is extended to a physical level, enabling high level easy-to-use embedded system development.

The EOC is presented, implemented and tested in this thesis. The conceptual idea of the embedded objects is implemented with the Atomi II Framework, which has been developed in this thesis. The framework specifies the following:

a) a scalable and inexpensive bus system, including novel arbitration, addressing, a flow-controlled data transfer method,
b) a generic logical interface,
c) physical dimensions for Atomi modules,
d) rules for expansion,
e) PCB specification,
f) compulsory components for each object
g) electrical properties,
h) layout recommendations (for example, grounding method, silk screen texts, assembly and test calibration markings)
i) default connector types and pin outs
j) default software structures
k) naming and versioning recommendations
l) production tester and testing principles
m) recommended sourcing principles.

The Atomi II Framework includes also the following Atomi template files for making new Atomi objects:

a) Schematic template with only compulsory components with pre-routed layout template
b) Templates based on a), appended with arbitration circuitry
c) A selection of templates based on b), appended with different microcontrollers, and software templates with bus software for each hardware template
d) VHDL code for the Atomibus, test waveforms and a sample project for Altera FPGA chips.
e) Design rule files for PCB design software.
f) Test software for testing Atomi objects from PC via USB-Atomi.

The Atomi II Framework contains several techniques for making the EOC a commercially feasible implementation. The most important techniques include the scalable and inexpensive bus system (see Paper VI) and the single board conversion (see Paper IX). The bus system is scalable, i.e. system can be expanded by adding more objects as required by the application. A single bus can contain up to 256 objects, but the buses can be split and thus any amount of objects can be used. Its implementation is inexpensive when compared to other techniques when external components need to be used for implementation. This is the case in MCU based modules. In FPGA based systems existing arbitration and addressing methods could be implemented in the chips logic, which would not cause a problem with costs. However, FPGA is not suitable for all modules. The arbitration and addressing method with Atomi system costs 0.26 euros total, while the next cheapest way to create arbitration and addressing, a CPLD which can implement a number of different arbitration and addressing methods, costs 0.91 euros plus side components (prices from Digikey, 1,000 pcs sets). While the bus system makes the implementation inexpensive and versatile, the single board conversion enables the direct conversion of a prototype to a mass producible device, lowering further the manufacturing costs of the final device. Both are combinations of existing and new techniques with a novel implementation.

The thesis shows several test cases which utilize the EOC and the Atomi II Framework. The test cases are actual implementation with real hardware. They mostly
relate to robotics, which is a direct result of the fact that this thesis has been made in the robotics group at the University of Oulu. The test cases show that real devices can in fact be built with the concept proposed and that the techniques developed function properly in real world applications. Furthermore, the test cases demonstrate different capabilities of Atomi objects by building a complex device, a low power device, a low cost device (using custom interfaced objects), and precision measurement devices (despite of restricted PCB layout).

1.5 Summary of original papers

This thesis consists of ten publications. Paper I introduces the concept of the embedded objects for the first time. It presents the ideology of considering bus-based electronic modules as objects, and applying object-oriented fundamentals to them. It presents an outline how the fundamentals can be mapped into embedded object domain, and discusses the possibilities for high level system development. It also introduces the first implementation of the embedded objects, called the Atomi objects.

Paper II presents a robot named Qutie. It is partially implemented with the first generation Atomi objects in conjunction with a PropertyService (Tikanmäki & Röning 2007). It is the first published test case where the Atomi objects have been used.

The term Embedded Object Concept (EOC) is introduced in Paper III. It presents an example case of the EOC in order to show how UML can be utilized for EOC-based designs. The example case is a simple, wall following cleaning robot. The UML applicability enables the EOC to be used with object-oriented design methods.

The technical architecture, details of implementation, and electrical and physical properties of the first generation Atomi objects are introduced in Paper IV. It presents the hardware of an Atomi, the magnet attachment, the logical interface, and the physical bus and its protocol, namely Atomibus. The Atomibus is the backbone of the implementation of the EOC, and in the first realization it is a simple asynchronous serial bus. The Atomibus is analyzed through the existing test cases, and several problems are pointed out, but still the EOC in general is found promising.

Paper V presents yet another test case of the EOC, a telepresence robot, which is a complex test case for the EOC. It is built in two versions. The first version operates together with a computer, and the second one being completely built with Atomi objects. This test case has several purposes. The main purpose is show the functionality of the EOC in a relatively complex embedded system. Furthermore, it also shows how the Atomi objects can be used as an extension to a computer, and
how Atomi object construction can also replace a computer. It also demonstrates how incremental device development can be applied to electronics via the EOC. Neither of the telepresence robot versions is completed in this paper. This paper also introduces the second generation Atomi objects for the first time. It mentions the new parallel connectable boards with the possibility to form single PCB layouts, and the new improved version of Atomibus. It also presents two new terms: passive and active objects.

Paper VI presents in detail the Atomi II Framework, which is a set of specifications to ensure scalable, flexible and inexpensive implementation of the EOC. The specification consists of physical, electrical, and functional parts. The paper derives the requirements for the new framework from the results of the previous papers, and shows an implementation that is a compromise of the requirements. The Atomi II Framework contains the new Atomibus II with novel arbitration and addressing methods.

Paper VII is a continuance paper to Paper V. Paper VII shows the completed version of the Atomi-based telepresence robot. Paper VII also discusses the benefits and potential applications of the EOC, presents test data from the Atomi II framework communication sequences, and discusses the experiences derived from the EOC test cases.

The novel low cost and scalable arbitration method is examined in Paper VIII. This arbitration method is the most significant part of the Atomibus II along with the addressing method.

Paper IX presents a test case for making a prototype with the Atomi objects and then generating the mass production version of it. The mass production version is a compilation of Atomi objects converted into a single PCB. The paper presents several techniques for enabling a feasible conversion board. It also presents two test cases for calculating the costs of the results and a comparison to a similar board made from individual components. The paper also discusses the time and savings that can be achieved with the Atomi objects.

Journal article X presents the overall picture of the EOC appended with the latest EOC related techniques and design process. It also shows a new test case, the Painmeter, which is the first battery powered low power device made with Atomi objects.
1.6 Author’s contribution to published work

The Author has been the main author and contributor in all the publications included in this theses except for II, where the he is the second author and contributor. In each article, professor Juha Röning has assisted in writing. Some Atomi objects that are used in test cases are not made by the Author. In the following test cases, the Author has acted as a technical support and a consultant: Painmeter, Qutie, Micromanipulation platform, and Biofilm.

Paper I: The manuscript is written by the author. Professors Janne Haverinen and Juha Röning contributed in the idea and ways of mapping object-oriented fundamentals into embedded objects.

Paper II: The Qutie robot and most of the writing was done by M. Sc. Antti Tikannäki. The author wrote the Atomi sections of the text and developed the Atomi objects used in the Qutie robot.

Papers III–IX: The research, designs, implementations and tests were made by the author.

Journal X: The research, designs, implementation and tests regarding the Telepresence robot and EOC was made by the author. The Painmeter test case was designed by M. Sc. Juha Angeria, professor Jukka Riekki and the author. The new Atomi objects for the application was designed and implemented by Juha Angeria.
2  Object-oriented principles and software development

The following chapters present separately the fundamental ideas of object-orientation and how they are used in software development. This separation is meant to clear the concept in order to make it easier to understand the contributions of this thesis, where the fundamental ideas of object-orientation are applied into the embedded objects.

2.1 Introduction to object-oriented system development

Object-oriented analysis and design is a modern way to model and organize systems, thus producing object-oriented systems. Object-oriented analysis and design evolved through the evolution of programming languages. First objects as programming entities were introduced already in the 1960s in Simula 67 (Dahl et al. 1968). Since then, several programming languages have contributed to the evolution of the object-oriented techniques. The actual boom of object-oriented programming languages and theories was in the 1980s and early 1990s. The object-orientedness became an alternative to structured techniques (e.g. Ward & Mellor 1985) that were introduced in the 1970s and 1980s. Since then, the object-oriented languages, tools and techniques have gained more and more success in computer software development. However, in embedded system software development, where the systems are more focused on functionality than data, it is still debated whether the object-orientation actually brings any benefits to the structured techniques. Also, the object-oriented languages tend to bring some overhead to the software, and thus produce less a efficient code than non-object-oriented languages, which may be significant in systems with little memory and processing power. In more complex systems, however, the benefits of object-oriented decomposition may be more important than the disadvantages. The current popularity of object-orientation is shown in the number of popular object-oriented programming and scripting languages (e.g. Java, C++, C#, Visual Basic, Python, PHP, Perl) and the emergence of object-based programming frameworks (e.g. J2EE and .NET). (Booch et al. 2007, Yourdon 1994).

The key benefits of object-orientation are the reuse of objects and maintainable architecture (Yourdon 1994). The reuse is made possible by the architecture: well made objects in a well designed architecture can be reused in further projects. This brings other benefits as well, such as improved quality, increased productivity and rapid prototyping. The quality improves because the objects have a longer life span,
and their bugs have more likely been discovered and fixed already in the earlier phase of their existence. The reuse increases productivity because things do not need to be made again. The productivity can also increase due to the architecture as it may better suit certain applications. Also prototypes of new systems can be done quickly by reusing existing objects, which enables the rapid prototyping. Especially the graphical user interface (GUI) objects are usable in prototypes that do not yet need to have the functionality of the application, but shows the generic outlook of a system for a customer in software projects. (Yourdon 1994)

The meaning of object-orientation in a more detailed level, i.e. what is considered object-oriented and what is not, has been defined by several different parties. Their own contribution to the development and definition of the concept has been given by Booch et al. (1991 and 2007), James Rumbaugh (1991), Martin & Odell (1992 and 1995) and Yourdon (1994), which each present the basic concept of object-orientation. The basic principles are the same with all writers but the differences are in the notation (before UML was developed) and in the elements or features of object-orientation that are considered most important. Object-oriented analysis and design is mostly used in software development and thus some of its concepts are closely software related. Therefore, many authors present the concept in software terms and with software examples. However, the basic principles in object-oriented systems are applicable to other development as well. They have been used for example in electronic system design and embedded systems. In the author’s opinion, Booch’s presentation distinguishes the best between the software development and the principles of the concept, making it easier to grasp the conceptual ideas and seeing how they can be applied to development other than software development. The following chapters present the object-orientation related subjects as defined by Grady Booch et al.

2.1.1 Fundamental properties

A system is considered to be object-oriented if it has the fundamental properties of object-oriented modelling. According to Booch et al. (2007), these properties are abstraction, encapsulation, modularity, and hierarchy.

Abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects, and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer (Booch et al. 2007) It is a mechanism and practice to reduce and factor out details so that one can focus on a few concepts at a time. For example, an abstraction of leather ball, rubber ball, and
bowling ball could be a ball. Abstraction is utilized in formulating the object-oriented architecture. Through abstraction, the system is decomposed to classes. Classes are the types of objects that the system will be constructed of.

Encapsulation is the process of hiding all of the details of an object that do not contribute to its essential characteristics (Booch et al. 1991). In other words, it means packaging the operations and data together inside an object such that the data is only accessible through its interface (Yourdon 1994). For example, a motor controller may provide an interface which enables the user to select the rotation speed and direction for the motor. Due to encapsulation, it hides the irrelevant information from the user, i.e. how the motor is driven. Thus, the user is only required to know how fast the motor must rotate, but the user is not required to know how the controller applies the current and voltage to the coils of the motor, or whether it is driven by PWM or some other method. Encapsulation and abstraction are complementary concepts. Abstraction focuses on the observable behaviour of an object, whereas encapsulation focuses on the implementation that gives rise to this behaviour (Booch et al. 2007). Encapsulation is important for the reusability of objects.

Modularity is a property of a system that has been decomposed into a set of cohesive and loosely coupled modules. Principles of abstraction, encapsulation, and modularity are synergistic. An object provides a crisp boundary around a single abstraction, and both encapsulation and modularity provide barriers around this abstraction (Booch 1991). For example, a C++ programmer may include several closely related classes into one namespace so that their reuse is as convenient as possible. In Java, a similar thing is done with packages.

Hierarchy is ranking or ordering of the abstractions (Booch et al. 2007). In the object model, there are two important types of hierarchy: inheritance and aggregation. Inheritance represents a hierarchy of abstractions, in which a subclass (child) inherits from one or more superclasses (parents) (Booch 1991). A subclass has the structure and behaviour of its superclass, which it augments or redefines. Inheritance is also called a generalization/specialization hierarchy. The superclasses are generalized abstractions. When the subclasses augment and redefine these abstractions, it is called specialization. For example, a parent (or generalized or super) class called 'cat' could be specialized into a class called Persian cat or tiger, by augmenting data and behaviour that is specific to each of these child (or specialized or sub) classes. Inheritance is also called the "is a" hierarchy: you can say that a Persian cat is a cat or a tiger is a cat.

Another form of hierarchy is aggregation. Aggregation means that one object is part of another object. Aggregation is also called the "part of" hierarchy. For example,
a cat has legs, head, tail and other body parts which each could be their own classes. Thus the hierarchy between the cat and tail is aggregation.

Hierarchy, especially inheritance and aggregation, are significant parts of creating the architecture of object-oriented systems. It provides the base for manageable architecture, reusability of objects, and maintainability for a system.

2.1.2 Other properties

Booch et al. (2007) define also three minor properties of the object model. Minor means that each of these properties may be useful but not essential part of the object model. The minor properties are typing, polymorphism, and concurrency.

Typing is the enforcement of the classes of the class of an object, such that objects of different types may not be interchanged, or at the most, they may be interchanged only in very restricted ways. Typing lets us express our abstractions so that the programming language in which we implement them can be made to enforce design decisions (Booch et al. 2007). Typing is particularly relevant as the complexity of our system grows by preventing wrong types to be used with typed structures or methods.

Polymorphism is a concept in type theory wherein a name (such as variable declaration) may denote instances of many different classes as long as they are related by some common superclass (Booch et al. 2007) or happen to have the same name (Yourdon 1992). Any object denoted by this name is thus able to respond to some common set of operations in different ways.

Concurrency means that several sequences of operations (or threads) are handled at the same time (Awad et al. 1996). Often, systems may run on a single CPU and they only archive an illusion of concurrently running threads, while in fact they may use some time-slicing algorithm. Systems running on multiple CPUs can allow a true concurrency. In object model, there may be concurrency both between and within objects. Concurrency allows events to be handled simultaneously, but it creates problems regarding the consistency of data, because the same data can be simultaneously accessed by two concurrent requests. This requires synchronization mechanisms to be used.
2.1.3 Development Process

An object-oriented system can be developed with many different development process types. Software development processes can be roughly divided into two categories: waterfall process and iterative and incremental processes.

Waterfall process is the traditional process for making software. There the design process flows from phase to phase, ending in the release of a completed project. Each phase is made in a sequential order, as shown in Fig. 6. The difficulty of returning to an earlier phase once it has completed has been compared to swimming up a waterfall, hence the name (Bennet et al. 2006). There are variations of what the phases are in a waterfall model. Bennet describes the phases as:

- system engineering, where the major software and hardware elements are identified,
- requirement analysis,
- design,
- construction (implementation),
- testing,
- installation, and
- maintenance.

In iterative and incremental development process, the functionality of the system is developed in a successive series of releases (iterative) of increasing completeness (incremental). A release may be external (available to the customer) or internal (not available to the customer). The selection of what functionality is developed in each iteration is driven by the mitigation of project risks, with the most critical risks being addressed first. The experience and results gained as a result of one iteration are applied to the next iteration. With each iteration, one gradually refine one’s strategic and tactical decisions, ultimately converging on a solution that meets the end user's real (and usually unstated) requirements, and yet is simple, reliable, and adaptable (Booch et al. 2007). Iterative and incremental processes include agile methods (i.e. methods using short timeboxed iterations with adaptive, evolutionary refinement of plans and goals (Larman 2003), such as extreme programming (XP) software engineering method (Steinberg et al. 2003) and Scrum project management method (Scrum Alliance 2008).

The benefit of the waterfall process in respect to iterative and incremental is that if it can be successfully applied, it is more predictable, produces a good documentation and it is easier to manage for large projects. In order to successfully
use the waterfall process, the system developed needs to be well understood, with a familiar scope and feature set, and the requirements must be fairly stable, which often may not be the case. (Booch et al. 2007)

The benefit of iterative and incremental processes is in its flexibility. The requirements for the end result may change during the development process, and it does not cause harm to the project. The key components of the project are built early in the development process, which makes it easier to identify reusable elements and key risks in the project, and tactical changes to the product are possible. Defects can be found early and corrected, since testing is performed during every iteration. Furthermore, project personnel may in some cases be employed more efficiently,
team members can learn along the development process from earlier iterations, and the development process can be refined and improved during each iteration. (Booch et al. 2007)

The division between the types of processes is not that clear in real life. It is common to include the iteration as a part of the development process, regardless whether it is iterative and incremental or resembles more the waterfall type. For example, the Rational Unified Process (Kruchten 2004) and the Spiral model (Boehm 1988) are waterfall derivative processes which include iterations. Stiller & LeBlanc (2002) calls these hybrid software development processes. Booch et al. (2007) use a term plan-driven process, meaning a waterfall type of process that is not necessarily iterative and incremental but can be.

Even though the processes vary from waterfall to iterative and incremental, the processes contain similar phases when using object-oriented approach. Each cycle of these phases produce a release of the system. The development process with OOAD contains the following phases (Booch et al. 2007):

- requirements,
- analysis & design,
- implementation,
- test, and
- deployment.

In the requirements phase, the system under development is studied and described in some manner. Usually, a requirement specification document is made in this phase. It describes what the system needs to do and how it is used, and gathers the requirements of the application for a release. There exist fact finding techniques that help to identify the requirements, and notations and methods of writing down the requirements. The Unified Modelling Language (UML) is one popular option for a standardized visual specification language for object modelling (Muller 1997) (http://www.uml.org/).

The purpose of analysis & design phases is to produce an architecture description and analysis/design model from the system. This phase contains the following activities:

- Identifying the elements (i.e. key abstractions, architectural partitions, classes, objects, and other components). The system is decomposed into architectural partitions, class hierarchies, and objects.
- Defining the collaborations between the elements. This activity describes how the identified elements collaborate to provide the system's behavioural requirements.
- Defining the relationships between the elements that support the element collaborations.
- Defining the semantics of elements. The behaviour and attributes of the identified elements are established, and the elements are prepared for the next level of abstraction.

The analysis and design phase is done in several levels of abstraction. It starts from the architecture and components of the whole system, and as the system is being decomposed, it proceeds further down to subsystems and subcomponents in the system. Alternatively, the architecture can also be refined from the bottom up. The end result is a detailed design realization of the original requirements. (Booch et al. 2007)

In the implementation phase, the final technical decision are made as the design is being actually realized. The implementation phase also includes unit tests and the integration of the design. The result of this phase is a working system.

In the test phase, the implementation is tested against the requirements. Its purpose is to validate through a concrete demonstration that the system functions as designed. Finally, the deployment phase ensures that the system is available for its end users.

2.2 Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is applying the object-oriented fundamentals into the software development. Booch (2007) defines the OOP as follows: "Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships." OOP uses objects, not algorithms, as its fundamental logical building blocks. Each object is an instance of some class, and classes may be related to one another via inheritance relationships.

OOP requires a support from the programming language. Currently, the support for object-oriented programming is in about all popular languages, for example C++, C#, Java, Visual Basic, Python, Perl, and PHP. Object-oriented programming itself is conducted according to the development process as described in the previous section.
Significant to OOP is the emergence of class/object libraries and object-oriented frameworks. Their availability has made the software development a lot faster and easier. For example, the Microsoft Visual Basic (Microsoft 2008b) has already since 1991 utilized the objects and object libraries in its user interface and coding fashion. The visual user interface with objects, such as ActiveX and COM objects (Microsoft 2008c), was a concrete example of how the objects could be utilized in a productive way. Another example is the Java programming language by Sun Microsystems, where the class libraries are in a significant role in (Sun 2008a). Java has a large set of library objects that can be used on different HW platforms (Sun 2008b), thus utilizing the reusability aspect of the objects in its extreme. A more recent object-oriented framework is the Microsoft .NET framework (Microsoft 2008d), which among other things, combines the easy-to-use graphical interface Visual Basic with the vast class libraries and portability of Java.

2.3 UML

The Unified Modelling Language (UML) is the primary modelling language used to analyze, specify, and design software systems, especially object-oriented systems. It is a standard issued by the Object Management Group (OMG). OMG is a consortium of methodologists and companies that creates and maintains standards for the computer industry. The first UML standard became in 1997 and now the development has proceeded to version UML 2.0. The latest UML specifications are available for public at http://www.uml.org.

UML has numerous types of diagrams, each providing a certain view to the system. The diagrams can be divided into two groups: structure diagrams and behaviour diagrams. The structure diagrams are used to show the static structure of elements in the system. They may depict such things as the architectural organization of the system, the physical elements of the system, its runtime configuration, and domain-specific elements of your business, among others.

Structure diagrams are often used in conjunction with the behavioural diagrams to depict a particular aspect of the system. The behaviour diagrams describe the dynamic behavioural semantics of the system, such as external events triggering an operation or objects sending messages to one another.

Use of the aforementioned diagrams in practice is usually case dependent. Although UML contains a wide set of diagrams, not every diagram is meant to be used at all times. Rather, a proper subset of this notation is usually sufficient to express the semantics of a large percentage of analysis and design issues. Thus, in
each project, depending on its type and depth of details, the diagrams are applied as the analysis and design requires.

2.4 Discussion

The object-oriented fundamentals and the related techniques have become a success in software development. They bring many benefits to software development. Significant benefits are for example the ease of use, quick development and robustness. The work for creating a new functionality needs to be made only one time. Once the functionality is made as an object, it can be reused as such or enhanced with more properties. This saves development time and increases the robustness of the resulting application. Furthermore, through the reusability of objects and manageable architecture, vast object libraries and object-oriented frameworks are possible. With the library objects, object-oriented framework, and intuitive graphical user interface, the software development can be as simple as collecting required objects to the application and the creating the high level control code, and still get perfectly useful applications for commercial use.

Modern software development tools integrate object-oriented ideas and an intuitive graphical user interface achieving a development tool that is very easy to use. The vision for the work in this thesis was to have a corresponding easiness for creating embedded systems through Lego-like, easy-to-use electronic blocks with object-oriented, easy-to-use development tools. The object-oriented fundamentals have been used in electronics before, at a description language level. In this thesis, the object-orientation is applied to a different level as the objects are considered to be entities that consist of both hardware and software. The goal was to apply the object-oriented fundamentals to these objects as they best fit, and thus develop an easy way to build embedded systems that still produces commercially feasible devices.

Paper I presents how the object-oriented fundamentals can be utilized in embedded systems. The paper was written in 2004 when the first generation of Atomi objects was made. The concept has evolved since, and the current version of Atomi objects is quite different and much more commercially oriented than the first versions. Thus, it may not be clear whether it is justified to call the embedded object concept with Atomi implementation actually object-oriented. For some people, it may simply be modular or object-based. This semantic difference is insignificant for the feasibility of Atomi objects or the EOC, but it is an interesting subject regarding further development of the concept. The question is how to utilize more of the object-
oriented ideas with this concept. That is where the analogies and mapping of ideas to concrete measures may be significant.

According to (Booch et al. 2007), a system is considered to be object-oriented if there are four major elements of this model: abstraction, encapsulation, modularity, and hierarchy, which mainly means inheritance and aggregation (terms were introduced in section 2.1). These elements realize in Atomi objects in a following way:

Abstraction, i.e. presenting the essential characteristics of an object that distinguishes it from other types of objects, realizes clearly in different types of objects. Each Atomi object has one functionality, which is the way how a system is decomposed into classes.

Encapsulation, the process of hiding all of the details of an object that do not contribute to its essential characteristics, realizes in the technical details and the implementation of an object. Each Atomi performs its function as described in its datasheet. The way it does that is hidden from the user i.e. user does not need to worry about that part. Of course, if the user wants familiarize himself with the interiors of an object, it can be possible.

Modularity, decomposing a system decomposed into a set of cohesive and loosely coupled modules, has an obvious realization with Atomi objects: each Atomi is also its own module. Creating modules as a collection of several objects, as in software, is not reasonable or relevant as the Atomi objects are physical entities. If an analogy to module as a collection of objects needs to be achieved, it could be compared to how the Atomi objects are packed and sold by the party that does business with them. It may be reasonable to store or sell Atomi objects as kits (i.e. modules), containing those objects that are closely related to each other in some application. For example, a HID controller kit could include USB Atomi for USB connection, an IO Atomi for attaching switches and an ADC Atomi for connecting joysticks to the controller device.

Hierarchy has two most important types: inheritance and aggregation. Aggregation, i.e. one object is part of another object, is realized in the architecture with the GroupAtomi. A new object (so called larger object) can be made out of several Atomi objects or other larger objects. The GroupAtomi provides the encapsulation by providing a new interface via a separate physical bus interface. This new interface represents the larger object entity, which is composed of several objects. Through the new interface, only those variables are exposed that are relevant for the functionality of the whole.
Inheritance, a hierarchy of abstractions, in which a subclass (child) inherits from one or more superclasses, is a more complicated issue. In software world, the most significant use on inheritance is in the reuse and managing changes. Modules with sufficiently similar interfaces can share (reuse) a lot of code, reducing the complexity of the program. This is realized by inheriting all classes from a common parent, i.e. a base class. When the basic interface needs to be changed for all classes, only the base class code needs to be changed. Then, after a new compilation of each object, the new interfaces have been implemented for all modules at the same time.

In Atomi objects, the inheritance is realized at many levels. Firstly, due to the Atomi II Framework specification, each Atomi must be of a certain size and have the Atomibus, etc. Thus, each Atomi derives from the same physical and electrical template, which is the equivalent of the base class. This is utilized in the PCB and software design: designing an Atomi can be started from a suitable template for both hardware (and software if applicable). The inheritance is further utilized already at the template level: there are specialized templates containing different MCU choices with already routed basic circuitry. For each template with an MCU, there exists a software template that realizes the basic Atomibus operations. Furthermore, each complete Atomi can in some extend to be further specialized. For example, an AD-converter Atomi can be further specialized for measuring current, certain specific voltage range or some specific sensor. The reuse is well exploited, but managing change does not realize as in software. There is no automatic compilation tool for that purpose i.e. a change to base class requires (at the moment anyway) a manual propagation of the change to each Atomi that is built on it. To avoid such changes, the base class in Atomi II Framework has been very carefully designed during several years.

Paper I presents also another level of inheritance, i.e. the high level inheritance. It shows how a specialized class can be made via a similar process to aggregation. Even though the depicted process provides a specialized class from the interfaces of the parent classes, this process could still be considered also as an aggregation.

A new application of inheritance has emerged after writing the first paper. It is the inheritance of the variable table interface. Each Atomi object has a variable table that is used to control its properties, events, and methods. These variables are organized in an object-oriented fashion according to the different levels of abstractions. For example, for a motor controller Atomi, there is a certain base set of variables. If an Atomi is a DC-motor controller or stepper motor controller, this interface is further specialized to contain more variables. This simple method is not significant for the Atomi developer but is it valuable for the Atomi user. A motor controller made by one technique can be directly replaced by another version without
any modifications to software. Furthermore, even a motor type can be changed without any modification to software. This is a valuable property for modular systems, such as robots.
3 Embedded system development

This chapter presents the current methods for developing embedded systems and their relationship to the EOC.

3.1 Traditional embedded system development

An embedded system development process is a slow, time-consuming and expensive process that requires a lot of expertise. Traditionally, embedded system development consists of the following phases (Berger 2002):

- product specification,
- partitioning of the design into its software and hardware components,
- iteration and refinement of the partitioning,
- independent hardware and software design tasks,
- integration of the hardware and software components,
- product testing and release, and
- on-going maintenance and upgrading.

The development process is depicted in Fig. 7. The process starts from creation and inspection of a product specification. The product specification can be further divided into two parts: requirement specification (or definition) and system specification. The first objective is to create a requirement specification, which collects data about the product or system that is under development. Its goal is to define what the product does exactly and how well (for example, required accuracy of measurements). The requirement specification serves as a basis for the development process. The system specification follows the requirement specification. It defines how the system operates in order to do what the requirement specification states. Often, the system specification includes input and output specifications, use cases, and user interface (UI) definitions. It also defines some higher level subsystems and functions that implement the required features. (Peckol 2008, Berger 2002)
Fig. 7. Traditional embedded system development process.
This phase is followed by the partitioning of the design into software and hardware components. The system is decomposed into functional blocks which communicate together to achieve the desired functionality. The functional blocks are then mapped into either HW or SW implementations. Decision between implementing functions with SW or HW depends on many things. Most importantly, they depend on the requirement and system specification, which gives the performance and pricing directives for this phase. The partitioning phase is essentially searching for the optimal implementation for different functionalities of the system. Often, SW implementation is preferred if possible due to its inexpensive costs and flexibility via programmability. HW implementations are usually needed in time critical parts and electrical interfacing parts. The partitioning phase produces a functional and an architectural design of the system. The functional design describes the functional blocks of the system and the architectural design maps these functions into hardware and software implementations. (Peckol 2008, Berger 2002)

The iteration and implementation phase is a blurred area between implementation and HW/SW partitioning. This phase presents the early design work before the hardware and software teams go their separate ways. Actually, this phase could also be presented as a part of the partitioning phase. For example, the partition phase includes selecting the main components for the device, most importantly the MCU, FPGA, or other processor or processors that the system is based on. In order to become familiar with the devices, this phase may often involve buying a development kit for the MCU and/or other devices and creating some test codes for them. Similarly, different electronic parts may be separately created and tested on a stripboard or prototyping boards before being selected for the design. In some cases, the whole system may be simulated with a computer and emulated on an FPGA-based system before actually locking any parts of the design. This is also the part where a lot of expertise is needed: there are several ways to implement a function, several components of which to choose from, each affecting the costs and performance differently. There are also several possibilities to go wrong. (Berger 2002)

HW and SW design activities are parallel processes. In the HW design phase, the details of the HW of the system are defined and then implemented. The HW design path is divided into detailed HW design, PCB assembly, and HW tests.

The detailed HW design involves drawing a detailed schematic of the PCB and completing the component selection by choosing the side components for the board. The resulting schematic is then converted into a PCB layout by choosing a suitable PCB type, placing the components on a PCB, and routing the connections between the pins of the components in a feasible way. Finally, the created PCB design is turned
into a part list and production files for the PCB, enabling the actual manufacturing of the PCB.

In the PCB assembly phase, the actual PCB is made according to the files and the components are purchased. Then, the PCB is assembled by connecting the components to the PCB with some soldering method.

In the HW tests phase, the resulting PCB is tested electrically in order to verify that the production of the PCB has been successful. The test phase may also involve designing testing methods and creating a tester device for the system.

SW design activities include creating the detailed SW design, implementation, and Simulator tests. In the detailed SW design, the architecture and/or flow chart type of design is created of the software. This is then implemented, i.e. programmed in the implementation phase and tested in the simulator tests. Often, the implementation and tests are simultaneous as the software is being tested part by part as its being implemented.

When HW and SW of the device are ready they are integrated, i.e. the software is placed into the actual target HW and tested. Often, this phase requires iteration on both software and hardware designs and implementation, because it is (almost) impossible to create either without any errors (or bugs) in the first try. This phase requires good debugging and measuring tools for determining and discovering the problems that occur.

Acceptance testing follows a successful integration of HW and SW. In this phase, the system is tested against its functional requirements. Test phase makes sure that the device does everything it is supposed to do and is stable. Thorough testing is important as releasing a defective product is many times more expensive than investing in proper testing. Successful acceptance testing leads to the release of the product.

In the maintenance and upgrading phase, the existing released product is being under development. The bugs found by customers are corrected and new features that are found to be necessary or useful by the customers are added to the product if it is possible.

### 3.2 HW/SW co-design

This section presents a more modern embedded systems development process called HW/SW co-design. More specifically, the section introduces the background, development process and object-oriented extensions of this approach.
3.2.1 Introduction to co-design

Hardware-software co-design simultaneously designs the software architecture of an application and the hardware on which that software is executed. (Wolf 1994). In contrast to the traditional process of creating the software and the hardware of a system in independent parallel tasks, in co-design these tasks are joint.

The motivation for co-design is to correct some problems in the traditional design process. In traditional development process, the hardware and software partitioning is more or less locked before the implementation phase, because the software and hardware implementation paths are independent and parallel after partitioning phase. In this approach, the hardware may be specified without fully appreciating the computational requirements of the software in terms of processor speed and memory capacity. In addition, often software development does not influence hardware development and does not track changes made during the hardware design phase. It is finally during system integration phase that the software and hardware are combined and tested as a whole, revealing the actual characteristics of the system. When problems are encountered, there are some typical ways of correcting them. In some cases, the revealed hardware inadequacies can be corrected with software. Sometimes, some additional hardware is added to the system. In a worst case, the problems require iteration in the whole design cycle, which is slow and expensive. (Kumar et al. 1996)

The co-design fixes this problem by designing the hardware and software together, finding an optimal partitioning for each function in a system. The design is divided into functional modules and the implementation of each module, both software and hardware, is evaluated against requirements with simulation models. The co-design uses a special design representation, which is unified for both software and hardware. Unity is achieved through abstractions that can be implemented as well with hardware as with software. The partitioning and evaluating cycle can be iterated several times, allowing the final partitioning decision to be made as late a stage as possible. Co-design utilizes performance modelling techniques, such as stochastic Petri nets, for evaluating the goodness of different implementation options. Automated synthesis methods are used for creating the actual implementations. (Kumar et al. 1996)

Co-design has been widely researched. Different tools, techniques, notations, and design methods have been developed, for example (Slomka et al. 2000, Castellano et al. 2000, Noguera & Badia 2002, Russel 2002, Savage et al. 2004, Thepayasewan & Doboli 2005).
Slomka et al. (2000) has developed a Co-design and Rapid-Prototyping System for Applications with Real-Time Constraints called Corsair. It is a tool for the analysis, synthesis, and rapid prototyping of distribute embedded real-time systems. It involves specifications with SDL and MSC languages augmented with real time extensions, and testing a prototype system in a real environment instead of simulation via reconfigurable hardware/software systems. Its goal is to shorten the design cycles for large embedded systems, especially real-time communication systems like base stations for mobile communication.

Castellano et al. (2000) have developed a hardware/software co-design environment called GACSYS (GAC’s Co-design System). GACSYS is oriented to data-dominated applications. GACSYS uses a specification language named VSS (VHDL-based System Specification), which supports high level statements to make system specification easily. GACSYS contains a compiler to translate from VSS code to VHDL one. GACSYS uses also an intermediate representation to support VHDL specifications. It is based on the Architectural Synthesis of Complex Integrated Systems (ASCIS) data flow graph. There is also a compiler to generate the intermediate representation from VHDL code. Furthermore the GACSYS contains a hardware/software partitioning tool that supports process-level pipelining and takes into account system power consumption. This enables the exploration of the design space to make latency, area and power trade-offs.

Noguea & Badia (2002) have researched HW/SW co-design techniques for dynamically reconfigurable architectures, i.e. for programmable logic devices that are reconfigurable at run time. They have developed a HW/SW co-design methodology with dynamic scheduling for dynamically reconfigurable architectures, a dynamic approach to dynamically reconfigurable logic (DRL) multicontext scheduling, and an automatic HW/SW partitioning algorithm for DRL architectures.

Russel (2002) has applied program slicing techniques to co-design. Program slicing can be used as a maintenance or reuse tool for activities such as program understanding, regression testing, and function extraction from an existing code. Co-design techniques can use program slicing analysis as a design tool.

Savage et al. (2004) have developed an extended genetic algorithm (EGA) for complex hardware software co-design problems. The EGA is used with a normal co-design process for optimizing the implementation for synthesis. The EGA is mostly valuable in cases where the problem space is too large for an exhaustive search, a linear programming solution would be too slow, a deterministic method would have to be either very complex or highly domain specific and a heuristic method would have to avoid the many local optima and allow for complex interactions among
variables. The EGA implements a selection method for the implementation with function scaling, adaptive crossover, and mutation. The EGA is targeted for dataflow oriented applications and synthesis on Field-Programmable Gate Arrays (FPGAs).

Thepayasuwon & Doboli (2005) have developed a layout conscious approach to hardware/software co-design of Systems on Chip (SoC) optimized for latency, including an original algorithm for bus architecture synthesis. Their co-design method improves the effectiveness of SoC system-level design. The bus synthesis algorithm creates customized bus architectures in a short time, depending on the data communication needs of the application and the required performance. Their novelty is that the method addresses layout-related issues that affect system optimization, such as the dependency of task communication speed on interconnect parasitic.

3.2.2 Co-design process

![Diagram of co-design process]

Kumar et al. 1996 describes the co-design process as follows (see Fig. 8): A typical co-design process starts with a functional system representation, independent of
hardware and software. Several system representations may be utilized, including finite state machines (FSM) and concurrent processes. In some circumstances, the system is described using a programming language (e.g. SystemC) which is compiled into an internal representation, such as data/control flow description. This description serves as unified representation, one that can represent software or hardware.

Hardware/software partitioning is the process of determining which functions are provided in hardware and which in software. It is similar to the traditional process except that it is performed on the unified representation. This representation can be an internal description or the system description itself. The partitioning can be accomplished manually or automatically. The result is a hardware/software partition, which specifies the functions to be implemented in hardware and in software. From this description, the hardware and software for the system are generated. Typically, hardware synthesis and software synthesis techniques are employed. Any interfaces required between communication software and special hardware components are synthesized, after which the system is integrated.

The goodness of the partitioning decision can be evaluated in HW/SW partitioning phase and system integration phase. Sometimes, estimation techniques are used to evaluate the partitioning, typically in terms of performance. If the result does not satisfy requirements, another SW/HW partitioning is generated. This is done iteratively until a satisfactory partitioning is developed.

The significant differences between traditional design and co-design processes include the use of a unified representation for describing hardware and software, and the ability to perform HW/SW partitioning iteratively.

3.3 Object oriented enhancements to embedded system design

The object-oriented techniques have been adopted in electronics design in different ways. The motivation is to benefit from the basic object-oriented benefits, especially reuse and managing complexity. Commonly, the object-oriented methods in hardware design aim to arrange the design as reusable hardware objects, where the hardware objects are abstract, hardware description language level components. This is because in ASIC and FPGA based embedded systems, the hardware is often modeled, simulated, and synthesized via hardware description languages. The programming techniques that can improve the design process of software systems are applied also to hardware description languages.

A clear main stream methodology does not exist yet. Instead, several methods and notations have been developed by individual research parties. The research
approaches to object-oriented hardware design can be divided into two major categories: In the first category, the system descriptions are based on existing object-oriented software languages such as Java or C++. These are augmented with missing hardware features such as concurrency, reactivity, and an appropriate timing model. The second category is based on existing hardware description languages such as VHDL (IEEE Computer Society 2002) or Verilog (IEEE Computer Society 2001). These are correspondingly augmented with object-oriented features.

Kumar et al. (1994) present how the hardware components can be modelled as classes using C++ syntax in conjunction with a co-design process. As an example, they create a class of a register which is then used to derive specialized classes, such as a program counter and stack counter. The resulting classes can then be implemented in either hardware or software using appropriate synthesizing methods.

Another C++ based approach is SystemC\(^7\) (Grötker et al. 2002). It is a C++ class library that introduces some missing typical hardware features in C/C++. It allows to model systems, including both hardware and software parts. For simulating these systems, it provides an integrated simulation kernel which is automatically linked to each executable SystemC model. A SystemC hardware model is very similar to an equivalent VHDL or Verilog model. Classes represent entities, ports, or signals and special notations allow the modelling of processes like in common HDLs. SystemC provides also predefined data types for bit, std_logic, bit-vector, std_logic-vector and arbitrary sized integers, which typically build the basis of the type system of a hardware description language. SystemC has been further enhanced by a number of researchers. For example, Grimpe (2001) has augmented the SystemC with a SystemC-plus extension, i.e. an additional class library and a coding style which specifies how to write synthesizable object-oriented hardware models. Correa et al. (2007) have developed an approach to translate UML 2.0 notations to SystemC language. Moinudeen et al. (2006) has developed a methodology to verify SystemC designs. The methodology is based on an automatic generation procedure of the finite state machine of the system from SystemC description.

Green et al. (1994 and 2000) have developed the Model-based Object-Oriented Systems Engineering (MOOSE) method that also includes its own notation. It was initially developed to target software and/or ASIC implementation. It is a co-design method that uses C++ and VHDL for implementation. The design process is quite similar to regular co-design process. It begins with analysis of requirements, proceeding then to uncommitted models. Uncommitted models are diagrams of the

\(^7\) SystemC URI: www.systemc.org
objects and their interactions in the system and its subsystems. The uncommitted models do not contain the decision of whether the system or its part is to be implemented by hardware or software. The uncommitted model can be rendered executable by coding each operation in a simple procedural language. The executable model can then be tested. After testing, the design proceeds to committed models, which determine the partitioning to software and hardware, and other implementation specific things. After the committed model meets the requirements, the implementation phase begins. The implementation of the detailed design involves mapping soft subsystems into C++ class skeletons and hard subsystems into VHDL.

Fleischman et al. (1998) presents a corresponding method of using Java as the system description language for dynamically reconfigurable embedded systems. The Java based specification is finally synthesized into VHDL code and Java bytecode.

As mentioned before, hardware description language VHDL has been extended with object-oriented features, both to manage the complexity increase and to augment the capabilities and expressiveness of VHDL. Djafri & Benzakki (1997) present the Object-Oriented VHDL, where they have added features such as inheritance, polymorphism mechanism, and communication via messages to VHDL. Chung & Kim (1990) have developed a system-level design environment that uses object-oriented approach for modelling VHDL entities.

### 3.4 Platform-based design

This section presents the currently predominant embedded systems development process called platform-based design. The section introduces the background and development process of this approach.

#### 3.4.1 Introduction to platform-based design

Platform-based design is a co-design approach that tries to solve the problems caused by the ever-increasing complexity and time-to-market pressure in embedded system design. It is motivated by the following (Carloni et al. 2005):

1. The disaggregation of the electronic industry: Today, the identification of a new market opportunity, the definition of the detailed system specifications, the development of the components, the assembly of these components, and the manufacturing of the final product are tasks that are mostly performed by distinct organizations. The integration of the design chain becomes a serious
problem whose most delicate aspects occur at the hand-off points from one company to another.

2. The pressure for reducing time-to-market of electronics products in the presence of exponentially increasing complexity has forced designers to adopt methods that favour component re-use at all levels of abstraction. Furthermore, each organization that contributes a component to the final product naturally strives for a position that allows it to make continuous adjustments and accommodate last-minute engineering changes.

3. The dramatic increase in Non-Recurring Engineering (NRE) costs has created the necessity of correct-the-first-time designs.

In platform-based design, the system is built with precisely defined layers of abstraction, a platform, through which only relevant information is allowed to pass. For example, a device driver provides a layer of abstraction between an operating system and a device. The idea has been exploited for years in the design of Personal Computers. (Horowitz et al. 2003)

A platform hides the details of several possible implementation refinements of the underlying layers. It is a library of elements characterized by models that represent their functionalities and offer an estimation of (physical) quantities that are of importance for the designer. The library contains interconnects and rules that define what are the legal composition of the elements. A legal composition of elements and interconnects is called a platform instance. (Carloni et al. 2005)

Platform-based design is a meet-in-the-middle process, where successive refinements of specifications meet with abstractions of potential implementations that are captured in the models of the elements of the platform. The meet-in-the-middle process is the combination of two efforts:

- top-down: map an instance of the top platform with constraints into an instance of the lower platform with appropriate constraints resulting from an appropriate propagation involving budgeting wherever needed;
- bottom-up: build a platform by defining its components and their performance abstraction (e.g. number of literals for technology independent optimization, and area and propagation delay for a cell in a standard cell library).
The basic idea of system platform-stack is captured in Fig. 9. The vertex of the two cones represents the combination of the API and the architecture platform. A system designer maps its application into an abstract representation that includes a family of architectures that can be chosen to optimize cost, efficiency, energy consumption, and flexibility. The mapping of the application into the actual architecture in the family specified by the Application Programming Interface (API) can be carried out, at least in part, automatically if a set of appropriate software tools (e.g. software synthesis, RTOS synthesis, device-driver synthesis) is available. This set of tools makes use of the software layer to go from the API platform to the architecture platform. The system platform effectively decouples the application development process (the upper triangle) from the architecture implementation process (the lower triangle).

Establishing the number, location, abstraction, and components of intermediate platforms is the essence of platform-based design. The trade-offs involved in the selection of the number and characteristics of platforms relate to the size of the design space to be explored and the accuracy of the estimation of the characteristics of the solution adopted. The larger the step across platforms, the more difficult is predicting performance, optimizing at the higher levels of abstraction, and providing a tight lower bound. (Carloni et al. 2005)
3.4.2 Design process

Keutzer et al. (2002) depict the design process, as shown in Fig. 10. The design starts by describing the functions of the system. A function is an abstract view of the behaviour of an aspect of the system. This set of functions is the input/output characterization of the system with respect to its environment. It has no notion of implementation associated with it. The description of the function the system has to implement is captured using a particular language that may or may not be formal.

The next stage is the evaluation of tradeoffs across the architecture/microarchitecture boundary. The structural compositions that implement the architecture are of primary concern. Terms architecture and microarchitecture have the following meanings: the architecture defines an interface specification that describes the functionality of an implementation, while being independent of the actual implementation. The micro-architecture defines how this functionality is actually realized as a composition of modules and components along with their associated SW. It is a set of interconnected components (either abstract or with a physical dimension) that is used to implement a function. The micro-architecture determines the final HW implementation and, hence, it is strictly related to the concept of platform that will be presented in greater detail later on.

In mapping process, the functions to be implemented are assigned (mapped) to the components of the micro-architecture. The mapping process determines the performance and the cost of the design. Based on estimations on the performance and cost of the implementation of different functions, the design space is explored in order to find a good micro-architecture.

The “function on microarchitecture“ -phase is entered once the mapped micro-architecture has been estimated as capable of meeting the design constraints. In this phase, the components of the chosen micro-architecture are implemented. This requires the development of an appropriate HW block or of the SW needed to make the programmable components perform the appropriate computations. This step brings the design to the final implementation stage. The HW block may be found in an existing library or may need a special purpose implementation as dedicated logic. In this case, it may be further decomposed into sub blocks until either we find what we need in a library or we decide to implement it by custom design. The SW components may exist already in an appropriate library or may need further decomposition into a set of subcomponents.
More detailed design processes have been studied by several other research groups. For example, Kangas et al. (2006) have developed an UML Multiprocessor SoC Design Framework called Koski that contains a complete design flow for multiprocessor systems-on-chips (SoCs), covering the design phases from system-level modelling to FPGA prototyping. The system is modelled in a UML design environment following a UML profile that specifies the practices for orthogonal application and architecture modelling. The design flow tools are governed in a single framework that combines the sub tools into a seamless flow and visualizes the design process. Wehrmeister et al. (2005) have developed an object-oriented platform-based design process for real-time embedded systems. The approach promotes a smooth transition from high-level UML specification to implementation, which is composed by hardware and software components. The transition from higher to lower abstraction levels is facilitated by the use of an OO real-time API, whose underlying facilities can be optimized according to the application needs and selected platform. An integrated toolset is used to support the intermediate steps of the design process.
3.5 Discussion

In both traditional and co-design processes, an embedded system is usually built from individual components and/or modules, which have different interfaces and various electrical properties, and require a PCB development in the implementation phase. The problems of this approach regarding expertise requirements, time, and costs were introduced in section 1.2.

The co-design process is of help in the design phase to make it more certain that a design, once it has been designed and is proceeding into implementation, has as few errors as possible. The co-design also improves the traditional design process with more efficient implementations, and improvements in overall system performance, reliability, and cost effectiveness. The co-design process relies heavily on electronic design automation tools, such as high-level hardware and software synthesis capabilities. It is mostly applicable with VLSI systems, involving ASICs and FPGAs, and is very usable with reconfigurable hardware platforms.

The co-design with object-oriented enhancements and platform-based design bring benefits to the system design, especially in reuse and managing complexity. They both give also some tools for creating ready-made modules. However, in the way they are commonly used, they apply for the design phase making the architecture more reusable and easily manageable.

The relationship of EOC to these approaches is as follows: the EOC and the aforementioned design processes can be considered as different level issues. Traditional design, co-design, and platform-based design processes can be used in developing an embedded object, and thus embedded objects can be considered as simply modules made with these processes. However, these processes can also be used in developing embedded system with embedded objects. On the system design level, the EOC is similar to co-design in that hardware and software are combined as a functional unit, and a system is built with such units. In platform-based design, the EOC could be viewed as microarchitecture.

The difference comes along the standardized modular structure with standardized physical and functional interfaces, and implementation related properties that the EOC defines. This standardization enables the possibility for ready-made implementation of these modules. As connecting the modules together does not require PCB design, assembly, interface design, or many other implementation level things, EOC can make implementing systems easier and faster (but also more limited). EOC also mixes the physical implementation with design phase. In co-design, the system is fully designed before actual physical
implementation, while in EOC, if all of the low level blocks are already available, the design can also be incrementally developed and tested. Incremental development is desirable in many situations. It allows one to develop a system without knowing how the end result should be. Testing a component or a module (for example, different radio types, sensors, or motor controllers) with a separate development kit is not needed, if the component or module is in Atomi format: it can be tested directly as a part of the system. In the optimal case, EOC does not require expertise in so many levels than co-design (PCB, manufacturing, etc). Furthermore, the EOC allows exploring the use of agile methods in building embedded systems.

EOC could be viewed as an extension to co-design or platform-based design, enabling a basis for higher level embedded system development through

- a modular structure or platform where the modules have standardized physical and functional interfaces, and standardized implementation related properties (i.e. Atomi II Framework),
- principles on the partitioning the system functionality into these modules (one functionality per module), and
- a mechanism for expanding the architecture (object-oriented complexity growing).
4 Modular embedded systems

This chapter shows some modular approaches that are related to embedded system development. Modular electronics and building devices from electronic blocks are not new ideas. Recently, the use of modules and modular systems has increased in electronics industry. For example, designs that are sensitive to their layout design, such as radio circuits, are commonly sold as modules. Many chip manufacturers sell their integrated circuits also as modules to enable quick prototyping of the chips. Modules and modular technologies are used in educational kits and for prototyping purposes as well. The following sections present some information and examples of these topics.

4.1 Electronic modules

Designs that are sensitive to their physical layout are often sold as modules. Circuits of this kind are for example Bluetooth modules⁸ ⁹, other radio modules such as Radiocrafts Zigbee¹⁰ (shown in Fig. 11A) or Trimble GPS¹¹, accelerometers by Analog Devices¹², or Optrex display modules¹³. The sensitivity to their physical layout makes the layout design an error prone task. Thus, it is feasible to use them as complete functional modules.

Fig. 11. A) Radiocrafts Bluetooth module. Printed with permission from Radiocrafts. B) Prototyping module for USB chip by FTDIChip. Printed with permission from Future Technology Devices International LTD.

---

⁸ Bluegiga Bluetooth modules URI: http://www.bluegiga.com/modules-1
⁹ Free2move modules URI: http://www.free2move.se/index.php?type=view&pk=1&pk_language=1
¹³ Optrex display modules URI: http://www.optrex.com/products/groupdetail.asp?g=monochrome-graphic
Most IC manufacturers sell their chips also as modules to enable quick prototyping of the chip. Such a module is shown in Fig. 11B. Often, the modules are connectible to 2.54mm raster DIP sockets, which is the most common prototyping connector. Thus the modules fit easily to prototyping boards that consist of socket matrix for inserting such modules or leads. They can also be quite easily integrated (with leads) into development kits that have an programmable MCU onboard. Such modules are for example USB modules by FTDIChip\textsuperscript{14} and Ethernet modules by WIZnet\textsuperscript{15}.

This type of electronic modules always requires a tailored PCB for use. The modules connect to the PCB with various connectors and protocols, i.e. there is no generic physical method for attaching them to PCB nor a common protocol and data format. Popular low level protocols are 4 or 8 bit parallel data, UART, SPI, and USB. The data formats used on these protocols are always proprietary.

4.2 Motherboard with extension cards

Many modular building block approaches are based on a motherboard with extension card. The motherboard has an MCU. The pins of the MCU are connected to a bus that connects to extension boards. Motherboard based solutions commonly do not contain any arbitration method as they are clear master-slave architectures. Some of them do not even have any addressing possibility. The physical architecture varies. Some architectures are stackable, some horizontally connectible, and some a mixture of both. Some examples of this type of devices are presented in the following.

PC/104 (or PC104) is an embedded computer standard controlled by the PC/104 Consortium\textsuperscript{16} which defines both a form factor and computer bus. PC/104 is intended for specialized embedded computing environments, for example extreme environments. PC104 is basically a PC in a small form. PC104 computers have a bus (ISA, PCI, or PCI-e) that can be used to expand the system with expansion boards. One PC/104 computer is shown in Fig. 12.

Tower System\textsuperscript{17} is a modular development system for designing and prototyping computational devices. Physically, the Tower consists of a primary foundation layer with a central processor. Additional circuit board layers can be stacked on top for added functionality, as a particular application requires. The Tower system is meant for modelling of system topology. (Gorton 2003)

\textsuperscript{14} FTDIChip modules URI: http://www.ftdichip.com/Products/FTEvaluationKits.htm
\textsuperscript{15} WIZnet URI: http://www.wiznet.co.kr/en/
\textsuperscript{16} PC/104 Consortium URI: http://www.pc104.org/
\textsuperscript{17} The tower: http://gig.media.mit.edu/projects/tower/
SimmSticks\textsuperscript{18} are electronic modules based on a SIMM memory card format. They can be inserted into a SIMM connector on a motherboard to create computer-type embedded systems. They are intended for hobbyists who want to build quick custom devices. SimmSticks do not contain any preprogrammed software.

Gumstix\textsuperscript{19} are Linux based motherboards with a wide range of extension boards. Their speciality is a lot of processing power with powerful MCUs on motherboards. Gumstix motherboards can be extended with one or two expansion boards.

Tibbo system\textsuperscript{20} consist of a motherboard, a separately selectable MCU, and extension boards. Tibbo is mainly targeted for data communication applications. Tibbo is programmed with BASIC language.

Microbrics\textsuperscript{21} are modules that consists of either hardware or both hardware and software. The system has a motherboard, which can be extended with modules. The modules can be connected to the motherboard and together either horizontally or vertically with screws and connectors. The Microbrics are based on BasicATOM controllers, i.e. they are programmed with BASIC language.

\textsuperscript{18} SimmSticks URI: http://www.simmstick.com/
\textsuperscript{19} Gumstix URI: http://www.gumstix.com/
\textsuperscript{20} Tibbo system URI: http://docs.tibbo.com/phm/index.html?em1000_tev.htm
\textsuperscript{21} Microbrics URI: http://www.microbric.com/
RoboBricks\textsuperscript{22} are another set of modules for robotic applications, quite similar to Microbrics. RoboBricks create master-slave-type architecture, and the blocks communicate with each other via UART.

Arduino\textsuperscript{23} are yet another set of electronic modules that can be extended with so called "shield" modules. Arduino modules can be programmed with Arduino language, which is regular C language with an extra library functions. Arduino is an open-source project. Arduino modules are quite limited in their extendibility.

Future-blox\textsuperscript{24} is a prototyping concept of Future Electronics distribution company. Their concept contains a base board that can be extended with stackable modules. The blocks uses components and modules that are distributed by Future Electronics. Future Electronics gives these boards away for free in order to quicken the development of mass products of their customers.

4.3 Enclosed interconnectable blocks

Another approach is to have enclosed interconnectable blocks of electronics. These modules often have very limited functions but they are very easy to use.

Lego Mindstorm\textsuperscript{25} is probably the most well known building blocks in this category. The Lego Mindstorm is a set of Lego blocks that contain electronics inside the blocks. The system is built around a main unit, which can be extended with actuator and sensor blocks. The Mindstorm set is targeted for building toy robots.

eBlocks\textsuperscript{26} are embedded system building blocks for useful monitor/controller systems. Their main idea is to enable very easy building of embedded systems for non-experts. They are meant for educational purposes and are quite bulky and costly compared with what can be done with them.

\begin{thebibliography}{9}
\bibitem{22} RoboBricks URI: http://www.gramlich.net/projects/robobricks/
\bibitem{23} Arduino URI: http://www.arduino.cc/
\bibitem{24} Future-blox URI: http://www.my-boardclub.com/future-blox.htm
\bibitem{25} Lego Mindstorm URI: http://mindstorms.lego.com/
\bibitem{26} eBlocks URI: http://www.cs.ucr.edu/~eblock/indicies/pubs_index.html
\end{thebibliography}
Logidules\textsuperscript{27} are small plastic boxes that snap together. The boxes contain logic circuits, microcontrollers, and other ICs. Logidules are meant for developing the test hardware of complex research projects.

Logiblocs\textsuperscript{28} are also electronic building blocks for non-experts. Logiblocs are mostly for educational and recreational purposes. Logicbloks are show in Fig. 13.

4.4 Chip and software

There are also modular approaches to embedded systems that do not consist of ready-made hardware but focus on the software. There exist a preprogrammed MCU module that can be expanded with other modules, but there is no ready hardware platform available. In these approaches, the system is usually built first around prototyping boards as the MCU module is in a DIP package format.

OOPics\textsuperscript{29} are PIC-microcontrollers with preprogrammed multitasking objects from a library of software objects, mainly for robotic applications. They use preprogrammed multitasking objects from a library to interact with the hardware. Higher level control is done by writing scripts in Basic, C, or Java syntax. During operation, the objects run continuously and simultaneously in the background, while

\textsuperscript{27}Logidules URI: http://diwww.epfl.ch/lami/teach/logidules.html
\textsuperscript{28}Logiblocs URI: http://www.logiblocs.com
\textsuperscript{29}OOPics URI: http://www.oopic.com
the scripts run in the foreground telling the objects what to do. Every aspect of the objects can be controlled by the scripts as the objects do their work with the hardware. The OOPic Object library contains objects that know how to interact with the most popular sensors and drive systems.

The BASIC Stamp is a microcontroller with a small, specialized BASIC interpreter (PBASIC) built into ROM. It is made by Parallax, Inc. and has been quite popular with electronics hobbyists since the early 1990s due to its low threshold of learning and ease of use (due to its simple BASIC language). The PBASIC language has easy-to-use commands for basic I/O, like turning devices on or off, interfacing with sensors, etc. More advanced commands let the BASIC Stamp module interface with other integrated circuits, communicate with each other, and operate in networks.

4.5 Discussion

Commonly, these modular approaches do not suit for mass production due to expensive structure and difficult automation possibility in manufacturing the devices. The structure becomes expensive (compared to a device made from individual components into an integrated board) for several reasons:

- Connectors are needed to connect the modules together (in all modules except some modules in 4.1). In integrated board, the connections go via the PCB.
- In electronic assembly, it is more expensive to manufacture several separate smaller boards than one large board.
- Connecting the separate boards together is an extra phase in the manufacturing process, costing more money. Furthermore, this connecting is often manual labour because normal assembly devices are not designed to connect together boards with connectors. To automate this, a customized assembly robot needs to be made.
- Enclosed interconnectable blocks have the unnecessary enclosure raising their costs.
- Some modules contain an interpreter for some programming language (such as the Tower System has a Logo interpreter). This reduces the efficiency of the modules processing power. Thus, some function may need a more expensive component in order to achieve similar processing power as a module that works without interpreter.

Some of the modular systems are merely educational toys, like Lego Mindstorm, eBlocks, Logidules and Logiblocs. They cannot be used for commercial applications due to their limited capabilities and costly structures.

More cost effective solutions are the chips with pre-programmed software, such as OOPics, and Basic stamp. These approaches, however, waste some processing power due to their interpreted language and they also have limitations in their functionality. Furthermore, they require PCB design in order to be used.

Complete modular approaches that are actually used in commercial products and do not require PCB design, include for example Gumstix, Tibbo, and PC/104. These are motherboard based solutions, i.e. they have a main board which can be appended with peripheral modules. They enable many real life applications and are a feasible option for low quantity products systems and research purposes, where the quick development time compensates the higher production price. To make a mass product out of them is costly, unless they suit the application as such without extra modules, and if their pricing is competitive. For example, some PC104 format computers are produced in high quantities and have a competitive price compared to their processing power and properties.

The electronic modules such as Bluetooth and GPS modules are a cost effective commercial solution even for mass products. Their problem, however, is the requirement of a PCB design in order to be used. These modules are very close to Atomi objects. They each usually perform one function, which is controlled via registers (variables). They are complete encapsulated entities having properties, events and methods. If these modules implemented the standardized physical and functional interface of Atomi objects instead of various different data protocols, they could be Atomi objects.

Table 1 shows a simplified overview of the differences between EOC and the other approaches introduced in chapter 3. The table shows only those properties that differ in the approaches. It categorizes these properties to true or false in order to give a clear picture of the overall differences, even though in many cases there are exceptions and the division is not that clear.

The main difference of the EOC and Atomi II Framework to existing modular systems is that the EOC combines the flexible modular development process with a mass producible cost effective end result. The technical differences culminate to the automatic integration of separate modules into a single PCB, and the inexpensive bus system.
Table 1. Property comparison between state-of-the-art and EOC.

<table>
<thead>
<tr>
<th>Properties \ Methods</th>
<th>Individual components</th>
<th>Electronic modules</th>
<th>Motherboards with expansion cards</th>
<th>Enclosed interconnectable blocks</th>
<th>Chip and software</th>
<th>EOC with Atomii Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OO system design</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scalable for complex devices</td>
<td>x x x x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Does not require expensive tools</td>
<td>x x x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Does not require expertise in electronics design (PCB design)</td>
<td>x x x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Does not require expertise in low level software development</td>
<td>x x x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Produces optimal solution per application</td>
<td>x x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implementation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reusable physical hardware</td>
<td></td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
</tr>
<tr>
<td>Incremental device development feasible</td>
<td></td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
</tr>
<tr>
<td>Prototyping with real hardware is quick</td>
<td></td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
</tr>
<tr>
<td>Short overall development time</td>
<td></td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
</tr>
<tr>
<td>Extending existing device is easy</td>
<td></td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
</tr>
<tr>
<td>Enables highly integrated applications</td>
<td></td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
</tr>
<tr>
<td>Extremely low power devices possible (e.g. watch)</td>
<td></td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
</tr>
<tr>
<td>Any shape of device possible</td>
<td></td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
</tr>
<tr>
<td>Commercial aspects</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Easily predictable development project (time, costs and outcome)</td>
<td></td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
</tr>
<tr>
<td>Produces commercially feasible devices in quantities &lt; 1k</td>
<td></td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
</tr>
<tr>
<td>Produces commercially feasible devices in quantities 1k - 50k</td>
<td></td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
</tr>
<tr>
<td>Produces commercially feasible devices in quantities &gt; 50k</td>
<td></td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
<td>x x x</td>
</tr>
</tbody>
</table>
5 The Embedded Object Concept (EOC)

This chapter presents the embedded object concept that has been developed in this thesis. The concept is introduced and the implementation of the concept (i.e. the Atomi II Framework) is presented. The implementation is then analyzed and discussed.

5.1 Background

The problem with the difficulty, slowness, and vulnerability for errors in the traditional electronics design process was encountered in robotic research at the University of Oulu. New electronics was being created for a telepresence robot which was under research. When new ideas were to be implemented and tested in it, new versions of an existing electronic board were always needed. Most of the board would stay the same and some new features needed to be added.

The first electronic boards anticipated the upcoming extensions by implementing extra IO pins as pin headers on the board. The pin headers enabled extensions to be added as separate boards. This approach is very common in general in many electronic boards, but it is not very versatile: either the pins are not usable for certain purpose or the software for a new feature may not fit into the MCU in existing board. In our case, this lead the thinking to extension boards with their own MCU, which implied that a communication protocol was needed for data exchange between the MCUs.

From this point on, the idea of modular building blocks of embedded systems started. At first, the concept was to create advanced Lego type of blocks that were quickly connected together with magnets. This idea mixed together with the object-oriented ideas that were used in software development. The easy-to-use aspects were, and still are, much more advanced in software development than in electronics development. A good example of an easy-to-use software development tool is Microsoft Visual C#: it utilizes modularity, component models, and object-oriented fundamentals in an intuitive way for high level graphical software development. There are ready made low level functionalities that have simple interfaces for generic use. These are, for example, ActiveX objects. The application can be developed by collecting suitable objects to the application and then adding some higher level control code to make the objects do what you want. The application can then be compiled into a regular executable and distributed normally. Creating an application with these objects takes a fraction of the time compared to what it would take if to
create everything with lower level tools by yourself. The outcome may not be utterly optimized for the resulting application, but in many cases this is not a problem. This is the kind of easy development that we wanted to achieve in embedded system development as well. To create new embedded systems, one would have ready-made objects that implement the low level functionalities in a generic way and have simple interface for generic use.

5.2 Basic principle

The basic idea of the EOC is to build new devices in a manner similar to building new toys with Lego blocks. A set of ready-made building blocks can be connected to each other to create a new device. In EOC, the building blocks are small electronic boards, called embedded objects, which interconnect with each other via a general-purpose bus. Each board has a different functionality, such as a servo control or sensor inputs. A new embedded system prototype is built by connecting together boards that have suitable functionalities for the new system, and then creating high-level control software to give the system intelligence. This software can be added either to the boards (they are programmable) or an external device (usually PC). Then, the system can be tested, and revised if necessary. Finally, a working prototype can be automatically integrated into a board consisting of single PCB, and a tester can be automatically compiled for production. This process is depicted in Fig. 14.

Fig. 14 also shows images of some implemented embedded objects, which we call Atomi objects. The embedded objects are encapsulated entities, i.e. they independently take care of their own function, and offer a generic interface for accessing their functionality. The object may contain an MCU, and they are usually programmable. Normally, they contain a low level software as such, but a user can also rewrite or add own software to them.

5.3 Architecture

Simple devices are built, as shown in Fig. 14. To give an example, one such device could be a robot motion controller. A motion controller for a three-wheeled robot base could consist of two motor controllers (i.e. DC-MotorAtomis), a power board (i.e. PowerAtomi), an input for infrared distance sensors (i.e. ADC Atomis), and a USB connection to a computer to give driving commands (i.e. USB Atomis). When these objects are connected together, the hardware for the device is ready. After writing the main control code into one of the Atomi objects, the whole device is ready.
Select Atomi objects – Each Atomi object has some basic functionality. Depending on which functionalities you need, you can pick up suitable objects. You can also - add more objects later - replace objects by another objects later - remove objects later

Advanced users can also - edit the code of the basic functionality - add own code - create own objects with a prototyping Atomi - create totally own objects (both HW & SW) with Atomi templates

Make control code and test – Most Atomi objects are programmable. You can program one or more objects and make the device do what you want. The objects can also be controlled from PC.

Generate production version Atomi objects can be integrated into an inexpensive single PCB version for mass production. Also a production tester can be automatically generated.

Assemble prototype and connect peripherals – Connect the Atomi objects together with the magnets, and attach the peripherals (e.g. sensors and actuators) to the connectors. The hardware of the device is then ready.

Fig. 14. Basic principle of using the EOC.

The concept is scalable, i.e. system can be expanded by adding more objects as required by the application. The expansion is made using the fundamental principle of object-oriented complexity growth, i.e. a hierarchy, where objects can be built out of objects. The device we just created, a robot motion controller, can be viewed as a new
larger object consisting of several basic objects. To make a more complex device, these larger objects can be combined with any other basic or larger objects.

Combining can be done with a group module (GroupAtomi), which has two or more bus interfaces. For example, there can be another device, a so-called net camera device, made of a camera object, an audio object, a net object, and a power object. The previously created robot motion controller object can be added to the net camera device with a group module, and the result is a combined robot object (see Fig. 15).

The group module exposes an object-specific interface to one bus while controlling its sub-objects with the other bus. In other words, with the group module the robot motion controller can be used similarly to any other object as a part of a device, even though it already consists of several objects. The same principle can be applied again and again: one can create objects that consist of several objects, which again consist of several objects. With this basic principle of growing complexity, the EOC can be applied to complex devices in a manageable manner.

Fig. 15. A robot device: actual PCB implementation with Atomi objects and a UML Class diagram of its structure. Reprinted with permission from IEEE press (Journal Article X).
5.4 Development process

**Fig. 16 Embedded Object-based development process.**
Fig. 5 showed the development process using Atomi objects in respect to the other ways to create an embedded system. Fig. 16 shows a more detailed development process. EO-based development process starts from requirement specification. The requirement specification captures the specifications for the application, especially the functionalities required, and the design constraints.

The design phase creates an object-oriented design of the device. There is no particular method defined for creating the EO-based design yet. Designs can be created in top-down, bottom-up, or meet-in-the-middle type methods. In the design phase, it is beneficial to keep in mind the existing object selection in order to exploit the reuse of objects as much as possible. However, if required, a new embedded object can also be made, but it requires more expertise and time than building with ready-made objects. When the architecture is known, the high level control code can be developed. UML diagrams can be used for notating the design, and Paper III presents one way of using UML with embedded objects. Currently, the UML is used just for notation, but it is possible to create UML based automation tools for the design phase, such as a tool for exploration of architectures that provide the required functionalities and meet design constraints.

Next, the system can be simulated and verified. Currently, the simulation tools have not been implemented. Then, the implementation takes place. If the whole system can be built of ready-made objects, this phase is completed in a matter of hours. If a new embedded object must be built to supplement the existing selection, it is a longer process. A new object can be built on a template or modified from an existing object (*hw & sw inheritance*). A new object requires also a new tester module, which can be similarly built on a template or modified from existing testers. There are no restrictions for tools or implementation other than the specification of an object for the used framework, such as the Atomi II Framework. To create a new embedded object, a development processes shown in chapter 3 can be used. However, the tester can be built out of existing objects with the EOC design process. In the implementation phase, each object and sub object can be tested and debugged independently before adding them to the final device. Also, a PC can be used to emulate an object in software development phase due to the universal bus interfacing. As soon as all the objects are ready and integrated the device is ready for testing.

Even though the phases of the design process are described above in a sequential manner, the development process can be iterative and incremental. The EOC enables for example one or more objects to be built complete in a separate iteration in the development process, bringing the benefits of iterative and incremental processes into the development process (see section 2.1.3 for details).
Finally, when the prototype made on discrete objects is found to be complete, the design can be converted to a mass-producible single PCB version (see section 5.5.7 and Paper IX) and a production tester can be generated based on the used objects.

5.5 Implementation – The Atomi II Framework

The basic idea of the EOC can be implemented in many different ways. The Atomi II Framework is one successfully tested implementation. It is defined by a technical specification ensuring universal physical, electrical, and functional compatibility between embedded objects. Most importantly, the framework specifies a scalable and inexpensive bus system, including arbitration, addressing, and a flow-controlled data transfer method. It also includes physical dimensions, rules for expansion, PCB specification, electrical properties, default protocols, default software structures, and more. While the specification defines a lot of things, many of them are merely suggestions for a default starting point. In fact, the framework gives a great deal of freedom for creating new objects, especially on the software side, in order to enable versatile selection of objects. The Atomi II Framework has been partly introduced in Paper VI. Below is a short overview of the main issues of the framework specification.

5.5.1 History

The basic idea of EOC resulted in the first paper and then the first generation of embedded objects in 2004 (Paper I). At first, the concept was raw and the technology not mature. The first Atomi generation was a stackable model where modules were stacked upon each other much like Lego blocks (see images in Paper I and II). The bus was a UART based serial bus, which caused a lot of processing time to be wasted on waiting and also made the MCU selections quite restricted. After testing and analyzing of the flaws in the first generation, the requirements for the second generation was defined. Based on the new requirements, new techniques were developed: arbitration, bus protocol, addressing, and physical appearance and measures. After a few iterations, these became the Atomi II Framework.

Several different Atomi objects were created according to the framework specification and used in real life tests. The technology was stable and its modifiability suited well for research projects. The cost of the resulting device was low enough for commercial devices, resulting in some commercial products using this technology.
5.5.2 Physical specification

The physical specification ensures that the Atomi objects are physically interconnectable. It defines Atomi objects as small pieces of electronic boards made on 2-layer 1.6-mm-thick PCB. The basic size of an Atomi is 48 mm × 14 mm. The 48 mm width is fixed, but the 14 mm length can be expanded if necessary. The board size is optimized according to the size of a euro PCB panel, which is often used in production. By connecting the boards in parallel, the Atomi objects form a PCB layout that can be produced as a single PCB. This enables mass production of Atomi-based devices. A bus runs through each board via bus connectors. The Atomi objects are attached together with the bus connector and strong magnets located in each corner of an Atomi. An Atomi object with its main measures is shown in Fig. 17.

![Fig. 17. An Atomi object. Printed with permission from Atomia Oy.](image)

5.5.3 Atomibus II

For physical interconnection, the framework specifies a bus called Atomibus II. The bus consists of 12 generic purpose IO lines, 2 voltage lines (3.3 V and unregulated directly from the power source) and 2 ground lines. All except one of the IO lines are shared between general purpose IO, addressing, data, or control functions. The IO lines provide analog bus connections enabling many different communication methods or protocols between objects. The number of IO lines is chosen to enable a
reasonable amount of different communication methods on the bus (such as an 8-bit parallel data transfer), but still keep the amount of lines reasonably low in order to save space on PCB. The objects do not necessarily use all the lines of the bus, but only those it needs for communication.

The bus is a hybrid of existing techniques appended with a new proprietary addressing and arbitration techniques. The arbitration technique is presented in the Paper VIII in detail. It allows any object to be connected to any other object in any order without centralized bus control. Creating such a bus-based modular system with existing bus techniques becomes expensive for commercial products, when external components need to be used for implementation. This is the case in MCU based modules. In FPGA based systems, existing arbitration and addressing methods could be implemented in the chips logic, which would not cause a problem with costs. However, FPGA is not suitable for all modules. Thus, Atomibus overcomes this problem with novel addressing and arbitration methods (including implementation), keeping the costs so low that this approach is commercially feasible. As a comparison, the arbitration and addressing method with Atomi system costs 0.26 euros total, while the next cheapest way to create arbitration and addressing, a CPLD which can implement a number of different arbitration and addressing methods, costs 0.91 euros plus side components (prices from Digikey, 1000 pcs sets). The arbitration method implements a geographic priority scheme with a possibility for dynamic priority.

The path leading to this bus selection is discussed in Papers IV and VI. Paper IV explains the problems with the serial bus that was used with the first generation Atomi objects. Some of these problems apply to all serial buses. Paper VI discusses the basic reasons for the current implementation.

5.5.4 Atomi interface

The framework specifies so called Atomi interface. It is a generic logical interface and a simple bus protocol for object intercommunication. To make it easy to use any object, the Atomi interface can be implemented into the default software or hardware of each object. The interface is based on the idea that each object can have properties, methods, and events. Properties, method triggers, and event setups are variables (or registers) inside an object, but they are presented as properties outside the object, accessible by get and set sequences via the Atomibus. Each variable represents some item of the object, for example, the value of an IO port or the speed of a DC motor. These variables are accessed by an object number, which identifies the Atomi, and an
index number, which identifies the variable. The variables of an Atomi can be expressed as a variable table, which lists all the variables in an Atomi, as shown in Table 2 presenting variables of a stepper motor controller Atomi.

In more electrical terms, each object has a bus address (i.e. an object number) and a set of registers (i.e. variables). The address is selected by the user by marking a selection into a special pattern on PCB. This pattern (the address selector) is shown in Fig. 18.

Fig. 18. Atomi address selector layout. Reprinted with permission from IEEE press (Paper IX).
Table 2. Variable table of a stepper motor Atomi.

<table>
<thead>
<tr>
<th>index</th>
<th>variable type</th>
<th>len [bytes]</th>
<th>dir</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>unsigned char [16]</td>
<td>16 (max)</td>
<td>IO</td>
<td>Name of the Atomi: &quot;Stepper_L293_1&quot;</td>
</tr>
<tr>
<td>1</td>
<td>unsigned char</td>
<td>1</td>
<td>I</td>
<td>Reset. Writing one to this variable makes a soft reset to the Atomi.</td>
</tr>
<tr>
<td>2</td>
<td>unsigned char</td>
<td>1</td>
<td>I</td>
<td>Enable. 0 = power off, 1 = power continuous on, 2 = auto-on: when motor is moving, power is applied. When target location has been achieved, power is switched off.</td>
</tr>
<tr>
<td>3</td>
<td>unsigned char</td>
<td>1</td>
<td>I</td>
<td>Driving mode. 0 = stop (default), 1 = forward, 2 = backward, 3 = drive with target coordinate</td>
</tr>
<tr>
<td>4</td>
<td>float</td>
<td>4</td>
<td>I</td>
<td>(Maximum) speed. Default value = 10 steps/second.</td>
</tr>
<tr>
<td>5</td>
<td>float</td>
<td>4</td>
<td>I</td>
<td>Acceleration value. Default value = 1 steps/second squared.</td>
</tr>
<tr>
<td>6</td>
<td>float</td>
<td>4</td>
<td>IO</td>
<td>Current position. Counts the current position of the actuator.</td>
</tr>
<tr>
<td>7</td>
<td>float</td>
<td>4</td>
<td>I</td>
<td>Target position. Sets the target position for driving in mode 3.</td>
</tr>
<tr>
<td>8</td>
<td>float</td>
<td>4</td>
<td>I</td>
<td>Relative position to drive. Adds the given position to target position and drives there.</td>
</tr>
<tr>
<td>9</td>
<td>unsigned char</td>
<td>1</td>
<td>I</td>
<td>Boundary switch 0 state. 1 = switch pressed, 0 = unpressed</td>
</tr>
<tr>
<td>10</td>
<td>unsigned char</td>
<td>1</td>
<td>I</td>
<td>Boundary switch 1 state. 1 = switch pressed, 0 = unpressed</td>
</tr>
<tr>
<td>11</td>
<td>unsigned char [8]</td>
<td>1</td>
<td>I</td>
<td>Driving sequence. If you want to make a custom sequence, you can replace the default sequence here with your own. The 4 lowest bits of each byte corresponds to A1, A2, B1 and B2.</td>
</tr>
<tr>
<td>14</td>
<td>unsigned char [2]</td>
<td>2</td>
<td>I</td>
<td>Boundary switch 0 event. If boundary switch 0 has been set, a data byte with value 0 is sent to given object number and variable index. Byte 0 specifies the target object number, and byte 1 the target variable index.</td>
</tr>
<tr>
<td>15</td>
<td>unsigned char [2]</td>
<td>2</td>
<td>I</td>
<td>Boundary switch 1 event. If boundary switch 1 has been set, a data byte with value 1 is sent to given object number and variable index. Byte 0 specifies the target object number, and byte 1 the target variable index.</td>
</tr>
</tbody>
</table>

The Atomi interface utilizes the concept of inheritance. The interface can be inherited. Each Atomi (with Atomi interface) contains the same Name of the Atomi variable in variable index 0, and reset in variable index 1. This can be considered as a base class for Atomi interface. A specialized class from this is for example the motor interface.
Each motor control Atomi has similar basic variables in indexes 2 - 8. Then, each type of motor Atomi is specialized from this interface. Thus, different motors can be driven with the same software. This makes, for example, changing a motor and also motor type from an existing device easier.

Every object does not need to implement the Atomi interface. An Atomi can also be implemented with the interface and protocol that is native to the functional chip on board. These Atomi objects are called custom interfaced Atomi objects. In these objects, the physical interfacing must still be arranged with the addressing circuitry and the IO lines of the Atomibus. The addressing circuit can usually be connected to the chip select line of a chip, and the IO lines to the communication pins of the chip. This requirement restricts the native interfaces to such interfaces that use maximum of 12 IO lines in 0–3.3 V range.

The custom interfaced Atomi objects are a valuable asset for the EOC. In very cost sensitive or speed sensitive devices, it may be feasible to implement an Atomi with the interface and protocol that is native to the functional chip on board. For example, if one is building a remote control with several buttons and an infrared receiver transmitter. One Atomi may contain the infra red (IR) function, while another could simple consist of button switches with the common key matrix circuit. The key matrix can be made available as a custom interfaced object consisting of only the address recognition circuit and the matrix. This setup would create a simple single MCU device, which is almost similar to creating the same device traditionally from individual components. The difference would be in one address recognition circuit more in the Atomi based device.

Another example could be a high speed Ethernet chip with high data transfer speed requirements. If an MCU was used to implement the Atomi interface, it would act as a protocol converter between the Ethernet chip control pins and the Atomibus. This would require a high speed MCU or FPGA in order to maintain high data transfer speed. At the same time, it would add delay to the data transfer route. In this case, the control pins of the Ethernet chip could be directly connected to the Atomibus with the address recognition circuit connected to a chip select line. This would enabled a direct control of the Ethernet chip from any Atomi object provided that each Atomi object willing to send data via Ethernet had a driver for the Ethernet chip in their firmware. This approach would not generate any excess delays to the data transfer route, but it would add more complexity to the device.

However, the lack of the generic Atomi interface reduces the independence and encapsulation of an Atomi: it will require installation of a special driver into some other Atomi in order to be accessed. This will make its portability more difficult,
since there must be a driver for each possible processor that might want to access the Atomi.

### 5.5.5 Object types

The Atomi objects can be divided into categories in two different ways. The first way, passive and active objects, were introduced in Paper V. These terms are defined for the software context by Booch et al. (2007). In EOC context, these terms are defined as follows: active object means that the object can initiate a bus communication sequence, i.e. reserve the bus and use it (usually for sending a data request). Passive objects cannot initiate the bus communication sequence, but only decode the address and enable its communication lines on address match. In addition to Booch et al.’s definition of these terms, our active object can also respond to bus communication requests by other active objects. Hardware designers might prefer terms master and slave, and system designers terms server and client. They both mean essentially the same as active and passive objects, if their definition is extended by stating that the masters/clients act also as slaves/servers.

It was shown in the development process that the definition of active and passive objects was not very meaningful. Instead, division into Atomi interfaced objects and custom interfaced objects emerged (introduced in section 5.5.4), which had more impact on designs. Using custom interfaced objects in some application seemed to lead to a situation where a certain pair of custom interfaced and an Atomi interfaced objects were always combined together as a master-slave system, and the other objects always used the custom interfaced object via the Atomi interfaced object. In practice, these two became a merged object that used the Atomibus for their mutual communication. Such custom interfaced objects were modified to Atomi interfaced objects.

Most inconvenience from custom interfaced objects was experienced in computer controlled applications, where computer accessed the objects via an USB Atomi. The USB Atomi acts as a bridge between USB interface and Atomibus. If an Atomi implements the Atomi interface, it can be plugged into an existing device without a need to program any object. Only the PC software requires changes as the PC accesses now the new Atomi as well with the unified API. If an Atomi does not implement the Atomi interface, the USB Atomi has to be reprogrammed with the driver and an interface to present the data to USB. This made the device development and modifications more difficult, as often the developers wanted to (or could) modify only the software on computer, but not Atomi objects.
On the other hand, the custom interfaced objects were a good choice for the Painmeter (introduced in section 6.2), which is a low cost low power device, and a simple single MCU system in its architecture.

5.5.6 Templates

The Atomi II Framework involves templates for hardware and software. The templates can be used as an inheritance mechanism for the Atomi objects. There is a base template which corresponds to Atomi base class. Each Atomi is derived, i.e. specialized from that class. There are more specialized templates each of which are specialized to different levels and/or functionalities. The specialized templates correspond to subclasses and their subclasses. The following sections present the templates for PCB design and software.

PCB design

Fig. 19 shows the different parts that a template for a PCB design may include. Each PCB for an Atomi must contain the items that are on the base template. These include the PCB outline, bus connectors, magnets, and the bus tracks that go through the PCB in a certain direction. A template can include also selection of the other shown components. For example, a little further specialized template contains addressing and arbitration circuitry, on top on which an MCU of a developer’s choice can be added. These templates have a pre-routed layout which can be rerouted if needed. There are also several templates that contain a pre-routed MCU on board. Current MCU selection contains AVR (Atmel 2009) and ARM (Atmel 2009d, NXP 2009, Analog Devices 2009) series MCU’s. Further specialized templates contain also functional circuitry with or without MCU. Templates with FPGA (Actel 2009, Altera 2009) are currently under development. A VHDL block containing the Atomibus functionality has been developed.
Software

MCU based Atomi objects are driven by some software. The software of an MCU based Atomi is described in Fig. 20. The HW layer contains the addressing, arbitration, and bus IO pins that the Atomi may contain. These can be controlled by the Atomibus software, functional SW, and optional custom SW. The Atomibus software contains the bus handling routines and it is controlled by the functional and custom SW. The variabletable holds the interface variables of the Atomi and can be accessed by Atomibus, functional and custom SW layers.

The software templates may contain one or more of the SW layers, depending on the corresponding hardware template. For a hardware template, there are one or more software templates available. For HW templates without MCU, there is a SW template with Atomibus Hardware Abstraction Layer (HAL), which requires the HW mapping to be made for the selected MCU. For HW templates with pre-routed MCU, there are templates consisting of readily mapped HW. For templates with functional software, the previously mentioned template is appended with a driver for the functional circuitry. In each case, of course, a less specialized SW template can be used with more specialized HW templates.
5.5.7 Single PCB option for mass products

The idea in the physical form of the Atomi objects is that a combination of interconnected objects can be manufactured as a single PCB for mass production. As the user connects together the Atomi objects to create a new device, the Atomi objects form the layout of the new device at the same time. This idea requires some techniques for making it feasible. The Paper IV discusses the single PCB subject in detail. This section describes shortly the details of this technique.

To enable feasible manufacturing, the physical dimensions of the Atomi objects have been adjusted to produce PCB sizes that are multiples of the common euro-sized PCB. The basic size of an Atomi is 48 mm × 14 mm. The 48 mm width is fixed, but the 14 mm length can be expanded, if necessary. The objects are connected with the 48 mm sides parallel with each other. The objects attach together with a bus connector and magnets. The reason for the size is that a single row of objects forms a 48-mm-wide board, which is half the height of a euro PCB (160 × 100 mm). Placing Atomi objects in one or more rows will always create board sizes that are producible on either a euro-sized PCB board or its multiples. This again optimizes board space usage for production.

In addition to the regular Atomi objects, there are additional pieces for completing the board layout. There are turn pieces, attachment pieces, adapter pieces,
connecting pieces, and board space filler pieces. By adding a turn piece, the Atomi row can form a U-turn and continue in the opposite direction, i.e. a second row of Atomi objects can be added. Two rows results in an Atomi construction that is 96 mm wide, which is a good size for production on a euro PCB. Attachment pieces contain screw holes for attaching the board into a case. Adapter pieces can jump the Atomibus from one place to another and are usually used in conjunction with group modules. Connecting pieces connect two objects together by filling in the tiny gap in the bus and grounding patterns between two objects. Board space filler pieces are used when the board has some unused space. This usually happens when the board is constructed in two or more rows. The rows often do not match exactly in length, and thus the shorter side may have some empty space. The filler piece is mainly intended for milled boards, where milling the copper away from a larger area would wear out the cutter unnecessarily.

The single PCB option is also considered in the address selector technique. Each object requires a unique address on the bus. The address selector does not require a component in either discrete Atomi objects or the single PCB version. Furthermore, it demands only minimal PCB space, which is very important for the single PCB application.

The address selector is shown in Fig. 18. It has a selector line that passes next to each data line on the bus. To select an address for the object, one of the data lines must be connected to the selector line. As the selector line passes very close to the data lines, the connection can be made with a regular pencil. This property is very useful for prototyping purposes, especially when the Atomi combination is frequently modified or when the same objects are use in different device setups. If a more firm connection than a pencil is required for the address selection, a drop of solder can be used for the connection. For the single PCB version, a real copper connection is required. For this purpose, the address selecting copper part is added to the layout when the layout of Atomi objects is combined into a single board.

5.6 Discussion

5.6.1 Modularity

As the EOC is modular, it naturally possesses the benefits of modularity. These benefits include modification and expansion of the prototype by plugging modules in and out, incremental device development (iterative develop-implement-and-test-
cycle), manageable architecture, and re-use of existing designs. The re-use of existing designs brings along the stability of the hardware and software, reliability of the device, and time savings in the design phase.

The modularity is also utilized in production testers. Each Atomi object has its own production tester modules. To construct a tester for an Atomi-based product, it can be built out of corresponding tester pieces for each Atomi. As the Atomi modules have a ready-made physical form, the tester can be built out of ready-made physical blocks. The tester development involves the following four phases: 1) selecting the Atomi objects that are included to the Atomi-based device, 2) defining their physical locations, 3) generating the production tester software and the design of the tester electronics, and 4) assembling the system together from physical building blocks. Normally, production testing for production quantities between 1k–100k (the range where Atomi objects are commercially feasible) involves a tester device such as 31. There the test system is built into a standard equipment rack, with either a PXI chassis with PXI controller, or an industrial PC as main computer. Most of the measurement instruments are PXI- or PCI-based acquisition devices or other modular instruments. Stand-alone measurement devices, like power supplies, are placed directly into the rack and can be controlled either through GPIB, Ethernet, or serial port. The device under test is connected to the tester via a test adapter or a bed of nails, which is developed for each device separately. The tests are programmed with LabVIEW 32 using a driver library for the measurement equipment, and the test results are saved into a database. Developing a tester starts from 60 k euros and 2 man-months with the example tester device. Developing the Atomi-based tester starts from 3 k euros and 1 man-week.

To achieve the physical modularity, the embedded objects need a specification for their physical dimension in order to ensure the universal compatibility. This prevents the objects to be shaped arbitrarily, which may be a problem in some cases. For example, some hand held devices may require certain hand fitting shapes, which cannot necessarily be done with the embedded objects.

5.6.2 Easy-to-use comparison

The ease of use is measured by comparing the skills that are required for each approach to create embedded systems. The less skills an approach requires, the

32 http://www.ni.com/labview/
easier it is considered to be. Table 3 shows a comparison of required skill levels between different approaches of making embedded systems. In the comparison, the term “basics of electronics” means that the developer knows electronics in a more common level than in the electronic design. For example, a developer with this knowledge can connect peripherals to digital or analog connectors correctly in order to measure some property. The comparison does not show how deeply the developer must know certain skills. For instance, the system design does not need to go so deep in electronics when using ready-made modules. Still, the comparison shows the difference in large scale between different approaches.

Table 3. Comparison of required skills in different approaches. X marks for the required skill and O marks for a skill that is not always needed.

<table>
<thead>
<tr>
<th>Skills \ Methods</th>
<th>Modular \ PC104-based electronics</th>
<th>Development kits</th>
<th>Individual components</th>
<th>Atomi with only ready-made objects</th>
<th>Atomi with ready-made and custom objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>System design</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Low level hardware related programming and/or hardware description languages</td>
<td>O</td>
<td>O</td>
<td>X</td>
<td>O</td>
<td>X</td>
</tr>
<tr>
<td>Higher level programming skills</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Basics of electronics</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Electronic design (schematics)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>PCB design (PCB layout, EMC, testability, etc.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Component sourcing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Assembling of electronics</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Production testing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Based on this comparison, the easiest option is to use modular or PC104-based electronics, development kits, or Atomi objects with ready-made objects. The most difficult ways of making electronics are the individual components and the Atomi objects with custom objects.

5.6.3 Quickness of the development

The comparison of time consumed in embedded system development is compared by the development phases that each approach involves. The division to phases
are based on a survey made by CMP media (2006). The division into detailed contents on each phase is based on (Peckol 2008), an interview with an embedded system project manager from Espotel, a Finnish embedded system R&D company.

The first phases are developing overall system specifications and conceptual design. These phases involve gathering the requirements and expectation for the end product, and producing the outline solution to the design problem. This design is taken into such a detail that it allows the cost and schedule estimations to be made. Its length is estimated to be similar to all approaches. According to (CMP media 2006), it takes approximately 28% of the project time.

The next phase is a detailed design. It involves the hardware and software design in detail. Table 4 shows details involved in this phase and their relevancy for each approach. The survey states that this stage takes 19% of overall project time. Systems based on ready-made hardware do not involve most of the HW-related tasks or low level software, making this design phase faster than the ones that do. A system built with ready-made Atomi objects and appended with custom Atomi objects involves all tasks for the custom Atomi related parts of the system, but not for the ready-made Atomi parts of the system. This makes this approach faster than building with individual components, but slower than approaches with only ready-made modules.

Table 4. Tasks in the detailed design phase of a project.

<table>
<thead>
<tr>
<th>Phase \ Approaches</th>
<th>Modular / PC104-based electronics</th>
<th>Development kits</th>
<th>Individual components</th>
<th>Atomi with only ready-made objects</th>
<th>Atomi with ready-made and custom objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>HW/SW partitioning</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Communication between partitions</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Detailed hardware design</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Detailed low level software design</td>
<td>O</td>
<td>O</td>
<td>X</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>Detailed high level software design</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Design for testability (hw)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Plan for simulation and prototyping</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Plan for unit testing</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Plan for integration testing</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Plan for production testing</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Component selection</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Schematic of the PCB</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>EMC-design</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
The simulation phase can involve simulating the system in a whole or simulating only software. Sometimes, simulation is not used at all if target hardware is already present. This depends on how the system is being developed. For systems with ready-made hardware and drivers, the simulation does not involve as many details to be simulated as with the individual component based systems, and thus takes a shorter time. According to (CMP media 2006), this stage took 9% of the overall project time.

Testing, debugging, and prototyping phases involve the implementation of the system and testing and debugging each part separately and finally together. The tasks in this phase are shown in Table 5. The length of these phases is 37% of the overall project time according to (CMP media 2006). Again, ready-made hardware does not involve all tasks related to this phase. Furthermore, creating assembly documents and doing the assembly itself is faster on modules than components, because the assembly document is less detailed with modules, and the assembly does not require special tools and materials, such as solder paste, reflow oven, or manipulator equipment.

Hardware tests and debugging is not required for ready-made modules as they are pretested by their manufacturer. Unit tests and debugging should also be a shorter task on ready-made modules, because the modules come with pretested firmware, and the unit tests are only needed for modified firmware or units that are built out of several modules.

Table 5. Tasks in prototyping, testing, and debugging phases of a project.

<table>
<thead>
<tr>
<th>Phase w Approaches</th>
<th>Modular / PC104-based electronics</th>
<th>Development kits</th>
<th>Individual components</th>
<th>Atomi with only ready-made objects</th>
<th>Atomi with ready-made and custom objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>layout design</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>assembly documents</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>component/module sourcing</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>PCB manufacturing</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>assembly</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>hardware tests and debugging</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>unit tests and debugging</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>integration test and debugging</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>EMC-testing</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
The final phase is sending to production. The tasks of this phase are shown in Table 6. The length of this phase is 6% of the overall project time according to (CMP media 2006). The modular or PC-based systems are not considered to be mass producible (see section 4.5 for the reasons). The development kit based approach assumes that the manufacturer of the kit will offer the testing services. The Atomi-based approaches generate the production tester automatically so it does not need to be developed. However, if the custom software functionality needs to be tested, some extra software needs to be developed. This makes the Atomi approach faster than the individual component approach where the tester may need to be developed manually.

Table 6. Tasks in the sending to production phase of a project. X marks for the required task and O marks for an optional task.

<table>
<thead>
<tr>
<th>Phase \ Approaches</th>
<th>Modular / PC104-based electronics</th>
<th>Development kits</th>
<th>Individual components with only ready-made objects</th>
<th>Individual components with ready-made and custom objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Production tester development</td>
<td>X</td>
<td>O</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>Production tester implementation</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Production tester tests</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Production testing and production related documentation</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

To estimate the actual time spent in each approach, we estimate the time consumption based on the number and length of tasks that an approach requires in comparison to the individual component approach. Furthermore, we estimate some time saving value to the tasks that we believe to be shorter for different approaches, as explained before. These estimates are based on our own experience, thus, they are only indicative. The actual development times also varies depending on the developers’ experience, expertise and learning curves, the amount of reuse in different design phases, complexity of the end product, and used tools. The estimated development times are shown in Fig. 21.
Developing overall system specifications and conceptual design are estimated to be similar to each project. For approaches with ready-made modules, the detailed design phase is estimated to be 50% faster, simulation phase 67% faster, testing, debugging, and prototyping 50% faster, and sending to production 92% faster, being in total 50% faster.

For Atomi-objects with both ready-made and custom objects, the detailed design phase is estimated to be 25% faster, simulation phase 33% faster, testing, debugging, and prototyping 25% faster, and sending to production 50% faster, being in total 25% faster.

---

**Fig. 21** Time spent in each phase of an embedded system development project. X-axis is percentage of the project time in a project built with individual components.
5.6.4 Costs (commercial feasibility)

The costs and commercial feasibility is discussed in Paper IX. The paper presents two test cases (three-wheel-based robot controller and force feedback joystick) that are built with both Atomi objects and from individual components. The prices of the resulting devices, including component and manufacturing costs but no tester or testing costs, are compared together. It is concluded from two test cases that in these particular cases a device built with Atomi objects is approximately 10% more expensive on its production costs than a device built with the traditional method from individual components.

On the other hand, the Atomi-based development process can save design time and costs considerably, and give other object-related benefits as well. A rough estimation according to our own experience is that to build the presented robot controller board from scratch (assuming the overall specification is made) with traditional methods would take 1 – 4 months, depending on the expertise of the designer and amount of time spent on waiting for example components and PCB from sourcing and manufacturer. With Atomi objects, this could be done in 1 – 4 weeks. Currently, however, we have not yet conducted any usability tests to validate these estimates.

In general, in terms of the overall costs of creating a product for a market, the Atomi-based system can be better with lower production quantities, whereas the individual component based system with an optimized solution is better in higher production quantities. The actual limit between ‘higher’ and ‘lower’ production quantities depends on the labour costs, time savings, and the expertise of the electronics designers. For example, if altogether 5,000 pieces of the product are sold, and the price difference between the single MCU and Atomi versions is 1.5 euros, it creates a 7.5k euros cost benefit for the single MCU version. However, if the development time with the Atomi version is, for example, one month and with the single MCU version, four months, this gives the Atomi version a cost benefit of a 3-month salary of the electronics designer, which could, for example, be around 15k euros. In this case, the production quantities where the Atomi objects can be better could be up to 10k pieces. Additionally, the tester development can save up to 50k euros in equipment costs, increasing the feasible quantity up to 43k pieces. Furthermore, with the Atomi version, the product would reach the markets three months sooner, starting the revenue for the product earlier. It is difficult to estimate the cost effects of this, but undoubtedly, it is a significant benefit for most companies.
5.6.5 Performance

The performance of the Atomi II Framework can be measured in terms of several subjects. In this section, we examine the performance in terms that we find the most meaningful, namely: Atomibus data transfer speed, processing power overhead, and power consumption.

Atomibus data transfer speed

The performance of the Atomibus data transfer speed is a twofold issue. By default, it is used with the Atomi interface protocol, which is an 8-bit parallel data transfer method with flow control, similar to Freescale M68K series MCUs (Freescale 1993). With the current MCU selection that we have tested the bus on, we have not reached a maximum limit for the data transfer speed. The fastest MCUs have been ARM7-based 80MHz MCUs with approximately 10 Mbytes/sec data transfer speeds. This may be quite close to the maximum reliable speed with this protocol.

The current implementation of the arbitration circuitry, using external LV-series logic components, limits also the maximum data transfer speed. Transmission of one data packet consists of arbitration and the actual data transfer. The maximum packet size has been limited to 127 bytes and the arbitration takes 40 ns when tuned to its fastest form (see Paper VII for tuning details). Thus, the theoretical maximum for data transfer speed with the default protocol using current arbitration implementation cannot exceed 127 bytes in 40 ns, i.e. 3.175 GB/s.

However, due to the possibility for custom interfaced objects on the Atomibus, any data transfer protocol can be used that works between 0 and 3.3 volts, requires at most 11 IO lines, and does not require constant bus connection. This kind of bus protocols are for example PCI Express with 500 Mbytes/sec speed (Intel 2007) or Serial ATA with up to 600 Mbytes/sec (SATA-IO 2008). The electronics they require are considerably more expensive than the Atomi interface electronics, which is why they have not been implemented as the default protocol for Atomi objects.

Processing power overhead

The processing power overhead is relevant in context of multitasking and inter-object communication. Compared to single MCU systems, the Atomi object system has the tasks divided into several MCU (parallel processing) where, as in single MCU system, all the processes run in one MCU (see Fig. 22). In single MCU system, the
processor time is wasted in switching between tasks, as a multitasking operating system is required. For example, a free multitasking operating system (OS) for AVR controllers, AvrX, consumes 211 clock cycles to switch between tasks (Barello 2008). In a hard real time application, the task may have to be switched in a 10 kHz frequency. With a 10 MHz clock frequency in MCU, it would consume 20% of the processing power of the MCU. However, the switching frequency can, of course, be made slower, reducing the time wasted in task switching. Situations requiring quick response from the system can in some cases be handled with interrupt functions, making a lower switching frequency feasible. This, however, is highly system dependent.

In Atomi system (with Atomi interfaced objects), there is no need for a multitasking OS since the tasks are distributed in separate modules, which each have their task specific implementation. OS can be used inside a single module, but that is usually not needed either, because the functionality of each module is very simple. Thus, no time is wasted in switching between tasks. Instead, the processing time is wasted on bus arbitration and data transfer when the modules communicate with each other. Where as a single MCU system can simply read another processes memory space (although often interface functions are used, which are not as efficient) the Atomi objects must go through a longer data transfer procedure.

We conducted tests to measure clock cycles spent on data transfers with Atmel AVR-based Atomi objects. In an Atomi based device, one module transfers data to another via AtomiSet or AtomiGet functions. The message sequence diagrams of these functions are shown in Fig. 23. We implemented the functions with C-language and the avr-gcc compiler (WinAVR 2009) and optimized them with assembly language. In an optimal situation, i.e. when Atomi is communicating with at least as fast Atomi as itself, the clock cycles are consumed as follows:

- AtomiGet takes 78 clock cycles for setting timeouts, arbitrating and addressing and 14 clock cycles per transferring a byte.
- AtomiSet takes 64 clock cycles for setting timeouts, arbitrating and addressing, and 14 clock cycles per transferring a byte.
A comparable procedure in a single MCU multitasking system would be an interface function that is used to pass data from one module or object to another. Such a function is as follows:

```c
void copyBuf(unsigned char *buf, unsigned char alen) {
    memcpy(localbuf, buf, alen);
}
```

In AVR processor, it takes 33 clock cycles for moving in and out of function, and 8 clock cycles per transferring a byte with the code above, compiled with the avr-gcc compiler. Thus, it can be seen that the Atomi interface function with pure software

---

**Fig. 22 a)** A single MCU system with multitasking OS switching between tasks and tasks communicate in physically shared memory, and **b)** An Atomi object based system with own processing unit for each task and tasks communicates with each other via Atomibus.

---

99
implementation uses approximately double the amount of clock cycles compared to a
multitasking system. If an MCU has some programmable logic (for example Atmel
2009b and Atmel 2009c), the Atomi interface can also be made as an independent
hardware implementation. In that case, the data transfer can be at the same level with
single MCU implementation.

Fig. 23 Message sequence chart of a) AtomiSet and b) AtomiGet sequence.

The time wasted in inter object communication can easily be reduced in an Atomi
system. Usually, there is no need for polling other objects through Atomibus as each
object can generate events. The object that requires polling some event in another
object can simply wait for an event message from other object, which effectively
means polling some internal variable inside local MCU. Thus, the processing time overhead is usually not an issue. However, optimizing the communication between objects is also highly system dependent. In a data transfer intensive applications such as live video feed, the Atomi system cannot optimize the data transfers much.

Comparing the processing power overhead between multitasking single MCU system and Atomi system is difficult because both overhead types depend on system architecture and can be optimized in different ways. Both have pros and cons, and there's no problem implementing applications in both ways. Significant difference does not exist.

**Power consumption**

In terms of power consumption, the basic circuitry for arbitration causes problems. If calculated according to the power consumption values in 20 degree centigrade given in component datasheets, its current consumption is 0.7 mA at its idle state. The current consumption of addressing is 10 uA. Any other circuitry in an Atomi has no Atomi related extra requirements. Thus, creating an extremely low power device, such as a wrist watch, is not reasonable with Atomi objects. However, a device that can be recharged in a certain period of time can be feasible. An example of this is the Painmeter, which is introduced in the test cases chapter.

Extremely low power devices can be made with Atomi objects if the arbitration circuit is left out of the Atomi objects. In that case, the power consumption drops to a very low level. However, leaving arbitration out causes a restriction for an Atomi-based device. The device must be made as master-slave architecture, i.e. only one object is allowed to open bus communication in order to avoid overlapping bus reservations. This option has been tested in a simple wireless sensor device, which consists of a Bluetooth, a battery pack, and AD-converter Atomi objects. The Bluetooth Atomi is the master object. Most of the time, the Atomi objects are in sleep mode consuming only 118 uA of current, measured in 20 degree temperature with a current meter. In one hour intervals, the Bluetooth Atomi polls the AD-converter Atomi for sensor reading and logs the data into its memory. Once in a week, the Bluetooth purges the data to a main computer. With this setup, the 1200 mAh battery pack should last for more than one year.
6 Test cases

Several different Atomi objects have been created according to the Atomi II Framework specification for real-life tests. The Atomi objects have been used in several projects at the University of Oulu, and they have proved to be usable. Some of the existing objects are listed in Fig. 14. In addition, there are several other communication, mixed signal, audio, video, actuator, and sensor Atomi objects.

The following test cases present how real devices can in fact be built with the proposed concept and that the developed techniques function properly in real world applications. Furthermore, the test cases demonstrate the capabilities of Atomi objects in building:

- a relatively complex device (Telepresence robot case),
- a low power and low cost device (Painmeter case), and an example of using custom interfaced objects,
- a precision measurement device utilizing a precision AD-converter Atomi, showing that precision analog electronics is possible despite of very restricted PCB layout of Atomi objects (micromanipulation platform).

6.1 Telepresence robot

The telepresence robot, or telerobot, was presented in detail in Paper V (the first version) and in Paper VII (the latest version). It is an application for being present in a remote location through a mobile robot. The system consists of a two-wheeled, child-height robot and its computer counterpart. The telerobot contains a video and audio exchange capability, and a controlling and balancing system for driving it with two wheels. It is controlled remotely via a WLAN connection by a computer counterpart, which is an ordinary PC with audio and video capabilities, an Internet connection, and robot control application software. In other words, this robot has the ability to see, hear and speak, and it can be remotely controlled from a PC. Suitable applications for such a robot could be remote guarding or remote meeting applications, for example.

The telerobot consists of several concurrently operating functions, which are all relatively complicated. This makes it a good test case for several EOC-related issues: the robot implements the Embedded Object Architecture (EOA) with the Atomi II Framework, the architecture of the robot is designed with an object-oriented design, and the robot is built using incremental device development.
6.1.1 High-level operation

The structure of the robot is shown as a UML object diagram in Fig. 24. The UML object diagram has been applied in a specific way for the EOC. The aggregation symbol shows that an object is part of another object. Peripherals attached to the Atomi objects are also depicted as objects. The stereotype property denotes the Atomi
hardware that is used for the object in question. The attributes and operations are used normally, showing the main variables and functions of each object. As it can be seen from the stereotype properties, almost all the Atomi objects are in their default form, i.e. they run with their generic default software. Only the Configurator and Movement objects are customized Atomi objects. This usually means that the default software has been supplemented with new functionalities, which is the case here, as well.

Fig. 25 Telepresence robot message sequence diagram.
As the aggregation symbols reveal, the Configurator contains the highest-level control in the system. The Configurator is basically a control software loop added to the basic software of a USB-Atomi. It is the head of the device, which consists of objects that handle data transfer, video, audio, and robot control.

The message sequence diagram of the system is shown in Fig. 25. The Configurator initializes the other Atomi objects when the robot is switched on. Then, it waits for someone to connect to the robot via the TCPIP-Atomi. After a successful connection has been established, it configures the Atomi objects to send and receive data to and from the TCP/IP connection via the TCPIP-Atomi. Each Atomi is responsible for its own data flows: the CameraAtomi sends video feed, the AudioAtomi exchanges audio feed, and the Movement object receives movement commands. The Configurator monitors the connection state at all times while the robot is being used. When a telepresence session closes, the Configurator returns the robot to idle state and starts to wait for a new connection. Since each module handles its own lower-level tasks, the high-level control does not need to be any more complex.

Fig. 26 Message sequence diagram of the movement object.
The Movement object handles the robot control. It consists of several sub-objects. The high-level software of the Movement object is implemented in a GroupAtomi, which as such does not contain any functional code other than bus management functions. In the telerobot, it contains the balancing and driving software that keeps the robot in balance and manages the translation and rotation movements of the robot. Its message sequence diagram is shown in Fig. 26. The software is a timed loop that polls the tilt sensor from the ADC-Atomi and motor encoders from the DC-MotorAtomis. The current angle, angular velocity, angular acceleration, and translation velocity are calculated from the polled values. The Movement object receives driving commands (translation and rotation) from the TCPIP-Atomi via the second Atomibus interface, which is connected to the upper object of the robot. Both the calculated values and received command values are fed into a balancing algorithm, which gives force values that are then applied to the DC Motors via the DC-MotorAtomi.

Fig. 27 shows the response time of the telerobot system. The delay from user interaction via PC and Internet to the TCP/IP-Atomi of the telerobot depends on the computer device used and the Internet connection and is thus unknown. The internal data transmission delays are from clock cycles and frequency used by the objects. The worst case scenario involves all possible other data transmissions to be sent before the actual data can be sent through, and the longest time it can take for an object to process data internally. Correspondingly, the minimum time is only the transmission time between two objects and the data processing time in the optimal case.

The driving data advances from TCPIP-Atomi to DC-Motors in 0.14–11.32 ms. The biggest factor is the 10 ms control loop interval. Camera data is sent to the TCPIP-Atomi and Internet as soon as it has been compressed to JPEG format, i.e. after grabbing the frame. Grabbing and compressing happens simultaneously with data transmission. There is a delay of approximately 67 ms before getting a complete frame. The transmission time of sending a frame (approximately 150 packets = 20 kilobytes) is 68.2 ms. This time is calculated from the utilization of the bus by all objects. Thus, the total delay for a video frame is 135.2 ms. The audio input is sent from TCPIP-Atomi to AudioAtomi in 236–1000 us, then decoded for 84 us, and finally, played via speaker for next 15 ms. Thus, the device causes a 15.3–16.1 ms delay in audio playback. Playback happens simultaneously with data transfer and decoding. Correspondingly, the audio is recorded for 15 ms, encoded 311 us, and transferred to TCPIP-Atomi 128–799 us. Recording happens
simultaneously with encoding and sending. The device creates a 15.4–16.1 ms delay in audio recording.

Fig. 27 Response times of the system: a) Driving commands in, b) Video feed out, and c) Audio feed in and out.
6.1.2 **PC software**

The telerobot system includes PC software that is used to control the robot remotely. The PC software consists of a DirectShow-based video screen and a console program for sending and receiving audio and control data via the Internet. The user interface in the PC is simply a video screen, which shows the video feed from the telerobot. The robot is controlled with the arrow keys of a PC keyboard. Fig. 24 (top row middle) shows an example of the video feed (640x480 MJPEG compressed and decompressed) received from the telerobot, and the telerobot driving in a hallway (top row right).

6.1.3 **Discussion**

The telerobot presents a relatively complex system. The functioning of telerobot demonstrated that such systems can be constructed with the Embedded Object Concept. The robot is made of objects, and the complexity growing principle is tested with the robot by creating a separate object from the balancing and motor control part of the device.

In the construction of the telerobot, the modularity was utilized. The robot was built piece by piece and upgraded several times. Different versions of it were also made by adding and removing modules.

In the telerobot, the Atomi II framework with all its properties is under a heavy test. As the data goes from TCPIP-Atomi to Audio- and MovementAtomis, and from Camera- and AudioAtomis to TCPIP-Atomi, the arbitration and addressing are in continuous operation, and the bus is reserved most of the time. Furthermore, since there are MCU’s with different clock frequencies continuously communicating with each other, the flow control method of the default transfer protocol is also being thoroughly tested. More detailed information of the telerobot is presented in Paper VII.

6.2 **Painmeter**

The Painmeter is another application made with Atomi objects. It was presented in Journal article X. The Painmeter is a part of a wireless system for pain monitoring. Patients use wireless devices to report the level of pain that they experience. The reported values are delivered in real time to a nurse’s mobile phone and stored in a server. The Painmeter is a hand-held battery-powered
device for the patient. It has six buttons with LEDs, a buzzer, a vibrator, and a short-range radio module. The Painmeter is packed in a nicely shaped enclosure. The sequence diagram of its basic operation is shown in Fig. 29. The Painmeter also has a counterpart, an access point, which forwards data between the Internet and the Painmeter. It consists of a few LEDs, a short-range radio, and a TCP-IP/Ethernet module. The sequence diagram of its basic operation is shown in Fig. 30. The enclosure and two circuit boards of the Painmeter and the circuit board of the access point are shown in Fig. 28. A UML diagram showing the structure and the Atomi objects of the Painmeter and the access point are also shown in Fig. 28.

The Painmeter is a good example that shows how the EOC and Atomi II Framework can be used in a battery-powered hand-held device. It uses some special features of the EOC, which are presented in the following section.

Fig. 28. Enclosure of the Painmeter (top left), two PCBs of the Painmeter (bottom left, first two), PCB of the access point (bottom left, third board), and the UML class diagrams of the structures of the Painmeter and access point. Reprinted with permission from IEEE press (Journal Article X) and Arto Liiti.
Both the Painmeter and its access point are intended to be commercial products, and so the resulting devices have been generated as single PCBs for mass production. This means the prototype that was made with separate Atomi objects was combined into a
single PCB and the excess connectors and other components were left out. This procedure makes mass production commercially feasible, and it is one of the key features of the EOC and Atomi II Framework.

As the Painmeter is a battery-powered device, its power consumption has been minimized. Even though the EOC is not optimal for this purpose due to the bus-based architecture, decent power consumption is still possible. The Painmeter utilizes the power-saving features of the MCU and radio components, and achieves a 30-hour lifetime before recharging with a 3 volt 50mAh lithium battery.

The Painmeter uses so-called custom interfaced objects. These are the ButtonLedAtomi, VibratorBuzzerAtomi, and BatteryAtomi. They do not have an MCU and require the controlling object to have a driver for using them. Using custom interfaced Atomi objects is not as flexible an approach for prototyping as using objects with the generic Atomi interface, but it may bring cost benefits in simple devices. The telerobot was made with Atomi-interfaced objects as it gains many benefits from the encapsulated functionality of the objects. The Painmeter, however, is very simple and intended for mass-produced products, which makes using custom interfaced objects a feasible solution.

6.3 Micromanipulation platform

The micromanipulation platform (shown in Fig. 31) is a device platform that can be used for different applications that require actuation and sensing in nanometer resolution. Presently nanoactuation devices on the market are very expensive, and often limited in applications. This platform is mostly built with off-the-shelf components and Atomi objects and thus enables a reasonable cost of the instrument.

The device is based on a generalized modular architecture which covers both device hardware and the control software in PC. The modular architecture enables a swift changing of the actuators, sensors and tools with minimal effort with reusable source code, thus being an ideal frame for various applications.

The platform consists of a haptic 3D controller, a piezoelectric actuator device, a probe, and a PC. The haptic 3D controller enables manual control over the robot for a user. The controller has a pen-like handle which can be moved with six dimensions of freedom. The pen has two buttons which can be assigned to different functions. The controller also provides a haptic feedback option that can be used to let the user feel what the robot feels.

The piezoelectric actuator device consists of three linear piezo stages with position encoders. This actuator moves the measurement. The device can be easily
changed. We have used SmarAct piezo actuator, which can move in a resolution of 5 nm in a 3 cm range.

The probe (measurement head) is connected to the actuator. It can be customized to every device separately. In our case, the probe is a silicon strain-gauge force sensor AE-801, which is attached to mechanically custom designed arm. The probe is connected to a voltage amplifier, and further to an AD-converter device made with Atomi objects. A specially designed needle is attached to the force sensor, to feel the

Fig. 31. Micromanipulation platform in use (top), a block diagram of its main parts (bottom). The sensor electronics are made with Atomi objects.
surroundings. In some applications, the probe can include an additional actuator. This actuator can be for example a micro gripper, which can also be controlled via Atomi objects. The Atomi setup of the AD-converter device is shown as a class diagram in Fig. 32. It consists of an USB-Atom, a PowerAtom, and a 24-bit precision AD-converter Atomi (Gärding 2008). The functionality of the device is shown as sequence diagram in Fig. 33. The PC initiates the AD-conversions with correct parameters, and reads a buffer of values in small intervals.

![Class diagram of the AD-converter device in micromanipulation platform.](image1)

![Sequence diagram of the AD-converter device.](image2)
6.3.1 Discussion

Some properties of the PCB layout of the Atomi objects are specified in the Atomi II Framework. Properties like the dimensions, Atomibus tracks and location on PCB, two layered PCB, and the grounding specification cause challenges for analog electronics. The AD-converter Atomi demonstrates a level that is proven to be achievable. The Atomi is based on Texas Instrument ADS1255 AD-converter (Texas Instruments 2009) that advertises up to 23 bits noise-free resolution when measuring 0–5 volt range. The Atomi was tested with two test signal sources: an ideal signal source was simulated with a 1.5 volt battery, assuming that the low output impedance of a battery eliminates all disturbances. A more realistic measurement situation was simulated by using an unshielded 50 cm flat cable with two 2.2 kOhm resistor dividers, driven by the 5 volts from the Atomi. Furthermore, the effect of the signalling on Atomibus was tested by doing the tests both with and without simulated traffic on the Atomibus. As a result, the Atomi was capable of effective resolution of 21.3 bits at 2.5 Sps rate, which is a very good result considering the difficulty level of precision analog electronics even without any restrictions. Furthermore, the signalling on Atomibus did not have any effect on the resolution. As a conclusion, the restrictions do not prevent directly from creating precision analog electronics with Atomi objects. A more difficult challenge is to find components that fits well with required routes and grounding points on the PCB space that and Atomi II Framework specification leaves for the functional circuitry. The AD-Converter Atomi and the research results are presented in detail in the diploma thesis by Antti Gärding (2008).
7 Conclusion

Building an embedded system from an idea to a product is a slow and expensive, as shown by (CMP media 2006). It also requires a lot of expertise: system design, digital and analog electronic circuits, PCB layout design, software design and languages, testing of electronics, and manufacturing. The developer has different options at his disposal, depending on the developer’s expertise, the required quantity, required price level of the final product, and the time and money to be spent for development. The device can be built on a computer, a development kit, assembled from modules, or it can be built from individual components. Generally, solutions requiring less expertise, time and money produce more expensive products.

This thesis presents the EOC (Embedded Object Concept) and the Atomi II Framework. The EOC utilizes common object-oriented methods used in software by applying them to small electronic modules, which create complete functional entities. The conceptual idea of the embedded objects is implemented with the Atomi II framework. The EOC and the Atomi II Framework combines some of the good properties of building embedded systems on modules and individual components. It makes the difficulty level of making embedded systems lower by enabling a use of ready-made modules to build systems. It enables automatic conversion of a device made from ready-made modules into an integrated PCB, lowering production costs and making the manufacturing of higher quantities of the device feasible. It enables automatic production tester generation due to its modularity. These properties lower the number of skills required for building and embedded system and quicken the path from an idea to a commercially applicable device. A developer can also build own custom modules if possesses the required expertise. This opens up a new business possibility for companies that currently provide starter kits demonstrating their chips, or have a module as their product: they could make their kits and modules as Atomi objects, enabling a faster integration of their product to a commercial device.

To enable flexible modular building block system, a framework is required. The framework needs to specify mechanical, electrical and logical boundaries, properties, and interfaces in order to ensure the interconnectivity of the modules. The framework needs to be flexible, limiting its applications as little as possible. It must enable low cost devices in order to be reasonable to use commercially. The Atomi II Framework was developed for this purpose. It combines several techniques for making the EOC a commercially feasible implementation. Most important techniques include the
inexpensive bus system (Papers VI and Paper VIII) and the single board conversion (Paper IX).

Properties of the EOC have been studied and presented in several research papers. The EOC has adopted a number of object-oriented concepts to benefit its conceptual idea of physical objects. The Atomi II framework has been established as a feasible way of building embedded systems by several test cases (Papers V, VII, X), and currently also companies that have built new products based on our technology. Both academic world and the industry have been interested in this concept. This concept has a great potential both commercially and academically, and we believe it will make its way into many commercial applications in the near future.
References


Original articles


Original publications are not included in the electronic version of the dissertation.
327. Härkönen, Janne (2009) Improving product development process through verification and validation
332. Risikko, Tanja (2009) Safety, health and productivity of cold work. A management model, implementation and effects
333. Virtanen, Markku (2009) Mathematical modelling of flow and transport as link to impacts in multidiscipline environments
337. Leononen, Jouko (2009) Analysis of OFDMA resource allocation with limited feedback
338. Tick, Timo (2009) Fabrication of advanced LTCC structures for microwave devices
341. García, Verónica (2009) Reclamation of VOCs, n-butanol and dichloromethane, from sodium chloride containing mixtures by pervaporation. Towards efficient use of resources in the chemical industry
342. Boutellier, Jani (2009) Quasi-static scheduling for fine-grained embedded multiprocesssing
Tero Vallius

AN EMBEDDED OBJECT APPROACH TO EMBEDDED SYSTEM DEVELOPMENT