Charles University in Prague

Faculty of Mathematics and Physics

**MASTER THESIS**

Dominik Škoda

# Simulink Block Library for LEGO NXT

Department of Distributed and Dependable Systems

| | |
|---|---|
| Supervisor of the master thesis: | Tomáš Bureš |
| Study programme: | Software Systems |
| Specialization: | Dependable Systems |

Prague 2014

I give a special thanks to my supervisor doc. RNDr. Tomáš Bureš, Ph.D. for his dedicated help and guidance. I also thank my parents and closest for their support that allowed me to create this work.

Název práce: Simulink Block Library for LEGO NXT

Autor: Dominik Škoda

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce: doc. RNDr. Tomáš Bureš, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Cílem této práce je vytvořit podporu platformy LEGO NXT ve vývojářském prostředí Simulink. Takováto podpora cílové platformy již existuje, ale má několik nevýhod. Především je určena výhradně pro operační systémy Windows a její implementace je uzavřená, tudíž se nedá rozšířit ani přizpůsobit. Důraz, v tomto projektu, je kladen především na podporu operačních systémů založených na Linuxu a na otevřenosti celého řešení umožňující rozšíření a přizpůsobení projektu. Modelem řízený vývoj systémů pro platformu LEGO NXT za pomoci tohoto projektu zahrnuje testování modelu pomocí simulace a generování kódu v prostředí Simulinku s využitím jeho standardních nástrojů a nasazování hotových programů na cílová zařízení. Systémy vygenerované s pomocí tohoto projektu se řadí mezi systémy reálného času.

Klíčová slova: Modelem řízený návrh, Simulink, LEGO NXT

Title: Simulink Block Library for LEGO NXT

Author: Dominik Škoda

Department: Department of Distributed and Dependable Systems

Supervisor: doc. RNDr. Tomáš Bureš, Ph.D., Department of Distributed and Dependable Systems

Abstract: The goal of this work is to create a support for the LEGO NXT platform in Simulink development environment. Such support of the target platform already exists, but it suffers from several disadvantages. At first it is provided exclusively for Windows operating systems, and the implementation is closed, therefor neither extensible nor customizable. The main premise of this project is the support of Linux operating systems. The project is also opened to ensure the extensibility and customizability. The model-driven development of systems for the LEGO NXT platform using this project comprises the model testing in a simulation and code generation in Simulink environment by using its standard tools, and deployment of completed programs to target devices. The systems generated with the help of this project are categorized as real-time systems.

Keywords: Model-driven development, Simulink, LEGO NXT

# Contents

# 1. Introduction

A model-driven development is a modern approach to design software. This method keeps the developer during the software design process on the highest abstraction level. The solution is being modeled in the problem space instead of the solution space which is on the lowest abstraction level. This approach to problem solving has evolved and was enabled over the last few years, as tools and technologies have evolved.

Another aspect of model-driven development is that the developer doesn't write code by hand. The code is automatically generated from the model. Such generated program is called prototype and is considered as a particular instance of the model. Code generation process accelerates the software development and brings some other benefits such as elimination of some common coding errors.

It is essential for model-driven development to have an environment for model creation and tools that can generate code from such created models. Since the model representation takes a form of a diagram it also eases the presentation and explanation of the solution to the problem. And it can also serve as a graphical documentation of the software behavior.

One of the environments supporting the model-driven development is Simulink [2]. It is a component of MATLAB. Simulink is tightly integrated with MATLAB. MATLAB is an interactive environment for numerical computation, visualization and programming. It has in has a vast documentation [1]. The name MATLAB is a composition whose origin is in the phrase Matrix Laboratory.

Simulink comprises a data flow graphical programming language. It allows modeling, simulating and analyzing multidomain dynamic systems. It is an environment allowing modeling block-based diagrams, simulation of the model behavior, signal processing, code generation and other tasks of the system development process.

There is an effort to teach the model-driven development approach students who are interested in the work with embedded and real-time devices. Simulink represents suitable environment for this task. It comprises the necessary functionality and allows the students to focus on the problems of the development method rather than on problems of the used tool. But still it is a tool which enables a fast development of professional systems.

However, in order to educate this subject there need to be a device for which the models are developed. Without such device students would be unable to exercise the prototype upload on the target, test it and tune the model parameters to make the device function.

The prototype performance on the target device may very differ from the model performance when simulated on computer. It may partially be due to the significantly lower hardware performance of the target device comparing to the modern computer and partially due to different target device architecture, such as lacking the ability of performing floating point arithmetic operations, and also due to the imperfections of the simulated environment that doesn't precisely express the real environmental conditions.

LEGO NXT is a low-cost but full-featured platform. It comprise a hardware configuration that was used in real products of automotive industry. In addition there are various compliant sensors that can be used when building a robot based on this platform. For this reasons it is a suitable hardware platform for teaching the model-driven development approach.

The LEGO NXT platform is supported in Simulink by an official block set and environment configuration. But this support is granted only on the Windows operating systems. It doesn't work on Linux. Another down side is that this provided support is not opened and therefor is not customizable and extensible. Already there are NXT sensors that has no block representation among the provided blocks.

The aim of this work is to devise a method how to design software using the model-driven development approach in Simulink for the LEGO NXT devices. It is desired to support and emphasize the educational aspect of the model design of real-time systems for embedded targets.

The solution should extend Simulink with seamless support of the LEGO NXT target that will work under Linux operating systems. There are the following steps that needs to be done to accomplish this goal. A set of blocks for Simulink must be created. These blocks will represent needed features of the LEGO NXT brick, sensors and actuators. There must be provided a System Target File (STF) which configures the Simulink environment for the LEGO NXT target. Such configured environment will support simulation and code generation tailored to the LEGO NXT platform. The code generation process needs to be supported via various scripts. And finally there must be supplied a program for software upload to the LEGO NXT brick that will work on Linux operating systems.

# 2. Model-Driven Development

In the latest sense of the conceptual modeling, a model is anything used in any way to represent anything else. In our perspective the model refers to the algorithms and equations used to capture the behavior of the modeled system. The model should be independent of the implementation concerns, for example, concurrency or data storage. The model becomes a stable basis for subsequent development of applications in the domain. The concepts of the model can be mapped into physical design or implementation constructs using either manual or automated code generation approaches. As systems have become increasingly complex, the role of modeling has dramatically expanded. It is reflected by the increasing effectiveness of capturing the system fundamentals in models.

A model-driven development is an approach to design a software which keeps the programmer on a higher level of abstraction during the software development process in the contrast to traditional code writing. The model-driven development approach is meant to increase the productivity by maximizing the compatibility between systems. One model may be used as a subsystem in another model which exploits its functionality. This allows the formation of standard models with universally needed and well defined behavior. It is the same concept as in the most programming languages are the libraries of functions and data structures.

The model-driven development simplifies the process of design and promotes the communication between individuals and teams working on the system. The modeling paradigm is considered effective if its models make sense from the point of view of the user who is familiar with the domain, and if they can serve as a basis for implementing systems.

The model describes a solution to the problem in the problem space. It enables the developer to think in concepts rather than in statements of some programming language. This method enables the developer to design the model of desired software. The model then serves as a template from which the source code of the software system is generated. A program compiled from the generated source code is an instance of the model and is called the prototype. A nice study about the model-driven development approach is done in [8].

Our interest is focused on model-driven development of embedded and real-time systems. An embedded system is a computer system with a dedicated function within a larger mechanical or electrical system. Embedded systems often go hand by hand with real-time computing constraints. Embedded systems contrast a general-purpose computers. An embedded device may interact and influence the environment where it is used. Information about the environment are acquired via various sensors. The gathered information is processed by the controller and the environment can be influenced by the device via actuators. A diagram of this general concept is in figure 2.1. As said before embedded systems are often subjected to real-time computing. A system is said to be real-time if the correctness of its function depends not only on its logical correctness, but also on the time in which it is performed. These systems must guarantee a response within strict time constraints. Real-time systems, as well as their deadlines, are classified by the consequence of missing the deadline.

- *Hard* – missing a deadline is a total system failure.

- *Firm* – infrequent deadline misses are tolerable, but may degrade the systems quality of service. The usefulness of a result is zero after its deadline.

- *Soft* – the usefulness of a result degrades after the deadline thereby degrading the systems quality of service.

Real-time systems are often governed by a real-time operating system that is responsible for planning the available tasks to run on the processor. Tasks may be periodical or triggered by an event. Tasks have an assigned priority which helps the real-time operating system to decide which task should be planned next. It needs to ensure that each deadline will be met regardless of the system load.

The development of embedded and real-time systems goes through well defined phases specific to the model-driven development. At first the problem needs to be specified and well described. If the problem depends on any input, e.g. from the environment, the representative sample of these data needs to be gathered and analyzed. This is important for the later adjustment of the model. It is also desirable to have the model of the environment. This model will be used in the simulation and test phase of the development to interact with the model of the system. After these steps the model of the desired system itself can be designed.

The model itself represents the behavior description of the designed system. The behavior of the model can be simulated on the computer and debugged before the prototype, generated from the model, is uploaded into the target device. It makes the development phase easier and faster.

After the design phase there follow test phases. There are many test techniques designed to ensure high quality of the final system. Most of the following procedures are used in the context of the development of control systems. It means software that interact with a mechanic system. Typically these are referred to as the *controller* and the *plant* respectively.

Model in the Loop (MIL) is a test where the model of the controller is simulated in the model of the plant (environment). The simulation runs entirely on the computer. Extremely fast development occurs at this stage as changes can be made to the model of the controller and the system can be immediately tested.

System in the Loop (SIL) is a test of the slightly more real controller. The model of the controller is transformed into a program (typically in C or C++). The program is compiled into an instance of the model (for the computer architecture), and inserted back into the overall plant simulation. This is basically the test of the coding system. The coding may be done either using code generation or hand writing the source code. At this stage the design iteration slows down slightly but coding failures become evident.

Processor in the Loop (PIL) is a test where the controller system no longer runs on the computer but is deployed to the target microprocessor. While the controller is running on the target microprocessor the Input/Output (IO) to the plant is connected to the plant simulation on the computer using some high speed bus. This test is designed to expose problems with execution in the embedded environment. Design iteration now slows noticeably since each change leads to recoding and deploying the system. This test exposes execution issues on the embedded processor.

Hardware in the Loop (HIL) tests the fully installed controller. The control system interacts with the plant through the proper IO of the controller. The plant is running on a real-time computer with IO simulations to make the controller believe that it is installed on the real plant. HIL is often used only for the software validation rather than development since the design iteration is very slow at this point.

There may be other test techniques but the above are the main that are commonly used. All these test techniques are wonderfully explained in [9]. After each test the design iteration involves the correction of found errors. Sometimes parameter tuning solves the exposed problem, sometimes there is a need to slightly alter the model. In extreme cases the test proves the model completely wrong and puts the design process to its start.

When the model is ready it can be used as a template for code generation. Code generation is an automated process of creating files with source code in some common programming language, typically C. C is particularly suited for this purpose because of its constructs efficiently map on typical machine instructions and a low-level hardware features can be easily exploited by C constructs.

The code generation has some advantages in the contrast to the code writing. The programmer can focus more on the model behavior instead of the code structure. It also eliminates some errors made by coding mistakes, such as using a wrong variable, forgetting data initialization and cleanup, etc ...

Typically the model can be created in a graphical form using some development environment dedicated to this purpose. The representation may use various notations. Some representatives of such notations are:

- Unified Modeling Language (UML) [11]
- Object-Role Modeling (OMR) [12]
- Object-Modeling Technique (OMT) [13]
- Integration Definition for Information modeling (IDEF1X) [14]
- Entity-Relation model (ER model) [15]
- Data Flow Diagram (DFD) [16]
- and more...

The idea of the model-driven development of embedded and real-time systems stands on the premise that the behavior of the target device depends directly on the input data from sensors. It is handy to model the device behavior with a data driven model, because it corresponds to the paradigm of embedded systems. An embedded system interacts with the environment using sensors and actuators. The behavior of the embedded system is provided by an controller (algorithm), that reads data from the sensors and based on the data drives the actuators. Data flow diagrams are well suited for the model design for embedded devices. A diagram of the interaction of an embedded system can be seen in figure 2.1.

Simulink is an development environment that uses Data Flow Diagrams to represent models. An example of such diagram is shown in figure 2.2. The diagram represents a model of discrete derivative. Each block in the model represents some functionality. The blocks are interconnected by lines that represent the data

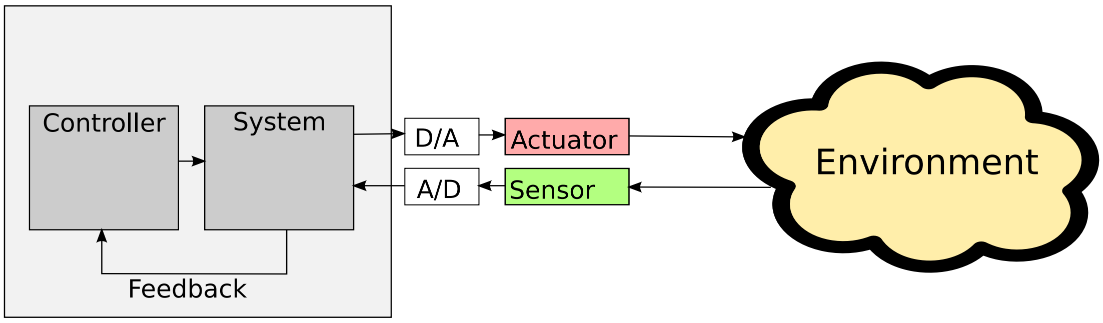


Figure 2.1: Embedded System Interaction

signals. A signal carries either a scalar value or a vector of values. The values change with the time.

The block `U` represents the input signal. The input signal represents some time-dependent function. The block `Y` represents the output signal, it carries the time-dependent value of the first derivative of the input signal `U`.

The block `TSamp` transforms the signal by dividing it by weighted sample time. It ensures correct computation of the derivative when the time steps are not uniform.

The block `UD` stands for unit delay. It remembers the input signal value in its inner state and outputs the signal with the previously stored value. In another words this block delays the signal for one time step.

The block `Diff` subtracts the value of the delayed signal from the value of the current signal. The result produced by this block represent the difference between the signal in this time step and the signal in the previous time step.

The last block `Data Type Duplicate` forces all input signals to have the same data type. It ensures that the output signal of this model has the same data type as the input signal has.

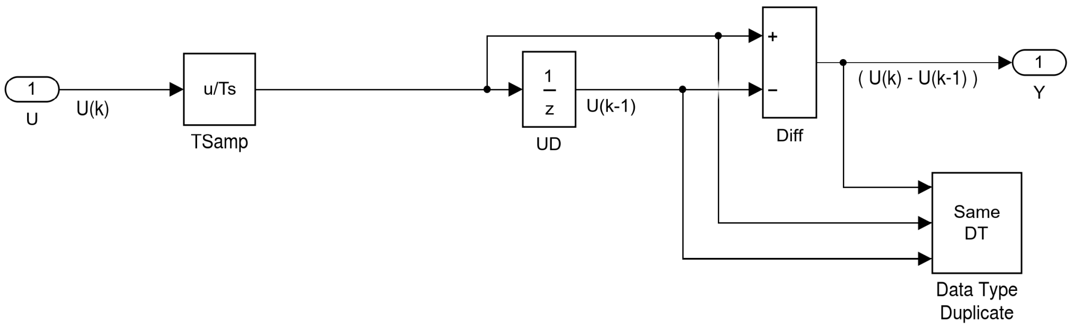


Figure 2.2: Discrete Derivative Model

Models such as the model in figure 2.2 can be encapsulated into a subsystem block. Then the whole functionality defined by the model is represented by a single block and can be easily used in another model. In fact this is exactly the case of the model of discrete derivative in figure 2.2. It is one of the blocks from the set of standard blocks supplied in Simulink block library.

## 2.1 Model-Driven Development in Simulink

As already said above there is an environment for model-driven development called Simulink in MATLAB. Simulink is focused on the model-driven development of embedded and real-time systems.

The model in Simulink is represented using data flow diagrams. It is a graphical representation of the elements of the system with their interconnections. The data flow diagram describes the system from the data flow point of view. Algorithms in the system are driven by the incoming data.

A model is there represented by a block diagram. Each block accounts for a physical component (e.g. sensor), or a simple functionality (e.g. constant value) or a subset of blocks which are gathered into a single block to provide a complex functionality. A block can have some inputs and outputs. In the model the blocks are connected output to input by links representing signals. An example of a simple model is in figure 2.3. The model consists of three blocks connected together. The first block `Sine Wave` generates the sine wave. It only has one output. The value of the output is time dependent and can be configured via various options. The value from the first block is transported to the next block via a data signal represented by the arrow connecting these blocks. The next block named `Gain` multiplies the value carried by the signal from the first block by a constant, in this case by the number two. Such modified signal is then passed to the last block in the diagram. The `Scope` block gathers all values carried by the signal during the simulation of the model and after then it can show a graph plotted from the gathered values with respect to the time when each value arrived. This block is very useful for visual evaluation of the simulation.

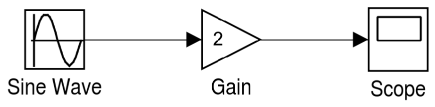
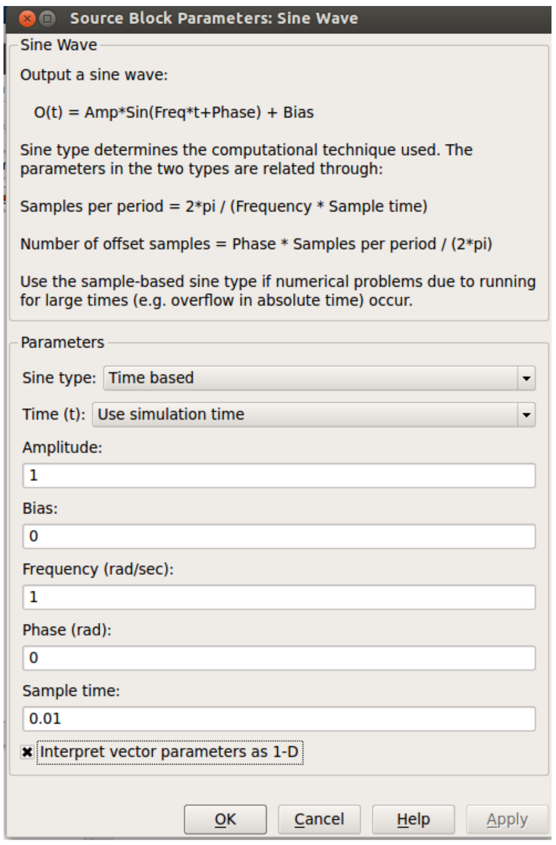


Figure 2.3: Sine Times Two

A block can have a number of associated parameters. These parameters can be comfortably set after double clicking the block with the left mouse button. Parameters of the `Sine Wave` block are illustrated in figure 2.4. There is also detail description of the block along with its parameters.

Another representation that can be used in Simulink is so called Stateflow diagram. Stateflow was developed by MathWorks and it is a control logic tool used to model reactive systems as state machines using a flow chart within Simulink model. Stateflow uses a variant of the finite-state machine notation, enabling the representation of hierarchy, parallelism and history within a state chart. Stateflow also provides state transition tables and truth tables. Stateflow is generally used to specify the discrete controller in the model. An example of the Stateflow diagram is in figure 2.5. The model contains two main states: `PowerOn` and `PowerOff`. These two states are exclusive, meaning the system can be in one of these states at a time. The exclusivity is represented by the full border line of the states. The `PowerOn` state is subdivided into three parallel sub-states. System is in all these states when it is in the state `PowerOn`. The parallel states are represented by the dashed border lines. The states `Fan1` and `Fan2` can be either in the


Source Block Parameters: Sine Wave

Sine Wave

Output a sine wave:

O(t) = Amp*Sin(Freq*t+Phase) + Bias

Sine type determines the computational technique used. The parameters in the two types are related through:

Samples per period = 2*pi / (Frequency * Sample time)

Number of offset samples = Phase * Samples per period / (2*pi)

Use the sample-based sine type if numerical problems due to running for large times (e.g. overflow in absolute time) occur.

Parameters

Sine type: Time based

Time (t): Use simulation time

Amplitude: 1

Bias: 0

Frequency (rad/sec): 1

Phase (rad): 0

Sample time: 0.01

[✓] Interpret vector parameters as 1-D

Figure 2.4: Sine Wave Parameters

state `On` or `Off`. The arrows between the states represent transitions. Each transition may be driven either by an event (`SWITCH`) or by an condition (`temp>=120`). In the diagram there are defined actions that are performed when the system is in the corresponding states. In the state `PowerOff` there is an assignment to a variable which happens only once when the system transits into this state (`entry`). When the system is in the state `FansOn` there is also an assignment to a variable but the assignment is performed at each time step when the system is at that state (`during`).

When the model is simulated it is the MIL test. During the simulation the `Scope` block records all the signal values it obtains. After the simulation the course of the signal can be studied. After double clicking the `Scope` block with the left mouse button the graph of the signal function is displayed. The graph is shown in figure 2.6. The `X` axis represents the time of the simulation. The `Y` axis represents the value of the signal.

The simulation of the model behavior in Simulink is possible thanks to the existence of solvers. Solvers are an important part of Simulink. A solver is an engine that runs the simulation of created model. The simulation runs under MATLAB environment and serves to test and evaluate the model. There is a great number of solvers in Simulink. The main difference between them is the principle how they evaluate the model. There are discrete and continuous solvers. Both sets of solvers rely on the model blocks to compute the values of any discrete state. Blocks that defines discrete states are responsible for computing the values of those states at each time step. However, unlike discrete solvers, continuous

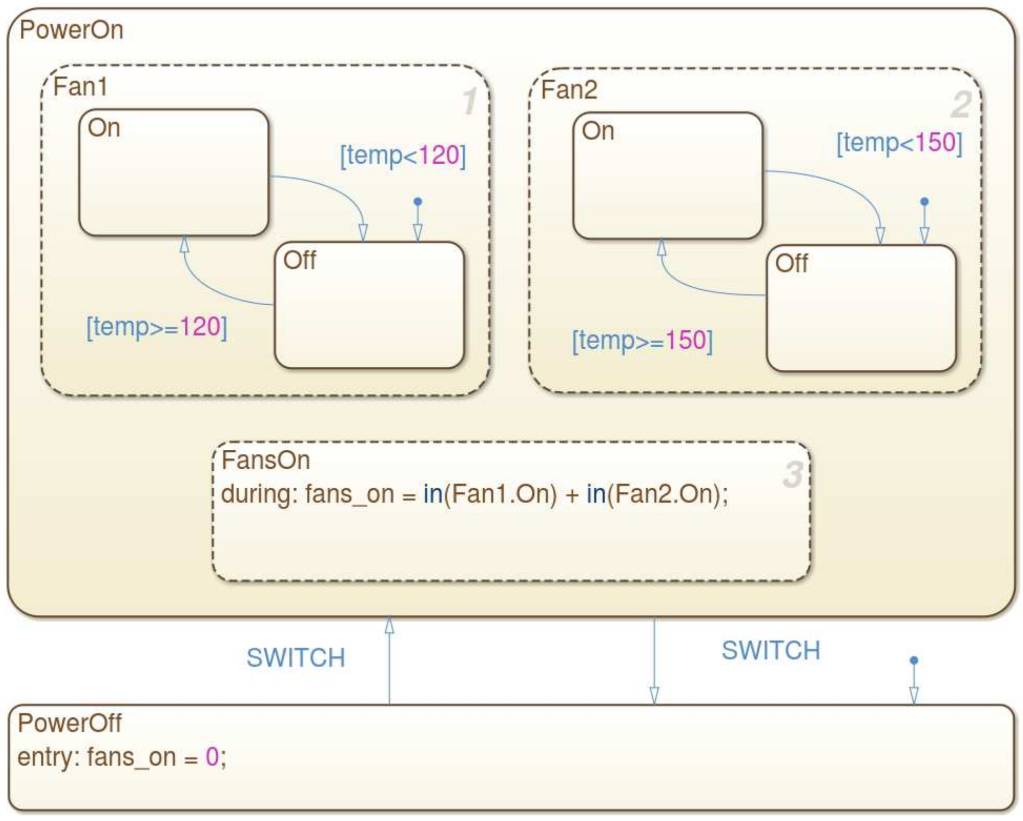


Figure 2.5: Stateflow Diagram

solvers use numerical integration to compute the continuous states defined by the blocks. These sets of solvers can be either fixed-step or variable-step, depending on the step characteristic of the simulation. Discrete solvers are suitable for digital systems, continuous solvers are suitable for analog ones.

As the name implies a fixed-step solver uses fixed time steps between the computation of the states of the model. In opposite to that a variable-step solver uses time steps with variable length between the computation of the states of the model. The variable-step solver increases or reduces the size of steps to meet the specified error tolerance.

There are many solvers in Simulink. They cover a range of methods that can be used to solve the model. For numerical integration there are the following methods: Euler's method, Heun's method, Bogacki-Shampine formula and others. Other methods include Numerical Differentiation Formulas, Jacobian method, etc ...

Simulink also supports the code generation. The component that handles this task is called a coder. The coder is used for code generation from the model into a prototype. Two main coders in Simulink are the Simulink Coder and the Embedded Coder. The Simulink Coder is the default coder included in the Simulink. The Embedded Coder [4] is provided separately but still it is an official component of Simulink. These two coders offers the automated generation of C and C++ source code. The Embedded coder generates compact code that is efficient enough for the use with an embedded device. A code generated from

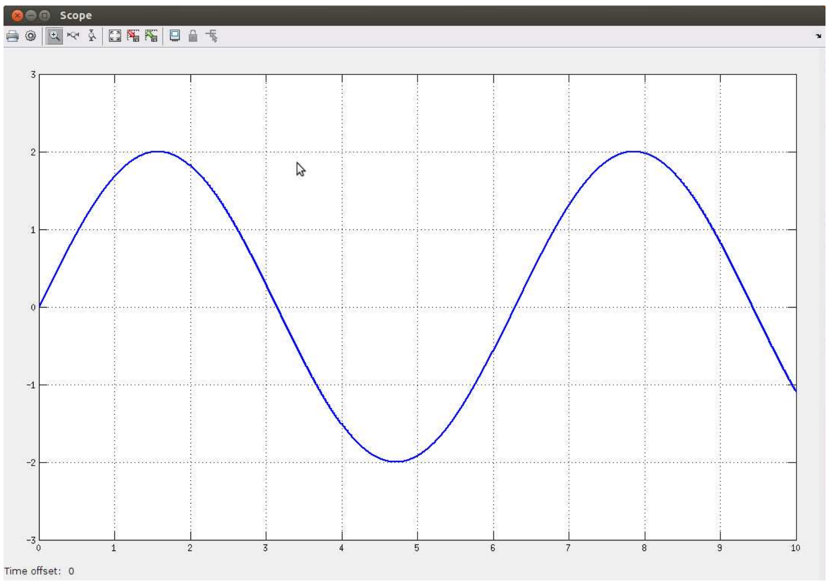


Figure 2.6: Sine Wave Scope

the model can be compiled into a binary program and uploaded into the target device and run.

If the code generation is intended for the embedded device, it is typical that the compilation needs to use a compiler toolchain to provide the binary code for the target platform. The code generation process then needs to be configured for the target platform. The Simulink Coder and the Embedded Coder supports various platform standards. Beside the generic real-time target there is the Automotive Open System Architecture (AUTOSAR) target, the Tornado (VxWorks) target and other.

# 3. NXT OSEK a framework for Embedded Coder

The LEGO NXT is a programmable robotics kit. The base kit is provided in two versions: the retail version and the education base set. The main component of the platform is a computer with the Advanced RISC Machines (ARM)[1] architecture, it is called the NXT intelligent brick. It can read signals from up to four sensors and drive up to three servo motors. The brick has an embedded monochrome Liquid-Crystal Display (LCD) and four buttons. The connectivity is ensured by a bluetooth and Universal Serial Bus (USB) (type B). There is also a speaker. The brick can run on the power from batteries inside. The embedded computer has the following hardware configuration: 32-bit ARM7TDMI-core Atmel AT91SAM7S256 microcontroller with 256KB of flash memory and 64KB of Random Access Memory (RAM), plus an 8-bit Atmel AVR ATmega48 microcontroller.

LEGO has released the firmware for the NXT intelligent brick as an open source, along with schematics of all hardware components. This gave the rise to alternative, unofficial firmwares that will be discussed later.

LEGO, the vendor of the platform, provides the associated programming environment NXT-G. It is a simple drag and drop graphical environment, that defines actions that can be put into a sequence to form the program. A simple programs can be created using this environment. The process of program creation is very fast and simple. LEGO provides this NXT-G environment only for the Windows and OS X operating systems. The main drawback of the NXT-G is that the programs created using this environment are not real-time. On the other hand the development process in this environment is model-driven since the developer creates a model of the program.

Over the time there was formed a great number of unofficial ways to program the NXT intelligent brick. Microsoft Robotics Development Studio [17] enables to program the NXT intelligent brick using the C# programming language. Bricx Command Center (BricxCC) enables programming it in the Next Byte Codes (NBC), Not Quite C (NQC) and Not eXactly C (NXC) programming languages. ROBOTC [18] is another programming language based on C. It was created for the software development for VEX Robotics Design System, but there was added the support for the development for the LEGO NXT. Most of the approaches above does not support the model-driven development and some of them are again available only for Windows. Simulink supports the model-driven approach, but the official MATLAB support of development for LEGO NXT is also provided only for Windows.

The nxtOSEK [5] is a project that implements the OSEK[2] standard for use with the LEGO NXT programmable robotics platform. It includes the TOPPERS/JSP real-time operating system for the ARM7 (ATMEL AT91SAM7S256), and the C/C++ Application Programming Interface (API) for the LEGO NXT

[1]Reduced Instruction Set Computing (RISC)

[2]Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen;
English: Open Systems and their Interfaces for the Electronics in Motor Vehicles

sensors, motors, and other devices. The nxtOSEK forms a framework for a development of software for the LEGO NXT devices. It provides a library of methods that ensures the communication with various LEGO NXT sensors and methods that allow manipulation with LEGO NXT actuators. Using the nxtOSEK the NXT brick can be programmed in a C and C++ programming languages. The final program is compiled together with an real-time operating system, this means that the NXT robot running such program will be compliant to real-time computing constraints. Since the development using the nxtOSEK requires C or C++ it allows the developer to use a low-level hardware features. This is considered as another benefit of the nxtOSEK. This development approach meets the requirements for real-time, but does not satisfy the modeling approach.

OSEK is a standard specification, described in [6], for an embedded operating system, communication stack, and network management protocol for embedded systems in automotive industry. OSEK was designed to provide a standard software architecture for the various electronic control units throughout a car. It is an open standard founded by the automotive industry.

A part of the OSEK standard is the OSEK Implementation Language (OIL) specification [7]. It is a concept for the description of the OSEK real-time systems, capable of multitasking and communication, which can be used for motor vehicles. The aim of OIL is to provide a mechanism to configure an OSEK application. In an OIL file there can be specified the Central Processing Unit (CPU), operating system, tasks, events, alarms, resources and others. All these together form the real-time application for an embedded target device.

## 3.1 Firmwares for LEGO NXT

The official LEGO NXT firmware can be used to upload and run only programs created in the official NXT-G environment. This forced the formation of unofficial firmwares that support programs created by other means. For example there is a firmware for the ROBOTC. This firmware is highly optimized and it allows the NXT brick to run programs very quickly. Like other NXT languages, ROBOTC requires this firmware to be downloaded into the NXT brick in order to run the programs created using the ROBOTC development tool.

The leJOS NXJ is another firmware. It includes Java virtual machine, which allows the NXT brick to be programmed in the Java programming language.

For the use with the nxtOSEK there are two firmwares. The first one is the Enhanced NXT firmware. It allows an upload of multiple nxtOSEK programs, but the program size is limited to 64 kB. The second one is the nxtOSEK NXT BIOS. There can be uploaded only a single nxtOSEK program using this firmware, but it can take up to 224 kB.

## 3.2 nxtOSEK application

An example of simple application for LEGO NXT brick using the nxtOSEK framework is discussed in this section.

The application written using the nxtOSEK framework needs only three files to be created. A main program file written in C or C++ programming language.

From now on, we will focus only on source files written in C. The main program file refer to the nxtOSEK files and exploits its functionality. The next file that is needed is the OIL file containing configuration of the application. And the last file is the Makefile which defines the build process. It must include source files and tools from the nxtOSEK. The build process uses the GNU ARM toolchain to cross compile the sources to the target ARM platform. The simplest application that will be described below is the hello world program, which is is also included as an example in the nxtOSEK project. The hello world project has the directory structure shown in figure 3.1.

```
example
|__nxtOSEK
|__helloworld
   |__helloworld.c
   |__helloworld.oil
   |__Makefile
```

Figure 3.1: Hello World Project Structure

The nxtOSEK directory is included in the hello world project structure to illustrate the need to include its files and to use its tools during the compilation.

```c
/* helloworld.c for TOPPERS/ATK(OSEK) */
#include "kernel.h"
#include "kernel_id.h"
#include "ecrobot_interface.h"

/* nxtOSEK hook to be invoked from an ISR in category 2 */
void user_1ms_isr_type2(void){ /* do nothing */ }

TASK(OSEK_Task_Background)
{
    while(1)
    {
        ecrobot_status_monitor("OSEK HelloWorld!");
        systick_wait_ms(500); /* 500msec wait */
    }
}
```

The main program file has some specific constructs required by the nxtOSEK framework. Above there is the source code from the helloworld.c file. The first part of the program file contains includes of files from the nxtOSEK. The `kernel.h` contains definitions and macros that are used to define Tasks, Events, Counters, etc. The `ecrobot_interface.h` file defines the functions that implements the functionality of the LEGO NXT features, sensors and actuators. Next there is a function `user_1ms_isr_type2`. It is a hook routine that is invoked from the nxtOSEK real-time operating system TOPPERS/JSP that is in conformity with the $\mu$ITRON4.0[3] specification [10]. This method is invoked every 1 ms from the interrupt service routine and typically there are counters, defined for

[3]The Real-time Operating system Nucleus (TRON)

the program in the OIL file, being incremented in this method. And finally there is the definition of task that specifies the program behavior. In this case the task contains infinite loop in which the hello world message is displayed on the NXT brick display. There is a half a second wait between refreshing the message.

The program written in the `helloworld.c` file needs to be configured for the nxtOSEK real-time operating system. There is a task used, which needs to be defined in the operating system in order to be handled as a task. This configuration is defined in the `helloworld.oil` file. The content of this file is shown bellow. The syntax of OIL files is defined by the OSEK Implementation Language Specification [7]. The OIL file contains definitions of OIL objects and their attributes. The CPU OIL object is the container for all the other OIL objects. It implies that for each CPU there is one OIL description. In the example below there is specified an operating system, application mode and the task. All these together will form the final system.

```
#include "implementation.oil"

CPU ATMEL_AT91SAM7S256
{
  OS LEJOS_OSEK
  {
    STATUS = EXTENDED;
    STARTUPHOOK = FALSE;
    ERRORHOOK = FALSE;
    SHUTDOWNHOOK = FALSE;
    PRETASKHOOK = FALSE;
    POSTTASKHOOK = FALSE;
    USEGETSERVICEID = FALSE;
    USEPARAMETERACCESS = FALSE;
    USERESSCHEDULER = FALSE;
  };

  /* Definition of application mode */
  APPMODE appmode1{};

  /* Definition of OSEK_Task_Background */
  TASK OSEK_Task_Background
  {
    AUTOSTART = TRUE
    {
      APPMODE = appmode1;
    };
    PRIORITY = 1; /* lowest priority */
    ACTIVATION = 1;
    SCHEDULE = FULL;
    STACKSIZE = 512;
  };
};
```

In this example there is no counter nor an alarm since the hello world task is not periodical. The task is planned once and it runs forever. If there was a periodical task it would be planned by the operating system in the period defined by the corresponding alarm in the OIL file, and the alarm would depend on some counter, which would be increased in the `user_1ms_isr_type2` hook routine.

The last file needed in the project is Makefile. The nxtOSEK project comes with a makefile hierarchy. There is defined the whole build procedure, using the GNU ARM compiler toolchain. In the makefile for the hello world project, there only needs to be defined the name of the final program, the list of source files outside the nxtOSEK and the OIL file. Finally there is included the makefile of the nxtOSEK that defines the whole build procedure. The content of the makefile of the hello world program follows.

```
# Target specific macros
TARGET = helloworld_OSEK
TARGET_SOURCES = \
    helloworld.c
TOPPERS_OSEK_OIL_SOURCE = ./helloworld.oil

# Don't modify below part
O_PATH ?= build
include ../nxtOSEK/ecrobot/ecrobot.mak
```

After compilation there is created a single binary file that can be uploaded into the NXT brick. The binary contains the real-time operating system together with the hello world task.

In order to use the nxtOSEK as an underlying framework for model-driven development there is the need to generate at least these three files: the main C program, the OIL description of the program and the makefile.

# 4. Related Work

In this chapter we will focus on other existing approaches. There exists two Simulink extensions that support the model-driven development of embedded and real-time systems exploiting the functionality of the development environment of Simulink. Both these solutions are targeted on the Windows operating systems, as well as is the official LEGO development environment that will be mentioned.

In this project we want to deliver the ability of software development for the LEGO NXT target platform in the Simulink development environment under Linux operating systems. We want to support NXT sensors and actuators that are used for the education at our university. The supported abilities of the development process must include the model simulation and code generation. This functionality should be seamlessly integrated into Simulink.

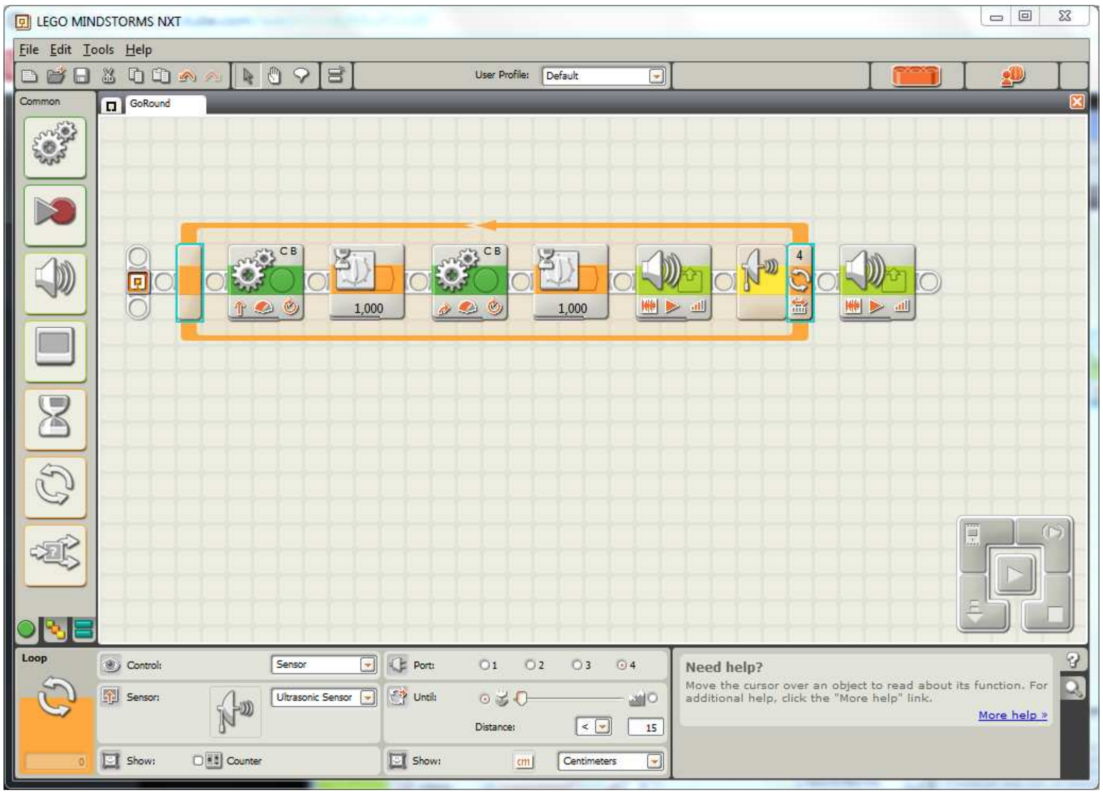
## 4.1 LEGO NXT-G

The official LEGO NXT-G programming environment allows the creation of programs for the NXT brick. The program is created by sequencing the blocks. The blocks don't represent sensors and actuators, but they represent actions. For example: move, wait, play a sound and so on. There are also blocks representing classical programming constructs such as if-else, switch-case, mathematical operations, writing and reading a variable, and so on. Creation of a basic programs is very simple and fast. Thanks to the presence of advance programming constructs, there can be developed more complex programs. The more complex the program is the less readable the diagram becomes. Figure 4.1 shows the NXT-G programming environment. The environment is simple, the left side panel contains all the blocks that can be used to create the model. The main area serves as a canvas where the model is created, using drag and drop method of placing the blocks. The bottom panel contains all the properties of currently selected block from the canvas. There is a group of buttons in the bottom right corner. These buttons can be used to download the program to the robot, to run or to stop the program in the robot.

The example in figure 4.1 represents a program for a robot with two motors and ultrasonic sensor. The behavior of the robot is programmed to do this sequence of actions: go strait, wait, turn right, wait, play sound. This sequence is repeated until the ultrasonic sensor detects an obstacle before the robot. The final action after the cycle is broken is playing different sound.

The NXT-G environment supports the model-driven development approach. The interface is rather simple and the created models are not represented in a standardize modeling language. One of the handicaps is also the absence of the support of Linux. Programs created using the NXT-G environment doesn't comply to the real-time computation constraints. There is no underlying real-time operating system that would manage the tasks of the program. The NXT-G environment doesn't support the simulation of the behavior of created models.

The ROBOTC environment supports the LEGO NXT platform and can interoperate with the NXT-G environment. It can emulate the NXT brick, into which the program from NXT-G environment can be downloaded. The downloaded



Loop

Control: Sensor

Sensor: Ultrasonic Sensor

Show: Counter

Port: 1 2 3 (✓ 4)

Until: Distance: < 15

Show: cm Centimeters

Need help?

Move the cursor over an object to read about its function. For additional help, click the "More help" link.

More help »

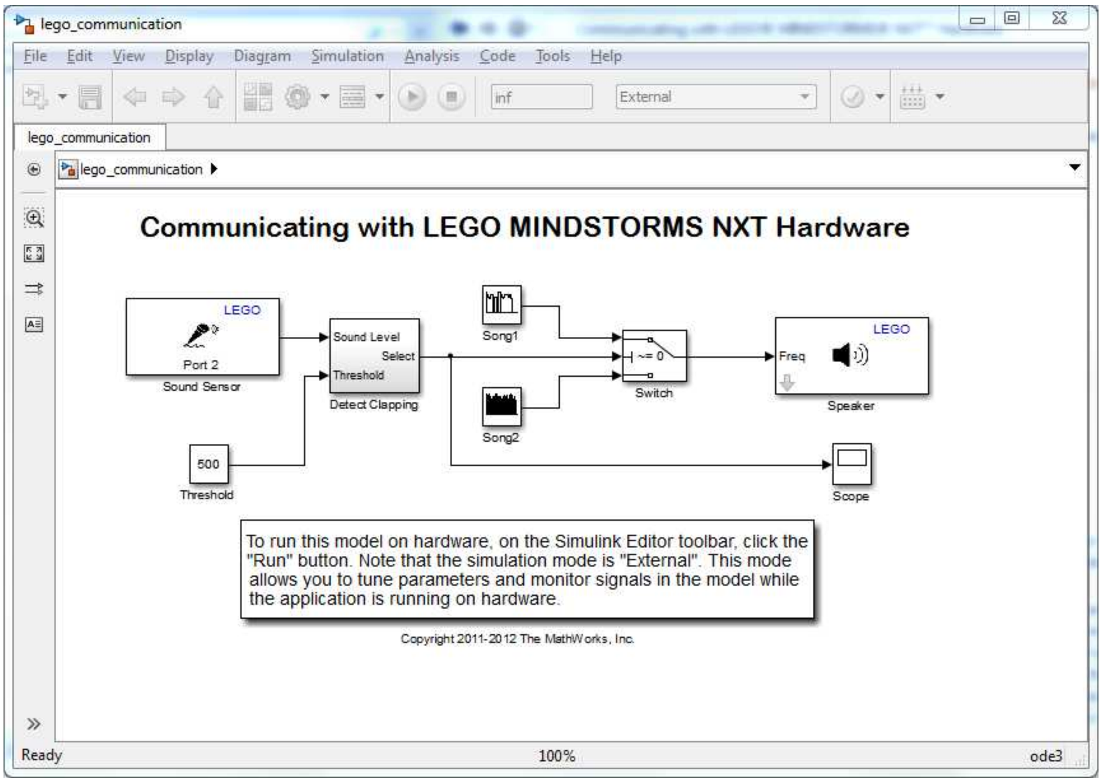
Figure 4.1: NXT-G Programming Environment

program then can be simulated in the ROBOTC environment.

## 4.2 Official Simulink support

There is an official Simulink support for developing models for LEGO NXT platform. After installing the package for LEGO NXT into Simulink, there appears a new set of blocks representing LEGO NXT sensors and actuators. Also there becomes available a new configuration of the overall development process for the NXT target. This configuration affects simulation and code generation. In figure 4.2 there is one of the example models from the LEGO NXT block library. The model drives the NXT robot to play two different sounds. It switches between them as the robot detects a laud sound (a clap).

The Simulink support of the NXT platform is provided only for Windows operating systems. It uses the nxtOSEK framework for the compilation of the generated code. This ensures the real-time requirements on the created systems. The provided solution is not open source, there are almost no source files, it is distributed mostly in the binary form. In addition when using the Simulink configuration for the NXT there cannot be kept the generated source files for further analysis or modifications. The created model can only be compiled, the source code files created in the intermediate step are deleted without any option to preserve them. Already there are sensors that are not included in the provided blocks. It becomes a problem especially for the HiTechnic color sensor version 2, which is used for the education at our university.



To run this model on hardware, on the Simulink Editor toolbar, click the "Run" button. Note that the simulation mode is "External". This mode allows you to tune parameters and monitor signals in the model while the application is running on hardware.

Copyright 2011-2012 The MathWorks, Inc.

Figure 4.2: LEGO NXT Official Simulilnk Support

## 4.3 Villanova University LEGO Real Time Target

Villanova University LEGO Real Time Target (VU-LRT) is an unofficial support for model-driven development of LEGO NXT brick in Simulink. It was created by Prof. James Peyton-Jones of Villanova University. This project was introduced for MATLAB version R2010a before MathWorks came up with the official support of the LEGO NXT target described above. This solution was targeted to work under Windows operating systems (32-bit version). Since there is the official support of the LEGO NXT target under Windows operating systems, this project was abandoned. From the beginning it was distributed as an open source. There was the support for model simulation and code generation. With the newer versions of MATLAB the VU-LRT project become less reliable.

The VU-LRT project also relies on the nxtOSEK framework. The compiled models are therefor could be proper real-time systems. For the upload of the binary into the NXT brick there is used a Fantom Driver, the official driver from LEGO which is compliant only with the Windows operating systems. The development process using the VU-LRT is the same as the development process using the official MathWork support. Figure 4.3 shows the blocks implemented in the project.

The handicap of the VU-LRT project is the way how an OIL file is generated. For all models the OIL file looks the same. It contains definition of only one task, that represents the behavior of the whole model. The task is not invoked by the real-time operating system in intervals but only once and then it runs until the

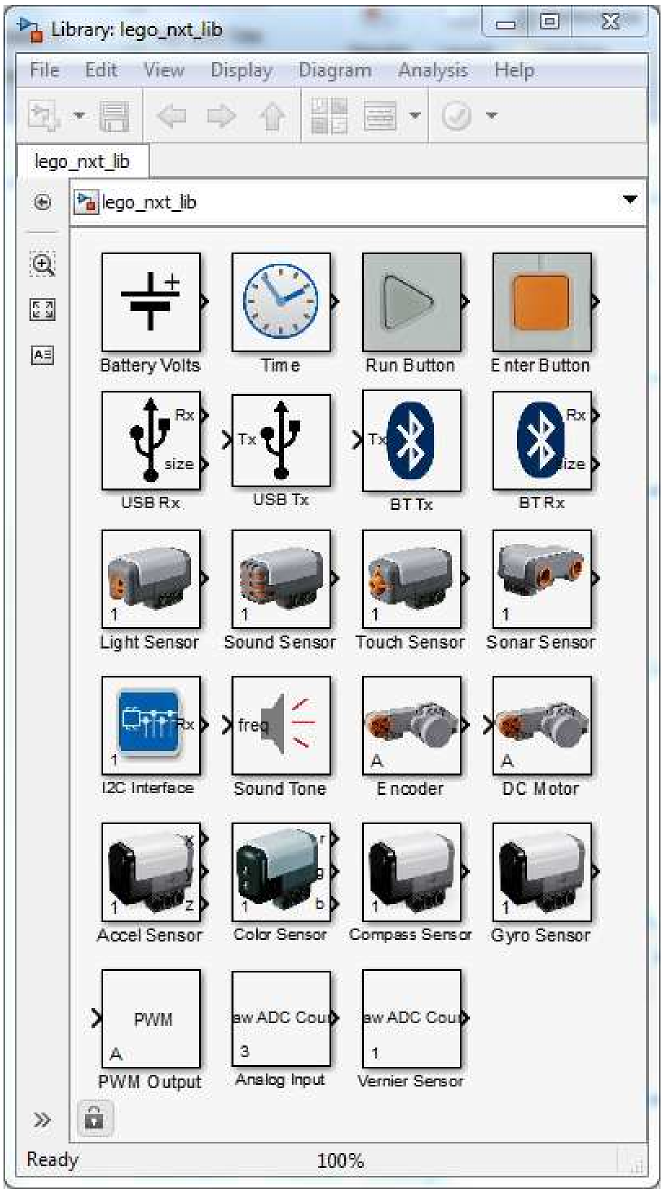


Figure 4.3: VU-LRT Block Library

NXT brick is turned off. If the model contains blocks that have a different period of sample time the generated prototype is multitasking. This fact is emulated in the main source file, which contains the definition of the only task. The program task cycles without any lags, meaning that the sample times configured in the model are ignored. This solution ties the hands of developers. During the education, students can't observe system runtime performance dependence on the sampling period of individual tasks. This approach also doesn't exploit the possibilities that are offered by the OIL configuration.

# 5. Model-Driven Development for Lego NXT

The model-driven development methods are independent of the system they are used for. Models are much less bound to the underlying implementation technology. However the development environment used for modeling must support the target platform. Since there exist a wide field of different hardware platforms and systems the producers of the development environments can't provide the support for all of these. This situation is therefor handled by different means. The development environment may be designed to be easily extended by its users to support systems they require. This is also the case of Simulink and it was the main prerequisite for this project.

Simulink generally contains a great number of blocks that can be used to create models. In order to support the LEGO NXT target there has to be supplied a set of blocks that represent sensors and actuators specific to this platform. Together with these new blocks it is possible do design models that exploits the functionality of the desired hardware. But the development process only begins with the construction of the model. There are many steps that separates the first model draft from the final working system deployed in the target device.

As described in chapter 2 the model can be and should be tested. There is described a number of test approaches that are specific to a different stages of the model-driven development process. Simulink by default offers the simulation of the model in the host computer, which is basically the MIL test. In addition Simulink supports the two standard approaches for verifying software: model-to-software verification and runtime error detection in the source code.

Model-to-software verification techniques like SIL and PIL testing can be used to confirm that the behavior of the software matches the behavior of the model. The Simulink Design Verifier brings the formal analysis methods and automates the generation of SIL and PIL tests from the model.

From these possibilities of testing the designed system this project implements the support for the simulation of the model in the simulated environment on the host computer CPU. This MIL test takes place at the early stage of model design and it represents an empirical approach to verification. It is critical to be able to do this on highly abstract and incomplete models that arise early in the development cycle, because this is when software designers make most of the fundamental design decisions. Using this testing method promises the fastest development iterations and helps discover the main design errors, mistakes and false assumptions. For educational purposes this testing has the higher value.

There are two sets of blocks provided in this project to ensure the support of model simulation. The relevant blocks come in two variants. One to design the model of the system and one to design the environment for the model necessary for the simulation. These two block variants are internally connected in such a way, that the input of one block becomes the output of the other. There is no need to explicitly connect these blocks in the model.

The next important step in the model-driven development process is automated code generation. It is essential to support the complete, rather than partial or just structural (skeletal), code generation to avoid the model becoming merely documentation of the system. Such models would have limited value since the documentation too easily diverges from reality.

We can attain the full benefits of model-driven development only if we fully exploit it potential for automation. The complete code generation simply means that the generated code is ready to be compiled into a binary without any need for adjustments and manual editing. For the code generation there are dedicated tools called *Coders* in Simulink. These tools are capable to generate complete source code from the model. The performance and effectiveness of the generated code depends on the used coder. Since our aim is to enable designing embedded, real-time systems for the LEGO NXT hardware platform we decided to use the *Embedded Coder* which is the most suitable for this purpose.

Because the code generation process is already dependent on the target system this project needs to provide the necessary support for the LEGO NXT hardware platform. The Embedded Coder produces code with the well defined interface. This project supplies scripts, that adjust the code generation for our target. It adds the generation of a *main* file within which are the tasks of the model called. There is also defined the generation of the OIL configuration file which is required due to the use of the nxtOSEK as an framework for the generated code.

Our concerns in this project aim also the issue of real-time systems. Embedded systems are most often also real-time systems. The model-driven development techniques are well suited for the design of real-time system, however, real-time systems require special attention during the code generation phase. The source code generated by the Embedded Coder defines tasks that are intended to be scheduled and executed in a real-time system, but they can be called within a simple program in a loop without any real-time awareness. The Real-Time Operating System (RTOS) that would host these tasks must be provided explicitly. One of the reasons why is the architecture dependence of operating systems. A detailed look at the subject of real-time operating systems is in [19].

For the reasons below we decided to rely on the nxtOSEK project. It comprises an real-time operating system TOPPERS/JSP. This system conforms to the $\mu$ITRON4.0 real-time kernel specification. It is de-facto industry standard in the embedded systems field. The $\mu$ITRON4.0 Specification was designed by the Kernel Specification Working Group under the $\mu$ITRON4.0 Specification Study Group. The TOPPERS/JSP is designed to be easily ported to different target processor architectures, and the LEGO NXT hardware platform is supported in the version of the TOPPERS/JSP included in the nxtOSEK project. This kernel excels in performance efficiency and low RAM usage, it is very important aspect since the hardware resources of the LEGO NXT platform are limited.

The nxtOSEK project also includes support for the OSEK standard. Using this support the target system can be configured in an OIL file. This file defines the main components of the system for the CPU of the target device. There is specified the used RTOS, resources, events, tasks, counters and alarms. The RTOS manages and governs all the components of the whole system. Tasks define the work to be performed. Resources are used to delimit critical sections in the program. Events can be used to trigger an action. Counters and alarms define periods within which to execute specified tasks.

All the above make the nxtOSEK very useful and powerful framework for developing real-time embedded systems for the LEGO NXT devices. In addition the nxtOSEK also provides the C and C++ API for LEGO NXT sensors, actuators and other devices. This API can be easily used in the source code generated from the Simulink blocks representing the NXT hardware, and this project takes the full advantage of this provided functionality.

In order to exploit some of the configuration abilities of the OIL file it was decided to constraint the structure of models that are targeted to the LEGO NXT platform. Simulink provides the parameter *Sample time* for many of its blocks. This parameter defines the period and optionally a time offset specifying the points in time when the block is simulated (executed). The value of this parameter is exactly the one needed to specify the period and time offset of tasks in the final system. To avoid different sample times for each block in the model and to ensure task compactness, the structure of the model at the top level is strictly dictated. All the top level blocks in the model are restricted to be `Subsystem` blocks. Semantically these blocks represent the tasks in the final system. There must be specified a valid sample time and optionally the time offset for these blocks. The rest of the model must be placed inside these top level subsystem block and all the blocks must inherit their sample time.

Such a constraint to the model structure is beneficial for one more reason. When students are designing the model, they have to keep in mind the real-time aspects of the system and they can visually express the tasks in the created system, and at the same time they have the way to configure the timing for the tasks.

The final step of the development process covered in this project is the program deployment, in this case an upload to the LEGO NXT brick. There is an program for software upload in the nxtOSEK project, but it works only on Windows. Therefor there was written a new program that handles this task on Linux. This program is called *appflash* and uses the USB to communicate with the LEGO NXT brick. The *appflash* program relies on the unofficial BIOS installed on the LEGO NXT brick. This, nxtOSEK NXT BIOS, is also a part of the nxtOSEK project, but it can be flashed on the LEGO NXT brick only in Windows.

The next chapter is devoted to the structure of this project, describing all the parts that are needed to enable Simulink to support the new target platform.

# 6. Project Structure

To support a custom target in Simulink there has to be developed a library of the target specific blocks. There must be specified the build process fitting the target. The Simulink model simulation engine needs to be configured with various parameters in such a way that the simulation corresponds to the target capabilities and behavior.

This project is composed of parts devoted to different function or role in the development process. These parts are separated into a directory structure described below. The root of the structure is named `siblilen` it is an abbreviation of the project name: **Si**mulink **B**lock **Li**brary for **LE**GO **N**XT. The directory structure of the project is shown in figure 6.1.

```
siblilen
├── doc
│   ├── src
│   ├── LEGO_NXT_Library.pdf
│   └── Tutorial.pdf
├── install
│   ├── +nxt_blocks
│   └── nxtOSEK_patch
│       └── altered_files
├── samples
├── scripts
├── src
│   ├── blocks
│   └── nxt_comm
├── nxt_remove.m
├── nxt_settings.m
└── nxt_setup.m
```

Figure 6.1: Project Structure

The `doc` directory contains documentation of this project, the `LEGO_NXT_Library.pdf` file, and the tutorial describing how to create the first model using the LEGO NXT support of this project in Simulink, the `Tutorial.pdf` file.

There are installation scripts in the form of MATLAB functions in the `install` directory. The installation process involves compilation of blocks, copying scripts, downloading additional tools and files. All these activities are separated into own MATLAB script file. In the `install` directory there are two other directories: `+nxt_blocks` and `nxtOSEK_patch`. The first one contains MATLAB functions for programmatically masking the blocks from this project. The second one contains patch files of the nxtOSEK that needs to be applied in order to support compilation of programs using the nxtOSEK under Linux operating systems. The included patches also add the support of the *HiTechnic Color Sensor version 2* into the nxtOSEK. The subdirectory `altered_files` includes all the altered nxtOSEK files that have the patch file in the super directory, these files are included only for the purpose of reviewing these files.

The `samples` directory contains the example Simulink models that are analyzed in section 6.3. These models illustrate the capabilities of this project.

The `scripts` directory contains a variety of scripts needed for the code generation process and for the configuration of Simulink for the new target.

Finally there is the `src` directory that contains source files of the blocks in the `blocks` subdirectory, and source files of the program utility *appflash* in the `nxt_comm` subdirectory. The upload utility is written in C programming language and handles the upload of programs into the LEGO NXT brick under Linux operating systems. The source code of a block is divided into two files: a C source code file defines the block behavior during the simulation of the model in Simulink, and a Target Language Compiler file defines the code generated during the code generation process that represents the block in the model.

The installation of this project is automated in a MATLAB script nxt_setup.m. It includes the download and installation of the GNU ARM compiler toolchain, download and patch of the nxtOSEK, compilation of the appflash utility, creation of the set of blocks from source files and their insert into Simulinks library of blocks. The installed project has the structure illustrated in figure 6.2. The installation process can be configured in the `nxt_settings.m` file, where is a section devoted to a user settings. The `nxt_remove.m` script contains the functionality to uninstall this project. How to use these scripts is described in the documentation file `LEGO_NXT_Library.pdf`.

```
NXT_LEGO_Library
 |__ blocks
 |__ documentation
 |__ gnuarm
 |__ nxtOSEK
 |__ samples
 |__ scripts
 |__ nxtLego.slx
 |__ slblocks.m
```

Figure 6.2: Installed Project Structure

During the installation there is a replacement of references to GNU ARM toolchain in the nxtOSEK and a replacement of references to nxtOSEK in the Template Makefile (TMF). These references are updated with the current path of the installed solution.

The `nxtLEGO.slx` file is the Simulink block library containing all the compiled and masked blocks from this project. The `slblocks.m` file is a MATLAB script, that registers the library in the Simulink Library Browser. MATLAB automatically loads the `slblocks.m` file when the Library Browser os opened for the first time since the MATLAB startup. Therefor the `slblocks.m` file must be placed on the MATLAB search path before the Library Browser is opened.

## 6.1 Parts of the Simulink Block Library for Lego NXT

The main component which Simulink needs to be supplied with in order to support a new target is a System Target File. This file contains the overall settings of Simulink environment for the target, and describes the code generation process. In this project there is the `nxt_ert.tlc` a System Target File for the LEGO NXT target. Code generation process specified by this file uses the Embedded Coder.

There is also a template of a makefile the `nxt_ert_default.tmf` file. During the code generation phase there is a makefile being generated from this template. Such generated makefile is used afterwards to compile the program. Except the generated sources there are included sources from the nxtOSEK framework and sources from MATLAB that are referenced within the generated files. The makefile imports other partial makefiles from the nxtOSEK framework where the GNU ARM cross compiler is specified. During the compilation there is also being invoked the parser of OIL files. This parser scans the OIL description of the project being compiled and composes the final program whose structure is defined in the OIL file. The makefile generated from the `nxt_ert_default.tmf` file is named `<model_name>.mk`.

The next file supplied in this project is the `nxt_main.tlc`. This file serves as template for generation of the program main file. In such main file there are definitions of individual tasks for the real-time operating system. From these tasks are called the task methods generated from the model. The file generated from the `nxt_main.tlc` file is named `<model_name>_main.c`.

The last template file which is part of this project is the template of an OIL file the `nxt_OIL.tlc`. The generation of the OIL file reflects the number of tasks specified by the model. For each task defined by the model there is a task defined in the generated OIL file. For each task there is a counter and alarm which is responsible for the periodical task invocation. The period and time offset of each task is also defined by the model. The OIL file generated from the `nxt_OIL.tlc` is named `<model_name>.oil`.

## 6.2 Implemented Blocks

There is a block library provided by MathWorks with Simulink with a variety of blocks such as mathematical operations, signal generating blocks, signal monitoring blocks and other signal processing blocks. These blocks are ready to use for a model design.

To create a block there must be written a C or MATLAB file that defines the behavior of the block during the simulation. Next there is a need for Target Language Compiler (TLC) file that defines the code generated for the block during the code generation process. These files utilize the nxtOSEK function library. The block is created by compiling the simulation code into a MATLAB Executable (MEX) file and adding it into a library. Thus created blocks can be afterwards masked for a more comfortable use. These two files (MEX file and .tlc file) together are needed in the library of blocks.

The blocks implemented in this project are separated into two groups. One group of blocks is used to create the model of desired system, while the other group is used to model the environment to simulate the model of the system. These groups are named `Model` and `Environment`. Each block from the `Environment` group is interconnected with the corresponding block from the `Model` group. The connection of these blocks is implicit, meaning that it is not represented in the model. The blocks from the `Environment` group are used only for the simulation of the model, they do not participate in the code generation process. The list of implemented blocks together with their description is placed below.

### 6.2.1 Model

The `Motor` block showed in figure 6.3 represents the LEGO NXT servo motor. It has one input signal that drives the speed and direction of the motor revolutions. The direction of the motor revolutions is defined by the sign of the signal value, the speed is defined in percentage. Therefor the signal value ranges from -100 to 100. In the parameters of the block there can be specified the port to which the motor is connected and the break mode. The break mode is either *break* or *coast*. Is the *break* value is selected the motor stops immediately when the input signal is 0. If the *coast* value is selected the motor fluently slows down and stops when the input signal is 0.

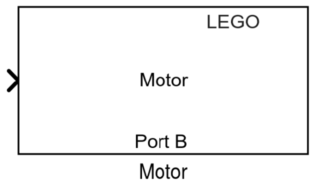


Figure 6.3: Servo Motor

The `Encoder` block showed in figure 6.4 represents the LEGO NXT servo motor encoder. The block has one output signal that carries the servo motor revolution count in degrees. The only parameter of this block is the port to which the motor is connected.

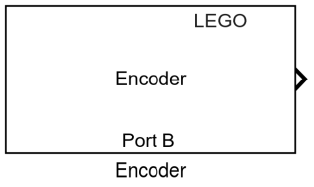


Figure 6.4: Encoder

The `Display` block is shown in figure 6.5. It represents the LCD display on the NXT brick. This blocks displays the value of the input signal alongside with some label on one line of the NXT brick LCD display. The type of the input signal has to be integer. The block has three parameters: *label*, *line* and *number format*. The *label* parameter specifies the label of the value to be displayed. The label is limited to eight letters. The *line* parameter specifies the number of the line on the NXT brick LCD display on which the labeled signal value will be displayed. There are eight lines on the display to be chosen from. The last parameter is the *number format*. It specifies the format of displayed values. It can be either *decimal* or *hexadecimal*.

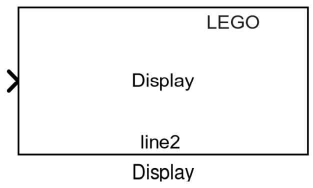


Figure 6.5: Display

Figure 6.6 illustrates the `Color Sensor` block. This block represents the LEGO NXT Color Sensor. This sensor can operate in different modes. These modes can be specified in the block parameters. There are two parameters that can be set to this block: *port* and *mode*. The *mode* parameter specifies the operation mode of the sensor. It can be selected from the following set:

- **ColorSensor** – the sensor provides Red, Green, Blue (RGB) components of measured color.
- **LightSensor_RED** – the sensor measures the color only in the red spectrum.
- **LightSensor_GREEN** – the sensor measures the color only in the green spectrum.
- **LightSensor_BLUE** – the sensor measures the color only in the blue spectrum.
- **LightSensor_WHITE** – the sensor measures the color in the white spectrum.
- **LightSensor_NONE** – the sensor doesn't use the illumination diode.
- **ColorSensor_DEACTIVATE** – the sensor is deactivated.

The sensor uses a RGB Light-Emitting Diode (LED) diode to illuminate the area that is scanned by the sensor. The color of the light produced by the diode depends on the selected mode. The second parameter of this block is the port into which the sensor is connected. If the mode of the block is *ColorSensor* then the block has three output signals. Each signal carries one color component (red, green and blue). In any other mode the block has only one output signal that carries the value of the color component defined by the mode.

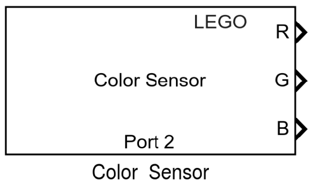


Figure 6.6: Color Sensor

There is a HiTechnic Color Sensor represented by the block in figure 6.7. This block has three output signals, each for one of the color component (red, green and blue). The HiTechnic Color Sensor brings better color recognition abilities compared to the LEGO NXT Color Sensor. The `HiTech Color Sensor` has one parameter specifying the port to which the sensor is attached.

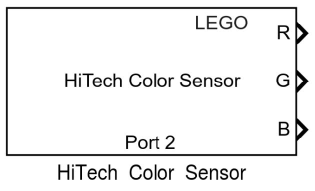


Figure 6.7: HiTechnic Color Sensor

There is also a HiTechnic Color Sensor version 2 provided by LEGO. It improves the performance and color range of the previous version. The block that represents this sensor is in figure 6.8. As the previous block this block has three output signals representing the red, green and blue color components. There is also the single parameter specifying the port into which the sensor is connected.

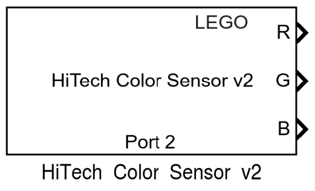


Figure 6.8: HiTechnic Color Sensor v2

### 6.2.2 Environment

The set of blocks in the `Environment` group is used to model the simulation environment of the model of the created system. Each block from this group is a complement block to the block of the same name from the `Model` group of blocks.

The `Motor` block in figure 6.9 represents the LEGO NXT motor in the simulation environment. It has two output signals: *speed* and *break*. The *speed* signal carries the value which is on the input of the `Motor` block from the `Model` group. The *break* signal is either 0 or 1, depending on the value of the parameter in the equivalent block from the `Model` group of blocks. This block has one parameter specifying the port of the motor. The value of this parameter must correspond to the value of the same parameter of the `Motor` block from the `Model` group used in the model.

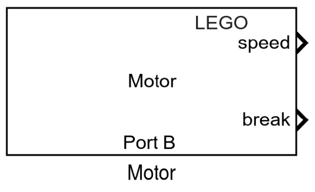


Figure 6.9: Servo Motor

The `Encoder` block in figure 6.10 is the mirror block to the `Encoder` block from the `Model` group. It has one input signal representing the number of motor rotations in degrees. The value of the signal is passed into the `Encoder` block from the `Model` group which has set the same port parameter value as this block has.

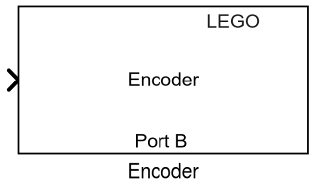


Figure 6.10: Encoder

Figure 6.11 illustrates the `Color Sensor` block from the `Environment` group of blocks. It has two parameters: *port* and *mode*. The *mode* parameter can have the same values as the `Color Sensor` from the `Model` group. If the *mode* is set to *ColorSensor* the block has three input signals for red, green and blue color. If the *mode* parameter has any other value there is only one input signal corresponding to the color specified by the *mode*. The values of input signals are passed to the `Color Sensor` from the `Model` group that has the same value of the *port* parameter.

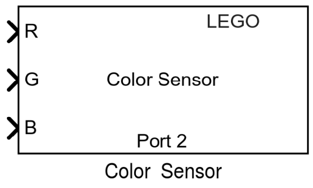


Figure 6.11: Color Sensor

The `HiTech Color Sensor` in figure 6.12 has one parameter *port* specifying the port of the `HiTech Color Sensor` in the model of the system. This block has three input signals for each for one of the red, green and blue component of the color spectrum. The values of the input signals are passed into the `HiTech Color Sensor` in the model of the system.

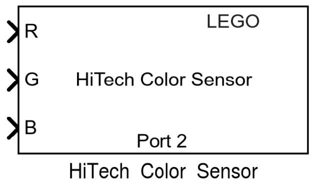


Figure 6.12: HiTechnic Color Sensor

The last of the blocks is the `HiTech Color Sensor version 2` illustrated in figure 6.13. The block has one parameter specifying the port of the corresponding block in the model of the system. There are three input signals in this block. They carry red, green and blue component of the color. These values are passed into the corresponding block in the model of the system.

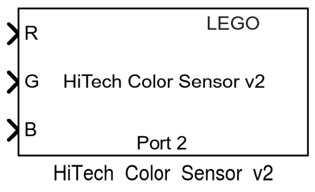


Figure 6.13: HiTechnic Color Sensor v2

## 6.3 Model Creation and Simulation

To create the model of desired system, we need to set up new Simulink model, we name it ”ExampleMdl”. The top level block in the model will be a `Subsystem` set to be treated as atomic. All the blocks describing the system will be placed into this top level block. The reasons for this structure are explained in section 6.4. The model that will be created will drive the robot in a strait line. To achieve this goal we will start with a constant connected to two motors. And for analysis we will insert encoder blocks connected to a scope. The model described here is shown in figure 6.14. We need to ensure signal type consistency. In order to do it we will change the constant output type from double to int8 (the input signal type of the motor blocks). Before we continue the model must be saved.

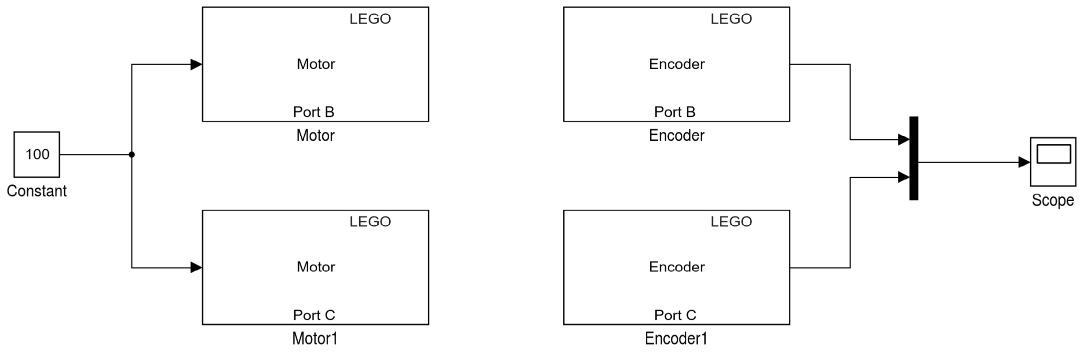


Figure 6.14: ExampleMdl Model

To simulate the created model we also need to model the environment in which the robot will operate. We will create second Simulink model named ”ExampleEnv”. We start by inserting two motor blocks. The break signal of either motor block will be ignored. And the speed signal will accumulate the motor revolution count in a memory block. This accumulated revolution count will be passed into corresponding encoder of each motor block. We will alter the signal from the second motor to a 90% of the original value, simulating physical difference between these motors. Finally we insert a model block that will reference our previously created model. The ”ExampleEnv” is shown in figure 6.15.

Such created solution is ready for simulation. Since we didn’t tune the sample time of individual blocks the simulation will be continuous. The preset simulation time is 10 seconds. After we click the *Run* button in the ”ExampleEnv” model the simulation will take place. After the simulation ends we can analyze the result. In the ”ExampleEnv” model we go into the block `Model` and we continue into the block `Scope`. The graph that has opened contains two linear functions

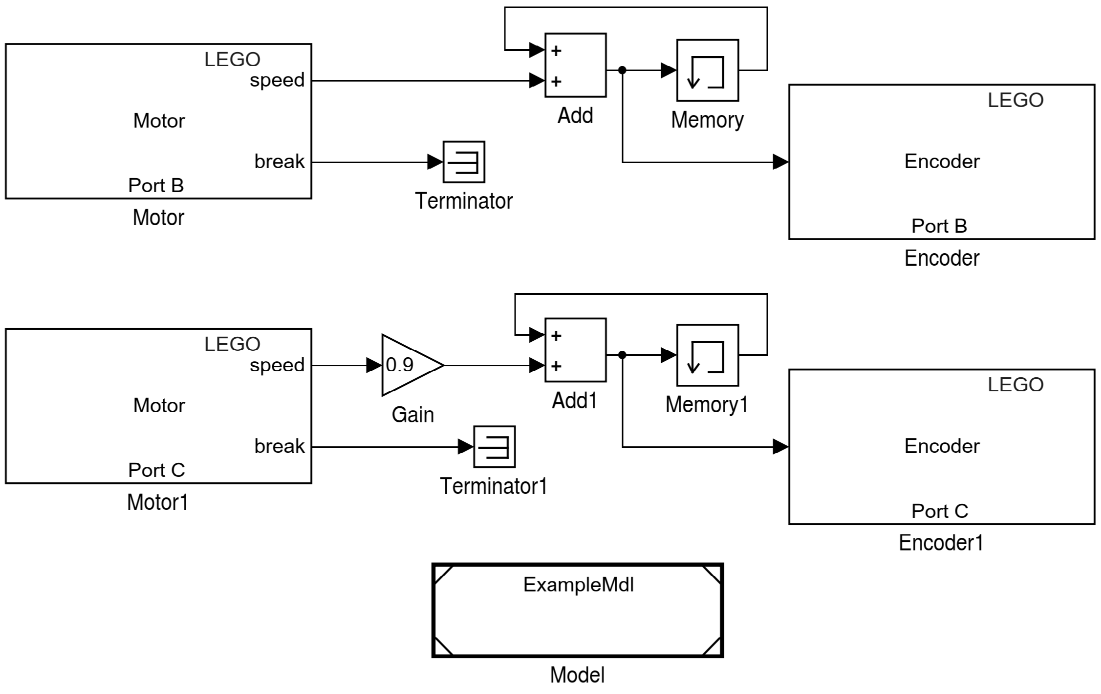


Figure 6.15: ExampleEnv Model

representing the number of revolutions of each motor. We can instantly see that one motor is faster than the other. The graph is shown in figure 6.16.

To fix this problem we need to introduce a *Proportional Integral (PI) controller* into the "ExampleMdl" system. The *PI controller* corrects the input values by adjusting the output values with respect to the accumulated error. To achieve this goal the *PI controller* uses the numerical integration. The *PI controller* contains two parameters ($K_i$ and $K_p$) that can be tuned to obtain the best results. The formula to calculate the correct value is in equation 6.1.

$$u(t) = K_p e(t) + K_i \int e(t)\,dt \tag{6.1}$$

Where the variables in the equation are the following:

- $u(t)$ – controller output signal as a function of time $t$.
- $K_p$ – controller proportional gain, a tuning parameter.
- $e(t)$ – current controller error, defined as $SP - PV$.
    - $SP$ – set point value.
    - $PV$ – measured process value.
- $K_i$ – controller integral gain, a tuning parameter.

We will create the *PI controller* subsystem in the "ExampleMdl" with the functionality defined above. And the signal produced by the subsystem will be used to correct the motor speed. The adjusted "ExampleMdl" model is shown in figure 6.17 and the *PI controller* subsystem is in figure 6.18.

Such altered model produces correct output during the simulation. After the simulation from the "ExampleEnv" model the signal values from the encoders are

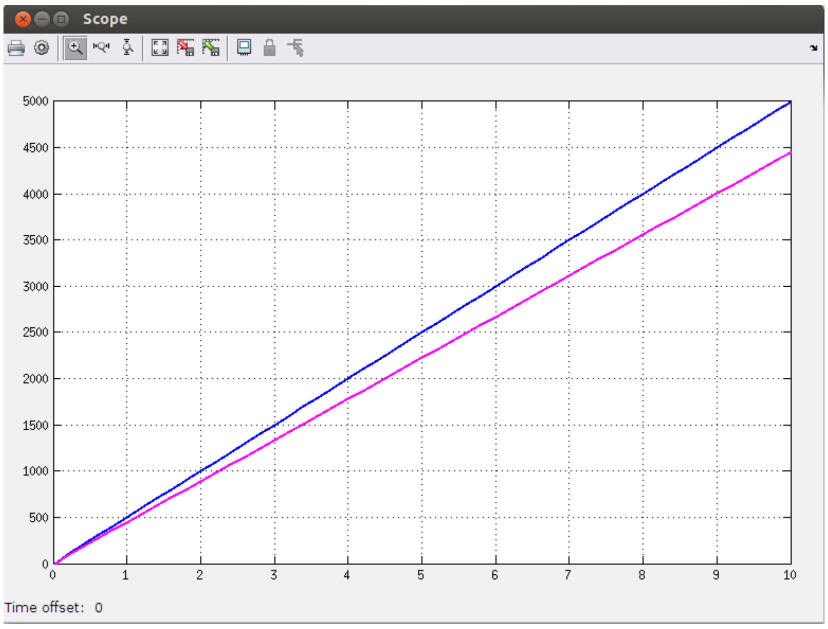


Figure 6.16: Encoder Values

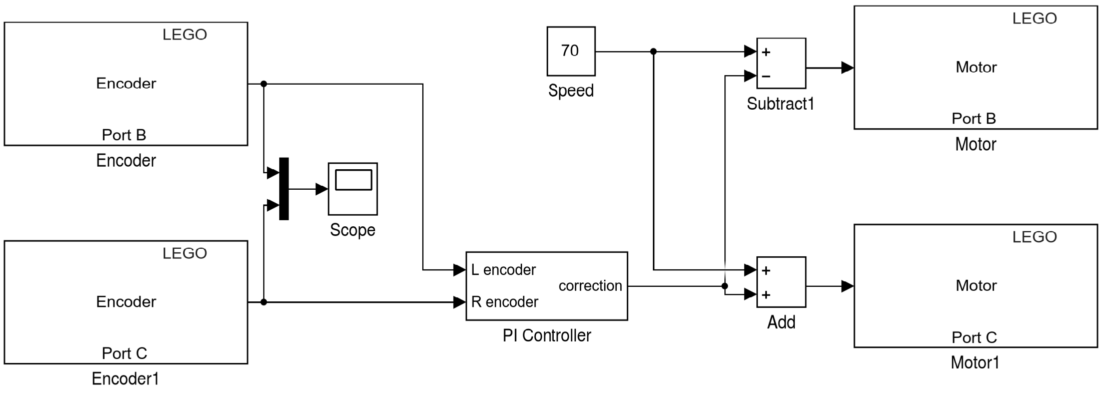


Figure 6.17: ExampleMdl Model with PI Controller

very similar. The robot driven by that model will go in a strait line. The model is still not completely correct though, because it operates in continuous time and the robot executes the task in discrete time steps. More about this is discussed in section 6.4.

To exploit and illustrate the ability of task configuration we add a new `Subsystem` block to the top level of the model and name this block *Task2*. We configure this block to be treated as atomic and to have a different sample time than the previous one. Figure 6.19 illustrates the shape of the model at its top level.

Inside the *Task2* subsystem we place two `Display` blocks representing different lines on the NXT LCD display. These blocks will display the value of the encoders. In order to accomplish that behavior we add two `Encoder` blocks, configure the ports they are connected to and connect them to the previously added `Display` blocks. Figure 6.20 illustrates the model in the newly added subsystem.

The generated code from this model will be more illustrative since there are two tasks, yet still simple to comprehend. Section 6.5 is devoted to the description

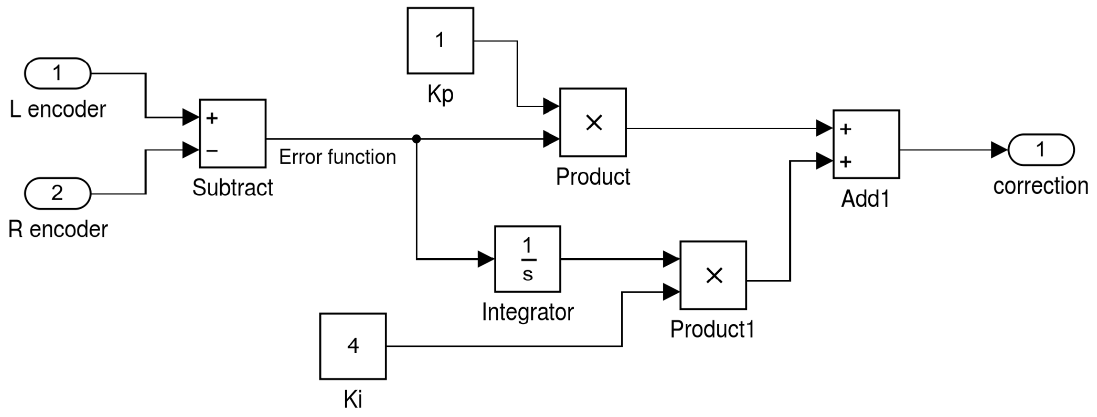


Figure 6.18: PI Controller

Task1
Task2

Figure 6.19: ExampleMdl Model Top Level

of the generated code.

## 6.4 Code Generation

The engine of the code generation process is a coder. There are many coders that can be used in Simulink. The basic one, that is shipped with Simulink, is the Simulink Coder. In this project we use the Embedded Coder. The code generated by the Embedded Coder is more efficient and is suitable for embedded systems.

The build process in Simulink employs more tools as it is divided into a several steps. The overall process of code generation is illustrated in figure 6.21 and the individual stages of the process are described below. The tool that drives all these steps is called Real-Time Workshop (RTW) and the command to invoke it is typically *make_rtw*.

1. RTW compiles the block diagram and generates a model description file, <model_name>.rtw.

2. RTW invokes the TLC [3] to generate target-specific code, processing <model_name>.rtw as specified by the selected STF.

3. RTW creates a makefile, <model_name>.mk, from the selected TMF.

4. make is invoked. make compiles and links the program from the generated code, as instructed in the generated makefile.

The Real-Time Workshop uses TLC files to translate the Simulink model into code. The Target Language Compiler uses two types of TLC files during the code generation and build process. The system target file, which describes how to generate code for a chosen target, is the entry point for the TLC program that

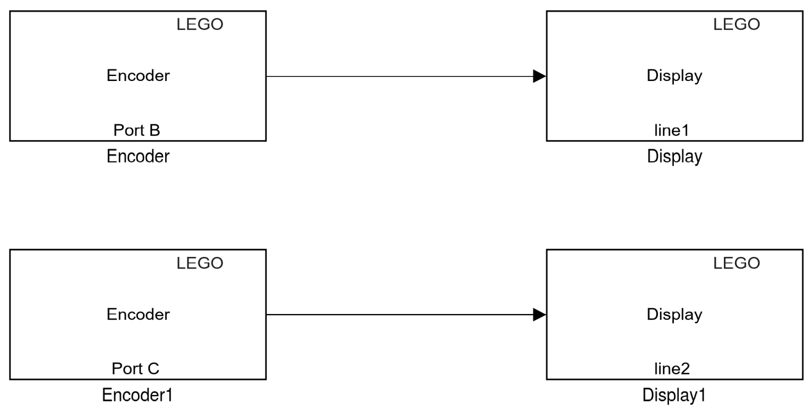


Figure 6.20: ExampleMdl Task 2

creates the executable. Block target files define how the code looks for each of the Simulink blocks in your model.

The code generation process has a lot of options, through these options the process can be adjusted to fit the target platform. There is the STF, which defines the coder that will be used for the code generation, callback handlers that perform variety of actions during the code generation, and additional files entering the code generation process. This project includes the STF named `nxt_ert`. It defines the code generation process for the LEGO NXT platform using the Embedded Coder. Selecting this STF configures the Simulink for the LEGO NXT target. The STF can be set on the *Code Generation* pane in the *Configuration Parameters* window (illustrated in figure 6.22). The `nxt_ert` defines the TMF used to generate the makefile for the generated code. Also there are included two additional source templates that enter the code generation process: `nxt_main.tlc` and `nxt_OIL.tlc`.

From these templates there are generated the <`model_name`>`_main.c` source file and the <`model_name`>`.oil` configuration file.

The `nxt_main.tlc` template is included in appendix 2. The file is written in the TLC language. The coder will generate code based on the description in this language. Each line in the TLC language starting with the %% sequence represents a comment. These lines are ignored by the coder and their significance is only to improve the readability and comprehensibility of the code. Each word starting with the % character is a directive for the coder similarly as keywords in classic programming languages. The expressions surrounded between %< and > are TLC tokens, and they are replaced during the code generation process. Ellipses (...) at the end of a line means the continuation of a statement and ensure that the line break is not printed in the code generation process. All other text in the TLC file, if it is not an argument to some TLC directive, is simply printed by the coder as is.

In the `nxt_main.tlc` file we can see that at first there is checked whether a valid tasking mode is specified. Following this check there is a directive that opens the file <model_name>_main.c and it also redirects the output of the coder into that file. Next there is a section that prints out the includes of necessary header files. Below these includes there are declarations of nxtOSEK constructs. There are counters and tasks, depending on the tasking mode, being defined. The

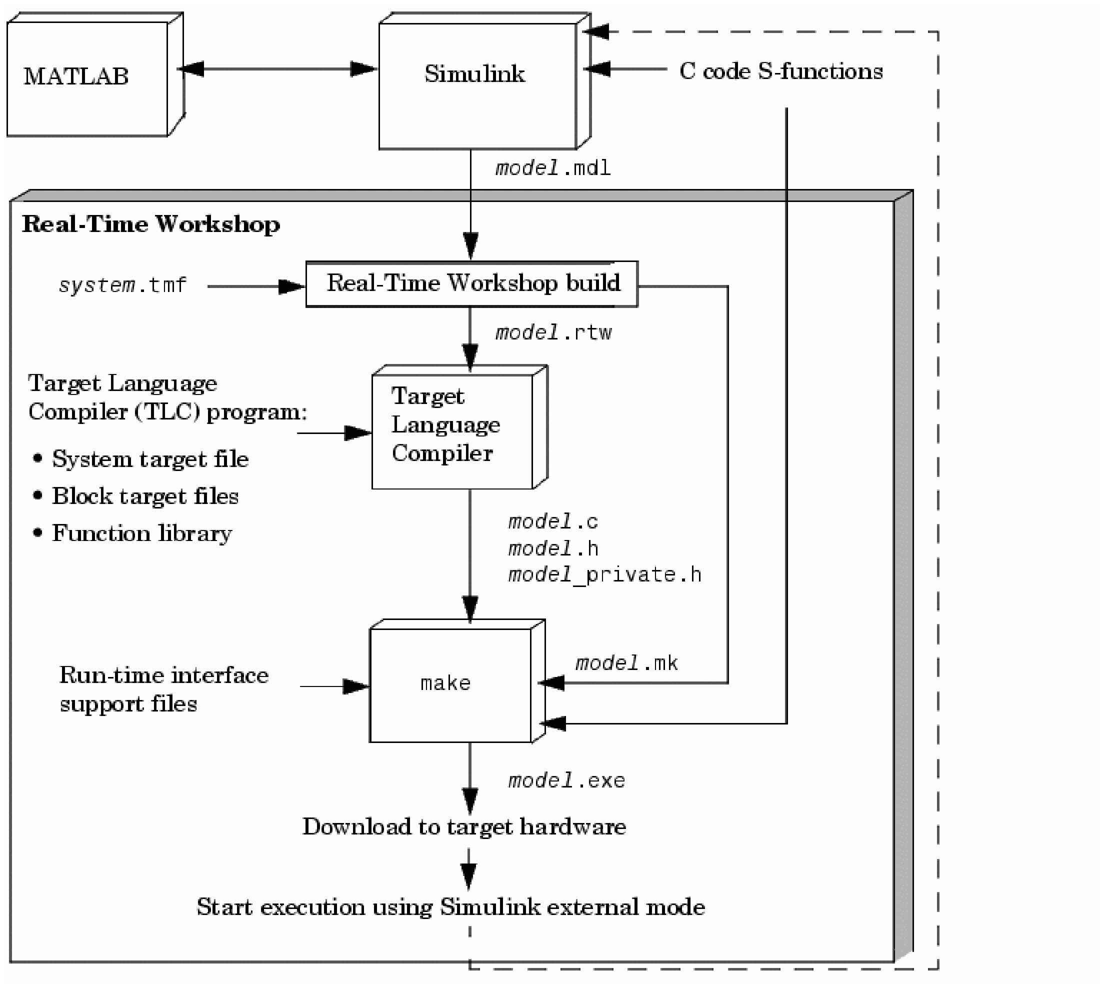


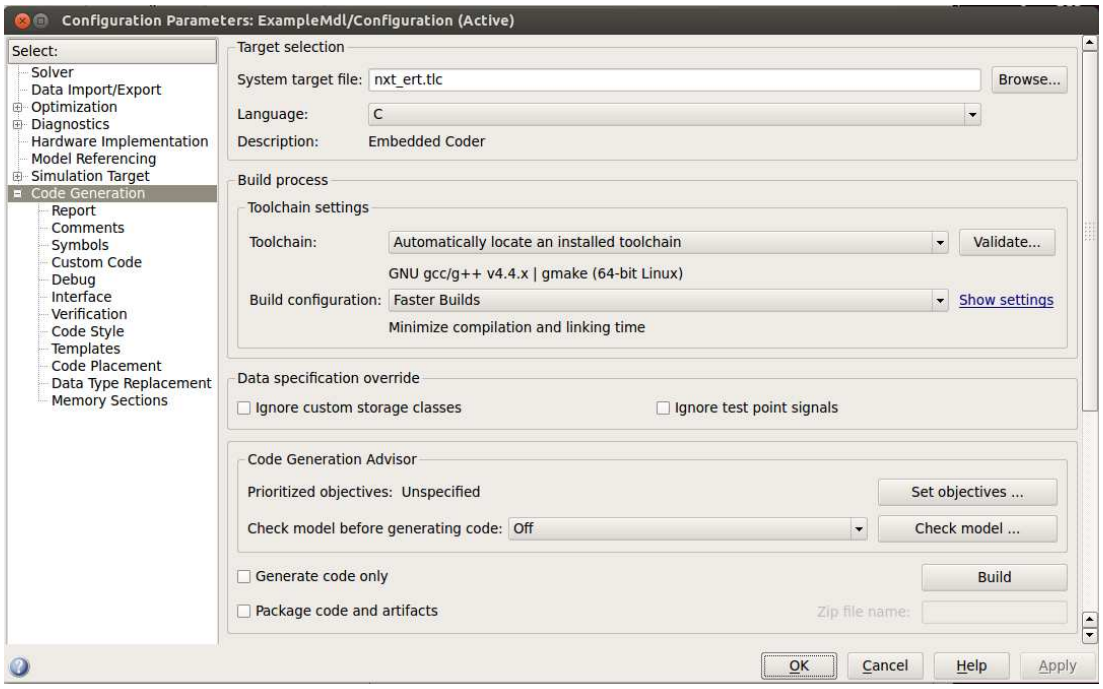
Figure 6.21: Code Generation Process

next section defines the nxtOSEK hook routines:

- `ecrobot_device_initialize`
- `ecrobot_device_terminate`
- `user_1ms_isr_type2`

The initialize routine is invoked by the RTOS once at the beginning of the program execution, and it simply calls the initialize method generated from the model. The terminate routine is invoked once at the end of the program execution, and it calls the terminate method generated from the model. The interrupt service routine (isr) is called by the RTOS every 1 ms and it increments the defined counter. The rest of the file is divided by the tasking mode of the model. If the mode is single tasking there is generated one task that calls the step function generated from the model, and terminates the task in order to allow its invocation in the next period. However, if the mode is multi tasking there are generated multiple tasks that calls the corresponding step functions.

At the end of this file there is the directive to close the <model_name>_main.c file. This directive redirects the output of the coder back to the location where it pointed before.


Configuration Parameters: ExampleMdl/Configuration (Active)

Target selection

System target file: nxt_ert.tlc

Language: C

Description: Embedded Coder

Build process

Toolchain settings

Toolchain: Automatically locate an installed toolchain

GNU gcc/g++ v4.4.x | gmake (64-bit Linux)

Build configuration: Faster Builds

Minimize compilation and linking time

Data specification override

☐ Ignore custom storage classes ☐ Ignore test point signals

Code Generation Advisor

Prioritized objectives: Unspecified

Check model before generating code: Off

☐ Generate code only

☐ Package code and artifacts

Zip file name:

Figure 6.22: Code Generation Options

The <`model_name`>`.oil` template is included in appendix 4. At the beginning of this file there is also a check whether the used tasking mode is supported. There follows the definition of local variable that defines the coefficient factor for the conversion of seconds into milliseconds. This coefficient is needed since Simulink defines the sample time of a block in seconds and the OIL configuration defines the task period in milliseconds. Next there is the directive for the redirection of the coder output into the <model_name>.oil file. The next lines define the configuration of the system and they are printed by the coder. There is definition of the RTOS, the counter, and the tasks and alarms for these tasks depending on the tasking mode of the model. The values of the `ALARMTIME` and `CYCLETIME` are taken from the model. At the end of the script the <`model_name`>`.oil` file is closed and the output of the coder is redirected back where it pointed before.

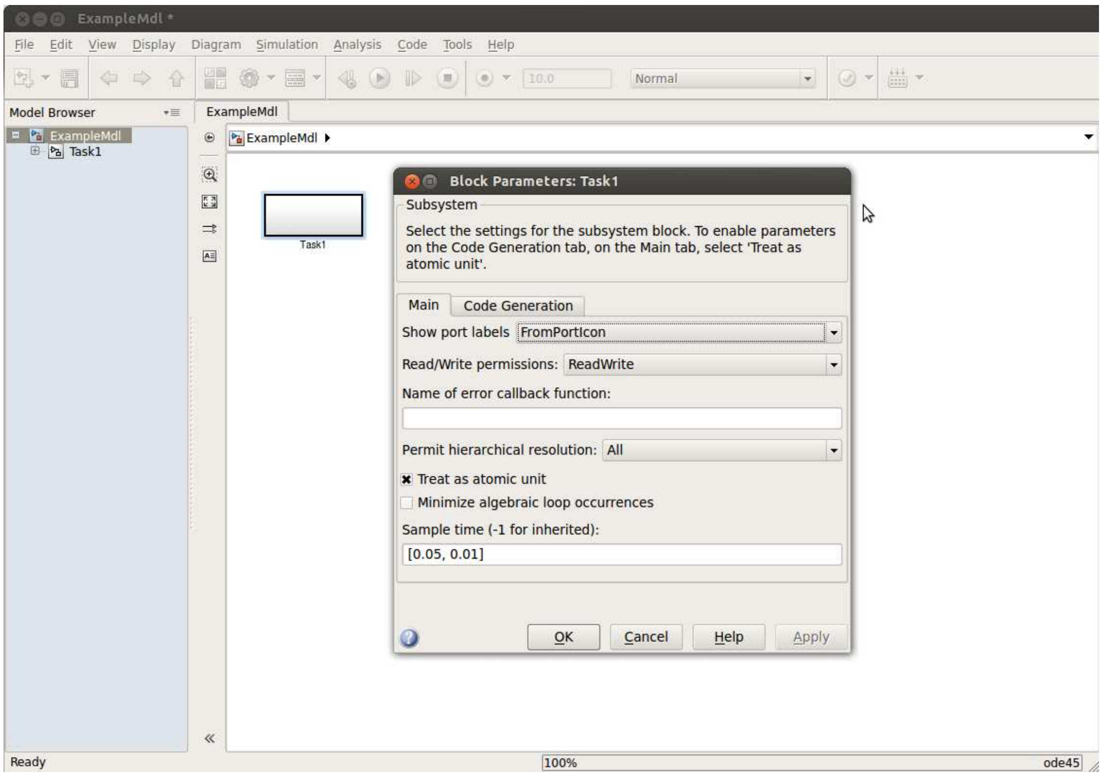
The coder generates the code from the model block by block. The support of the blocks from this project (discussed in section 6.2) is ensured by the TLC files (each for every block). These files define the code generated for those blocks. For each different sample time in the model, the code generation process creates functions for handling the process running in the sample time period. These functions are:

- <`model_name`>`_initialize`
- <`model_name`>`_step`
- <`model_name`>`_terminate`

There needs to be generated a main file that will define the tasks for the real-time operating system and will call the corresponding step functions in those tasks.

To exploit the power of an OIL file to define multiple tasks the structure of the model is bounded to have all top level elements the `Subsystem` blocks that are set to be treated as atomic. Each this block represents an task in the final. system. Each this block has defined a sample time and optionally a time offset. The sample time directly defines the period of the task it represents. Every block in the model, except the top level `Subsystem` blocks, has to inherit its sample time. During the code generation process there is an callback handler that checks the model. It verifies that the top level blocks are `Subsystem` blocks and that these blocks have valid sample time. It also checks that every other block in the model inherits its sample time. If any of the constraints above is not true, the code generation process is aborted.

Let's name the top level `Subsystem` block in the model "ExampleMdl" `Task1`. To generate code for the model, we must set a discrete sample time to the block `Task1`. For such altered model the simulation won't work now, because the `integrator` block (figure 6.18) requires continuous time. To correct the model we must replace the `integrator` block by the `Discrete-Time integrator` block. In figure 6.23 there are shown the options of the `Task1` block with defined sample time and time offset. The value of the *Sample Time* option is set to $[0.05, 0.01]$ where the first number defines the period and the second number defines the time offset. Both these values are in seconds. If the time offset value is omitted there is only one number specified (without the brackets) standing for the period.



Block Parameters: Task1

Subsystem

Select the settings for the subsystem block. To enable parameters on the Code Generation tab, on the Main tab, select 'Treat as atomic unit'.

Main | Code Generation

Show port labels: FromPortIcon

Read/Write permissions: ReadWrite

Name of error callback function:

Permit hierarchical resolution: All

[✓ Treat as atomic unit]

Minimize algebraic loop occurrences

Sample time (-1 for inherited):

[0.05, 0.01]

OK | Cancel | Help | Apply

Figure 6.23: Task1 Sample Time

The multitasking ability of the system can be influenced within the setting of the model. In the *Configuration Parameters* window there is a pane *Solver*. On this pane there are options for the simulation solver of the model. The option *Tasking mode for periodic sample times* can be either *MultiTasking*, *SingleTasking* or *Auto*. The value of this option influences the code generation, if there are defined more than one task in the model. If the option is set to be *MultiTasking* then the generated main and the OIL file contains one task definition for each task in the model and the tasks are called by the real-time operating system according to the rules of real-time constraints. The period and time offset of each task corresponds to the values of the task in the model. If the *SingleTasking* value of the option is selected than there is configured only one task in the main and in the OIL file. This task has the period of the greatest common divisor of all the sample times. All the tasks defined in the model are called within this one generated task, but the period of invocation of concrete task corresponds to the period defined in the model. Finally if the option if set to *Auto* the code generation process creates a single task if there is only one distinct sample time and it creates a multiple tasks if there is more distinct sample times defined in the model.

After the code is generated it can be compiled together with the real-time operating system from the nxtOSEK. The compilation can be invoked automatically after the code generation. Whether MATLAB starts the compilation depends on the option *Generate code only*. This option can be found in the Code Generation Options showed in figure 6.22. If the option is selected MATLAB doesn't invoke the compilation, but it stops after the code is generated. Otherwise the make is called within MATLAB automatically. In the makefile generated in the project there are definitions, that are read by MATLAB, and that configures the invocation. There is specified the make command, the host OS and other options.

In the case when the user selects the *Generate code only* for any reason. The generated project can be easily compiled from the command line. To accomplish the manual build there needs to be called the following command within the directory with the generated project.

```
make all -f <model_name>.mk
```

The build procedure will parse the OIL configuration file, configure and compile the kernel of the RTOS and the source files of the generated project. After the process finishes the resulting binary can be uploaded into the NXT robot using the *appflash* utility. During the compilation there is created also an script, that calls the *appflash* program with correct parameter and it knows where is the *appflash* located. It is convenient for the user because he/she aren't concern with this details. The script is named *appflash.sh* and it is always generated in the project folder. The robot needs to be prepared for the program upload as described in the nxtOSEK documentation as well as in the documentation of this project.

## 6.5 Code Structure

During the code generation process the coder creates a directory <model_name>_nxt_ert_rtw where it puts all the generated source code files. The suffix of the directory is unique for the LEGO NXT target, and is composed from three parts. The `nxt` stands for the target platform LEGO NXT. The `ert` expresses the used coder: the Embedded Coder. The `ert` stands for the Embedded Real-Time. And finally the `rtw` which stand for Real-Time Workshop. The Real-Time Workshop is legacy, it is former Simulink Coder. The structure of the generated project is illustrated in figure 6.24.

```
<model_name>_nxt_ert_rtw
 ├── <model_name>.h
 ├── <model_name>.c
 ├── <model_name>_data.c
 ├── <model_name>_private.h
 ├── <model_name>_types.h
 ├── <model_name>_main.c
 ├── <model_name>.oil
 └── <model_name>.mk
```

Figure 6.24: Generated Project Structure

There are other files generated from the model that contain necessary types and data structures, but the files above contain the principal implementation of the model. The description of these files follow:

- **<model_name>.h** – Declares model data structures and a public interface to the model entry points and data structures.
- **<model_name>.c** – Contains entry points for all code implementing the model algorithm.
- **<model_name>_data.c** – It contains the declarations for the parameters data structure and the constant block I/O data structure. If these data structures are not used in the model, this file is not generated.
- **<model_name>_private.h** – Contains local macros and local data that are required by the model and subsystems.
- **<model_name>_types.h** – Provides forward declarations for the real-time model data structure and the parameters data structure.
- **<model_name>_main.c** – This file is generated from the template in this project. It contains definitions of tasks and hook routines for the RTOS. From this file there are called the entry point methods from the **<model_name>.c** file.
- **<model_name>.oil** – This file is also generated from the template in this project. It contains configuration of the final system.
- **<model_name>.mk** – The makefile is generated from the TMF in this project. This makefile defines all the generated source files that needs to be compiled and references the dedicated makefile in the nxtOSEK project, which prescribes the overall compilation process.

There is an example of generated source code of the main file in appendix 3. This file is taken from the example model in section 6.3. At the beginning of the file there is a section with necessary includes. There is included the header file generated for the model, in that file there are defined the entry point methods to the model initiation, step execution and termination. The next two includes define the header files of the RTOS, its features are used in the main file as well. The last include concerns the nxtOSEK header file with the method interfaces for the LEGO NXT sensors and actuators. The next section takes care of the declaration of tasks for the RTOS. There is defined a counter, that counts the time, and since the model was multi tasking there is a multiple declared tasks. Follows the section with RTOS hook routines. The `ecrobot_device_initialize` method is called once at the beginning of the program execution. This method calls the initialize method on the generated code of the model. The `ecrobot_device_terminate` method is called once at the end of the program execution. This method calls the terminate method on the generated code of the model. The `user_1ms_isr_type2` routine is called by the RTOS each 1 ms and it serves to increment the defined counter. At the end of the file there are definitions of the tasks. Each task calls the step function on the model code with the process id as a parameter. The `Task1` and the `Task2` performs the behavior of the tasks in the model. But the `Task0` only switches flags that indicate which task should run next. This functionality is however covered by the present RTOS, and therefor the flags are ignored.

There is also an example of generated OIL configuration file in appendix 5. This file is taken from the example model in section 6.3 as well. At first there is an include of the `implementation.oil` file. This file is generated at the beginning of the compilation process and it contains the OIL version and implementation standard. The rest of the configuration in this file is encapsulated in the definition of CPU. This way it is ensured by the standard that there is one OIL configuration file per target CPU. As the first item of the following configuration there is the specified the used RTOS. Next there is an application mode, which is empty. We don't need to configure any application modes for our purposes. There is the definition of timer counter. This counter is updated in the interrupt service routine placed in the main file mentioned above. This single timer counter is used for all the alarms defined below. Now there are pairs of task and alarm. These pairs are identified by the same anding number in their name. The alarm is always used to activate the corresponding task when the time for its execution occurs. The `Task0` is only a task that handles flags indicating the next task to be executed. Therefor the cycle time of this task is the greatest common divisor of the cycle times of the remaining tasks. And the cycle time of each remaining task is the sample time defined in the model. The definitions in the OIL configuration files corresponds to the declarations in the main file.

Appendix 1 shows a simplified example of the generated makefile. This makefile is from the example model in section 6.3. At the beginning of the makefile there is a variable defining the location of the nxtOSEK project installed on the computer. This reference is substituted into the TMF during the installation of this project. The next section of the file is prefaced with a comment that describes the items in this section. There are definitions that are parsed by MATLAB after the code generation process. These definitions configure the build process and MATLAB is able to invoke the makefile and start the compilation of the project. The rest of the makefile contains definition of the source files that needs to be compiled. The last statement includes the makefile from the nxtOSEK project which defines the compilation process.

The compilation of the generated project creates other files that are mixed among the source files. The additional files are shown in figure 6.25. The description of these files follows:

```
<model_name>_nxt_ert_rtw
 |__ build
 |__ appflash.sh
 |__ <model_name>_rom.bin
 |__ <model_name>_rom.elf
 |__ <model_name>_rom.map
 |__ implementation.oil
```

Figure 6.25: Compiled Files

- **build** – Directory containing all the object files compiled from the generated source code file and from the sources that are required from the nxtOSEK project, including the sources of the RTOS.
- **appflash.sh** – Shell script that invokes the *appflash* utility program for the upload of the created program.
- **<model_name>_rom.bin** – The binary program that is the result of the compilation. This program encapsulates the RTOS together with the tasks defined by the model. It is targeted to the LEGO NXT hardware architecture.
- **<model_name>_rom.elf** – The Executable and Linkable Format (ELF) version of the compiled program targeted to the LEGO NXT hardware architecture.
- **<model_name>_rom.map** – The virtual memory map file describing the layout of the compiled binary program.
- **implementation.oil** – Defines the OIL version and implementation standard.

# 7. Evaluation

The goal of this project was to enable developing software system for LEGO NXT robots using the Simulink development environment exploiting the model-driven development approach. The support of the LEGO NXT target platform was required to work seamlessly within the Simulink under Linux operating systems. There was required to provide a set of Simulink blocks that employ sensors and actuators of the LEGO NXT robotics kit. Simulink models using these blocks must support the simulation and code generation steps of the model-driven development process.

There are two sets of blocks provided in this project. The first set of blocks represents the LEGO NXT sensors and actuators which are used to model the system for the NXT robot. These blocks constitute the sensors and actuators of the robot during the model simulation, and are used for the code generation when the prototype of the model is created. The second set of blocks also represents the sensors and actuators of the NXT robot, but are opposite to the blocks from the first set. These blocks are used to model the simulation environment for the model of the created system. These blocks are interconnected with the corresponding blocks from the first set. For example if the `Encoder` block from the `Environment` set of blocks acquire some signal the `Encoder` block from the `Model` set of blocks will provide the same signal in the model of the system. The condition which must be satisfied is that both these blocks have to have set the same port in their parameters.

The code generation process is supported by the blocks as mentioned above, but there is more about the code generation process to be supplied to ensure that the generated code defines complete system that can be compiled into binary. There are provided needed files which guides the code generation of the program main source file, the OIL configuration file and the makefile. The overall course of the code generation process is governed by the supplied System Target File. This file also defines the setting of needed parameters of Simulink to adjust the environment to LEGO NXT target platform. The code generation and subsequential compilation is realized within the Simulink development environment using the means dedicated to this purpose.

For the software systems generated from the models created in Simulink there is used nxtOSEK project as a framework. The nxtOSEK defines functions interfacing the NXT sensors and actuators, that are used in the generated code to interact with those peripheries. The nxtOSEK also provides the real-time operating system (TOPPERS/JSP) which is compiled together with the generated code and handles the resource management and scheduling of the tasks from the model. The nxtOSEK assures that the created systems are executed as real-time tasks.

There is a patch of the nxtOSEK in this project. The patch adds the support of the `HiTechnic Color Sensor version 2` and modifies the nxtOSEK in such a way to preserve its functionality under Linux operating systems.

The final part of this project brings an utility application *appflash* that is able to upload the created binary of the model into the NXT robot using the USB connection. This application works under Linux operating systems, thereby supplies the functionality of the *Fantom* driver provided by LEGO which works only in Windows and OS X operating systems. The upload of the program into the NXT robot is triggered from the command line by calling the *appflash* script in the directory of the compiled program.

This project enables developers to use the model-driven development approach to design systems for the LEGO NXT target platform within Simulink development environment in Linux operating systems. There is support of this development steps: system modeling, model simulation, code generation, compilation and deployment. System modeling is supported by the standard Simulink means and by the supplied set of blocks. Model simulation can be used, because there are two sets of LEGO NXT blocks, to represent the sensors and actuators in the model and in the environment, and because the behavior of the block in the simulation is defined in the code behind each block. Because there is a TLC file for each supplied model block the code generation is supported as well. Compilation is possible thanks to the patch of the nxtOSEK, and that there is a makefile being generated for each model. The *appflash* application ensures the deployment of compiled systems. These all steps of model-driven development are supported and enabled by this project. Using this project students can create, test and deploy their real-time embedded system designs under Linux operating system without the need to write a single line of code.

The software of this project is provided along with this thesis on the included CD. Appendix 6 summarizes the content of the CD.

# Bibliography

[1] MATLAB Documentation,
http://www.mathworks.com/help/matlab/index.html

[2] Simulink Documentation,
http://www.mathworks.com/help/simulink/index.html

[3] Target Language Compiler,
http://www.mathworks.com/help/pdf_doc/rtw/rtw_tlc.pdf

[4] Embedded Coder Getting Started Guide,
http://www.mathworks.se/help/pdf_doc/ecoder/ecoder_gs.pdf

[5] nxtOSEK Documentation,
http://lejos-osek.sourceforge.net/

[6] OSEK/VDX Operating System
http://portal.osek-vdx.org/files/pdf/specs/os223.pdf

[7] OSEK Implementation Language
http://portal.osek-vdx.org/files/pdf/specs/oil25.pdf

[8] The Pragmatics of Model-Driven Development,
http://www.computer.org/csdl/mags/so/2003/05/s5019.pdf

[9] Model-Driven Development Testing Methods,
https://www.linkedin.com/groups/Difference-between-MIL-SIL-PIL-109866.S.167229094

[10] $\mu$ITRON4.0 Specification,
http://www.ertl.jp/ITRON/SPEC/mitron4-e.html

[11] Introduction to OMG's Unified Modeling Language,
http://www.omg.org/gettingstarted/what_is_uml.htm

[12] Halpin, Terry. Object-Role Modeling: an Overview
http://www.orm.net/pdf/ORMwhitePaper.pdf

[13] Rumbaugh, James; Blaha, Michael; Premerlani, William; Eddy, Frederick; Lorensen, William. Object-Oriented Modeling and Design. Prentice Hall, 1990. ISBN 0-13-629841-9.

[14] Federal Information Processing Standards Publication 184 , 1993.
http://www.idef.com/pdf/Idef1x.pdf

[15] Chen, Peter. The Entity-Relationship Model - Toward a Unified View of Data, 1976. doi:10.1145/320434.320440.

[16] Bruza, P. D., Van der Weide, Th. P. The Semantics of Data Flow Diagrams. University of Nijmegen, 1993.

[17] Microsoft Robotics Documentation,
http://msdn.microsoft.com/en-us/library/bb881626.aspx

[18] ROBOTC Documentation,
`http://www.robotc.net/wiki/Main_Page`

[19] Liu, Jane W. S. Real-Time Systems Prentice Hall, 2000. ISBN-10: 0130996513.

# List of Abbreviations

**API** Application Programming Interface. 12, 22

**ARM** Advanced RISC Machines. 12, 13, 16, 25, 26

**AUTOSAR** Automotive Open System Architecture. 11

**BIOS** Basic Input/Output System. 13, 23

**BricxCC** Bricx Command Center. 12

**CD** Compact Disc. 44

**CPU** Central Processing Unit. 13, 15, 21, 22, 41

**DFD** Data Flow Diagram. 6

**ELF** Executable and Linkable Format. 42

**ER model** Entity-Relation model. 6

**GNU** GNU's Not Unix. 13, 16, 25, 26

**HIL** Hardware in the Loop. 5

**IDEF1X** Integration Definition for Information modeling. 6

**IO** Input/Output. 5

**$\mu$ITRON** Micro Industrial TRON. 14, 22

**JSP** Just Standard Profile. 12, 14, 22, 43

**LCD** Liquid-Crystal Display. 12, 27, 33

**LED** Light-Emitting Diode. 28

**MATLAB** Matrix Laboratory. 2

**MEX** MATLAB Executable. 26

**MIL** Model in the Loop. 5, 9, 21

**NBC** Next Byte Codes. 12

**NQC** Not Quite C. 12

**NXC** Not eXactly C. 12

**OIL** OSEK Implementation Language. 13–16, 19, 22, 23, 26, 36–39, 41–43

**OMR** Object-Role Modeling. 6

**OMT** Object-Modeling Technique. 6

**OSEK** Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen; English: Open Systems and their Interfaces for the Electronics in Motor Vehicles. 12, 13, 15, 22

**PI** Proportional Integral. 32

**PIL** Processor in the Loop. 5, 21

**RAM** Random Access Memory. 12, 22

**RGB** Red, Green, Blue. 28

**RISC** Reduced Instruction Set Computing. 12

**RTOS** Real-Time Operating System. 22, 36, 39–42

**RTW** Real-Time Workshop. 34

**SIL** System in the Loop. 5, 21

**STF** System Target File. 3, 34, 35

**TLC** Target Language Compiler. 26, 34, 35, 37, 44

**TMF** Template Makefile. 25, 34, 35, 40, 41

**TOPPERS** Toyohashi Open Platform for Embedded Real-time Systems. 12, 14, 22, 43

**TRON** The Real-time Operating system Nucleus. 14

**UML** Unified Modeling Language. 6

**USB** Universal Serial Bus. 12, 23, 43

**VU-LRT** Villanova University LEGO Real Time Target. 18, 19

# Attachments

## 1 Generated Makefile: ExampleMdl.mk

```
nxtOSEK_ROOT = /opt/MATLAB/NXT_LEGO_Library/nxtOSEK

# ------------------- Macros read by make_rtw -------------------
#
# The following macros are read by the build procedure:
#
#  MAKECMD          - This is the command used to invoke
#                     the make utility
#  HOST             - What platform this template makefile
#                     is targeted for
#                     (i.e. PC or UNIX)
#  BUILD            - Invoke make from the build procedure
#                     (yes/no)?
#  SYS_TARGET_FILE - Name of system target file.

MAKECMD          = make all
HOST             = UNIX
BUILD            = yes
SYS_TARGET_FILE = nxt_ert.tlc


# ---------------------- Makefile Content ----------------------


MATLAB_ROOT = /opt/MATLAB/R2013b

MODEL = ExampleMdl
MODULES = ExampleMdl_main.c rtGetInf.c rtGetNaN.c
EXT_MODE = 0
RT_NONFINITE = \
    $(shell test -e rt_nonfinite.c && echo 1 || echo 0)


# Target specific macros
TARGET = ExampleMdl
TARGET_SOURCES = \
    $(MODEL).c \
    $(MODULES)
TOPPERS_OSEK_OIL_SOURCE = ./ExampleMdl.oil
USER_INC_PATH = \
    $(MATLAB_ROOT)/extern/include \
    $(MATLAB_ROOT)/simulink/include \
    $(MATLAB_ROOT)/rtw/c/src
```

```
ifeq ($(RT_NONFINITE),1)
  TARGET_SOURCES += rt_nonfinite.c
endif

# Don't modify below part
O_PATH ?= build
include $(nxtOSEK_ROOT)/ecrobot/ecrobot.mak
```

## 2 Main File Template: nxt_main.tlc

```
%% Check that allowed solver is set
%if CompiledModel.Solver != "FixedStepDiscrete"
    %error "Only fixed step discrete solver allowed."
%endif
%if (CompiledModel.FixedStepOpts.SolverMode != "SingleTasking" ...
    && CompiledModel.FixedStepOpts.SolverMode != "MultiTasking")
    %error "Only MultiTasking and SingleTasking options allowed."
%endif

%openfile nxt_main = "%<CompiledModel.Name>_main.c", "w"

#include "%<CompiledModel.Name>.h"
#include "kernel.h"
#include "kernel_id.h"
#include "ecrobot_interface.h"

/*----------------------------------------------------------------*/
/* OSEK declarations                                              */
/*----------------------------------------------------------------*/
DeclareCounter(SysTimerCnt);

%if CompiledModel.FixedStepOpts.SolverMode == "SingleTasking"
    DeclareTask(Task1);
%elseif CompiledModel.FixedStepOpts.SolverMode == "MultiTasking"
    %foreach index = CompiledModel.NumSampleTimes
        DeclareTask(Task%<index>);
    %endforeach
%endif

/* nxtOSEK hook */
void ecrobot_device_initialize(void)
{
    %<CompiledModel.Name>_initialize();
}

/* nxtOSEK hook */
void ecrobot_device_terminate(void)
{
    %<CompiledModel.Name>_terminate();
}

/* nxtOSEK hook to be invoked from an ISR in category 2 */
void user_1ms_isr_type2(void)
{
    StatusType ercd;
    /*
     *  Increment OSEK Alarm System Timer Count
     */
    ercd = SignalCounter( SysTimerCnt );
    if( ercd != E_OK )
    {
        ShutdownOS( ercd );
    }
}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                                        %%
%%                  SINGLE TASKING TASK                   %%
%%                                                        %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%if CompiledModel.FixedStepOpts.SolverMode == "SingleTasking"

    /* Background Task */
    TASK(Task1)
    {
        %% ecrobot_status_monitor("OSEK HelloWorld!");
        %<CompiledModel.Name>_step();
        TerminateTask();
    }

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                                        %%
%%                  MULTI TASKING TASKS                   %%
%%                                                        %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%elseif CompiledModel.FixedStepOpts.SolverMode == "MultiTasking"

    %foreach index = CompiledModel.NumSampleTimes

        /* Background Task */
        TASK(Task%<index>)
        {
            %<CompiledModel.Name>_step(%<index>);
            TerminateTask();
        }

    %endforeach

%endif

%closefile nxt_main
```

## 3 Generated Main File: ExampleMdl_main.c

```c
#include "ExampleMdl.h"
#include "kernel.h"
#include "kernel_id.h"
#include "ecrobot_interface.h"

/*--------------------------------------------------------------------------*/
/* OSEK declarations                                                        */
/*--------------------------------------------------------------------------*/
DeclareCounter(SysTimerCnt);
DeclareTask(Task0);
DeclareTask(Task1);
DeclareTask(Task2);

/* nxtOSEK hook */
void ecrobot_device_initialize(void)
{
  ExampleMdl_initialize();
}

/* nxtOSEK hook */
void ecrobot_device_terminate(void)
{
  ExampleMdl_terminate();
}

/* nxtOSEK hook to be invoked from an ISR in category 2 */
void user_1ms_isr_type2(void)
{
  StatusType ercd;

  /*
   *  Increment OSEK Alarm System Timer Count
   */
  ercd = SignalCounter( SysTimerCnt );
  if (ercd != E_OK ) {
    ShutdownOS( ercd );
  }
}

/* Background Task */
TASK(Task0)
{
  ExampleMdl_step(0);
  TerminateTask();
}
```

```
/* Background Task */
TASK(Task1)
{
  ExampleMdl_step(1);
  TerminateTask();
}

/* Background Task */
TASK(Task2)
{
  ExampleMdl_step(2);
  TerminateTask();
}
```

## 4 OIL File Template: nxt_OIL.tlc

```
%% Check that allowed solver is set
%if CompiledModel.Solver != "FixedStepDiscrete"
    %error "Only fixed step discrete solver allowed."
%endif
%if (CompiledModel.FixedStepOpts.SolverMode != "SingleTasking" ...
    && CompiledModel.FixedStepOpts.SolverMode != "MultiTasking")
    %error "Only MultiTasking and SingleTasking options allowed."
%endif

%% conversion factor between seconds and milliseconds
%assign s2ms = 1000

%openfile oil_file ="%<CompiledModel.Name>.oil", "w"

#include "implementation.oil"

CPU ATMEL_AT91SAM7S256
{
    OS LEJOS_OSEK
    {
        STATUS = EXTENDED;
        STARTUPHOOK = FALSE;
        ERRORHOOK = FALSE;
        SHUTDOWNHOOK = FALSE;
        PRETASKHOOK = FALSE;
        POSTTASKHOOK = FALSE;
        USEGETSERVICEID = FALSE;
        USEPARAMETERACCESS = FALSE;
        USERESSCHEDULER = FALSE;
    };

    APPMODE appmode1{};

    COUNTER SysTimerCnt
    {
        MINCYCLE = 1;
        MAXALLOWEDVALUE = 10000;
        TICKSPERBASE = 1;
    };

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                                           %%
%%                   SINGLE TASKING TASK                     %%
%%                                                           %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%if CompiledModel.FixedStepOpts.SolverMode == "SingleTasking"
    %assign alarmCycle = ...
        %<CompiledModel.FixedStepOpts.FixedStep> * %<s2ms>
    %assign alarmCycleNum = CAST("Number", alarmCycle)

    TASK Task1
    {
        AUTOSTART = FALSE;
        SCHEDULE = FULL;
        PRIORITY = 5;
        ACTIVATION = 1;
        STACKSIZE = 512;
    };

    ALARM cyclic_alarm
    {
        COUNTER = SysTimerCnt;
        ACTION = ACTIVATETASK
        {
            TASK = Task1;
        };
        AUTOSTART = TRUE
        {
            %% Alarm offset
            ALARMTIME = %<alarmCycleNum>;
            %% Alarm cycle
            CYCLETIME = %<alarmCycleNum>;
            APPMODE = appmode1;
        };
    };

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                                                      %%
%%                         MULTI TASKING TASKS                          %%
%%                                                                      %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%elseif CompiledModel.FixedStepOpts.SolverMode == "MultiTasking"

%foreach index = CompiledModel.NumSampleTimes
    %assign sampleTime = ...
        %<CompiledModel.SampleTime[index].PeriodAndOffset>
    %assign alarmCycle = %<sampleTime>[0] * %<s2ms>
    %assign alarmCycleNum = CAST("Number", alarmCycle)
    %assign alarmOffset = %<sampleTime>[1] * %<s2ms>
    %if alarmOffset == 0
        %assign alarmOffset = alarmCycle
    %endif
    %assign alarmOffsetNum = CAST("Number", alarmOffset)

    TASK Task%<index>
    {
        AUTOSTART = FALSE;
        SCHEDULE = FULL;
        PRIORITY = 5;
        ACTIVATION = 1;
        STACKSIZE = 512;
    };

    ALARM cyclic_alarm%<index>
    {
        COUNTER = SysTimerCnt;
        ACTION = ACTIVATETASK
        {
            TASK = Task%<index>;
        };
        AUTOSTART = TRUE
        {
            %% Alarm offset
            ALARMTIME = %<alarmOffsetNum>;
            %% Alarm cycle
            CYCLETIME = %<alarmCycleNum>;
            APPMODE = appmode1;
        };
    };

%endforeach
%endif
};

%closefile oil_file
```

## 5 Generated OIL File: ExampleMdl.oil

```
#include "implementation.oil"

CPU ATMEL_AT91SAM7S256
{
    OS LEJOS_OSEK
    {
        STATUS = EXTENDED;
        STARTUPHOOK = FALSE;
        ERRORHOOK = FALSE;
        SHUTDOWNHOOK = FALSE;
        PRETASKHOOK = FALSE;
        POSTTASKHOOK = FALSE;
        USEGETSERVICEID = FALSE;
        USEPARAMETERACCESS = FALSE;
        USERESSCHEDULER = FALSE;
    };
    APPMODE appmode1{};

    COUNTER SysTimerCnt
    {
        MINCYCLE = 1;
        MAXALLOWEDVALUE = 10000;
        TICKSPERBASE = 1;
    };
    TASK Task0
    {
        AUTOSTART = FALSE;
        SCHEDULE = FULL;
        PRIORITY = 5;
        ACTIVATION = 1;
        STACKSIZE = 512;
    };
    ALARM cyclic_alarm0
    {
        COUNTER = SysTimerCnt;
        ACTION = ACTIVATETASK
        {
            TASK = Task0;
        };
        AUTOSTART = TRUE
        {
            ALARMTIME = 10;
            CYCLETIME = 10;
            APPMODE = appmode1;
        };
    };
    TASK Task1
    {
        AUTOSTART = FALSE;
        SCHEDULE = FULL;
        PRIORITY = 5;
        ACTIVATION = 1;
        STACKSIZE = 512;
    };
    ALARM cyclic_alarm1
    {
        COUNTER = SysTimerCnt;
        ACTION = ACTIVATETASK
        {
            TASK = Task1;
        };
        AUTOSTART = TRUE
        {
            ALARMTIME = 30;
            CYCLETIME = 30;
            APPMODE = appmode1;
        };
    };
    TASK Task2
    {
        AUTOSTART = FALSE;
        SCHEDULE = FULL;
        PRIORITY = 5;
        ACTIVATION = 1;
        STACKSIZE = 512;
    };
    ALARM cyclic_alarm2
    {
        COUNTER = SysTimerCnt;
        ACTION = ACTIVATETASK
        {
            TASK = Task2;
        };
        AUTOSTART = TRUE
        {
            ALARMTIME = 50;
            CYCLETIME = 50;
            APPMODE = appmode1;
        };
    };
};
```

# 6 Content of Included CD

```
│ README.txt
├── Thesis_Dominik_Skoda.pdf
└── siblilen
    ├── doc
    │   ├── src
    │   ├── LEGO_NXT_Library.pdf
    │   └── Tutorial.pdf
    ├── install
    │   ├── +nxt_blocks
    │   └── nxtOSEK_patch
    │       └── altered_files
    ├── samples
    ├── scripts
    ├── src
    │   ├── blocks
    │   └── nxt_comm
    ├── nxt_remove.m
    ├── nxt_settings.m
    └── nxt_setup.m
```

The `README.txt` file contains the basic information how to get started using this project. The `Thesis_Dominik_Skoda.pdf` file is the electronic version of this text. The `siblilen` directory contains the software project that is the subject of this work.