Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Control Engineering

MASTER THESIS

RTW target for Linux with CANopen support

Author: Bc. Lukáš Hamáček

Supervisor: Ing. Libor Waszniowski, Ph.D. Prague, 2009

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra řídicí techniky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Lukáš Hamáček**

Studijní program: Elektrotechnika a informatika (magisterský), strukturovaný
Obor: Kybernetika a měření, blok KM1 - Řídicí technika

Název tématu: **RTW target pro Linux s podporou CANopen**

Pokyny pro vypracování:

1. Seznamte se s principy CANopen, nástrojem RTW a implementací CANopen stacku CanFestival.
2. Navrhněte a implementujte základní podporu CANopen do RTW targetu pro Linux.
3. Demonstrujte použití na vhodné aplikaci.

Seznam odborné literatury:

Dodá vedoucí práce

Vedoucí: Ing. Libor Waszniowski, Ph.D.

Platnost zadání: do konce zimního semestru 2009/10

[illegible]

prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta elektrotechnická
-6-
L.S.

[illegible]

doc. Ing. Boris Šimák, CSc.
děkan

V Praze dne 27. 2. 2009

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

V Praze, dne 22.5.2009

Lukáš Hamáček

Poděkování

Tímto bych rád poděkoval všem, kteří mi přispěli cennými radami a pomocí při vypracování diplomové práce, zejména mému vedoucímu ing. Liboru Waszniowskému, PhD. Zvláštní poděkování patří také mé rodině, která mi během celého studia poskytovala podporu jak materialní tak duševní.

Abstract

The goal of this master thesis was to create the support of CANopen communication protocol in Linux target for the Real Time Workshop tool of Matlab. First of all, the toolset (target) for Real Time Workshop was prepared to enable code generation, compilation and execution at the MPC5200 based single-board computer running Linux operating system. The architecture and particular scripts of this target are described in the first part of this thesis. The main task of this work was integration of CanFestival, the open source implementation of the CANopen stack, into the code automatically generated from the Simulink model. The CANopen blockset, providing protocol API in the Simulink model, has been developed for this reason. The CANopen blockset is described in the second part of this thesis. Moreover, examples of using the target and the blockset are attached to simplify the quick start of using these tools.

Abstrakt

Cílem této diplomové práce bylo vytvořit podporu komunikace po protokolu CANopen v kódu generovaném pomocí nástroje Real Time Workshop, který je součástí programu Matlab. Dokument popisuje nejprve takzvaný target, což je sada nástrojů umožňující generování, překlad a spustění kódu na cílové platformě tvořené počítačem BOA5200 s operačním systémem Linux. Hlavní část diplomové práce se poté zaměřuje na integraci ovladače protokolu CANopen do kódu generovaného z Matlabu. K tomuto účelu je vytvořena sada Simulinkových bloků (CANopen blockset), která zpřístupňuje rozhraní ovladače pro použití v modelu Simulinku. Obě části práce jsou doplněny ukázkovými aplikacemi pro rychlé pochopení práce s nástroji.

Contents

Chapter 1

Introduction

1.1 Work motivation

As an introduction to this work I will write a few words about code generation and I will try to explain the gain of code generation from a model. I will focus on control system applications executed by an embedded computer. Applications in control engineering have some common features. The application is based on running periodic task or tasks that contain new output calculation in each step. It means that all the applications have similar architecture and code generation is reasonable in this case.

The automated code generation technique became very popular because it has a lot of advantages over the manual programming. Firstly, the code generation enables very fast application design. While having the required hardware supported by the code generation tool, the application can be done in a few hours. Moreover, the correct working of the application kernel is ensured and does not need to be tested anymore.

The other significant advantage is that code generation isolates low level programming from application design. Professional control system engineer is then able to create embedded controller without having extensive programming skills.

Finally, all the code fragments that appear in the final program are written just once while developing the code generation tool. It faces to more reliable code which is written more rigorously as it is expected to be used in many applications.

Very popular application for control systems design is Matlab-Simulink which creates environment for controller tuning and simulation. *Real Time Workshop* [Mat] is a part of Matlab which provides tools for code generation from the Simulink model. Real Time Workshop defines C-language code representation of each block of Simulink block library and places it into the proper part of model source while generating the code. Moreover, main function of the generated application ensures model synchronization and communication with original Simulink model via TCP connection.

The code generation obviously depends on the target platform defined by computer type and operating system. Real Time Workshop supports several platforms, however, processor PowerPC 5200, which is used at the Department of Control Engineering is not supported. The support for BOA5200 computer (equipped by PowerPC 5200 processor) [AM] and Linux operating system was prepared by Pavel Jelínek in his master thesis [Jel08].

The BOA computer is equipped by the CAN bus port [CiAa]. The aim of this work is to create Simulink support for CANopen protocol communication [CiAb] and enable code generation of real-time application using CANopen communication.

1.2 Employed technologies

The following section describes briefly the hardware, software and protocols used in this thesis.

1.2.1 BOA 5200 computer

BOA5200 is single-board embedded computer based on the PowerPC 5200 processor. It is equipped by 128MB of RAM, 32MB of FLASH memory and Ethernet, RS232 and two CAN peripherals. This computer is running Linux 2.6 having SocketCan [Ber] CAN driver included in its kernel in our application. The file system image includes SSH server to enable using SSH connection and file transfers from the host computer to BOA. Images of both Linux kernel and file system have been already prepared in [Jel08].

1.2.2 Matlab - Simulink

Simulink is a part of Matlab software which provides graphical user interface for numerical simulations. It contains basic blocks that provide signal routing, simple mathematical operations and data display into charts. Simulink is enhanced by blocksets like *Control System Blockset*, *Aerospace Blockset* and many others. These blocksets provide blocks of more sophisticated functions.

Simulation model is created by connecting particular blocks together. This model is then simulated in discrete or continuous sample time by Simulink engine.

Simulink provides also the possibility to create new blocks. The user defined block functionality is written in so called *S-function* which contains the code that should be performed at the initialization phase, simulation step, and so on. This S-function is attached to the S-function block in the Simulink model. The user interface of the block can be customized by the block mask.

Note The blockset has been developed and tested at the Matlab version 2008a. However, it should work at the newer versions as well.

1.2.3 Matlab - Real Time Workshop

Real Time Workshop is a Matlab tool which is able to generate C language code from the Simulink model. This code can be compiled with any C compiler and while running it performs the same operations as the Simulink model during simulation. The generated code is equipped by feature called *External mode*. It is possible to connect from the Simulink model to the running generated application via TCP and the External mode. The simulation in Simulink can be then performed in the same way as without generated code with the difference that it is physically running the external program and the data exchange and control signals are transfered via the External mode.

It is obvious that code generation and compilation is platform dependent. It means that code generated for PC and Windows will not work at embedded computer running Linux and vice versa. The tool which customizes the code while generating and prepares *Makefile* for its compilation is called *target*.

1.2.4 CANopen protocol

CANopen protocol [Wik] is designed to enhance CAN bus communication protocol. It creates in fact the application layer to the CAN protocol according to the OSI model (*Open System for Interconnection*). CANopen uses CAN messages to transmit its data and defines meaning of particular message IDs.

CANopen network consists of particular nodes (devices). Each node is defined by its *Object Dictionary* (OD). This dictionary is a data structure which keeps all the information about communication, node and device settings as well as the state variables of the device. It means, for example, that the CANopen thermometer has the current temperature value stored in its OD. Read CANopen protocol description at [CiAb] for detail information.

1.2.5 CanFestival driver

CanFestival is an open-source driver for CANopen network. This implementation of CANopen stack has been chosen due to the fact that it supports Linux OS as well as the SocketCan CAN bus driver used by our Linux kernel. The driver code architecture is obviously well-considered and I have not noticed any serious limitations in using it. On the other hand, the driver is still in development and it yields a lot of problems. The author team has not released any official version of the driver for a long time. It means that the only way how to obtain the current version is the CVS checkout. However, the code in the CVS repository is continuously changing and not only by fixing bugs and adding new features but by changing existing API functions as well. Moreover, the changes and the code itself are not documented very well. It makes following the current driver version absolutely impossible.

Approximately in the same time as I started working on this project, my colleague started customizing the driver to be used at 16-bit micro-controller without operating system. On the basis of experience with CanFestival described above, we decided to stop updating the driver from CVS repository and fix a stable version in our repository. It does not mean that updating CanFestival version used by CANopen blockset is not possible. It would just need more effort than pure update of repository. Read section 3.2.4 for more information about updating CanFestival version.

1.3 Solution concept

The master thesis goal is to create CANopen protocol support into Real Time Workshop to enable generating applications using CANopen communication from Simulink model. The application has to be compatible with the BOA5200 computer running Linux.

Solving a few tasks allows reaching the goal. First of all, the *target* has to be prepared to enable generating, building and executing the code at the BOA computer. Then, the CANopen blockset has to be created to provide CANopen API to be used in Simulink model. CANopen blocks provide high level simulation of data exchange and events occurrence and generation of code interfacing the CANopen stack.

1.3.1 Embedded target creation

The target customizing the code generated by Real Time Workshop to be used at the BOA computer have been prepared by Pavel Jelínek [Jel08]. The target is supplemented by a set of tools that provide code compilation in Windows and uploading the program executable into the BOA machine. MSYS environment running in Windows is used to execute cross-compiler based on GCC, which compiles generated code and creates Linux executable. This executable file is transfered by SCP to BOA machine and started via SSH connection.

The tools described above are already done and working properly. However, the target for Real Time Workshop is of type GRT (*Generic Real-time Target*) which uses basic type Real Time Workshop for code generation. Matlab provides enhanced version of this tool called *Real Time Workshop Embedded Coder* generating optimized code. The target based on ERT (*Embedded Real-time Target*) has to be used to enable generating code by RTW Embedded Coder. Creation of this target for the desired platform is the first goal of this thesis. Moreover, the documentation of compiler tools and environments should be joint with the new target documentation to provide unified user guide. The target creation and the unified documentation are described in the chapter Linux ERT Target 2 of this document.

1.3.2 CANopen blockset

The main idea of CANopen [CiAb] support is to integrate CanFestival driver [LOLb] into the code generated by ERT target and create Simulink blocks that provide API of particular driver functions. The integration itself means in fact enhancing the generated *Makefile* so that the driver libraries are linked together with the generated code while building the executable file. If the appropriate driver header files are included into the generated code, driver functions can be used.

To implement the driver API in Simulink it is necessary to create blocks that will generate code making the driver to do the required operations. Each block can add its code into the model *Start* and *Terminate* function to perform the necessary settings. The main function of each block is placed in its *Output* function which is performed periodically in the simulation steps or asynchronously after some event occurrence.

Additionally, the driver itself has to be initialized and started. The CANopen node block (section 3.4) will be created to control the driver. This block will be the main block of the blockset and it will have to be used just once for each CANopen node in the model (the blockset will support creation of more nodes as BOA has 2 CAN ports). This block will read the EDS file defining the node behavior and set the block configuration according to the file content. Moreover, inputs and outputs of the block will match the PDO data messages mapping and provide direct access to synchronously transfered data objects.

Support for synchronization messages generation will be provided by a special block which will send the message in each simulation step. Asynchronous events like node state change, emergency message reception, etc. are handled by callback functions in CanFestival driver. Callbacks block will convert these functions to Simulink function call signals that will activate particular Function call subsystems. Callbacks parameters will be provided by parameters parser blocks to the model.

The blocks described above provide the ordinary working of the node. Asynchronous operations like SDO messages, Network management, etc. will be supported by an Asynchronous block (section 3.8). This block will perform a defined set of asynchronous operations. It will be designed mainly for use in the function call subsystem to handle events.

1.3.3 Simulation support of CANopen blocks

The main goal of CANopen blockset project is to create a tool for code generation for model using CANopen communication. However, it is very useful to provide at least a basic simulation capability to enable controller tuning just in Simulink while designing the embedded device. There are a few ways how to handle the simulation. They differs in the level of CANopen network simulation.

The most universal way of network simulation is the simulation of CAN bus and the complete message traffic. This would enable to simulate all the CANopen features including Network management, emergency messages and so on. On the other hand, this simulation support would take a lot of work on something which is not a goal of the project at all as we do not want to create CANopen network simulator.

The other way is based on simulation inputs and outputs of particular blocks. It means that if some block reads some value from network communication and provides it at his output, a simulation input will be created and its value will be used instead of the received message in case of simulation. Almost all data transfers can be simulated using this method, however, the nodes do not use any object dictionary to store their values. It means that the data integrity between nodes in the model is not ensured during simulation.

The compromise solution is to enhance the previous method by storing object dictionary content of each node into the Matlab workspace. CANopen node and Asynchronous blocks will work with these data objects. The remote node can be simulated by using extra CANopen node block, which will create its OD in the workspace and other block can simulate reading or writing to the remote node OD. Simulation of this type will be used in the blockset. Read particular sections in block descriptions to learn more about the simulation.

Chapter 2

Linux ERT Target

2.1 Introduction

Real Time Workshop embedded coder is a Matlab tool which provides the code generation of any Simulink model. This is very useful for quick application development. User can create, for example, some controller model in Simulink and just click *Generate code* to get source code of the controller in C language. Obviously, the code has to be customized for particular hardware and eventually operating system (Target with upper case *T*). This customization of generated code is controlled by targets (target with lower case *t*). Matlab contains a set of targets. Each target defines some platform (Target) which it is designed for (PC + Windows, Unix, etc.).

Linux ERT target is based on generic Unix target and is design to control code generation for Target platform BOA5200 computer [AM] and Linux. This computer is equipped by two CAN peripherals [CiAa], so the target contains support for CANopen blockset. It is Simulink blockset which integrates the CANopen [CiAb] driver to the generated code and enables the user to develop embedded applications using CANopen communication. However, the blockset is described in a separate documentation. The simple diagram of Real Time Workshop principle is shown in the figure 2.1.

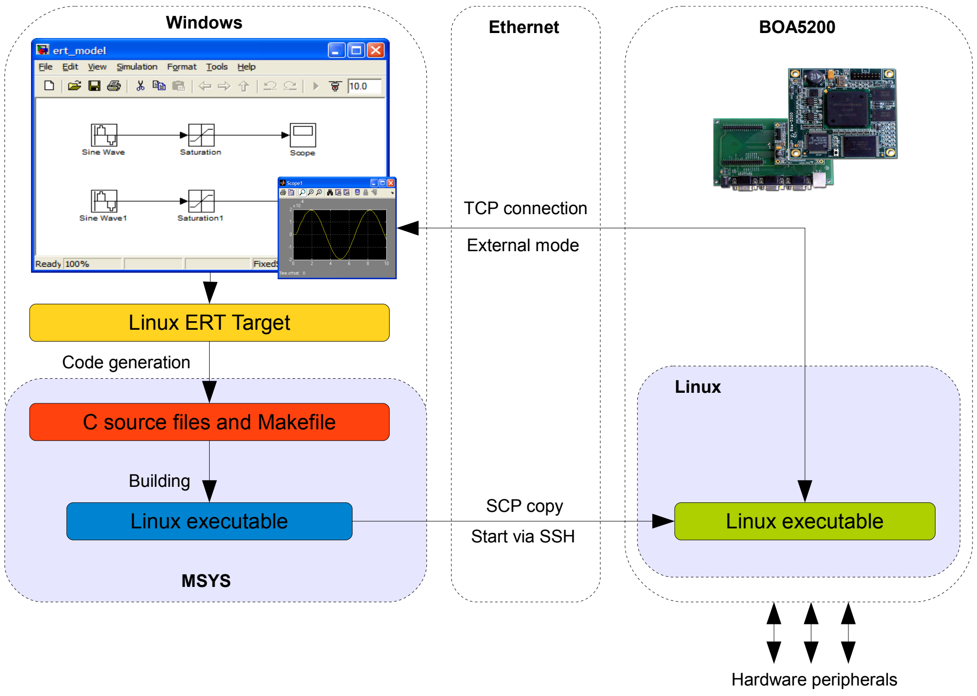


Figure 2.1: Real Time Work Target principle

This document describes everything which is necessary to know for getting started working with the target. The first chapter 2.2 describes the target installation and installation of all supporting tools like compilers on Windows host PC. Additionally, the BOA software equipment is defined and its preparation is described. The architecture of the target itself, including particular files description (section 2.3), and generated code appearance (section 2.4) is explained later in the document. Finally a simple tutorial (section A) on using the target is attached to make the first steps with target as easy as possible.

2.2 Target installation

The Linux ERT Target is designed to be used in Windows operating system. However, the Target machine BOA5200 [DoCE] is running OS Linux so that the generated code has to be compiled for Linux and PowerPC processor (BOA5200 is based on the PowerPC5200). This type of compilation is called *cross-compilation*. The tool chain for building programs for BOA target was prepared at the Department of Control Engineering at CTU [DoCE]. Pavel Jelínek has ported this tool chain for working at MinGW environment as a part of his Master thesis [Jel08]. I have just used the complete parts to enable compilation and execution of code generated by this target.

In the aim of creation the complete documentation for the target, I will describe particular steps of installation everything needed for the work with target and BOA computer.

2.2.1 Installation prerequisites

For work with Linux ERT target you have to have PC with Matlab installed. The blockset has been developed and tested at the Matlab version 2008a. However, it should work at the newer versions as well. The target was developed and tested in the Matlab of release 2008a. The Matlab installation has to contain Simulink and Real Time Workshop Embedded Coder tools. The target distribution should contain installation files of MinGW environment, compiler tool chain and BOA Linux image. Bellow is the list of all files necessary to install and use the target. These files should be attached to the target distribution.

- Tftpd32-3.22-setup.exe
- winscp421.exe
- gcc-powerpc-603e-linux-gnu-addwin.tar
- gcc-powerpc-603e-linux-gnu.tar
- MinGW-5.1.3.exe
- minires-1.01-1-MSYS-1.0.11.tar.bz2
- MSYS-1.0.10.exe
- openssh-4.6p1-MSYS-1.0.11.tar.bz2
- openssl-0.9.8e-3-MSYS-1.0.11.tar.bz2
- zlib-1.2.3-MSYS-1.0.11.tar.bz2
- MPC5200-2007.02.01-redbootROMRAM.srec
- myfs.jffs2
- zImage.elf

2.2.2 MinGW environment installation

MinGW is a collection of freely available and freely distributable Windows specific header files and import libraries, augmenting the GNU Compiler Collection (*GCC*) and its associated tools (*GNU binutils*) [min]. It is supplemented by *MSYS* which is a *Minimal SYStem* providing a POSIX compatible Bourne shell environment, with a small collection of UNIX command line tools [min]. These tools together provide the Windows compatible environment of GNU utilities for application building and necessary file management.

There are steps of MinGW installation and its setup to fit all our needs in the following text. All the files mentioned bellow should be present nearby this document in *software* folder. If their are not, it is no problem to download them from the Internet. Moreover, you may want to use newer versions of the programs that will have been probably released.

Note I recommend to install everything just into *C:\* or into folder without space or diacritic marks in its path at least. This could raise some problems.

1. Run *MinGW-5.1.3.exe* and install MinGW in its minimal configuration. No language support is necessary. Note that this installer will download MinGW from the Internet during installation.
2. Run *MSYS-1.0.10.exe* and install MSYS. Proceed the post-install guide started at the end of the installation.
3. Unpack *minires-1.01-1-MSYS-1.0.11.tar.bz2* to the MSYS installation folder (e.g. *C:/MSYS/1.0*).
4. Unpack *openssh-4.6p1-MSYS-1.0.11.tar.bz2* to the MSYS installation folder.
5. Unpack *openssl-0.9.8e-3-MSYS-1.0.11.tar.bz2* to the MSYS installation folder.
6. Unpack *zlib-1.2.3-MSYS-1.0.11.tar.bz2* to the MSYS installation folder.

Note Unpacking *.tar.bz2* archives in Windows can be done e.g. in Total Commander [Ghia] while having the proper packer plugin installed [Ghib]. The other way ho to unpack these archives is to just copy them in the proper location in Windows, run MSYS console and unpack them by the *tar* command.

```
$ tar -xjf soubor.tar.bz2
$ tar -xvf soubor.tar
```

Cross-compiler installation

The working MinGW and MSYS environment with SSH communication support is installed now. Steps of installation the cross compiler for PowerPC machine and Linux will mentioned bellow. The tool chain creation and customization is described in [Jel08] and [DoCE]. I will describe just using the prepared archives here.

1. Unpack *gcc-powerpc-603e-linux-gnu.tar* to the MinGW installation folder.
2. Unpack *gcc-powerpc-603e-linux-gnu-addwin.tar* to the MinGW installation folder.
3. Add the compiler path to the system paths by adding the following line at the end of the file */etc/profile*.

```
    export PATH=$PATH:/mingw/usr/bin
```

2.2.3 Linux ERT Target installation

Installation of the Linux ERT Target itself is very simple. Just unpack the *linux_ert_target.zip* somewhere and run *linux_ert_target_setup.m* script in Matlab. It will add the target path into Matlab paths. The target should be visible in the Simulation configuration dialog *Real Time Worshop* pane of the Simulink model after restarting Matlab. This will install just the target without CANopen blockset which has to be installed separately. Without having the blockset installed, the CANopen support option must not be enabled.

2.2.4 BOA machine preparation

The target is designed to work with BOA5200 computer [AM] based on PowerPC processor. From the software point of view, there has to be OS Linux with SSH support and SocketCan [Ber] CAN bus driver installed. The SSH support is secured by adding DropBear SSH server [Joh] into Linux file system. The SocketCan driver is included in the Linux kernel. Creation of the Linux kernel and file system is described in [Jel08] and [DoCE] again. The Linux target should be supplemented by prepared Linux kernel and file system image and I will only describe the steps of installing these images into the BOA computer.

The images will be transfered via TFTP service which is much faster than serial bus. We will set up the network connection and install TFTP server now. First of all, install the TFTP server by running the file *Tftpd32-3.22-setup.exe*, start it and set its work directory to the directory *BOA5200* of the target distribution.

We will configure the BOA network connection now. First, connect the BOA computer to the host PC by RS232 cable. Open Windows Hyperterminal with this configuration: *Baud Rate: 38400, Bits: 8, Parity: No, Stop Bits: 1* and turn the BOA on. Something should be written at the console. Wait until the information about executing the boot script appears and press *Ctrl+C* to avoid the script execution. Now the RedBoot console should be active.

RedBoot [Hat] is a software started just after computer power up and it is used to maintain the flash memory and boot the operating system. It is required to have RedBoot software in version 2007-02-01 or later to run Linux kernel 2.6. You can recognize the current RedBoot version from console output after BOA power up. If RedBoot version of your BOA is 2007-02-01 or later, you can skip the RedBoot update section. You will have to configure your RedBoot to connect to the network. The best way how to do it is to follow the steps later in this chapter.

Updating RedBoot

The following paragraph summarizes the steps of updating RedBoot according to [DoCE] or [Jel08]. You should have your BOA running the RedBoot console and the TFPT server started at our host computer. Note, that the RedBoot console is not case sensitive, but the image names are! Mind the RedBoot image name case.

1. Type this command to the console

```
RedBoot> load -v -b 0x100000 MPC5200-2007.02.01-redbootROMRAM.srec
```

2. After finishing the transfer save the new RedBoot image to the flash memory and reset the computer by typing these commands

```
RedBoot> fis cr RedBoot
RedBoot> reset
```

Now, the computer should started with the new RedBoot version. Avoid running the boot script and continue with installing the Linux

Installing Linux kernel and file system

To install Linux at the BOA computer it is necessary to store kernel and file system images into the flash memory and set up the RedBoot to boot Linux after startup. The following steps describe the file transfer using TFTP and their storage into the memory.

1. Format BOA flash and clear RAM by these commands

```
RedBoot> fis init
RedBoot> mfill -b 0x100000 -l 0xFF0000 -p 0xFFFFFFFF
```

2. Transfer the file system image *myfs.jffs2* by typing this command.

```
RedBoot> load -v -r -b 0x100000 myfs.jffs2
```

3. After successful finish of the transfer save the image into the flash memory.

```
RedBoot> fis cr JFFS2 -l 0xFF0000
```

4. Transfer the kernel image *zImage.elf* by typing this command.

```
RedBoot> load -v -b 0xFF0000 zImage.elf
```

5. After successful finish of the transfer save the image to the flash memory.

```
RedBoot> fis cr Linux
```

The Linux kernel and file system are stored in the flash memory and Linux can be started by typing following commands.

```
RedBoot> fis load Linux
RedBoot> exec
```

It is possible to write these commands to the RedBoot boot script and enable automatic starting of Linux. This is described in the next paragraph.

Configuring RedBoot

RedBoot configuration can be performed by typing *fconfig* command in the RedBoot console. It starts simple configuration dialog. The following listing shows the example settings. I have added comment lines started by # to explain particular values.

```
RedBoot> fconfig
# enable boot script
Run script at boot: true
# old boot script commands
Boot script:
.. fi lo Linux
.. ex
Enter script, terminate with empty line
# loading Linux image from flash
>> fi lo Linux
# executing the image
>> ex
>>
# timeout before starting the boot script in RedBoot
Boot script timeout (1000ms resolution): 5
# dynamic IP address assignment (DHCP) is disabled
Use BOOTP for network configuration: false
# IP address of the network router
Gateway IP address: 192.168.123.254
# IP address of BOA computer, will be used by SSH connection
Local IP address: 192.168.123.199
Local IP address mask: 255.255.255.0
# IP address of the computer running TFTP server
Default server IP address: 192.168.123.101
FEC Network hardware address [MAC]: 0x00:0x00:0x00:0x00:0x00:0x03
GDB connection port: 9000
Force console for special debug messages: false
Update RedBoot non-volatile configuration - continue (y/n)? y
# reset the Target
RedBoot> reset
```

With this configuration BOA will set its IP address permanently and start Linux. Mind that it is necessary to set the same IP address in the Linux configuration file once more according to the next paragraph.

Customizing Linux

Some more settings is necessary to perform in the Linux itself to make network connection and SSH server working. First of all edit */etc/init.d/config-eth0* by *vi* editor which is installed in the BOA file system and change the IP address to match the local address configured in RedBoot.

Note There are two ways how to create and edit files and directories in the Target file system. You can connect via SSH to the Target and use standard Linux shell commands and *vi* editor which is installed at the Target. The other way is to use the WinSCP program at your host PC and connect it to the Target using *root* user. It opens the Target file system in the window and supports file manipulations in the similar way as the Total commander does.

Then generate the keys for SSH server by typing following set of commands.

```
$ cd /etc/ssh
$ /usr/local/bin/./dropbearkey -t rsa -f dropbear_rsa_host_key
$ /usr/local/bin/./dropbearkey −t dss −f dropbear_dss_host_key
```

You can change the supervisor password by typing the command *passwd root.*

We will create user profile for executing the generated programs at the BOA machine. However, setting thread priority in the main function (see section 2.4.1) requires root authority, so the program will work but all the threads will have the same priority which is not optimal especially in the multitasking mode. The solution is to execute the program from the root profile. The following set of commands creates user called *rtw* and set the folder */home/rtw* to be its home folder.

```
$ cd /
$ mkdir home
$ cd home
$ mkdir rtw
$ adduser −h /home/rtw rtw
$ chown rtw:rtw rtw
```

After the next boot up, the network should work correctly and it should be possible to connect to BOA computer via SSH service as both *root* and *rtw* user.

Finally, the CAN ports have to be configured to use 1Mbps bit rate and to be turned on after the Linux boot up. To do this create script called *can* in */etc/init.d* with the following content.

```
#!/bin/sh
echo 660000 >/sys/class/net/can0/can_baudrate
echo 660000 >/sys/class/net/can1/can_baudrate
ifconfig can0 up
ifconfig can1 up
```

To run the script while Linux is booting enable *can* script execution by typing *chmod a+x /etc/init.d/can* and add the following line into */etc/inittab.*

```
::once:/etc/init.d/can
```

How to make the generated model to be started automatically

While developing embedded application, it is often required to make it running at the Target machine without having the host computer connected. It means that it should start up just after the machine boot up without running the *go* script and connecting the *external mode* at the host computer.

The model has to be generated without *external mode* support. It can be set in model configuration parameters *Real Time Workshop/Interface* pane. We have to upload the model executable at the Target after the generation. This is done automatically by the *go* script just after the code compilation. Once the executable file is placed in the Target file system, we can create a simple script which will start it after each boot up. Open WinSCP application and connect to the Target as the *root.* Create a new file called *run* in the root directory and open it for editation (*F4*). Copy the following commnands into it.

```
#!/bin/sh
while true
do
```

```
/model -tf inf
done
```

Note */model* is the full path of the model executable. The file name is the same as the Simulink model name. The path is the root directory / in case of using *root* user name in Simulink model configuration parameters *Real Time Workshop/Target* pane. In case of using *rtw* user name, the file will be copied into the *home/rtw* directory and this path has to be used in the command.

Now, save the file and enable its execution by changing the permissions in its *Properties* in WinSCP or by typing *chmod a+x /run* command in the Target shell.

Finally, the following line has to be added into the */etc/inittab* file at the end of *::once:* section.

```
::once:/run
```

2.3 Target architecture

I will describe particular files of the ERT target in this chapter. What the files do and when they are called are the main things that is necessary to know to understand the target architecture. It is usual to name all the target files with the prefix same as the target name. It means that in our case all the files has the prefix *linux_ert_target*.

The diagram of all the target scripts is in the figure 2.2. It shows the process of target selection and configuration customization by user in the upper part. The code generation process is shown in the lower part of the figure.

2.3.1 Target configuration scripts

This group of scripts performs the target integration into Matlab environment and customization of the configuration dialog. It also defines the variables used by code generator and set them to the default values.

linux_ert_target.tlc

The basic file of the whole target is *linux_ert_target.tlc* which defines the appearance of the Simulation configuration dialog of the Simulink model while the target is chosen for the code generation. This file was created by customization of *matlabroot/rtw/c/ert/ert.tlc* file which defines the generic ERT target and is set to be an inheritor of it. It means that it inherits the settings of the generic ERT target and adds the specific features of the Linux ERT target.

sl_customization.m

This file contains the *sl_customization* function which adds TCP/IP communication support for external mode into the target. It has to be placed somewhere in the Matlab path and it is invoked by Simulink during the startup.

linux_ert_target_select_callback_handler.m

Select callback script is called after selecting the target in configuration dialog. This script set up default values of configuration and disables some of them to be changed by user. The aim of this is to set up as much as possible of the configuration automatically and disable the user to set unsupported values.

linux_ert_target_canopen_support_callback_handler.m

This is the callback script after changing the value of *Enable CANopen support* in the *Real Time Workshop/CANopen support* configuration pane. It tries to search the CanFestival *lib* and *include* folders. They are placed in the *CANopen blockset* folder and if it is correctly installed (the blockset folder is in the Matlab path), it should be found. The searching is based on file called *cfpointerfile.txt* which is placed in the *canfestival* blockset subfolder. Its only purpose is the path searching.

The *canfestival/include* folder contains the header files of CF API that are necessary for compilation of program using CanFestival. The folder has to be added into include path for compilation in Makefile if the CANopen blockset support is enabled. The *canfestival/lib* folder contains the CanFestival libraries that are necessary for program linking. If the

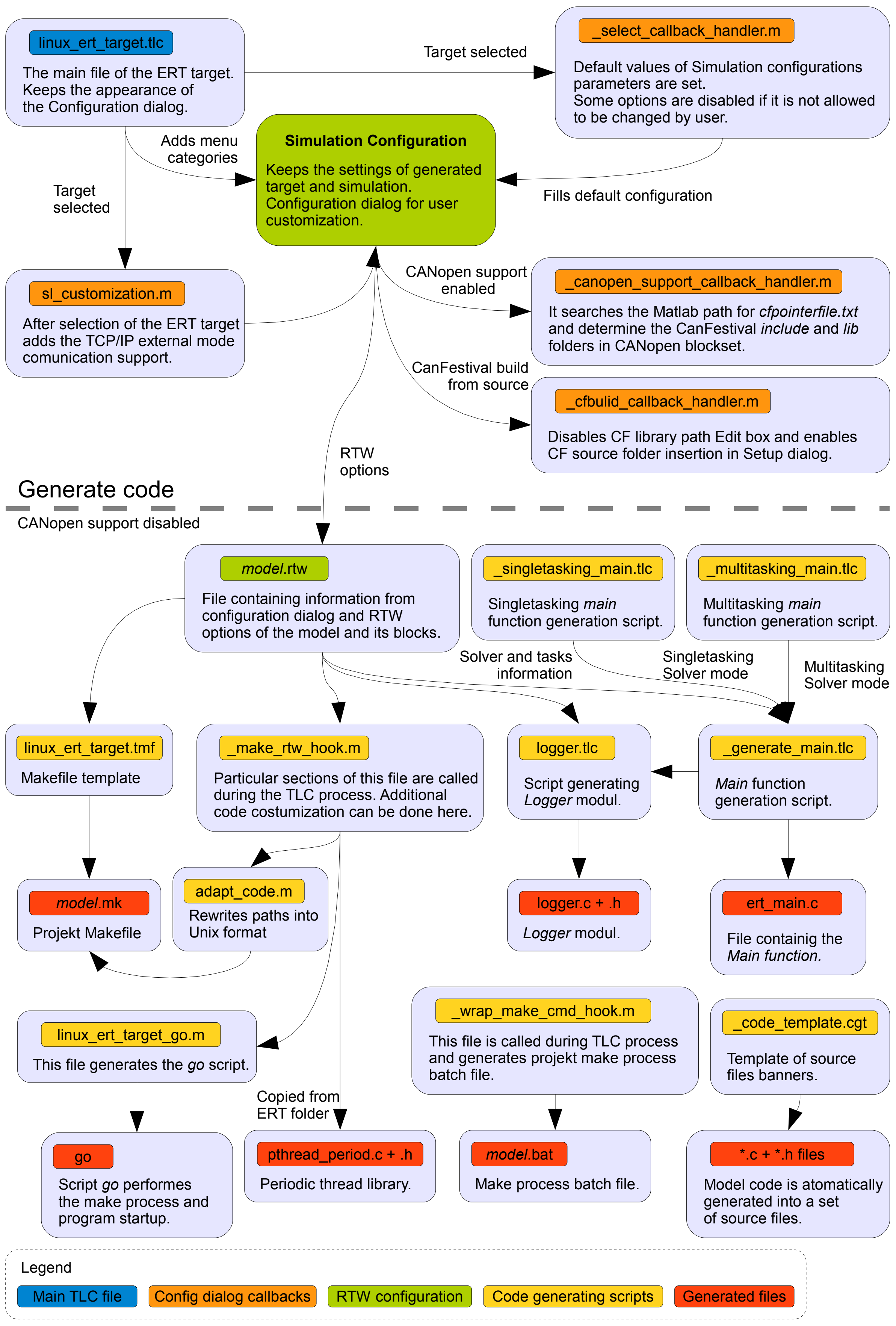

linux_ert_target.tlc
The main file of the ERT target. Keeps the appearance of the Configuration dialog.
Target selected
_select_callback_handler.m
Default values of Simulation configurations parameters are set. Some options are disabled if it is not allowed to be changed by user.
Adds menu categories
Simulation Configuration
Keeps the settings of generated target and simulation. Configuration dialog for user customization.
Fills default configuration
Target selected
sl_customization.m
After selection of the ERT target adds the TCP/IP external mode comunication support.
CANopen support enabled
_canopen_support_callback_handler.m
It searches the Matlab path for cfpointerfile.txt and determine the CanFestival include and lib folders in CANopen blockset.
CanFestival build from source
_cfbulid_callback_handler.m
Disables CF library path Edit box and enables CF source folder insertion in Setup dialog.
RTW options
Generate code
CANopen support disabled
model.rtw
File containing information from configuration dialog and RTW options of the model and its blocks.
_singletasking_main.tlc
Singletasking main function generation script.
_multitasking_main.tlc
Multitasking main function generation script.
Solver and tasks information
Singletasking Solver mode
Multitasking Solver mode
linux_ert_target.tmf
Makefile template
_make_rtw_hook.m
Particular sections of this file are called during the TLC process. Additional code costumization can be done here.
logger.tlc
Script generating Logger modul.
_generate_main.tlc
Main function generation script.
model.mk
Projekt Makefile
adapt_code.m
Rewrites paths into Unix format
logger.c + .h
Logger modul.
ert_main.c
File containig the Main function.
linux_ert_target_go.m
This file generates the go script.
_wrap_make_cmd_hook.m
This file is called during TLC process and generates projekt make process batch file.
_code_template.cgt
Template of source files banners.
Copied from ERT folder
go
Script go performes the make process and program startup.
pthread_period.c + .h
Periodic thread library.
model.bat
Make process batch file.
*.c + *.h files
Model code is atomatically generated into a set of source files.
Legend
Main TLC file
Config dialog callbacks
RTW configuration
Code generating scripts
Generated files

Figure 2.2: Target architecture with scripts calling diagram

Build CanFestival from source option is enabled this libraries are not used and they are obtained by compilation the CanFestival source code.

linux_ert_target_cfbuild_callback_handler.m

This callback function is invoked by changing the *Build CanFestival from source* option and it only switches between insertion of CanFestival library folder or source code folder. Moreover, it enables or disables *CAN message console logging* option as the CanFestival has to be built from source to turn it on.

2.3.2 Code generating scripts

This group of scripts is used by Real Time Workshop during the code generation. The code generation has predefined some scripts that are called in specified phases of the process.

Before describing the target scripts I will mention the *model.rtw* file (*model* is generally the name of the model so that it may be called differently). This file is generated automatically at the beginning of the TLC process and it contains all the information about model, its blocks and simulation setting necessary for program generation, compilation and execution.

linux_ert_target_generate_main.tlc

This is the TLC script which generates the *main* function file *ert_main.c* according to solver mode and sample times settings of the model. This file is registered in the *Real Time Workshop/Templates* pane in the Simulation configuration. It is set by *linux_ert_target_select_callback_handler.m* script and the user change is disabled.

This script creates the *ert_main.c* source file and calls TLC function for single or multi-tasking solver mode. Finally, it calls TLC script for generation of the system logger module.

linux_ert_target_singletasking_main.tlc

This script generates the content of the *ert_main.c* file for the singletasking mode. All the model code is executed from a single thread running at the base sample time. The single tasking mode differs according to the number of sample times used in the model.

Singletasking-singlerate mode means that just one sample time is used. Singletasking-multirate mode means that more sample times are used in the model but everything has to be executed from a single thread. It is handled by the rate scheduler which is automatically generated into the *model.c* file and the main function is the same for both single and multi rate case.

Read in Code generation chapter for more information about the singletasking main function.

linux_ert_target_multitasking_main.tlc

This script generates the content of the *ert_main.c* file for the multitasking mode. The multitasking mode is only reasonable (and allowed as well) for the model with more different sample times. The multitasking mode is realized in the way that each of the sample times has separate thread where its loop is running. The cycles of each loop is controlled by the main loop running at the base sample time (the greatest common divisor of all sample times).

Read in Code generation chapter for more information about the multitasking main function.

logger.tlc

The logger module of the generated code performs runtime logging of the program events. It supports two types of logging - console output and log file output. Both types can be disabled or set onto three different levels according to the amount of logged messages. This is set in the *Real Time Workshop/Runtime logging* pane in the configuration dialog. This file generates the source and header files of the logger module.

linux_ert_target.tmf

This file is the template of the Makefile used for building the generated program. The result of this is the Makefile *model.mk*, where *model* is the name of the model. The template is based on the generic template for Unix ERT target which can be found in *matlabroot/rtw/c/ert/ert_unix.tmf*. The template is only changed to enable linking CanFestival driver into the program. Linking of the driver is subject to the *Real Time Workshop/CANopen support* settings in the Simulation configuration.

If the CANopen support is disabled, the Makefile does not need any parts of the driver for linking. If it is enabled and it is not required to build the driver from its source code, it adds include folder to the *include* paths and *lib* folder to the libraries for linking. These folders are set in the configuration dialog. If building the driver from source is required *(*configure) and *make* commands are applied at the inserted source folder. Then the libraries obtained by actual compilation are used for the final linking.

linux_ert_target_make_rtw_hook.m

File with this name is called during the code generation process if it exist in the target path. It contains sections that are called in a different moments of the generation. It performs several tasks. It rewrites CanFestival folders paths into old DOS format by *rtw_alt_pathname* Matlab function. This command removes spaces and diacritical marks from the given paths.

The copying of several source files and libraries is performed after the TLC process. The source and header files of periodic thread library are copied into generated code folder. If the CANopen support is enabled and build CanFestival is not required three library files of CanFestival driver are copied there as well.

Finally there are executed to scripts in the *exit* section. The script *adapt_code.m* customizes generated Makefile for running in MSYS environment and the script *linux_ert_target_go.m* generates the *go* script.

adapt_code.m

This script customizes already generated Makefile *model.mk* for running in the MSYS environment. It rewrites all backslashes \ with slashes /. All the paths are rewritten into the Unix format this way. However, there are some backslashes in the Makefile used as line continuation mark that have to be kept. It is secured in the way that all these backslashes are represented by *BACKSLASH* keyword in the Makefile template. After rewriting the paths this keyword is replaced by regular \ mark.

linux_ert_target_go.m

This script creates the *go* script and place it into the working directory. This script is the only user interface of program compilation and execution. Its content differs according to the settings in *Real Time Workshop/Target* configuration pane. There is set in this dialog whether the project should be just compiled, compiled and copied onto the Target machine or executed at the Target machine as well. Information about the Target machine IP address is used as well. Moreover if CANopen support is enabled the script inserts into the *go* script commands for copying the canfestival dynamic library onto the Target computer.

linux_ert_target_wrap_make_cmd_hook.m

The *model.bat* file is generated by Real Time Workshop. This file contains the execution command of *make* program. This is done automatically, but if there is the *linux_ert_target_wrap_make_cmd_hook.m* script in the target path, it is called instead and the *model.bat* content can be customized.

linux_ert_target_code_template.cgt

This file contains template for source file banner of all the generated *.h* and *.c* files. It is used by Real Time Workshop while generating the files.

2.4 Generated code

This chapter describes the code generated by Real Time Workshop using Linux ERT target. The folder *model_linux_ert_rtw* is created in the working folder and all the generated files are placed into it. Generally, there are several types of generated files. First of all the file *ert_main.c* with main function contains the code used for model synchronization. Additionally the files containing the code of all the blocks used in the model are generated. The interface between the step synchronization in the main function and calculations of the model blocks is created in the file *model.c*. This is an interface between Linux ERT target specific code (main function) and general block code generated by RTW automatically as well.

Finally the Makefile (*model.mk*) and make execution batch file (*model.bat*) are generated. In the working directory, the *go* script is created to enable the user to run building and execution of the code by a single command.

2.4.1 Main function

The main function is completely generated by the target and is customized according to the solver mode and sample times setting of the Simulink model. The target support all the solver modes (singletasking - singlerate, singletasking - multirate and mutlitasking). However, the base part of the main file is always the same.

The base part of the main file contains three functions. The main function which performs the basic setting of the simulation and creates two threads running the *main_loop* and *rt_OneStep* functions. The main loop thread is set to have the highest priority and it produces the base sample time steps. The main loop controls the execution of step function using a semaphore. There may be more step function threads in case of multitasking mode.

Main loop

Each simulation independently of the solver mode has just one base sample time. This base sample time is the greatest common divisor of all used sample times of the model. The main loop function is started in a thread with the highest priority and it is liable for generating the base step intervals with as high precision as possible.

The timing is realized by periodic thread implemented in *pthread_periodic* library created by Pavel Píša at the Department of Control Engineering at CTU [Píš05]. This library has a very simple API for creation periodic timer and waiting for the timer expiration. It is secured by the period timer that the main loop cycle is executed just in the base sample time frequency. The most important code of the main loop is shown in the figure 2.3.

The step cycle waits for unblocking the step semaphore. This semaphore is unblocked in each main loop cycle. From the step cycle the *mode_step* function placed in the automatically generated *model.c* file is called. This function contains the code of simulation step calculations.

```
\* setting the priority is just pseudo code here *\
set_thread_priority(MAX_PRIORITY);

\* start = actual time, period = base sample time *\
pthread_make_periodic_np(pthread_self(), &start, &period);
```

```
while(!finished) {
    \* waiting for timer expiration *\
    pthread_wait_np();
    \* checking the overrun *\
    sem_getvalue(&step_semaphore, &step_sem_value);
    if (step_sem_value) {
        rtmSetErrorStatus(test2_M, "Overrun");
        break;
    }
    \* performing the step *\
    sem_post(&step_semaphore);
    sem_post(&step_semaphore);
    \* external mode communication *\
    rtExtModeCheckEndTrigger();
}
```

Figure 2.3: Main loop cycle

Overrun checking

It is obvious from the simulation synchronization principle that if the execution of the model code called from the *rt_OneStep* function took more time than is the sample time, the real time feature of the simulation would be violated, an overrun would occur. It is not possible to guarantee that the code would be fast enough and so it is necessary to check the overrun occurrence at least.

This is done by a simple mechanism. Linux semaphore can be set to zero (than it is blocked) or any positive number (than it is unblocked). Each command *sem_post()* increments the semaphore value and command *sem_wait()* decrements the value or wait for the semaphore relaxation if it is zero. The main loop unblocks the semaphore twice and so set its value to number two (see code fragment 2.3). The step cycle is unblocked, decrements the semaphore value from two to one and starts the model step code. The semaphore is decremented once more and its value is set to zero after finishing the model step (figure 2.3).

Obviously, if the semaphore value is non-zero just before start of the next simulation step, the model code does not finished yet and the overrun had occurred. The whole mechanism is shown in the figure 2.5.

```
while(!finished) {
    sem_wait(&step_semaphore);      /* sem_val = 1 */
    model_step();                   /* performing the model step code */
    sem_wait(&step_semaphore);      /* sem_val = 0 */
}
```

Figure 2.4: Semaphore decrement in the step cycle

Singletasking case

The singletasking - singlerate case means that the model uses just one sample time for all its blocks. The code for calculation of the simulation step is generated into a single function *model_step()* located in the *model.c* source file. This step function is called from *rt_OneStep()* thread loop. This loop is synchronized from the main loop running in the period thread using a semaphore. The mechanism for singletasking case is shown in the figure 2.5.

The case that more than one sample time is used in the model and the solver mode is

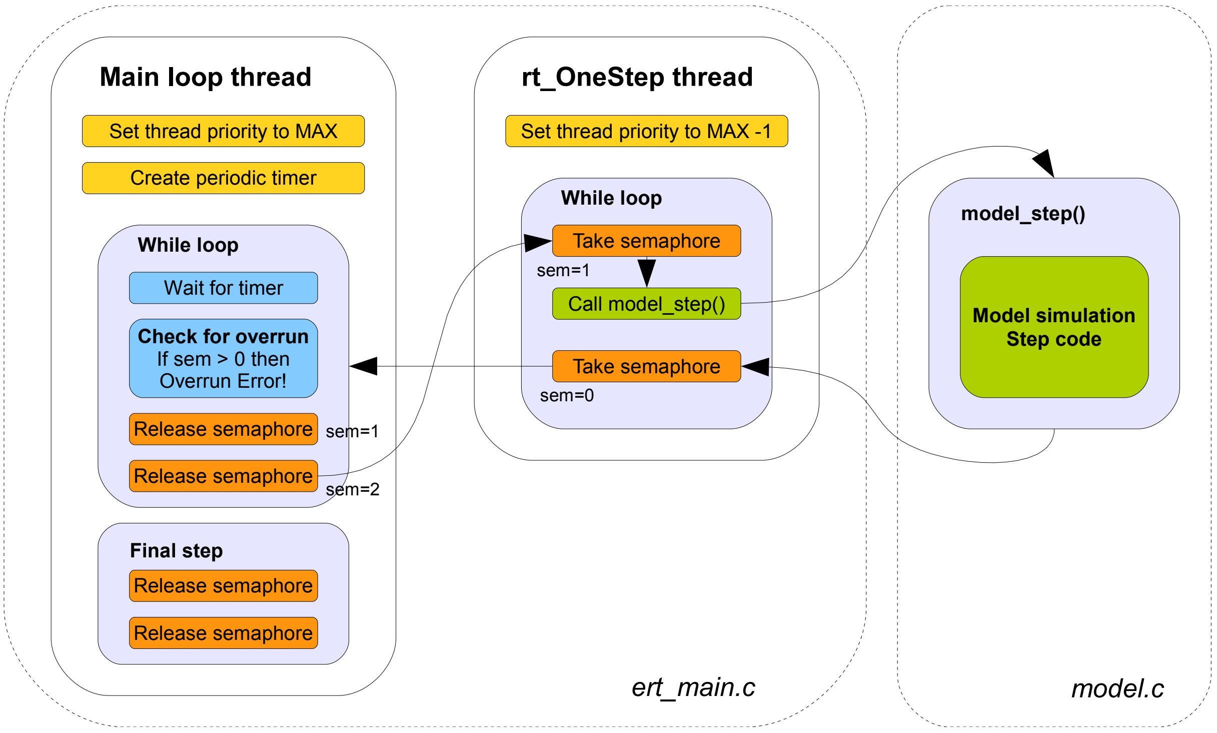


Figure 2.5: Overrun checking

required to be singletasking is called singletasking - multirate. The main function is the same as in the classic singletasking mode and the rate scheduling of the particular sample times is performed in the *model.c* file. The diagram is shown in the figure 2.6. Note that double using of semaphores (described in Overrun checking paragraph) is abandoned in this figure to simplify the diagram.

Multitasking case

Usually if more than one sample times are used in the model, the multitasking solver mode is chose. The advantage over the singletasking - multirate case is that each sample time operations are running in the separate thread. The threads are generated into the *ert_main.c* file. Each contains the loop which is synchronized by a semaphore.

The rate scheduling is controlled by the main loop with calling scheduler functions from *model.c*. The main loop is running at the highest (base) sample time. It calls the *model_SetEventsForThisBaseStep(eventFlags)* function in each cycle. It fills the array of logical variables *evenFlags* so that it contains an information whether particular sample time cycle is scheduled into this base time cycle or not. This decision have been made in the last base sample cycle by calling the rate scheduler inside the *model.c* file.

While having the information whether to run particular sample time steps or not, appropriate semaphores are released. Base sample time (always numbered as 0) is performed in each main loop cycle. The diagram of function calls in multitasking model is shown in the figure 2.7. Note that double using of semaphores (described in Overrun checking paragraph) is abandoned in this figure to simplify the diagram.

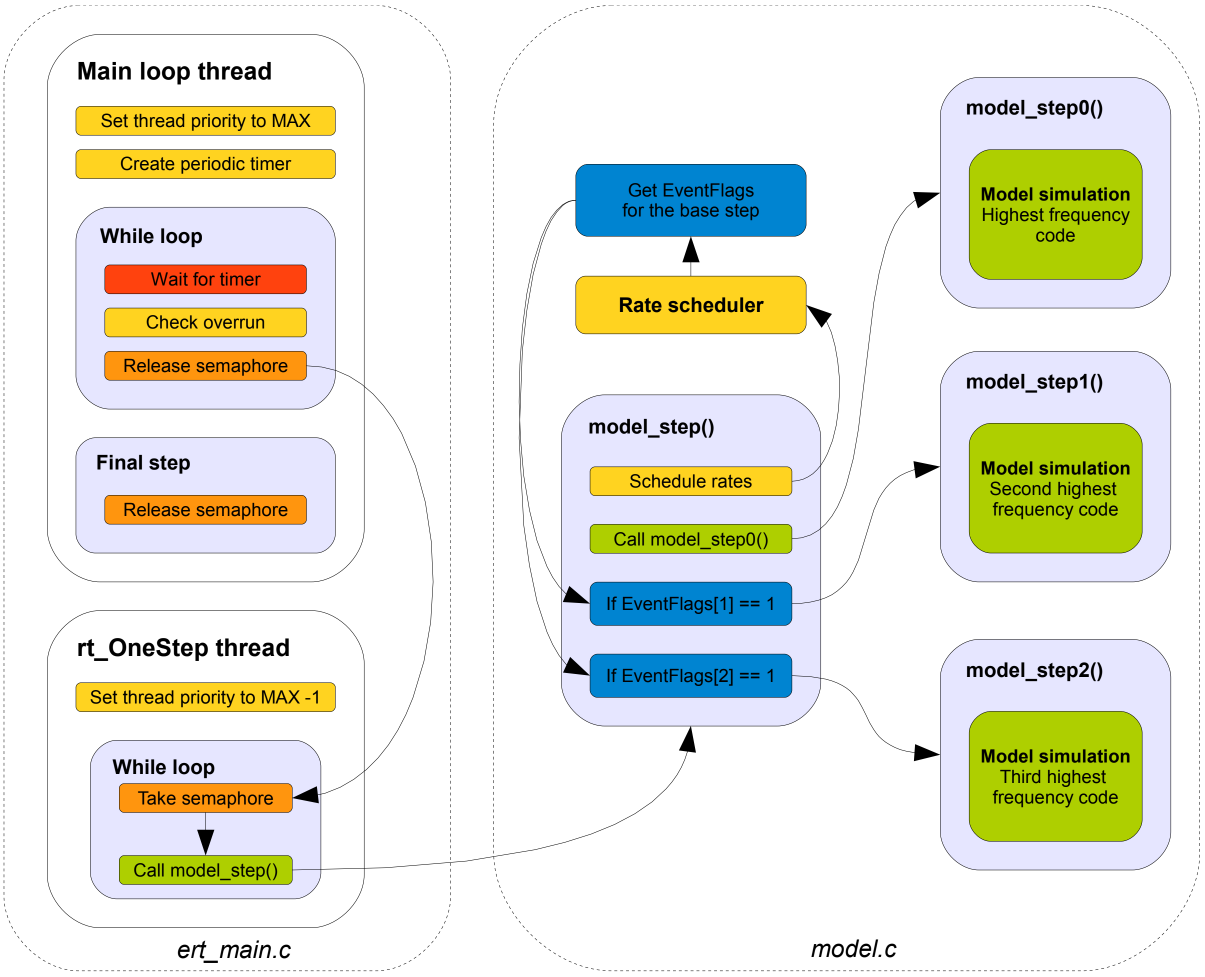


Figure 2.6: Singletasking - multirate model synchronization

Thread priority assignment

Separation of the model code into threads according to sample times was described in the previous paragraphs. The scheduling of these threads is managed by operating system, Linux in this case. The scheduling priority can be set to particular threads. Then the preemptive scheduler is able to interrupt running thread in case that some thread with higher priority is ready to run. The highest priority at all is set to the thread running the main loop 2.4.1. The second highest priority is then set to the thread running the base sample time step (*rt_OneStep*). Although both loops runs at the same frequency, the main loop controls the synchronization and needs to have the highest priority to check the eventual overrun.

Priorities of *rt_OneStepX* threads are set to match the step sample time order in the multitasking case. It means that threads running faster sample times are set to have higher priority. But still the highest priority is always set to the main loop.

2.4.2 Makefile

The Makefile *model.mk* (*model* is the model name) is created by filling the template *linux_ert_target.tmf*. Parameters stored in the RTW options configured in Simulation

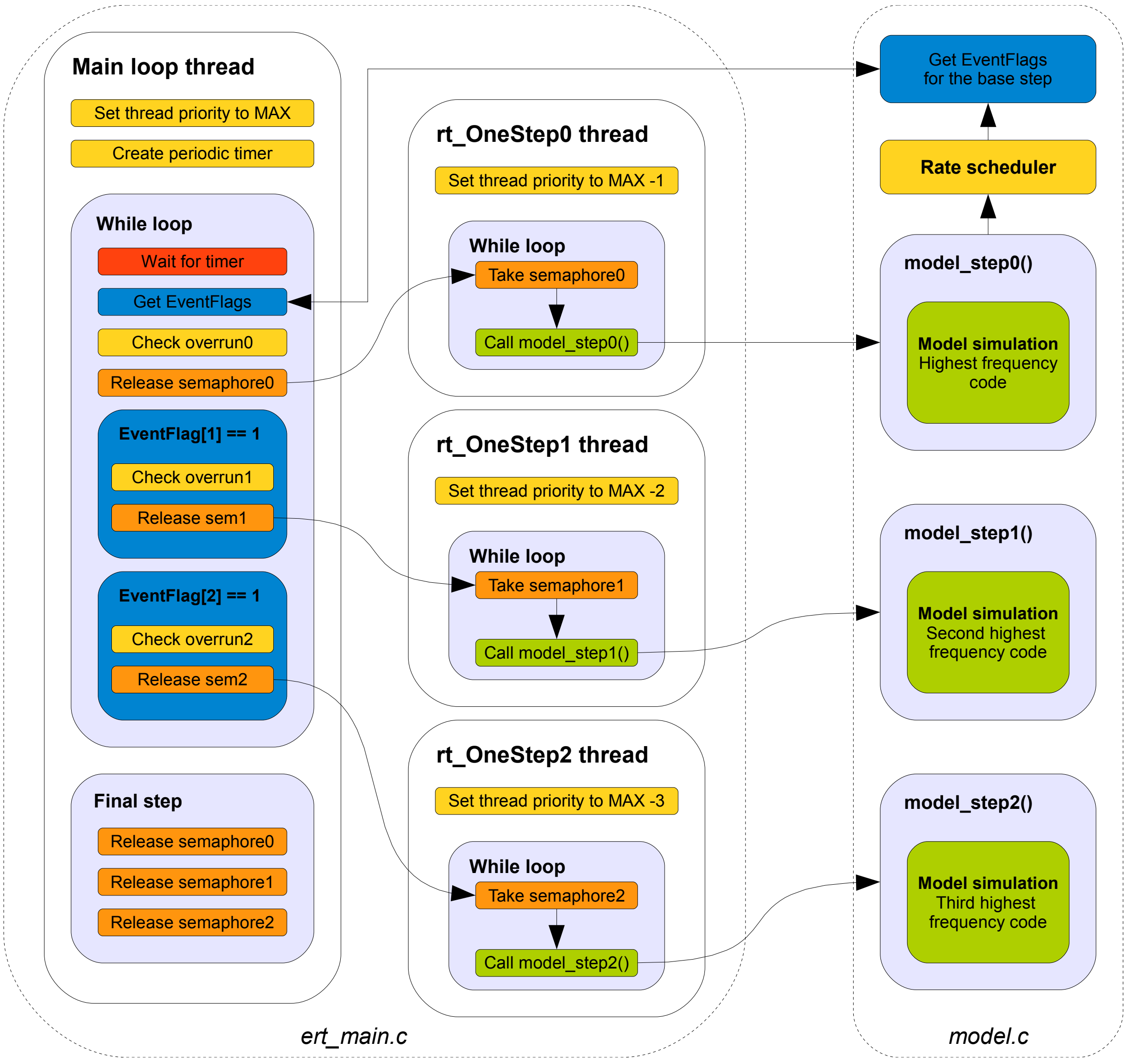


Figure 2.7: Multitasking model diagram

configuration dialog are filled into prepared variables of the Makefile. CANopen support variables were added to the general RTW options. These variables are used in the Makefile to control the compilation of code using CanFestival API and the compilation of the driver itself. CANopen support variables and their usage in the Makefile is shown in the table 2.1.

Real Time Workshop forwards CanFestival paths and Matlab paths from the configuration to the Makefile. The problem is that Windows uses backslash as folder separator and Linux and MSYS too the classic slash. All the paths forwarded to the Makefile has to be rewritten to the Unix format. MSYS is able to work with Windows address beginning with the disk identifier, the only thing which is necessary to be changed in the paths are the slashes. It means that if Matlab path *C:\MATLAB\R2008a* is rewritten into *C:/MATLAB/R2008a*, it works in MSYS environment.

The Matlab path is customized by *linux_ert_target_wrap_make_cmd_hook.m* script. The CanFestival address are rewritten in the *linux_ert_target_make_rtw_hook.m* script in the section called just before TLC process.

RTW variable	**Values**	**Influence on the Makefile**
CANOPEN_SUPPORT	0	No CanFestival parts are compiled or linked
	1	CanFestival headers and libraries are used
CANFESTIVAL_BUILD	0	CF libraries are copied from library path
	1	CF source path compiled and libraries used
CANFESTIVAL_INCLUDE	Path	Folder with CF header files
CANFESTIVAL_LIB	Path	Folder with precompiled CF libraries
CANFESTIVAL_SOURCE	Path	CF source folder prepared to *make* command
CANFESTIVAL_CONSOLE	0	No CanFestival driver console output
	1	CAN messages logging (*–debug=MSG* option)

Table 2.1: CANopen variables in the Makefile

2.4.3 Script *go* and the build process

This script is generated into the working directory. It contains commands for building the generated program, copying to the Target machine and starting it. The setting of the Target IP address and other connection information can be customized in the *Real Time Workshop/Target* pane of the Simulation configuration.

Chapter 3

CANopen blockset

3.1 Introduction

The CANopen blockset provides Simulink API of the CANopen communication protocol [CiAb]. It is designed mainly for generating the code by Linux Embedded Real Time Target which is customized for Linux OS and BOA5200 computer [AM]. The blockset integrates CanFestival driver [LOLb] into the code generated by the target. Particular blocks provide particular API functions of the driver to be used in Simulink model. It implements entire CanFestival API which enables to create CANopen node with common features.

Note The blockset has been developed and tested at the Matlab version 2008a. It should work at the newer versions as well.

3.1.1 Simulation capabilities

The blockset has the simulation support as well, but it is designed only to enable running the simulation during the embedded controller tuning in Simulink. It does not simulate CANopen network and its communication at all. The simulation support is realized mainly by the simulation inputs and outputs of the blocks that are used to obtain data from the model instead of the CANopen network during the simulation.

Moreover, *CANopen node* and *CANopen asynchronous* blocks works with Object dictionary object stored in Matlab workspace and use it for data exchange. This feature enables to simulate asynchronous data transfers via SDO messages. Finally, simulation of CANopen event callbacks including parameters parsing is supported as well. On the other hand, Network management, Heartbeat service or Network synchronization are not simulated at all. Read simulation capabilities of particular blocks to learn more about this topic.

While simulating the model in Simulink some simulation blocks are always added. While generating the code all model blocks are generated including these used just for the simulation. You can use *Environment controller* switch to avoid generating parts of the model connected to simulation ports. See blockset usage tutorial in section B.3.2 for example of doing this.

3.1.2 Blockset parts

Particular blocks of the blockset are defined in model file *linux_ert_target_blockset.mdl*. This model is a library and is set to be part of Simulink Library Browser by the code in file *slblocks.m*. It secures that the blockset will be visible in the browser and its blocks will be usable in Simulink models. Read [Mat] to understand the following text about blockset parts well.

Block mask

The block user interface enabling the parameters settings is realized by the block mask. The mask defines the appearance of the block and forwards the parameters to the block S-function. If some more complex code is necessary to process the parameters or customize the block appearance, callbacks on parameter change are stored outside the model in *mask_callbacks* folder. This folder is added into Matlab path by blockset installation script. This simplifies the code maintenance and enables code re-usage.

Block S-function

Each block has just one S-Function which is written in C language. The S-functions are then compiled into *MEX* file with the extension *.mexw32*. The S-function is a set of callback functions in fact. There are functions performing block ports and sample time settings, writing code into *.rtw* file while generating code and performing the simulation in Simulink environment as well.

Block TLC script

Each block has the TLC (*Target Language Compiler*) script called by the same name as its S-function. TLC scripts are stored in subfolder *tlc_c* and are reliable for the code generation. Similarly to the S-function, the TLC script is based on callback functions that are called by the Real Time Workshop while generating code. The code for generating source files and their content is placed in the proper callback functions. There are more TLC scripts than is the number of blocks, however just the main ones are called automatically by RTW and the other are used as libraries and called from the main scripts.

Block help

The block help is realized by *HTML* script with the same name as the S-function. The help files are linked to the block mask so that the help is accessible directly by block help button. The help is based on parameters description which is the same as Block parameters sections in this document. Images used in help pages are stored in *help_img* subfolder.

CanFestival driver

The blockset is based on CanFestival driver for the CANopen network. It is possible either build the CanFestival from source while generating code or using prepared libraries. The libraries are stored in the *canfestival/lib* subfolder of the blockset and are used while *Build CanFestival from source* option in model configuration is disabled. In the same way CanFestival header files are handled. They are prepared in folder *canfestival/include* and in case of building CanFestival, the headers from the source folder are used instead.

ObjectDictionary class

An instance of the *ObjectDictionary* class is created by each *CANopen node* block used in the model. It is stored in the workspace and keeps the node Object dictionary. The OD content is read from the EDS file and parsed by the EDS parser which is a part of the class called from the class constructor. This Matlab class is defined in the *ObjectDictionary.m* file in the blockset folder.

3.2 Installation

This chapter describes installation of the CANopen blockset and all the other programs necessary to work with the blockset. As the blockset is designed to be used with the Linux ERT Target, it is supposed to have this target already installed according to its help. The target distribution contains all the necessary tools for working with the Target computer [AM] and generating code for this platform.

3.2.1 Object Dictionary editor installation

For creating the CANopen node in Simulink model it is essential to have the EDS file and source files of the Object Dictionary prepared (see section 3.3). Creation of the EDS file and generation of the source files has to be performed by the Object Dictionary Editor distributed with the CanFestival driver sources. It is also available in the blockset distribution in the subfolder *canfestival/objdictgen*.

The editor needs to have Python, WxPython and Gnosis Utils installed for running. The following installation files should be available in the blockset distribution or you can download the current versions from the appropriate web sites.

- python-2.6.1.msi
- wxPython2.8-win32-unicode-2.8.9.2-py26.exe
- Gnosis_Utils-1.2.2.win32.exe

While having the installation file prepared you can install the Python environment by running them one after another. Then the Object Dictionary Editor can be started by running *objdictedit* command in Matlab command line. Existing *.od* file can be passed as a function parameter. Read section 3.3 for more information about working with the editor and EDS files.

3.2.2 CANopen blockset installation

The installation of the blockset itself is very simple. It has to be just unpacked somewhere and the *canopen_blockset_setup.m* script has to be performed from the Matlab command line. The script just adds the blockset folder and subfolders to the Matlab search path.

3.2.3 CANopen support in the target

The blockset can be used only with the Linux ERT Target. To enable the blockset usage in the target the *CANopen support* option has to be checked in the model configuration in the *Real Time Workshop/CANopen support* pane. If the blockset is properly installed the paths of CanFestival libraries and includes will be automatically filled in after checking this option.

The menu offers compiling CanFestival from the source as well. However, the path of the source folder is not filled automatically. The CanFestival sources should be distributed together with the target as well or it can be obtained from the CanFestival web sides [LOLb] (see section 3.2.4 for more information). If the building is required, the target writes commands for CanFestival compilation into the Makefile. It performs the *configure* script on the source folder first. The following parameters are used to set up the desired target platform.

```
./configure --timers=unix --can=socket --cc=powerpc-603e-linux-gnu-gcc --arch=ppc --os=↩
    linux --target=unix
```

Moreover, if the *CAN message console logging* option is enabled the *–debug=MSG* parameter is added. Then the *make* command is applied to the CanFestival source folder and the libraries are copied into the generated code folder to be used by linker. If you are starting work with the driver, visit [DoCE] to learn the very basics in using it.

3.2.4 Updating CanFestival version

The current version of CanFestival can be obtained from project CVS repository (visit [LOLb] for the information about the repository). Now, you can set *Build CanFestival from source* option in the configuration parameters of your model and fill the new CanFestival folder. The target is prepared to configure and build the CF source, however, there are some problems that have to solved before starting the code generation.

Firstly, the *drivers/can_socket/can_socket.c* file of CF source includes some headers of SocketCan driver distribution. This headers are placed in the *include/linux* folder of our local CanFestival source. Copy this folder into the new CF source *include* folder and you will not have to install SocketCan driver.

In the time you decide to update the CanFestival, the driver API will have been probably changed. It means that you will have to change the particular TLC scripts of the blockset to generate the API function calls according to the current prototypes. Visit [LOLa] doxygen sites for the information about the current driver API.

Finally, you will have to rebuild your Object dictionary created in the old version of *objdictedit* as it will not be probably compatible. You have to load your *.od* file into the new *objdictedit* and build the dictionary by the menu command once more.

After performing the steps described above, the blockset should work with the new CanFestival version correctly. However, according to my experience with using the driver, I do not believe it very much. While integrating the driver I had to solve some problems and fix some bugs. I could not wait until the bugs were fixed by the authors in the repository because some changes in driver API would be made as well.

The most significant bug or may be chaos is in the endianity solution. It has been changed many times, so it is not possible to follow changes until these days very well. Macros called *UNS16_LE* and *UNS32_LE* are applied to OD entries to switch the value bytes into little endian form. However, this causes that CANopen message IDs of little endian are compared with masks of big endian at machine using big endian notation and so the IDs are not recognized. Practically, it means that some messages defined to be sent are not sent and received messages are not processed. If you noticed such problems, have a look at the endianity solution and messages proceeding in the current version.

3.3 EDS file and Object Dictionary

The CANopen network is based on particular nodes. Each node has its own Object Dictionary (OD) [CiAb] which keeps all the configuration variables of the node and the attached device as well as the data defining the state of the device. Some entries (indices) of the OD have meaning defined by the CANopen standard, some are device specific. The OD fully defines the behavior of the node in the network.

The content of the Object Dictionary can be saved in a text EDS file (*Electronic Data Sheet*). Many universal CANopen devices supports loading its configuration from such EDS file. The main block of the blockset does it as well. The custom EDS file can be created by CanFestival OD editor.

3.3.1 CanFestival OD editor

CanFestival OD editor is a Python script called *objdictedit.py* which can be found in CanFestival source folder in *objdictgen* subfolder. It can be started by running *objdictedit* command in Matlab command line. Existing *.od* file name can be passed as a function parameter. Python and WxPython have to be installed according to section 3.2 to run it properly. The editor has graphical user interface for creating or customizing Object Dictionary. The OD can be then exported into EDS file and C language source and header files that contain the C structure representation of the OD and have to be linked with the generated code.

The Node name is specified in the OD editor. This name is used as a node identifier in the model in case of using more than one nodes. The name has to be the same as the name of EDS file and *.c* and *.h* source files. These files are then placed in the model folder and the name is set to all the CANopen blocks that should work with this dictionary. The example of using OD files with OD (node) name *myOD* is shown in the figure 3.1.

In case of using more than one CANopen node block in the model, EDS, *.c* and *.h* files with the unique name has to be created for each node.

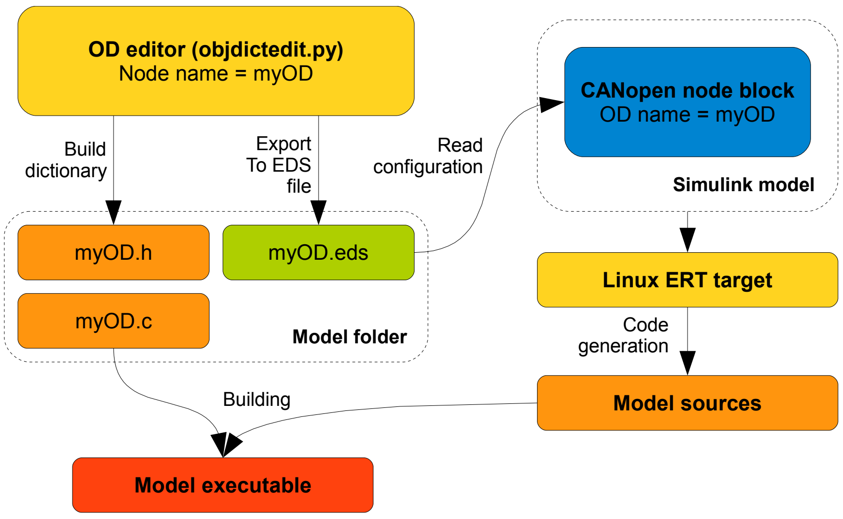


Figure 3.1: Principle of creating and using OD in the CANopen node

3.4 CANopen node block

The CANopen node block is the main block of the CANopen blockset. It keeps the Object Dictionary content of the node and it has to be used just once for each node created in the model. The dependency of each other block on the main block and OD name match is checked. This block uses its *Object Dictionary name* parameter as a base name of EDS file. It reads the EDS file and fills the block mask according to its content (see section 3.4.2). Finally, it adds the OD *.c* and *.h* files to the model source to be linked with generated code (see section 3.3).

The block is defined in *linux_ert_target_blockset.mdl* file and support files are mentioned in the table 3.1.

S-function source	canopen.c
S-function mex	canopen.mexw32
Help file	canopen_help.html
TLC scripts	canopen.tlc (main) cf_canopen.tlc
Mask callbacks	canopen_mask_edsFile.m canopen_mask_generateSYNC.m canopen_mask_initialisation.m canopen_mask_loadMapping.m canopen_mask_modelSYNC.m

Table 3.1: Block files

3.4.1 Block parameters

The following text describes particular parameters of the block mask. It is the help of using the block in the model. The parameters are configured in the block mask which is shown in the figure 3.2.

Node ID

Node ID is the ID of the local CANopen node (1 - 127).

CAN board number

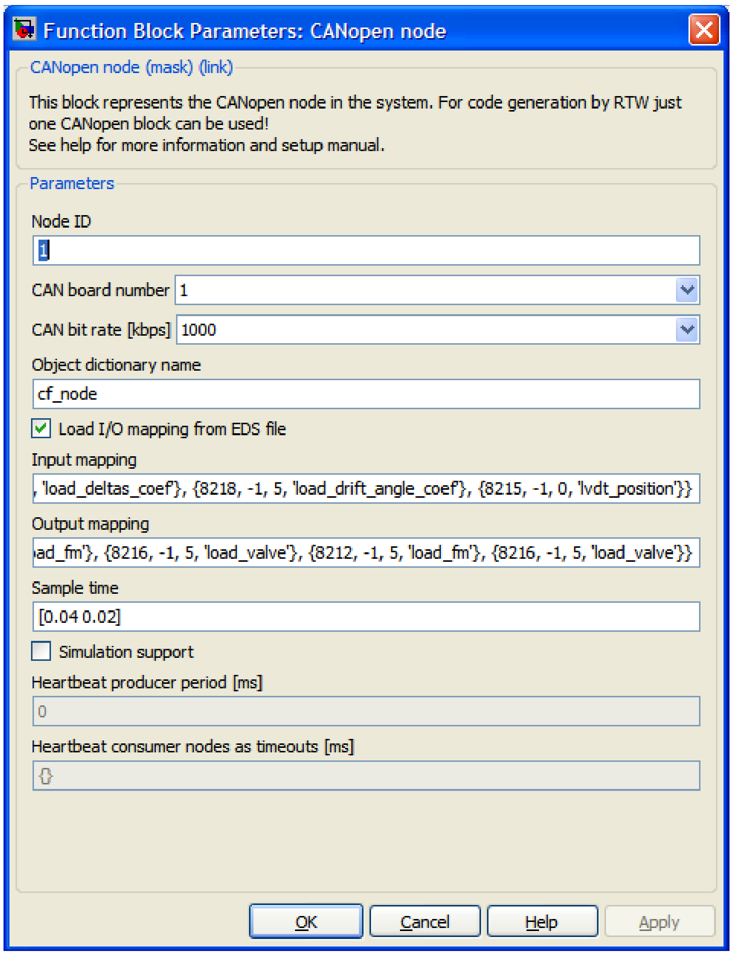
BOA5200 computer is equipped with two CAN ports. By setting CAN board parameter you choose which of these ports to use.

CAN bit rate

Bit rate of used CAN bus can be set into predefined values only.

Object Dictionary name

Name of the CanFestival OD of this node. It is either the name of *.eds, .c, .h* CanFestival files and the OD structure identifier defined there. The *EDS file* and the Object Dictionary definition *.c* and *.h* files can be edited and generated by CanFestival Objdictedit and have to be placed in the model directory. The name of the dictionary included


Function Block Parameters: CANopen node

CANopen node (mask) (link)

This block represents the CANopen node in the system. For code generation by RTW just one CANopen block can be used!
See help for more information and setup manual.

Parameters

Node ID: 1

CAN board number: 1

CAN bit rate [kbps]: 1000

Object dictionary name: cf_node

[✓] Load I/O mapping from EDS file

Input mapping: , 'load_deltas_coef'}, {8218, -1, 5, 'load_drift_angle_coef'}, {8215, -1, 0, 'lvdt_position'}}

Output mapping: ad_fm'}, {8216, -1, 5, 'load_valve'}, {8212, -1, 5, 'load_fm'}, {8216, -1, 5, 'load_valve'}}

Sample time: [0.04 0.02]

[ ] Simulation support

Heartbeat producer period [ms]: 0

Heartbeat consumer nodes as timeouts [ms]: {}

Figure 3.2: Mask of the CANopen node block

in these files has to be the same as the name of these files. The reason is that all the OD variables use the name as prefix. For example: the CanFestival OD name is *myNode*. Then insert *myNode* in the edit array and place OD files *myNode.eds*, *myNode.c* and *myNode.h* into the model directory.

While more than one CANopen node blocks are used in the model this identifier determines which model block belongs to each CANopen node.

Load I/O mapping from EDS file

If checked the cells of input and output mapping are filled automatically according to the RPDOs a TPDOS mapping read from EDS file.

Input/output mapping

Input mapping defines which entries of Object Dictionary are connected to block inputs. Output mapping defines which entries of Object Dictionary are connected to block outputs.

The only accessible entries are the Manufacturer specific variables defined in EDS. These entries are generated by OD editor (see section 3.3.1) as variables into the C code. The entries are accessed just through these variables in IO mapping. The variables have the name which they are represented by in the code.

Input/output mapping is an array of inputs/outputs of the block in the following form.

```
{{OD_index, OD_subindex or -1, Entry_data_type, 'Entry_name'}, ...}
```

Example

```
{{8193, -1, 7, 'cpd_velocity'}, {8195, -1, 5, 'cpd_controlword'}}
```

This array is filled automatically by OD entries mapped into RPDO (output mapping) and TPDO (input mapping). The array can be modified by user. To return to the default values check the *Load I/O mapping from EDS file* option. According to the input/output mapping the number of block inputs and outputs is set. Each input is supplied by simulation output and each output is supplied by simulation input. In case of simulation these inputs/outputs are used for obtaining values instead of PDO communication. *Simulation support* has to be enabled to create simulation I/O.

Sample time

The sample time and offset of the block I/O refresh in seconds. It is set in the form of *Sample time* = [*sampleTime offset*]. In the case that just a single number is set, it is considered to be a sample time and the offset is set to 0.

Mind that it has nothing in common with the CANopen SYNC messages. This is just synchronization of the block inputs and outputs (block Update and Output functions).

Simulation support

The block is mainly useful for generating code, but it is also able to perform simulation of synchronous communication. If *Simulation support* is checked simulation output is created to each data input and simulation input to each data output. While simulating values from simulation inputs are copied to data outputs and from inputs are the values copied to corresponding simulation outputs.

Heartbeat producer period

Heartbeat service setting is load from EDS file and it is displayed by the GUI just for the information, but cannot be changed here. Heartbeat producer period is the interval (milliseconds) in which the heartbeat message is sent by the local node. If it is set to zero, no messages are produced. This setting is load from the Object Dictionary index 1017 hex.

Heartbeat consumer nodes and timeouts

This information is load from the EDS file at the index 1016 hex of the Object Dictionary. It defines the nodes (heartbeat producers) that should be monitored and the periods of heartbeat messages they are sending. It is displayed as a cell array of pairs node ID and period in milliseconds. If the array is empty no heartbeat consumer is established. This settings cannot be changed using this GUI. If some node does not send the message in the given interval, the Heartbeat error callback is called. Therefor the CANopen callbacks block should be used to handle this error while using Heartbeat service.

3.4.2 *ObjectDictionary* class and EDS file parser

While having the Object Dictionary name inserted in the CANopen block mask, the appropriate EDS file is loaded. It is given to the *ObjectDictionary* class constructor which uses EDS parser do read the EDS file. The resulted OD object is placed into the *objDict* structure in the field with the same name as the OD name. This structure is stored in the Matlab workspace and all used CANopen blocks have their ODs in it. The OD object keeps all the indices of the given EDS file, their default values and data types (both CanFestival and Matlab representation).

3.4.3 Block functionality

The CANopen node block generates the code controlling the CANopen driver (CanFestival) and the local CANopen node as it is the main block of the blockset. It means that it starts the driver at the simulation startup and stops it at the simulation termination. After the startup the local node state is set to *Initialisation* and according to the CANopen standard [CiAb] it switches to the *Pre-operational* state automatically. However, it is necessary to switch the state to *Operational* manually to start the CANopen node function. There is shown the diagram of the block functionality in the figure 3.3.

Output and *Update* functions of the model are performed in each simulation step. This block copies particular entries of the Object Dictionary to its output ports in the *Output* function and updates its OD by new inputs in the *Update* function. Read appropriate paragraph in the Block parameters section for more information about mapping input and output block ports on the OD entries.

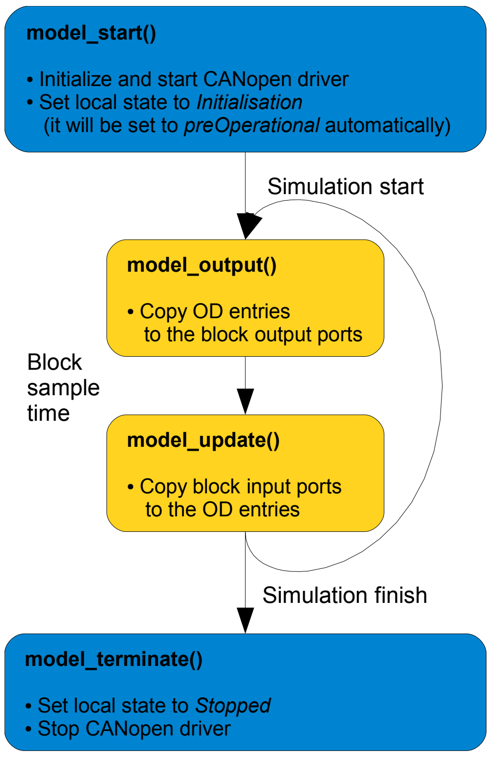

model_start()
• Initialize and start CANopen driver
• Set local state to *Initialisation*
(it will be set to *preOperational* automatically)
Simulation start
model_output()
• Copy OD entries to the block output ports
Block sample time
model_update()
• Copy block input ports to the OD entries
Simulation finish
model_terminate()
• Set local state to *Stopped*
• Stop CANopen driver

Figure 3.3: CANopen node block functionality in the model functions

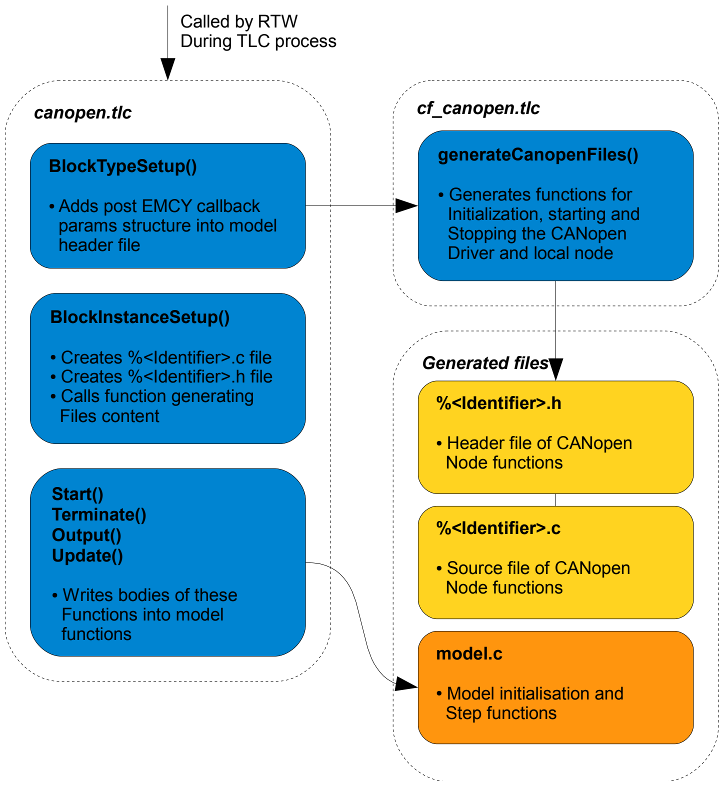

Called by RTW
During TLC process
canopen.tlc
BlockTypeSetup()
• Adds post EMCY callback params structure into model header file
BlockInstanceSetup()
• Creates %<Identifier>.c file
• Creates %<Identifier>.h file
• Calls function generating Files content
Start()
Terminate()
Output()
Update()
• Writes bodies of these Functions into model functions
cf_canopen.tlc
generateCanopenFiles()
• Generates functions for Initialization, starting and Stopping the CANopen Driver and local node
Generated files
%<Identifier>.h
• Header file of CANopen Node functions
%<Identifier>.c
• Source file of CANopen Node functions
model.c
• Model initialisation and Step functions

Figure 3.4: TLC function call diagram in CANopen block code generation

3.4.4 Simulation capabilities

If the simulation directly in Simulink is required, the *Simulation support* option in the block mask has to be checked. While having this option checked simulation inputs and outputs of the block are created. The *ObjectDictionary* object is created in Matlab workspace. This object is stored in *objDict* structure in the field of the same name as is the OD name parameter. It means that if the OD name is e.g. *controller*, the proper *ObjectDictionary* is stored in the field *objDict.controller* in the workspace. In each simulation step, entries of this OD are updated by block input values and the block outputs are read from the OD object as well. It means in fact that the Object Dictionary realizes the block state. This data object in the workspace is used for data exchange by CANopen asynchronous blocks during the simulation.

3.4.5 Block code generation

While generating the code, TLC functions of *canopen.tlc* file are called by Real Time Workshop [Mat] as it is demonstrated in the figure 3.4. Except writing into default model functions, the pair of *.h* and *.c* files is created. These files have name given by block identifier which is build from block name in Simulink (e.g. files *CANopennode1.c* and *CANopennode1.h* are created for the block called *CANopen node 1*). This code module is added to the model sources and is linked with the model code. It contains the function performing the CanFestival driver initialization, CANopen node startup and termination and are called from *model.c* in the proper time of model execution.

3.5 Callbacks block

This blocks creates the API of CANopen event callbacks implemented in CanFestival driver. It allows the user to handle asynchronous events like Emergency message reception, synchronization or Heartbeat error. Read section 3.5.1 for more information.

Each callback function is performed in separate thread with priority lower than the priority of periodic model threads. It means that long duration of callbacks execution cannot break the model synchronization.

This block implements *SS_OPTION_ASYNCHRONOUS* option in its S-function. It says to Simulink that subsystems connected to the block output port are asynchronously executed. Using *Asynchronous rate transition* block (see section 3.10) at each input and output of these subsystems is required and checked by Simulink engine.

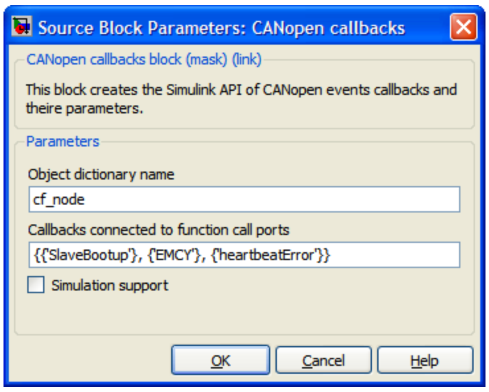
The block is defined in *linux_ert_target_blockset.mdl* file and support files are mentioned in the table 3.2.

S-function source	canopen_callbacks.c
S-function mex	canopen_callbacks.mexw32
Help file	canopen_callbacks_help.html
TLC scripts	canopen_callbacks.tlc (main) callbacks.tlc
Mask callback	callbacks_mask_initialisation.m

Table 3.2: Block files

3.5.1 Block parameters

This section describes the parameters of Callbacks block and possibilities of usage as well. The mask of the block with parameter settings dialog is shown in the figure 3.5.


Source Block Parameters: CANopen callbacks

CANopen callbacks block (mask) (link)

This block creates the Simulink API of CANopen events callbacks and theire parameters.

Parameters

Object dictionary name: cf_node

Callbacks connected to function call ports: {{'SlaveBootup'}, {'EMCY'}, {'heartbeatError'}}

☐ Simulation support

Figure 3.5: CANopen callbacks block mask

Object dictionary name

The Object dictionary name is used to say the block which OD it should work with if more than one CANopen node blocks are used in the model. The OD name has to be the same as the OD name of the appropriate CANopen main block.

Callbacks connected to function call ports

You can define the callbacks of CANopen node that are to be used as function call ports of the block. The callbacks and corresponding function calls can be used for activation of another block for example Asynchronous operations block or even the main CANopen node block. It is defined by the cell of the list of callback names (and sometimes configuration parameters) to be used by the port. The syntax of the cell is:

```
{{'callback1_name'}, {'callback2_name', {'params'}}, ...}
```

Example

```
{{'SYNC'}, {'EMCY'}, {'OD',{'1800', '1'}}, {'Initialisation'}}
```

The function call output port of the block is created (always the first output port) and its width is set to be the same as the number of defined callbacks. If using more than one callbacks the Demux Simulink block has to be connected on the output port and split the signals into appropriate number of outputs. The number of Demux outputs has to be set by user manually and fit the number of defined callbacks. Finally, on the Demux ports any Function call subsystems can be connected.

Additionally some callbacks are supplemented by a parameter or parameters. For example EMCY callback has 3 parameters defining the received EMCY message: node ID, Error code and Error register content. These parameters are stored into the global memory while executing the callback and can be accessed by appropriate parser block. For example EMCY callback parameters can be decoded by EMCY parser block. This parser block should be placed in the function call subsystem connected to the appropriate callback. All the supported callbacks are described in the following text.

SYNC messages callback

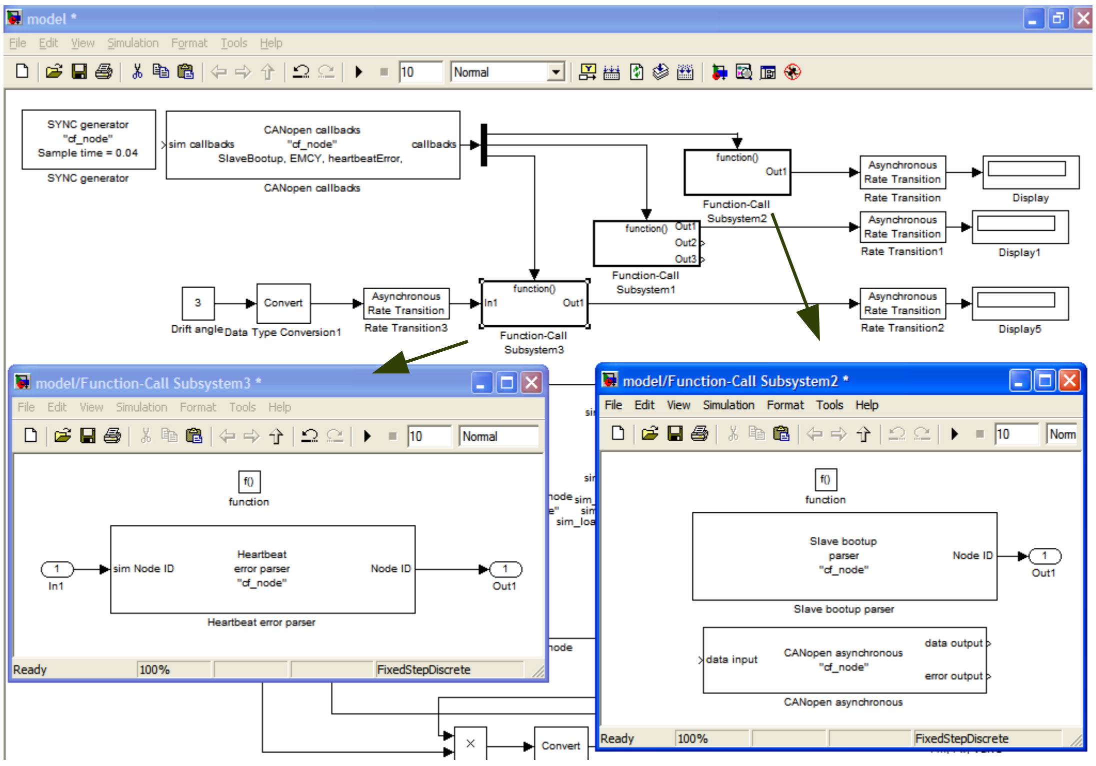
This callback is called after SYNC message reception or transmission. No parameter is set to the callbacks parameters port.

```
{'SYNC'}
```

EMCY messages callback

This callback is called after emergency message reception. This callback has three parameters defining the received EMCY message: node ID (UINT8), Error code (UINT16), Error register (UINT8). Use EMCY parser block to get these parameters from the memory.

```
{'EMCY'}
```



model *
SYNC generator "cf_node" Sample time = 0.04
CANopen callbacks "cf_node" SlaveBootup, EMCY, heartbeatError,
Function-Call Subsystem2
Function-Call Subsystem1
Function-Call Subsystem3
Asynchronous Rate Transition
Display
model/Function-Call Subsystem3 *
Heartbeat error parser "cf_node"
model/Function-Call Subsystem2 *
Slave bootup parser "cf_node"
CANopen asynchronous "cf_node"

Figure 3.6: Working with callbacks and their parameters

TPDO messages callback

This callback is called after PDO transmission. No parameter is set to the callbacks parameters port.

```
{'TPDO'}
```

Initialisation state callback

This callback is called after entering the Initialisation state by local node. No parameter is set to the callbacks parameters port.

```
{'Initialisation'}
```

PreOperational state callback

This callback is called after entering the PreOperational state by local node. No parameter is set to the callbacks parameters port.

```
{'PreOperational'}
```

Operational state callback

This callback is called after entering the Operational state by local node. No parameter is set to the callbacks parameters port.

```
{'Operational'}
```

Stopped state callback

This callback is called after entering the Stopped state by local node. No parameter is set to the callbacks parameters port.

```
{'Stopped'}
```

Slave boot up callback

This callback is called after reception of Slave boot up message (COBID = 0x700 + NodeID). This callback has one parameter defining the node ID (UINT8). Use SlaveBootup parser block to get the value from the memory.

```
{'slaveBootup'}
```

Heartbeat error callback

This callback is called after detection of CANopen heartbeat error. This callback has one parameter defining the node ID (UINT8). Use Heartbeat error parser block to get the value from the memory.

The heartbeat service has to be defined in EDS file used for node configuration. This setting can be done in Object Dictonary editor (see section 3.3.1). If the node is set to be a heartbeat producer, the heartbeat request is sent to the network with the given period. If some node does not aswer it in the defined timeout, the heartbeat error callback is called at the producer node.

```
{'heartbeatError'}
```

OD entry callback

This callback is called after change of data in the defined OD entry. It can be set to any existing OD entry. No parameter is set to the callbacks parameters port.

The callback has to be predefined in CanFestival *Objdictedit* by checking the *Have callback* option of the appropriate OD entry!

```
{'OD', {'index', 'subindex'}}
```

Simulation support

The block is mainly useful for generating code, but it is also able to perform simulation of CANopen events activating the callbacks. If *Simulation support* is checked simulation input is created. This input has as many signals as defined callbacks number. If one of the *boolean* signals is set, particular callback is activated.

3.5.2 Block functionality

The Callback block does not perform any periodic tasks. It does not have output and update functions in the generated code. Its only function is to register required callbacks at the program startup. This registration saves pointers to generated callback functions into CanFestival structure. If some callback is activated by the driver, code of the connected subsystem is called by the callback function (figure 3.7). Callback parameters are stored into the global memory structure and can be accessed by the appropriate parser block (section 3.6). Read section 3.5.1 for exact information about particular callback types.

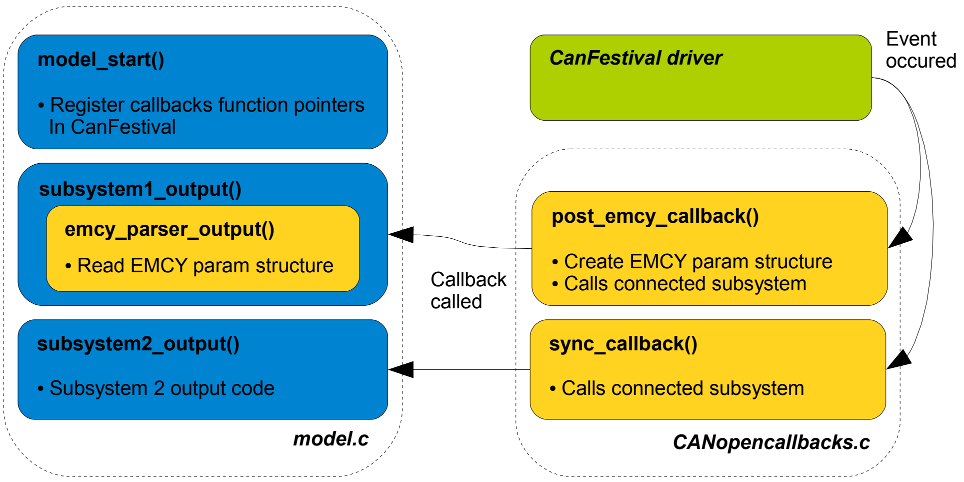

model_start()
• Register callbacks function pointers In CanFestival
subsystem1_output()
emcy_parser_output()
• Read EMCY param structure
subsystem2_output()
• Subsystem 2 output code
model.c
CanFestival driver
Event occured
post_emcy_callback()
• Create EMCY param structure
• Calls connected subsystem
sync_callback()
• Calls connected subsystem
Callback called
CANopencallbacks.c

Figure 3.7: Example of block code function while having Emergency and Sync callbacks employed

3.5.3 Simulation capabilities

Simulation capabilities of these block are very simple. While the *Simulation support* option is enabled, a single input port is created. Its width is the same as the number of defined callbacks. Each input signal is a trigger of particular callback function call. It is logical input that activates particular function call when enabled.

The callback parameters can be employed in the simulation by enabling the simulation support of particular parameters parser blocks (see section 3.6). Then the simulation inputs of that blocks are created and the parameter values can be passed via signals.

3.5.4 Block code generation

This block uses two TLC scripts - *canopen_callbacks.tlc* and *callbacks.tlc*. The scripts creates the pair of header and source files called by the block identifier which is based on the block label in the model. It means that each block has its own source files. The callback functions are generated into these files and the code for callbacks registration is placed into the model start function. The diagram of TLC functions calls is shown in the figure 3.8.

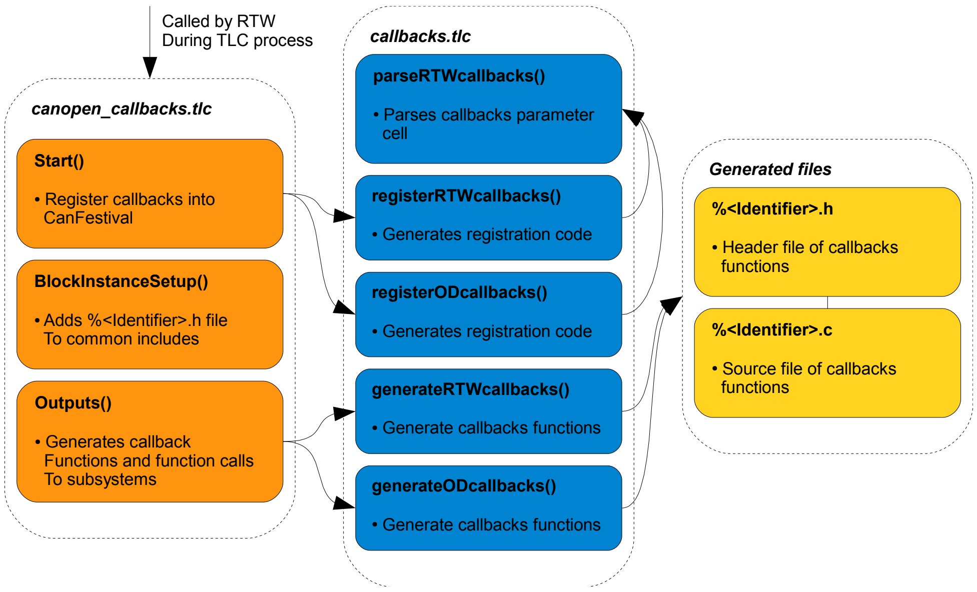

Called by RTW
During TLC process
canopen_callbacks.tlc
Start()
• Register callbacks into CanFestival
BlockInstanceSetup()
• Adds %<Identifier>.h file To common includes
Outputs()
• Generates callback Functions and function calls To subsystems
callbacks.tlc
parseRTWcallbacks()
• Parses callbacks parameter cell
registerRTWcallbacks()
• Generates registration code
registerODcallbacks()
• Generates registration code
generateRTWcallbacks()
• Generate callbacks functions
generateODcallbacks()
• Generate callbacks functions
Generated files
%<Identifier>.h
• Header file of callbacks functions
%<Identifier>.c
• Source file of callbacks functions

Figure 3.8: Diagram of block code generation

3.6 Callback parameters parser blocks

A group of blocks that parses callback parameters provided by Callbacks block (see section 3.5) was created to enable working with parameters of CANopen driver callbacks. These blocks are designed to be placed in function call subsystem activated from the callback block. The Callbacks block stores the callback parameters into the global memory while processing a callback having parameters (EMCY, Slave boot up and Heartbeat error). The parser block can read these parameters from the global memory and set them to its output ports. If there is more than one CANopen node defined in the model, each can have its own callbacks and parsers block. They are determined by their Object dictionary name parameter. The dependency of the parsers on the Callbacks block is checked.

For more information about the callbacks and the events that activate the callbacks read Callbacks block section 3.5 and CanFestival driver documentation [LOLb].

3.6.1 Parser blocks parameters

All the parser blocks has the same two mask parameters. Its the reason for describing the parameters generally for all the block together.

Object dictionary name

The Object dictionary name is used to say the block which OD it should work with if more than one CANopen node blocks are used in the model. The OD name has to be the same as the OD name of the appropriate CANopen main and callbacks blocks.

Simulation support

While the *Simulation support* option is checked three input ports are created. Each of these ports corresponds to one of the outputs. In case of Simulink simulation values of the simulation inputs are copied to the output ports in each block step.

3.6.2 Simulation capabilities

While having the *Simulation support* mask option enable in some parser block, simulation input ports are created. Each of these inputs corresponds to some block output port and callback parameter as well. In case of Simulink simulation, values from the simulation inputs are copied to the outputs in the block *Output* function. Together with callback event simulation by Callbacks block the event execution can be completely simulated in the Simulink.

3.6.3 Emergency callback parameters parser block

This block parses the Emergency message parameters. It can be used only in the co-operation with the CANopen callbacks block (with the same Object Dictionary name) having the EMCY callback employed. The parser block is designed to be placed in function call subsystem connected to the appropriate callback of the Callbacks block.

The block has three outputs according to the number of EMCY callback parameters. These outputs provide values of last received EMCY message processed by the CANopen callbacks block. This Callbacks block writes message parameters to the global memory and activates the EMCY callback function call.

The block is defined in *linux_ert_target_blockset.mdl* file and support files are mentioned in the table 3.3.

S-function source	emcy_parser.c
S-function mex	emcy_parser.mexw32
Help file	emcy_parser_help.html
TLC scripts	emcy_parser.tlc

Table 3.3: Block files

3.6.4 Heartbeat error callback parameters parser block

This block parses the Heartbeat error event parameters. It can be used only in the co-operation with the CANopen callbacks block (with the same Object Dictionary name) having the Heartbeat error callback employed (see section 3.5.1).

The parser block is designed to be placed in function call subsystem connected to the appropriate callback of the Callbacks block. The block has a single output according to the number of Heartbeat error callback parameters. This output provides the value of the last Heartbeat error event processed by the CANopen callbacks block. The parameter is the node ID of the node which produced the error. The Callbacks block writes the parameter to the global memory and activates the Heartbeat error callback function call.

The block is defined in *linux_ert_target_blockset.mdl* file and support files are mentioned in the table 3.4.

S-function source	heartbeat_error_parser.c
S-function mex	heartbeat_error_parser.mexw32
Help file	heartbeat_error_parser_help.html
TLC scripts	heartbeat_error_parser.tlc

Table 3.4: Block files

3.6.5 Slave boot up callback parameters parser block

This block parses the Slave boot up message parameters. It can be used only in the co-operation with the CANopen callbacks block (with the same Object Dictionary name) having the Slave boot up callback employed. The parser block is designed to be placed in function call subsystem connected to the appropriate callback of the Callbacks block. The block has a single output according to the number of Slave boot up message callback parameters. This output provides the value of the last Slave boot up message processed by the CANopen callbacks block. The parameter is the node ID of the node which produced the message. The Callbacks block writes message parameter to the global memory and activates the Slave boot up callback function call.

The block is defined in *linux_ert_target_blockset.mdl* file and support files are mentioned in the table 3.5.

S-function source	slave_boot up_parser.c
S-function mex	slave_boot up_parser.mexw32
Help file	slave_boot up_parser_help.html
TLC scripts	slave_boot up_parser.tlc

Table 3.5: Block files

3.7 SYNC message generator block

This block can be used for generating SYNC messages in the CANopen network. The SYNC generation can be performed by the CanFestival driver automatically if it is correctly set in the Object Dictionary. However, the automatical SYNC generation has one big disadvantage that it is not synchronized with the model sample time. It means that the time offset between the network synchronization and the model step is not known. This is solved in the way that the automated SYNC generation is disabled and it is performed by SYNC generator block if it is used in the model. If the SYNC generator block for the given Object Dictionary is not placed in the model, the SYNC generation is performed automatically by the CanFestival driver according to the OD settings. It means of course that if neither the SYNC generator is used nor the SYNC generation is defined in OD, no SYNC generation is performed and the node is considered to be the SYNC slave.

This block calls the driver command for sending SYNC message in its *Output* function if the node state is *Operational*. It secures the synchronization of the model with the CANopen network communication. SYNC message ID is usually 80 hex, however, it is possible to change it by the block parameter. The message ID is stored into the Object Dictionary at the model startup. The block functionality diagram is shown in the figure 3.9.

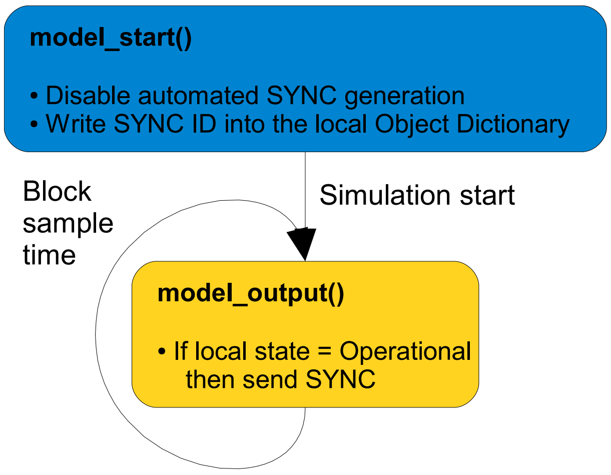

model_start()
• Disable automated SYNC generation
• Write SYNC ID into the local Object Dictionary
Block sample time
Simulation start
model_output()
• If local state = Operational then send SYNC

Figure 3.9: Function diagram of SYNC message generator block

As described above, generating SYNC messages by the separate block is very useful, however it is necessary to think about the Simulink work in this case. Functions of particular blocks are performed in each simulation step. If a group of blocks has the same sample time, all the blocks are performed one after another in the same step function and the order of blocks is established just by the coincidence. It means that some very small time delay can occur, which could be unwanted in case of SYNC messages that synchronize all the network. Calling the block code as soon as possible after the timer tick is ensured by implementing *SS_OPTION_PLACE_ASAP* in the S-function as well.

The block is defined in *linux_ert_target_blockset.mdl* file and support files are mentioned in the table 3.6.

S-function source	sync_generator.c
S-function mex	sync_generator.mexw32
Help file	sync_generator_help.html
TLC scripts	sync_generator.tlc

Table 3.6: Block files

3.7.1 Block parameters

The following text describes the parameters of the SYNC generator block.

Object dictionary name

The Object dictionary name is used to say the block which OD it should work with. The OD name has to be the same as the OD name of the appropriate CANopen main block.

Sample time

Block sample time and offset can be set in the same way as in each Simulink block.It is set in the standard Simulink format [*sampleTime offset*]. If it is just a single number, the offset is considered to be 0.

SYNC message ID

The ID of the generated SYNC message. The default value is 0x80 (80 hex, 128 dec). It is not recommended to change it in the CANopen network, because it is defined by CANopen standard. The ID number has to be set in the C language form (0x prefix for hex numbers).

3.7.2 Block code generation

This block has a single TLC script *sync_generator.tlc* which generates its code during the TLC process. It does not generate any files. It just writes the code for SYNC message ID setup into the model start function and the code for SYNC message sending into the model output function.

3.8 Asynchronous operations block

This block realizes a group of asynchronous transmissions defined by the parameter. It performs all the transmissions in the given order in its Output function. The block is useful for network management or asynchronous communication. It is designed mainly to be placed in the function call subsystem and called in some CANopen event callback (see section 3.5). In this case the block sample time has to be set to -1 (inherited). However, it is possible to set the periodic sample time and run the block periodically straight in the model. As an example of the common usage we can mention this block connected to the *Pre-operational* callback and performing setting the node state to *Operational* or may be sending *Start-Node* command to other nodes as well.

The block is defined in *linux_ert_target_blockset.mdl* file and support files are mentioned in the table 3.7.

S-function source	cf_asyn.c
S-function mex	cf_asyn.mexw32
Help file	cf_asyn_help.html
TLC scripts	cf_asyn.tlc

Table 3.7: Block files

3.8.1 SDO transfers

Most of the operations that are supported by this block do not need to wait for the end of the message transfer. It means that they are local operations that are performed immediately or CANopen transfers that are not confirmed and so do not wait for the answer. The only confirmed service of the CANopen protocol is SDO (*Service Data Object*). This service provides confirmed reading or writing to the remote object dictionary and are used mainly for the node setup and not for the regular data transfer.

This service is supported by operations *readNetworkDict* and *writeNetworkDict* in this block. The answer to the message is handled by the CanFestival driver and a callback function is called at the end of the transfer. It is necessary to wait for this callback and read the transfer result. If the answer is not delivered in time, the timeout error is announced so no infinite blockage can occur. You can avoid waiting for the answer by setting appropriate parameter (see section 3.8.3). However, while reading the network dictionary, it will always wait for the result as the required value is there.

The principle of this block work while some SDO transfers are used is shown in the figure 3.10. If no SDO transfers are used, the *checkSDO* function is not necessary and all the code is performed in a single function and in a single iteration.

3.8.2 Block inputs and outputs

The block has one input port and two output ports. The input port is a vector of width given by the highest value X of the *inputX* mark used in the operation definition cell. If the highest input used is 5, the width of input port is set to 6 (0..5). The data output is created in the same way according to the highest outputX value used. The second output port is the error output. Its width is the same as the number of defined operation. Each signal of the port keeps the error code of the particular operation. The order is the same

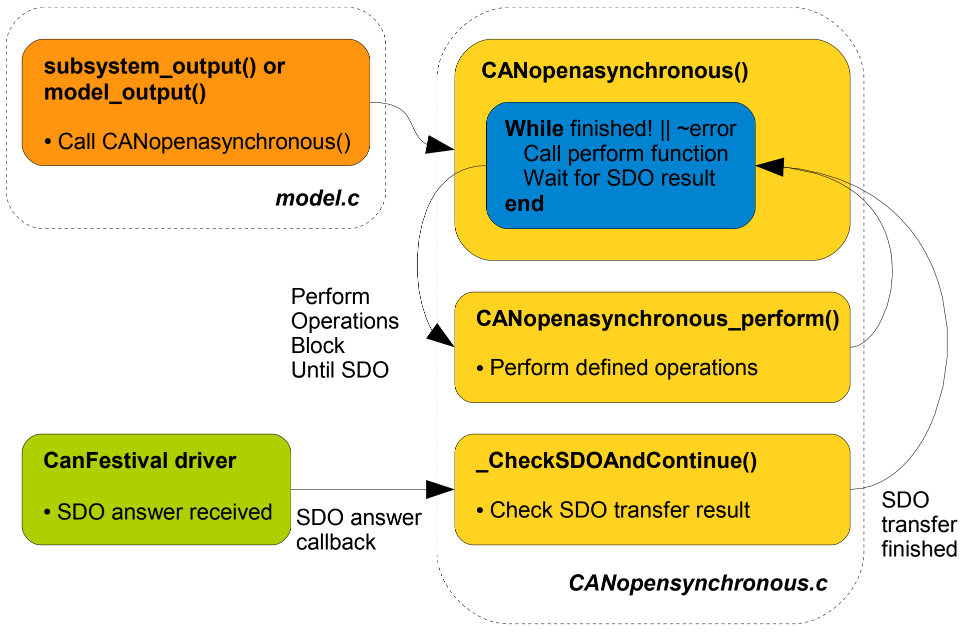

subsystem_output() or model_output()
• Call CANopenasynchronous()
model.c
CANopenasynchronous()
While finished! || ~error
Call perform function
Wait for SDO result
end
Perform Operations Block Until SDO
CANopenasynchronous_perform()
• Perform defined operations
CanFestival driver
• SDO answer received
SDO answer callback
_CheckSDOAndContinue()
• Check SDO transfer result
SDO transfer finished
CANopensynchronous.c

Figure 3.10: Asynchronous block function diagram

as the order of defined sequence. The error code 0 means that the operations terminated correctly. See operations description bellow to get information about particular error codes.

Using block input/output as operation parameter

You can use *inputX* mark as an operation parameter (X is the number of signal in the input port vector $< 0, N >$). This will read the value of particular field in input port vector and use it as the function parameter. In the same way *outputX* can be used in operations of reading OD to copy the value to the output port instead of some internal variable.

3.8.3 Block parameters

Block mask parameters are described in the following section.

Object dictionary name

The Object dictionary name is used to say the block which OD it should work with if more than one CANopen node blocks are used in the model. The OD name has to be the same as the OD name of the appropriate CANopen main block.

Do not wait for SDO transfer result

In case of SDO transfer, the block waits for the answer before continuation. This can be over outweighed by checking the *Do not wait for SDO transfer result* check box. Mind that in case of readNetworkDict the program always wait for the answer (value)!

Continue the transfer sequence in case of error

You can check the *Continue the transfer sequence in case of error* for ignoring the operation failure and performing the sequence to the end.

Sample time

The sample time and offset of the block I/O refresh in seconds. It is set in the form of [*sampleTime offset*]. In the case that just a single number is set, it is considered to be a sample time and the offset is set to 0.

The sample time setting can be used to run the block periodically, however, the block is designed mainly to perform the asynchronous operations in CANopen event callback. In this case the sample time should be set to inherited (-1) and the block should be placed in *Function call subsystem*.

Simulation support

In contrast to other blocks the *Simulation support* option does not create any simulation inputs of the block. It enables or disables using Matlab workspace objects for data exchange during simulation. This feature makes the simulation slow. It is recommended to disable this option while simulation of asynchronous operations is not required.

Operation parameters cell

The cell contains cells of particular operations of the order which they should be performed in. The cell of one operation then contains a string of operation type and a cell of particular parameters. The operation type name corresponds to the function name of particular CanFestival function. Each operation type has different number of parameters. All the parameters are set as strings even if it is a number. Numbers are written in the same form as in C language (e.g. 100, 0x10). Various data sources and destination can be used by the operations. First of all, it can be the pair of index and subindex in case of accessing the Object dictionary. Then, it can be a block input or output represented by *inputX* or *outputX* string. Finally, it can be a constant number or almost anything which is the user sure to be defined in the generated code. For example it can be a variable defined in the *Manufacturer specific* section of Object dictionary.

```
{{op1\_Name, {op1\_Parameter1, op1\_Parameter2}},
{op2\_Name, {op2\_Parameter1, op2\_Parameter2, op2\_Parameter3}}}
```

Example

```
{{'sendPDOevent'},
{'writeNetworkDict', {'3', '0x1802', '1', 'input2', 'UNS8'}},
{'masterSendNMTstateChange', {'3', 'NMT\_Start\_Node'}},
{'writeLocalDict', {'0x1802', '1', '255', 'UNS32'}},
{'writeNetworkDict', {'4', '0x1803', '1', '255', 'REAL64'}}}
```

Supported operations

The following text describes the asynchronous operations of CANopen protocol which are supported by this block.

Variable copy

This just copies the value, variable or block input into the target variable or block output. While using variable name it has to be defined in the CanFestival node source file generated by Objdictedit. The variable set may be used before PDO event operation to start the transmission of PDOs which are transmitted just after its value change.

This is a simplified version of read/writeLocalDict operation which works only for the variables defined in the node source. This operation always finishes with error code 0.

Example

```
{'VarCopy', {'var8192\_0', 'var8193\_1'}}
```

copies the value of variable var8192_0 to variable var8193_1, or

```
{'VarCopy', {'0x15', 'output2'}}
```

copies the constant 0x15 to the block output 2, or

```
{'VarCopy', {'input3', 'var8192\_0'}}
```

copies the value of the input signal 3 into variable var8192_0.

PDO event

This operation invokes PDO transmission of PDOs that are to be sent after data change. Only PDOs that satisfy this condition are transmitted. This operation always finishes with error code 0.

Example

```
{'sendPDOevent'}
```

Set local node state

This operation sets local CANopen node state. The states can be *Initialisation*, *Stopped*, *Operational* or *Pre-operational*. Mind that the names are case sensitive!! This operation always finishes with error code 0.
See the CANopen protocol documentation for more information.

Example

```
{'setState', {'Operational'}}
```

Set remote node state

This operation sets the state of remote CANopen node with given ID. The states can be *NMT_Reset_Node*, *NMT_Reset_Communication*, *NMT_Stop_Node*, *NMT_Start_Node* or *NMT_Enter_PreOperational*. Mind that the names are case sensitive!! This operation always finishes with error code 0.

Example

```
{'masterSendNMTstateChange', {'3', 'NMT\_Start\_Node'}}
```

Write to local dictionary

This operation writes value into local Object Dictionary. The parameters of the operation are these: index, subindex, value, value data type. If the given OD entry has no subindices set the subindex parameter into '-1'. This operation always finishes with error code 0. The data types are specified using these strings:
INTEGER8, INTEGER16, INTEGER24, INTEGER32, INTEGER40, INTEGER48, INTEGER56, INTEGER64
UNS8, UNS16, UNS32, UNS24, UNS40, UNS48, UNS56, UNS64
REAL32, REAL64

Example

```
{'writeLocalDict', {'0x1802', '1', '255', 'UNS32'}}
{'writeLocalDict', {'0x1802', '1', 'input0', 'UNS32'}}
```

Read entry of the local dictionary

This operation reads an entry of the local Object Dictionary. The parameters of the operation are these: index, subindex, target variable, value data type. If the given OD entry has no subindices set the subindex parameter into '-1'. This operation always finishes with error code 0. The data types are specified using these strings:
INTEGER8, INTEGER16, INTEGER24, INTEGER32, INTEGER40, INTEGER48, INTEGER56, INTEGER64
UNS8, UNS16, UNS32, UNS24, UNS40, UNS48, UNS56, UNS64
REAL32, REAL64

Example

```
{'readLocalDict', {'0x1802', '1', 'var1', 'UNS32'}}
{'readLocalDict', {'0x1803', '0', 'output1', 'UNS32'}}
```

Write entry of the network dictionary

This operation writes value into Object Dictionary of remote node. The parameters of the operation are these: node ID, index, subindex, value, value data type. If the given OD entry has no subindices set the subindex parameter into '-1'. The data types are specified using these strings:
INTEGER8, INTEGER16, INTEGER24, INTEGER32, INTEGER40, INTEGER48, INTEGER56, INTEGER64
UNS8, UNS16, UNS32, UNS24, UNS40, UNS48, UNS56, UNS64
REAL32, REAL64

The SDO transfer uses confirmation. If the *Do not wait for SDO transfer result* check box is unchecked the operation waits for the remote node confirmation of the SDO and checks the errors. In case of error the SDO abort code is written to the error output port of the block. See CANopen specification for more information about SDO error codes.

Example

```
{'writeNetworkDict', {'4', '0x1803', '1', '255', 'REAL64'}}
```

Read entry of the remote node dictionary

This operation reads an entry of the remote node Object Dictionary. The parameters of the operation are these: nodeID, index, subindex, target variable, value data type. If the given OD entry has no subindices set the subindex parameter into '-1'. The data types are specified using these strings:
INTEGER8, INTEGER16, INTEGER24, INTEGER32, INTEGER40, INTEGER48, INTEGER56, INTEGER64
UNS8, UNS16, UNS32, UNS24, UNS40, UNS48, UNS56, UNS64
REAL32, REAL64

After sending the SDO read command to the remote node, the program has to wait for the SDO answer with the requested value. The *Do not wait for SDO transfer result* check box has no influence on the wait time in the case of reading! In case of error the SDO abort code is written to the error output port of the block. See CANopen specification for more information about SDO error codes.

Example

```
{'readNetworkDict', {'2', '0x1802', '1', 'var1', 'UNS32'}}
{'readNetworkDict', {'4', '0x1803', '0', 'output1', 'UNS32'}}
```

Sleep

Generates the sleep command which makes the current thread wait for the given amount of seconds. Mind that the thread may be the sample time loop thread in case that the block is not called in the Function call subsystem activated by Callbacks block. This operation always finishes with error code 0.

Example

```
{'sleep', {'5'}}
```

USleep

Generates the usleep command which makes the current thread wait for the given amount of microseconds. Mind that the thread may be the sample time loop thread in case that the block is not called in the Function call subsystem activated by Callbacks block. This operation always finishes with error code 0.

Example

```
{'usleep', {'100'}}
```

3.8.4 Block code generation

The block code is generated by the *cf_asyn.tlc* script. All the blocks of this type have the common header file called *cf_asyn.h*. Each block has its own source file called by the same name as its name in Simulink model. See the figure 3.11 for the graphical representation.

3.8.5 Simulation capabilities

The block supports Simulink simulation of data transfer operations (*readLocalDict*, *writeLocalDict*, *readNetworkDict*, *writeNetworkDict*). The other operations defined in the cell are ignored in simulation mode. Local dictionary operations reads or writes values to the *ObjectDictionary* object stored in the *objDict.(odName)* structure in the Matlab workspace. This object is created and used by the CANopen block so that the data exchange is ensured.

In case of network dictionary operations (SDO transfers), the block tries to find the Object Dictionary of the node with the given ID in the *objDict* structure. If the proper object exists (CANopen node block of this OD is used in the model), it reads or writes the data in the same way as in the local node case.

This feature makes the simulation process very slow. It is recommended to disable *Simulation support* option in the parameters if asynchronous operation simulation is not required.

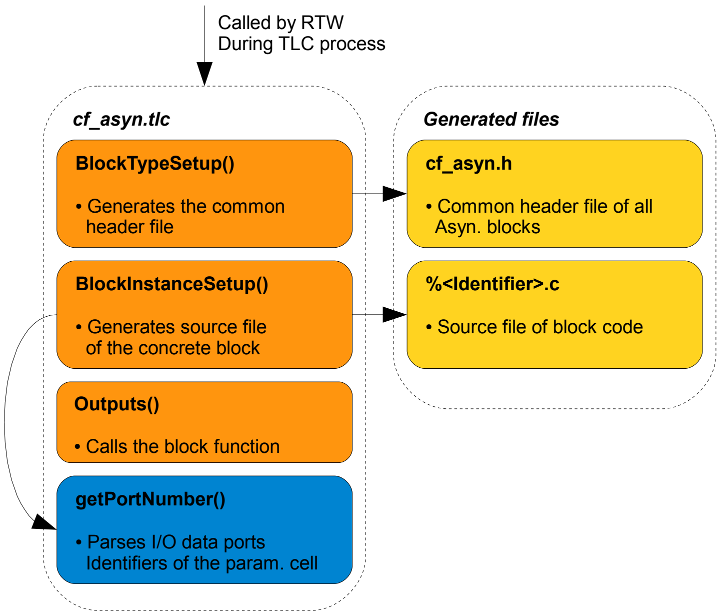

Called by RTW
During TLC process
cf_asyn.tlc
BlockTypeSetup()
• Generates the common header file
BlockInstanceSetup()
• Generates source file of the concrete block
Outputs()
• Calls the block function
getPortNumber()
• Parses I/O data ports Identifiers of the param. cell
Generated files
cf_asyn.h
• Common header file of all Asyn. blocks
%<Identifier>.c
• Source file of block code

Figure 3.11: Block code generation process

3.9 Function call offset block

This block is not a part of CANopen blockset at all. It does not need CANopen support enabled in the model and it can be used with general function call blocks of Simulink. It is designed to be called from within the function call subsystem. It then waits for the given time and activates its own function call port. The timing is realized by periodic thread [Píš05] and is performed in a separate thread to avoid model blocking (figure 3.12). This block has a single parameter which is a value of required offset in seconds.

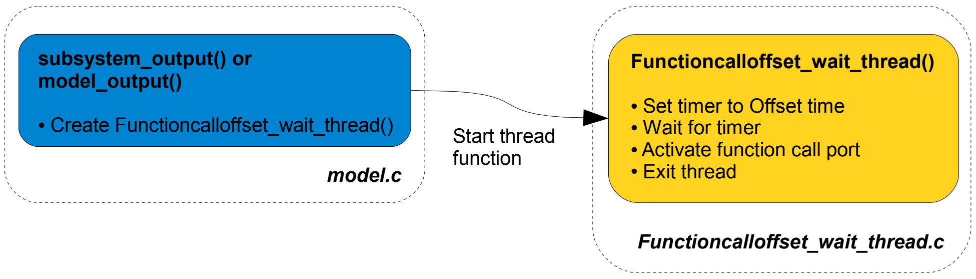

subsystem_output() or model_output()
• Create Functioncalloffset_wait_thread()
model.c
Start thread function
Functioncalloffset_wait_thread()
• Set timer to Offset time
• Wait for timer
• Activate function call port
• Exit thread
Functioncalloffset_wait_thread.c

Figure 3.12: Function call offset code functionality

The block is defined in *linux_ert_target_blockset.mdl* file and support files are mentioned in the table 3.8.

S-function source	fc_offset.c
S-function mex	fc_offset.mexw32
Help file	fc_offset_help.html
TLC scripts	fc_offset.tlc

Table 3.8: Block files

3.9.1 Block code generation

While generating the model code *fc_offset.tlc* script is called (figure 3.13). It creates common header file of all blocks of this type and one source file for each of them.

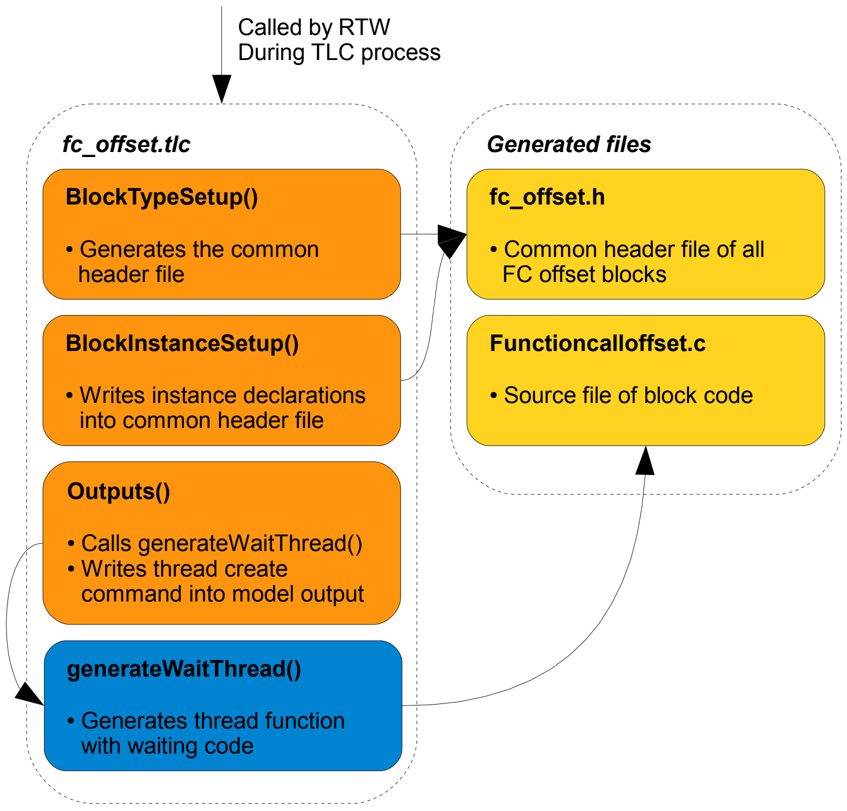

Called by RTW
During TLC process
fc_offset.tlc
BlockTypeSetup()
• Generates the common header file
BlockInstanceSetup()
• Writes instance declarations into common header file
Outputs()
• Calls generateWaitThread()
• Writes thread create command into model output
generateWaitThread()
• Generates thread function with waiting code
Generated files
fc_offset.h
• Common header file of all FC offset blocks
Functioncalloffset.c
• Source file of block code

Figure 3.13: Function call offset code generation

3.10 Asynchronous rate transition block

This block is not a part of CANopen blockset at all. It does not need CANopen support enabled in the model and it can be used with general function call blocks of Simulink. This block has to be used while connecting synchronous and asynchronous section of the model by a data signal. It creates a critical section and ensures data integrity.

In case of simulation it just copies input data into the state vector in its Update function and sets the data onto the output port in its Output function. It means that it needs two sample hits to get the value from input to output. Its sample time should be set to the sample time of appropriate synchronous section of the model. While generating code it locks these data transfers by mutex.

The block supports all data types and any port width. The output port type and width is set automatically according to the input signal.

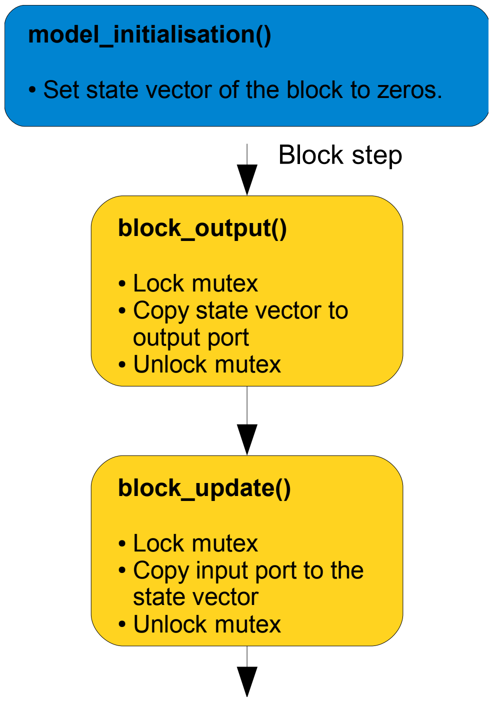

model_initialisation()
• Set state vector of the block to zeros.
Block step
block_output()
• Lock mutex
• Copy state vector to output port
• Unlock mutex
block_update()
• Lock mutex
• Copy input port to the state vector
• Unlock mutex

Figure 3.14: Asynchronous rate transition block functionality

The block is defined in *linux_ert_target_blockset.mdl* file and support files are mentioned in the table 3.9.

S-function source	async_rate_transition.c
S-function mex	async_rate_transition.mexw32
Help file	async_rate_transition_help.html
TLC scripts	async_rate_transition.tlc

Table 3.9: Block files

3.10.1 Block code generation

While generating the model code *async_rate_transition.tlc* script is called (figure 3.15). It writes definitions into common model header file and Output and Update code into the proper place according to block usage in the model.

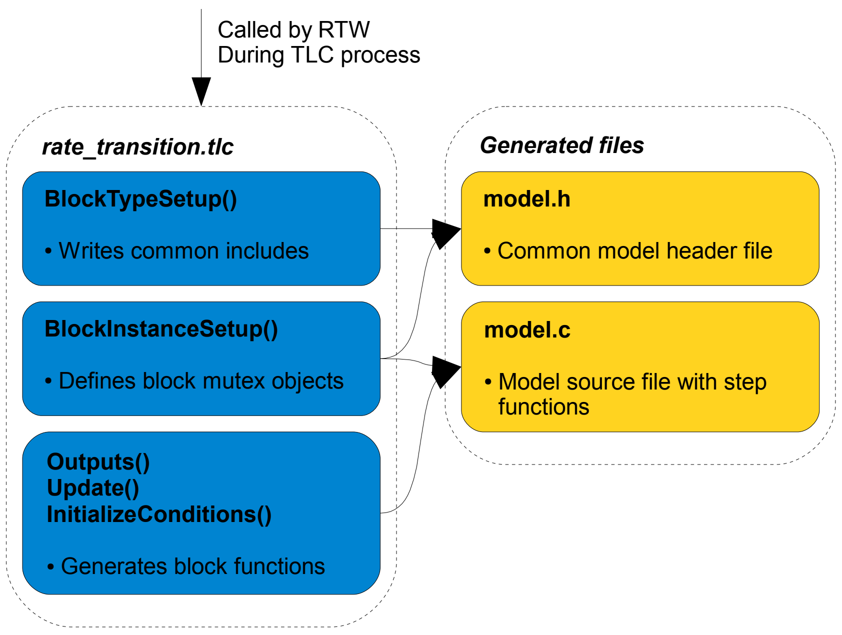


Figure 3.15: Asynchronous rate transition block code generation

Chapter 4

Conclusion

I will summarize the work and describe achieved results at the end of the thesis. The main goal of the thesis was creation of the CANopen protocol support for code generated from Simulink by Real Time Workshop.

The generated code has to be customized to be used at embedded hardware. The desired platform (computer BOA 5200 and Linux operating system) is supported by Linux GRT target prepared in [Jel08]. I decided to redesign the GRT (*Generic Real Time Target*) to the ERT type of target. The ERT (*Embedded Real Time Target*) target uses enhanced optimization features of *Real Time Workshop Embedded Coder* for code generation. Moreover, the new Linux ERT Target is designed to generate multi-tasking application and work with CANopen driver. The target controls the code generation and prepares scripts for code compilation. The compilation itself is performed by a tool chain created in [Jel08].

Linux ERT target features

- Supports BOA5200 computer running Linux operating system
- Generates single-tasking (single-rate and multi-rate) and multi-tasking applications
- Precise simulation steps timing using periodic threads
- Task threads priorities assignment according to task sample times
- Simulation step overrun checking
- Optional CANopen support, CANopen driver does not have to be linked
- Adapts Windows paths to be used in MSYS environment (supports paths including spaces as well)

The CANopen blockset is based on integration of CanFestival driver into the code generated by Real Time Workshop. The libraries of the driver are linked together with the generated code by Linux ERT Target if the CANopen support is enabled.

Particular blocks of the blockset provide particular API functions of the driver to the Simulink environment. The blockset supports both synchronous and asynchronous communication and enables asynchronous events handling. Moreover, the simulation support of the main functions has been created as well. The aim of this was not to create CANopen network simulator for Simulink, but just enable basic functionality simulation while tuning embedded controller in Simulink.

The simulation is provided partially by simulation inputs and outputs of the blocks and partially by using Object Dictionary objects saved in Matlab workspace. Such object is created by each CANopen node block in the model and data transfers are simulated by writing into OD of the other node. Remote SDO transfers can be simulated as well if the remote node is created in the model. To be honest, using workspace during simulation makes it very time-consuming. The simulation support of each block can be switched off and it is reasonable to use it only if it is necessary. While tuning the controller the simulation of the CANopen node block should be sufficient. The following list describes all the capabilities of the blockset.

CANopen blockset features

- Integrates CanFestival driver into the code generated by Linux ERT Target
- Loading node configuration from EDS file
- Possibility to create more then one node in a single model
- Optional program and CAN messages logging
- Supported features of CANopen protocol:
  - PDO transfers (also in simulation mode)
  - SDO transfers (also in simulation mode)
  - NMT for remote nodes control
  - SYNC generator
  - Callbacks on driver events: EMCY, SYNC, Heartbeat error, OD entry change, PDO transfer, Slave boot up, Node state change (also in simulation mode)
  - Event parameters reading (also in simulation mode)
  - Heartbeat service support
  - Asynchronous rate transition block for secured data exchange between synchronous and asynchronous parts of the model
  - Function call offset block

Finally, I will point out the things that can be enhanced in the future. The BOA computer is running Linux kernel without real-time patch applied. Moreover, the Linux kernel uses 4 milliseconds as the system timer resolution. It means that using sample time lower than 4ms is not possible at all. On the other hand, applications running sample time of multiples of 4ms should work properly. I have performed some experiments and I have not noticed any problems while running model on sample time of any multiple of 4ms. However, the reliability cannot be guaranteed as it is not a real-time system and good results were achieved just due to the high performance of the processor which is not fully employed by the simple simulation. Generally, this system is not very suitable for running applications requiring sample time of a few milliseconds. However, the experiences from another MPC5200 based computer used at the Department of Control Engineering show that Linux kernel with real-time patch can ensure reliable behavior in all situations. This is promising direction for the system improvement.

Bibliography

[AM] Analogue and Micro. Boa5200 specification. *http://www.analogue-micro.com/powerpc/BOA5200.pdf*.

[Ber] BerliOS. Socketcan driver. *http://developer.berlios.de/projects/socketcan/*.

[CiAa] CiA. Can bus. *http://www.can-cia.org/index.php?id=170*.

[CiAb] CiA. Canopen network. *http://www.can-cia.org/index.php?id=171*.

[DoCE] CTU FEE Department of Control Engineering. Boa wiki. *http://rtime.felk.cvut.cz/hw/index.php/Boa5200_HOWTO*.

[Ghia] Ghisler. Total commander. *http://download.totalcommander.cz/*.

[Ghib] Ghisler. Total commander packer plugin. *http://www.totalcmd.net/plugring/targzbz2.html*.

[Hat] Red Hat. Redboot guide. *http://ecos.sourceware.org/docs-latest/redboot/redboot-guide.html*.

[Jel08] Pavel Jelínek. Podpora simulace s hardware ve smyčce http://dce.felk.cvut.cz/dolezilkova/diplomky/2008/dp_2008_jelinek_pavel/dp_2008_jelinek_pavel.pdf. Master's thesis, Faculty of Electrical Engineering, Czech Technical University, 2008.

[Joh] Matt Johnston. Dropbear ssh. *http://matt.ucc.asn.au/dropbear/dropbear.html*.

[LOLa] LOLITech. Canfestival api documentation. *http://www.canfestival.org/api/*.

[LOLb] LOLITech. Canfestival driver. *http://www.canfestival.org/*.

[Mat] Mathworks. Real time workshop embedded coder documentation. *http://www.mathworks.com/access/helpdesk/help/toolbox/ecoder/index.html*.

[min] Mingw web. *http://www.mingw.org*.

[Píš05] Pavel Píša. Periodic threads. *http://dce.felk.cvut.cz/por/hlavni_uloha/motor.html*, 2005.

[Wag] Wago. Wago 750-348. *http://www.wago.com/wagoweb/documentation/index_e.htm*.

[Wik] Wikipedia. Canopen. *http://en.wikipedia.org/wiki/CANopen*.

Appendix A

Target usage example

This chapter describes step by step the process of generating model code using Linux ERT Target and its execution on the BOA machine. In the aim of creating as simple example as possible, we will use very simple model without CANopen support. However the model will use two different sample times to try the multitasking program (see section 2.4.1).

Before starting the model creation and code generation it is necessary to have the Linux ERT target installed as well as the MinGW environment (see section 2.2). The BOA machine has to be configured according to section 2.2.4 and turned on.

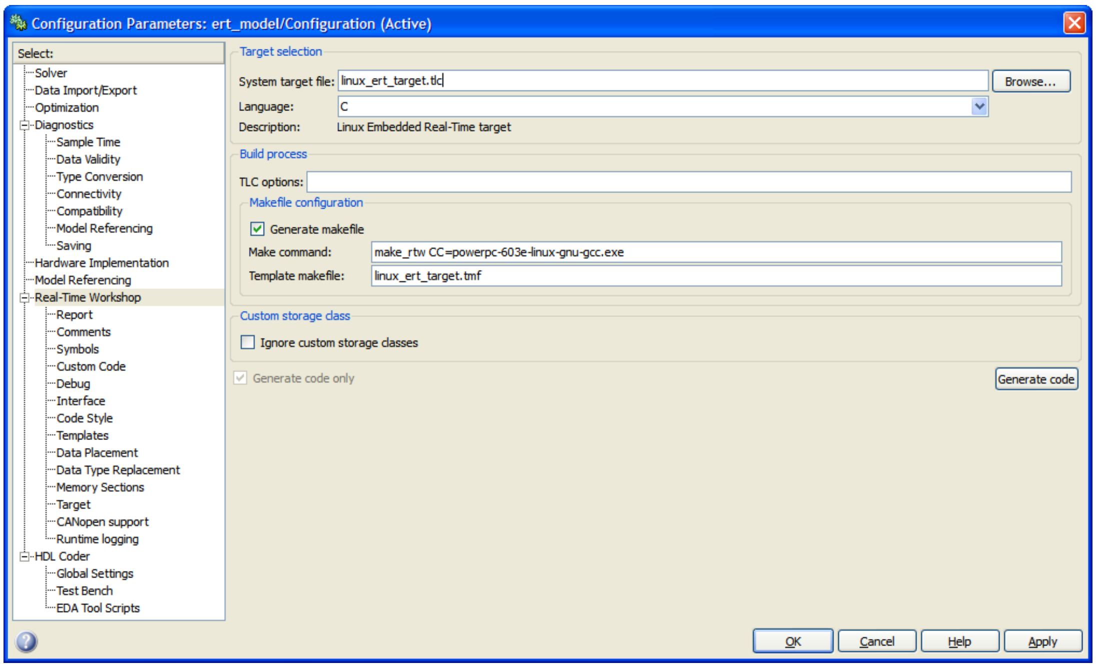
A.1 Model target configuration

First of all create a folder called lets say *ert_example* and new Simulink model (called *ert_model.mdl*) inside it. Click *Simulation/Configuration parameters* in the model menu. In *Real Time Workshop* section click *Browse* and choose *linux_ert_target.tlc* from the given list (see figure A.1). The target contains callback after its choice. The default settings is loaded in this callback function so that some simulation parameters are set automatically. Nevertheless we will have a look at the settings important for the code generation.

Click on the *Solver* pane in the configuration dialog. The fundamental step size is set here. It is the sample period of the main loop (see section 2.4.1). If it is set to the value *auto* it is calculated automatically as a greatest common divisor of all sample times used in the model. The other important option is the *Tasking mode.* It is recommended to keep the *auto* value as well. Then the singletasking mode is chosen if just one sample time is used or multitasking for more sample times.

The other important setting is the *External mode* configuration in *Real Time Workshop/Interface* pane (figure A.2). The external mode provides the TCP connection between Simulink and the model running on the Target machine and transfers the simulation signals to the Simulink window during the simulation progress. The *MEX-file arguments* have following meaning: 192.168.123.199 is the IP address of the BOA module, 1 enables the communication state messages and 17725 is the communication port of TCP connection.

Previous paragraph described the external mode settings which is in fact the communication settings on the side of host computer. There is the configuration of BOA module side communication in *Real Time Workshop/Target* pane. This dialog contains


Configuration Parameters: ert_model/Configuration (Active)

Target selection

System target file: linux_ert_target.tlc

Language: C

Description: Linux Embedded Real-Time target

Build process

TLC options:

Makefile configuration

☑ Generate makefile

Make command: make_rtw CC=powerpc-603e-linux-gnu-gcc.exe

Template makefile: linux_ert_target.tmf

Custom storage class

☐ Ignore custom storage classes

☑ Generate code only

Figure A.1: Model configuration to use Linux ERT Target

information like name and password of the user which should be used for login to the BOA computer (use *root* according to the section 2.2.4 to avoid authority problems), BOA IP address and model executive options. This options have to contain parameters *-w -port 17725* to enable external mode and set its port to 17725.

The last option defines what operations should be done with the generated code. It can be just generated and compiled or copied to the BOA Target or finally executed there too. We will set it to the *compile_copy_execute* option.

The next configuration pane is called *CANopen support*. The support of CANopen blockset is configured here. However we do not want to use it and so we will keep the support disabled.

The last pane is *Runtime logging*. You can set the console and file event logging level here.

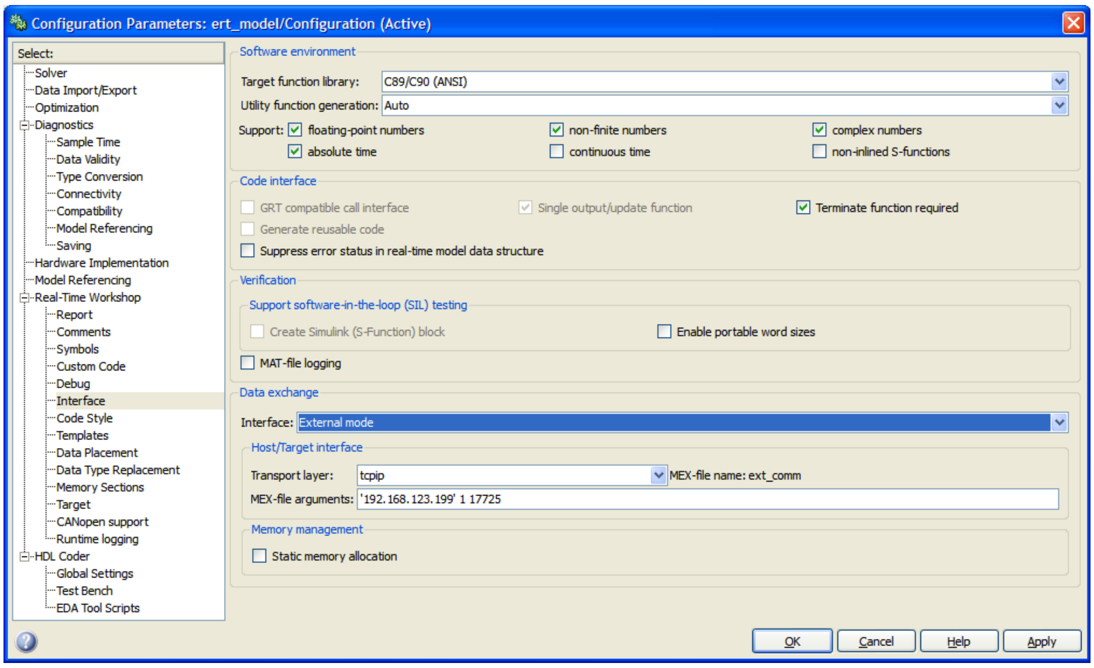
The target configuration is finished now. Press *Apply* to store the values and switch to the model window.

A.2 Model creation

While having the configuration done we have to create the Simulink model itself to have something to be generated. As we want to have a simple model but using two sample times, we will create two discrete sinus wave generators. Output values of these generators will be cut by saturation blocks and displayed by scopes.

To do this open Simulink Library and drag two *Sine Wave* blocks from *Sources* section to the model. Set the sample time of the first Sine wave to 0.2. Then set the sample time of the second Sine wave to [0.5 0.1] (0.1 is the time offset of the block).

Now drag two *Saturation* blocks from the *Discontinuous* section and two *Scope* blocks from the *Sinks* section to the model and connect everything according to the figure A.4.



Software environment

Target function library: C89/C90 (ANSI)

Utility function generation: Auto

Support: ✓ floating-point numbers ✓ non-finite numbers ✓ complex numbers

✓ absolute time ☐ continuous time ☐ non-inlined S-functions

Code interface

☐ GRT compatible call interface ✓ Single output/update function ✓ Terminate function required

☐ Generate reusable code

☐ Suppress error status in real-time model data structure

Verification

Support software-in-the-loop (SIL) testing

☐ Create Simulink (S-Function) block ☐ Enable portable word sizes

☐ MAT-file logging

Data exchange

Interface: External mode

Host/Target interface

Transport layer: tcpip MEX-file name: ext_comm

MEX-file arguments: '192.168.123.199' 1 17725

Memory management

☐ Static memory allocation

Figure A.2: External mode settings

The settings of these blocks can be left as it is.

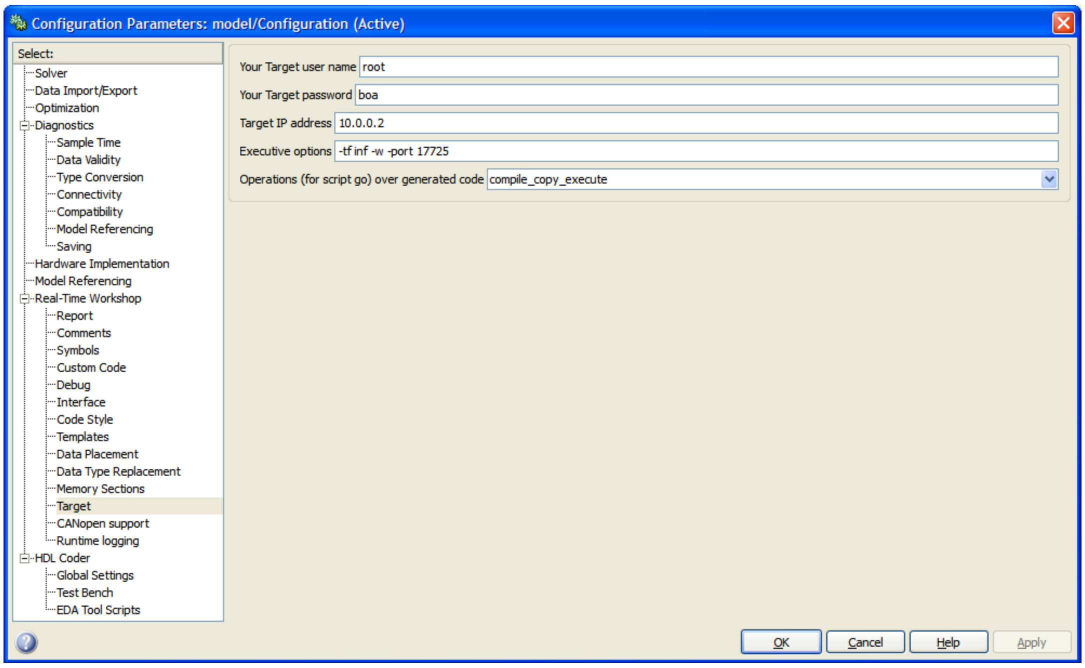
Now switch back to the Configuration dialog and press *Generate code* in the *Real Time Workshop* pane. The code generation will start and after it finishes, *model_linux_ert_target* folder will be created in the working directory. This folder contains generated files and Makefile prepared to be build (see section 2.4).

A.3 Code building and execution

While having the code generated run the MSYS console and open the model folder in it. The *go* script (see section 2.4.3) is created there. This script contains commands for build and execution of the code according to *Real Time Workshop/Target* pane of the configuration dialog. We have chosen *compile_copy_execute* option. It means that the *go* script will first compile the code and obtain model executable. Then it will be copied to the BOA machine and finally executed.

Note It is possible to use real Windows paths in MSYS but with slash instead of backslash as folder separator (e.g. C:/work/ert_example).

Run the *go* script. If the building finishes successfully the *ert_model* executable file will be created. The connection with the BOA machine is established, the executable is send and started at the Target computer. If no error either in building or communication occurs, the program should be running now. Switch to the model, set *Simulation/External* to enable external mode and click *Connect To Target*. While the connection is established the *Start real-time code* button should become enabled. Click it and start the simulation. You can open the scopes and check the process. To finish the simulation click the *Stop* button. Then the code running at the Target terminates and has to be started again to


Configuration Parameters: model/Configuration (Active)

Select:
- Solver
- Data Import/Export
- Optimization
- Diagnostics
  - Sample Time
  - Data Validity
  - Type Conversion
  - Connectivity
  - Compatibility
  - Model Referencing
  - Saving
- Hardware Implementation
- Model Referencing
- Real-Time Workshop
  - Report
  - Comments
  - Symbols
  - Custom Code
  - Debug
  - Interface
  - Code Style
  - Templates
  - Data Placement
  - Data Type Replacement
  - Memory Sections
  - Target
  - CANopen support
  - Runtime logging
- HDL Coder
  - Global Settings
  - Test Bench
  - EDA Tool Scripts

Your Target user name	root
Your Target password	boa
Target IP address	10.0.0.2
Executive options	-tf inf -w -port 17725
Operations (for script go) over generated code	compile_copy_execute

Figure A.3: Target settings

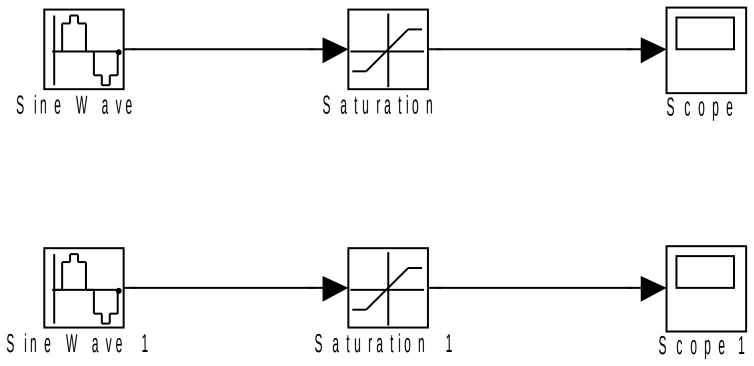


Figure A.4: Example model

run the simulation once more. It can be started by *go* script or manually after connecting to the BOA via SSH by the following commands.

```
$ ssh root@192.168.123.199
$ ./model -tf inf -w -port 17725
```

This is end of the example. Mind that after making any change of the model, the code has to be regenerated and the *go* script has to be performed as well to build the program and start at the BOA.

Appendix B

CANopen blockset usage example

This tutorial describes step by step the creation of a simple controller with CANopen communication. This example is called *controller* and all the files used in this tutorial should be available in the blockset distribution.

Note The CANopen blockset and this example as well are based on model using Linux ERT Target. It is essential to be familiar with using it before starting using the blockset. Performing the target tutorial first will help you.

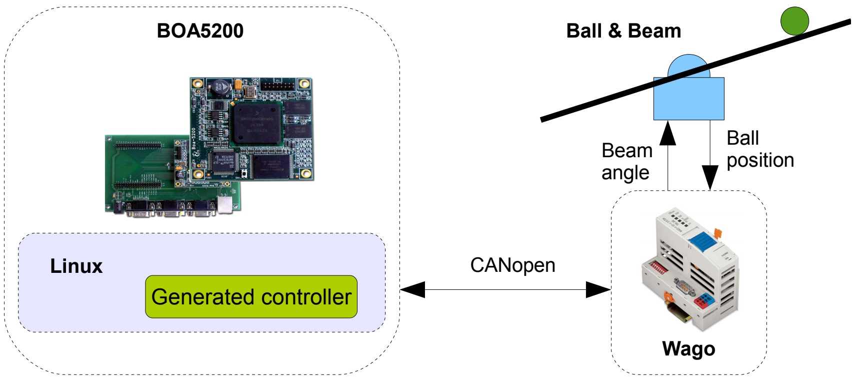


Figure B.1: Control example

B.1 Definition of the control problem

Imagine that you have *Ball and Beam* model and you want to control the ball position at the beam (figure B.1). The output of the system is the ball position measured as voltage and you can control the beam angle by changing the motor input voltage. It means that the model is the SISO (*Single-Input Single-Output*) system described by the following discrete transfer function with sample time $T_s = 0.05s$.

$$G = \frac{0.000304z^2 + 0.00096z + 0.00018}{z^3 - 2.364z^2 + 1.731z - 0.3665}$$

This system will be controlled by filtered *PD* controller with following transfer function and the same sample time as the system.

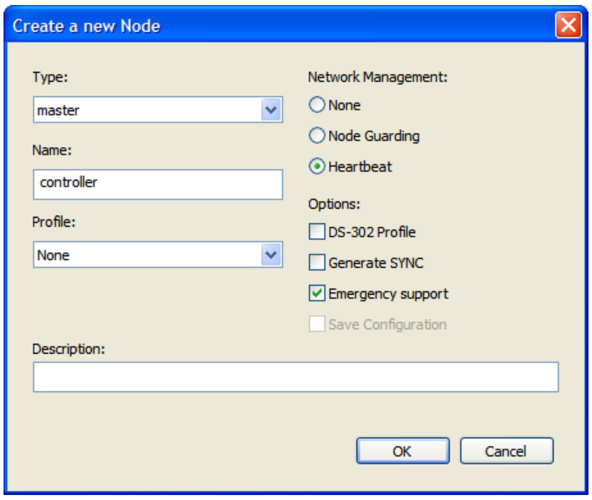
$$C = \frac{5.76z - 5.452}{z - 0.7408}$$

We will use BOA5200 computer [AM] to implement the controller. The interface between model analogue input and output and the computer CAN peripheral will be realized by Wago distributed IO in version 750 with CANopen support [Wag]. We have Wago 750-348 main module and Wago 750-459 (4 analogue inputs) and Wago 750-559 (4 analogue outputs). However, we will use just one input and one output as the model is SISO. This device uses one PDO message to transfer all 4 inputs and one for all 4 outputs. This is the PDO number 2 which has message ID *280hex + NodeID* in the direction from Wago to controller and ID *300hex + NodeID* in the opposite direction. We will set the Wago node ID to 2.

After defining the task we can start with the controller design. It is necessary to have the Linux ERT Target and CANopen blockset correctly installed (see section 3.2).

B.2 Creation of the node Object Dictionary

First of all, we have to create CANopen node object dictionary in OD editor (*objdictedit.py*) and generate the necessary files (see section 3.3.1). Open the OD editor and create a new project. Fill the setup dialog according to the figure B.2. This says that the node will be master in the CANopen network, the Object Dictionary will be called *controller* and the node will use *Heartbeat* service to secure the network functionality. Click *OK* after having this filled.


Create a new Node

Type: master

Name: controller

Profile: None

Network Management: ( ) None ( ) Node Guarding [✓ Heartbeat]

Options:
- [ ] DS-302 Profile
- [ ] Generate SYNC
- [✓] Emergency support
- [ ] Save Configuration

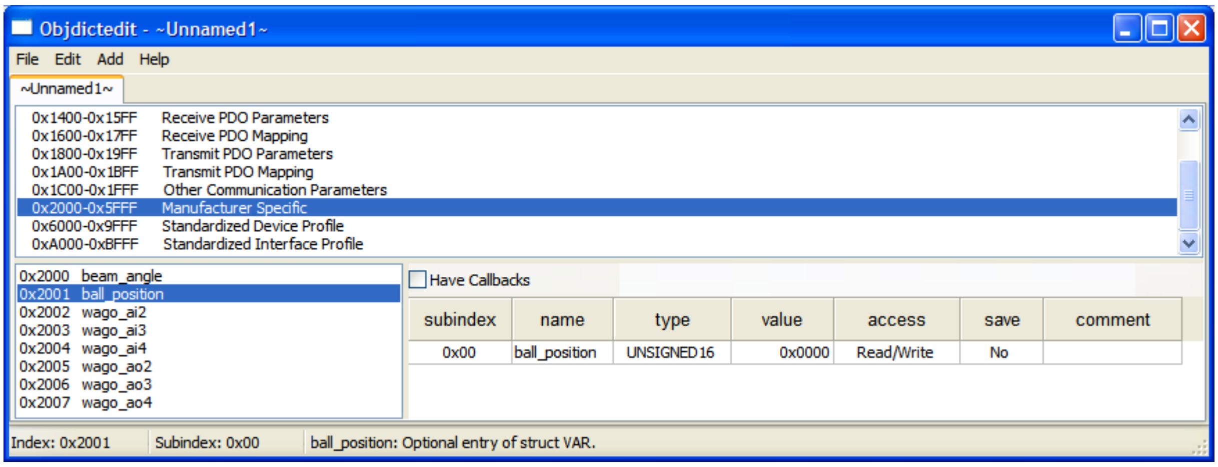
Description:

Figure B.2: Creating new Object Dictionary

We will set up the *Heartbeat* service now. Open *Edit/DS-301 profile*, move index 0x1016 (*Consumer Heartbeat Time*) to the right pane and click *OK*. Open *Communication Parameters* in the main window. Indices 0x1016 and 0x1017 should be available there. Set the value of 0x1017 (*Producer Heartbeat Time*) to 1000ms (0x03E8). This will produce the Heartbeat message every second. Additionally, set the value of index 0x1016 (*Producer Heartbeat Time*) to 0x00020064. This says that if the answer from the node ID 2 (Wago) was not received in 100ms (0x64), the Heartbeat error should be announced. We will handle this error by the appropriate callback in the model.

We will use one SDO message to configure the Wago behavior and we have to register it in the OD now. Click *SDO parameters* and add a SDO client. Set its subindices 1 (Transmit SDO) and 2 (Receive SDO) to 0x602 and 0x582.

Next, we will create internal node variables in Manufacturer specific section. First, create two variables *beam_angle* and *ball_position* that will represent the input and output of the model. However, PDOs defined in Wago contains values of all 4 analogue inputs and outputs so that we have to create variables that will be mapped onto the rest of the message. Create 3 more variables for the 3 inputs and 3 more variables for 3 outputs. Set the data type of all the created variables to *UNSIGNED16*. The variable definition should look like in the figure B.3.



0x1400-0x15FF	Receive PDO Parameters
0x1600-0x17FF	Receive PDO Mapping
0x1800-0x19FF	Transmit PDO Parameters
0x1A00-0x1BFF	Transmit PDO Mapping
0x1C00-0x1FFF	Other Communication Parameters
0x2000-0x5FFF	Manufacturer Specific
0x6000-0x9FFF	Standardized Device Profile
0xA000-0xBFFF	Standardized Interface Profile

0x2000	beam_angle
0x2001	ball_position
0x2002	wago_ai2
0x2003	wago_ai3
0x2004	wago_ai4
0x2005	wago_ao2
0x2006	wago_ao3
0x2007	wago_ao4

subindex	name	type	value	access	save	comment
0x00	ball_position	UNSIGNED16	0x0000	Read/Write	No	

Index: 0x2001 Subindex: 0x00 ball_position: Optional entry of struct VAR.

Figure B.3: Costume variables definition in Object dictionary

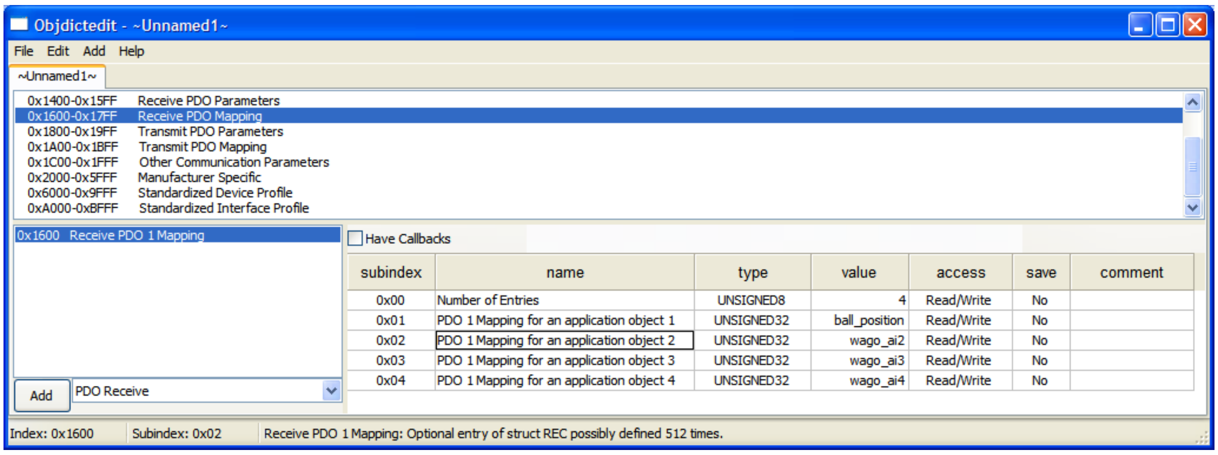
Finally, we have to define receive and transmit PDO messages and their mapping on the variables. Add one *Receive PDO* in the *Receive PDO parameters* section. Then, set its ID to 0x282 and the *Transmission type* to 0x01 (synchronized by SYNC messages). Now, click on the *Receive PDO mapping* section, add one mapping object and add 4 subindices by right-clicking the single entry. Choose variables created as inputs in the mapping objects value column. First object should be the variable representing the ball position as the first input of the Wago module (figure B.4). Create *Transmit PDO* in the same way as the *Receive PDO* and use 0x302 message ID. Use output variables in the mapping with the beam angle variable at the first position.

The Object Dictionary is completely defined now. Save it as *controller.od* and click *File/Export to EDS file* and *File/Build Dictionary*. This will produce *controller.eds*, *controller.c* and *controller.h* files. You can close the OD editor now and continue by creating the Simulink model.

B.3 Creation of the Simulink model

Create a new model in Simulink and save it with the name *model.mdl* (model name has to be different from OD name). Place *controller.eds*, *controller.c* and *controller.h* files created in the previous section into the model folder.

First of all, we will set the target in *Simulation/Configuration parameters* menu. Choose Linux ERT target (*linux_ert_target.tlc* file) in the *Real Time Workshop* menu. Then, check *Enable CANopen blockset support* option in the *CANopen support* pane. Library and include folders should be determined automatically, if the target is installed correctly. Finally, enable and set the external mode parameters in *Target* and *Interface*



0x1400-0x15FF	Receive PDO Parameters
0x1600-0x17FF	Receive PDO Mapping
0x1800-0x19FF	Transmit PDO Parameters
0x1A00-0x1BFF	Transmit PDO Mapping
0x1C00-0x1FFF	Other Communication Parameters
0x2000-0x5FFF	Manufacturer Specific
0x6000-0x9FFF	Standardized Device Profile
0xA000-0xBFFF	Standardized Interface Profile

0x1600 Receive PDO 1 Mapping

PDO Receive

subindex	name	type	value	access	save	comment
0x00	Number of Entries	UNSIGNED8	4	Read/Write	No	
0x01	PDO 1 Mapping for an application object 1	UNSIGNED32	ball_position	Read/Write	No	
0x02	PDO 1 Mapping for an application object 2	UNSIGNED32	wago_ai2	Read/Write	No	
0x03	PDO 1 Mapping for an application object 3	UNSIGNED32	wago_ai3	Read/Write	No	
0x04	PDO 1 Mapping for an application object 4	UNSIGNED32	wago_ai4	Read/Write	No	

Index: 0x1600 Subindex: 0x02 Receive PDO 1 Mapping: Optional entry of struct REC possibly defined 512 times.

Figure B.4: Receive PDO mapping onto internal variables

pane according to your BOA and network settings. See Linux ERT Target documentation for more information about this.

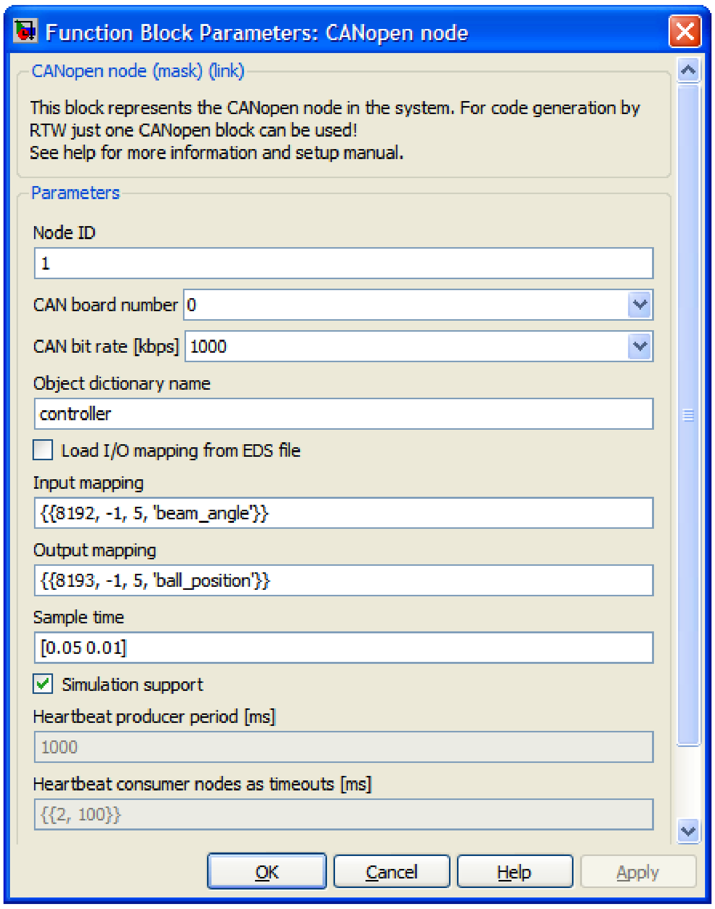
We will create the model itself now. Open the Simulink Library Browser and drag the *CANopen node* block from *Linux ERT Target blockset/CANopen blockset* section into the model. Open Block parameters dialog and set it according to the figure B.5. If you insert the OD name (*controller*) and click *Load I/O mapping from EDS file* the mapping is loaded automatically in the way that each variable mapped into some PDO is used as input or output of the block. However, we need only one input and one output variable so that you can delete the 3 more entries in both cells. The block sample time will be $0.05s$ as the rest of the model and the network communication as well. Finally, enable the Simulation support to create the simulation input and output. This will enable testing the controller just in the Simulink without using hardware.

After having the main block configured, we can employ other CANopen blocks. Firstly, drag the *SYNC generator* block to the model and set its OD name to *controller* and sample time to [0.05 0.04]. This will produce SYNC messages in the CANopen network with the period of $0.05s$ delayed $0.04s$ after the model update. It means in fact that the model update will be performed $0.01s$ after the SYNC message so that the data transfers are surely finished.

B.3.1 CANopen callbacks

We need to handle some asynchronous events of the CANopen network. To do this we will use *CANopen callbacks* block. Drag it to the model and set it to provide three types of callbacks - Boot up of the Wago device, *Heartbeat error* occurrence and stopping the local node at the end of the simulation (figure B.6).

According to the settings the Callbacks block has one output port of the width 3. It means that the port has to be split by *Demux* block into 3 separate signals first. The output port is the function call port that should be used to trigger Function call subsystem. The callback parameters are stored in the global memory by the block and can be processed by appropriate parsers. Drag 3 *Function-Call subsystem* blocks into the model and connect their triggers to the 3 output signals of the *Callbacks* blocks. Then open each subsystem and create there the code handling particular event.


Function Block Parameters: CANopen node

CANopen node (mask) (link)

This block represents the CANopen node in the system. For code generation by RTW just one CANopen block can be used!
See help for more information and setup manual.

Parameters

Node ID: 1

CAN board number: 0

CAN bit rate [kbps]: 1000

Object dictionary name: controller

☐ Load I/O mapping from EDS file

Input mapping: {{8192, -1, 5, 'beam_angle'}}

Output mapping: {{8193, -1, 5, 'ball_position'}}

Sample time: [0.05 0.01]

☑ Simulation support

Heartbeat producer period [ms]: 1000

Heartbeat consumer nodes as timeouts [ms]: {{2, 100}}

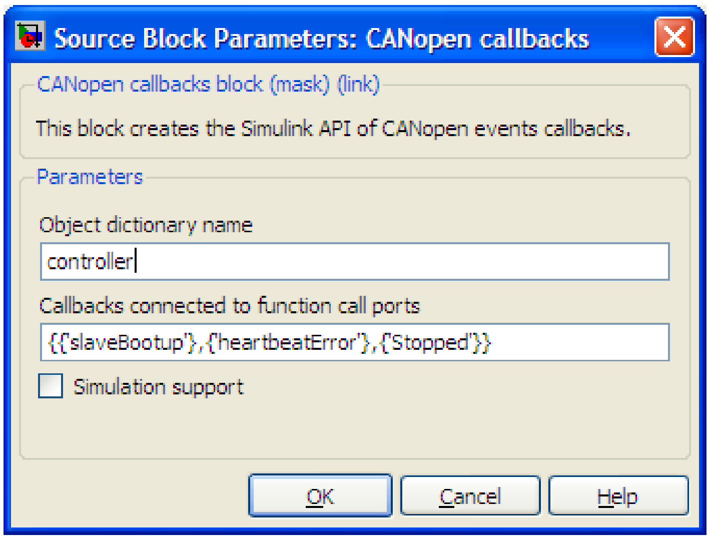
Figure B.5: CANopen node block parameters

First function call subsystem will be connected to the first Callback block output signal that is *Slave boot up* event. This callback has a single parameter - ID of the node which booted up. To get this parameter we have to place *Slave boot up parser* block into the subsystem and display its input value.

We will perform Wago device initialization and start both nodes after reception of the boot up message from Wago. *CANopen asynchronous* block is designed to do such operations. Drag this block to the subsystem and open the parameters dialog. The OD name is *controller* again and the *Operations parameter cell* should contain the following text.

```
{{'writeNetworkDict', {'2', '0x1801', '2', '1', 'UNS8'}}, {'masterSendNMTStateChange', ↩
    {'2', 'NMT\_Start\_Node'}}, {'setState', {'Operational'}}}
```

The first operation sends SDO message to the Wago and enables PDO transmission after SYNC message reception. The second operation sets Wago into *Operational* state and the last sets local node to *Operational* state as well. It is a good idea to check SDO transfer failure. To do this split error output of the block into 3 signals and display the value of the first one by the *Display* block. The subsystem content is shown in the figure B.7. Moreover, it is necessary to insert *Asynchronous rate transition* block between the subsystem outputs and displays to ensure data integrity between synchronous and asynchronous part of the model. Set the sample time of rate transitions to 0.05 as is the synchronous model sample time.


Source Block Parameters: CANopen callbacks

CANopen callbacks block (mask) (link)

This block creates the Simulink API of CANopen events callbacks.

Parameters

Object dictionary name

controller

Callbacks connected to function call ports

{{'slaveBootup'},{'heartbeatError'},{'Stopped'}}

☐ Simulation support

Figure B.6: Callbacks block parameters

The second callback is activated after Heartbeat error. To handle it place *Heartbeat error parser* block into the appropriate function call subsystem and set its OD name to *controller* again. Display its output at the *Display* block. It will show the ID of the node which produced the heartbeat error (did not answer on the server request). Do not forget about the *Asynchronous rate transition* block.

The last subsystem will perform what should be done after stopping the local node at the end of the simulation. We will stop Wago by the *CANopen asynchronous* block and the *masterSendNMTStateChange* operation.

B.3.2 Creating the controller

We have already implemented all the required functionality of the CANopen node. Now, we will create the controller which will communicate using this node. The Wago analogue output produces signal of voltage in range $< 0V; 10V >$. This is, however, transmitted in PDO messages in data type *16-bit unsigned integer*. This data type is used for CANopen node block ports as well. Moreover, controlling the ball rolling at the beam requires using negative voltage (this would have to be solved electronically). We have to create converter from double value in range $< -5V; 5V >$ to unsigned value in range $< 0; 65535 >$ (16-bit) and the same in the opposite direction. These converters are shown in the figure B.8.

We can use *LTI system* block from *Control System Toolbox* of Simulink to implement the controller. Drag this block to the model and set its parameter to the controller discrete transfer function.

```
tf([5.76  -5.452],[1  -0.7408],0.05)
```

To enable testing the controller in Simulink we will create Ball and Beam mathematical model in the same way as the controller with the following transfer function.

```
tf([0.000304  0.00096  0.00018],[1  -2.364  1.731  -0.3665],0.05)
```

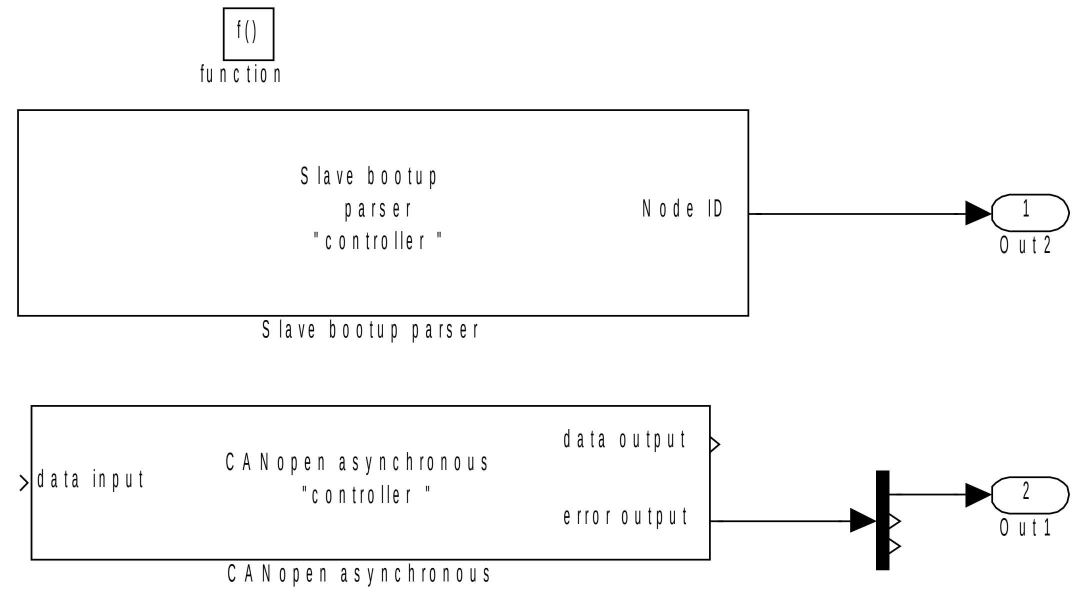


Figure B.7: Function call subsystem triggered by *Slave boot up* callback

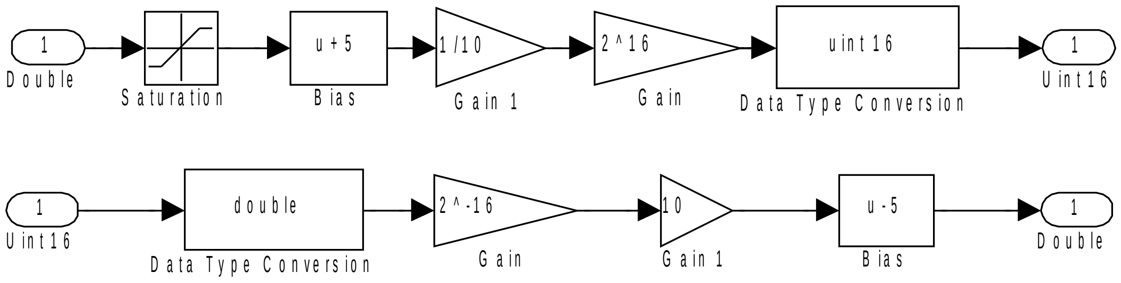


Figure B.8: Signal converter subsystems

Note We do not want to generate code of these blocks used just for the simulation. To avoid generating the simulation loop use *Environment controller* blocks to switch the CANopen block simulation input and output between simulation loop (simulation case) and *Ground* (code generation case).

Finally, we can connect the controller with the data types converters in the closed control loop using input and output of the CANopen node block. In the same way the simulation model will be connected to the simulation I/O. As a reference signal we can use e.g. Step function. The complete model is in the figure B.9 and the colors marks sections with the same sample time (right-click the model and enable *Format/Sample Time Colors*). Now, you can press the *Start simulation* button and see the simulation results in the scope window.

B.4 Generating and compiling the code

To generate the model code click *Generate code* button in the simulation parameters (or press *Ctrl+B*). This will generate the necessary files as well as the *go* script which performs the code compilation and copying into the BOA computer. This is described

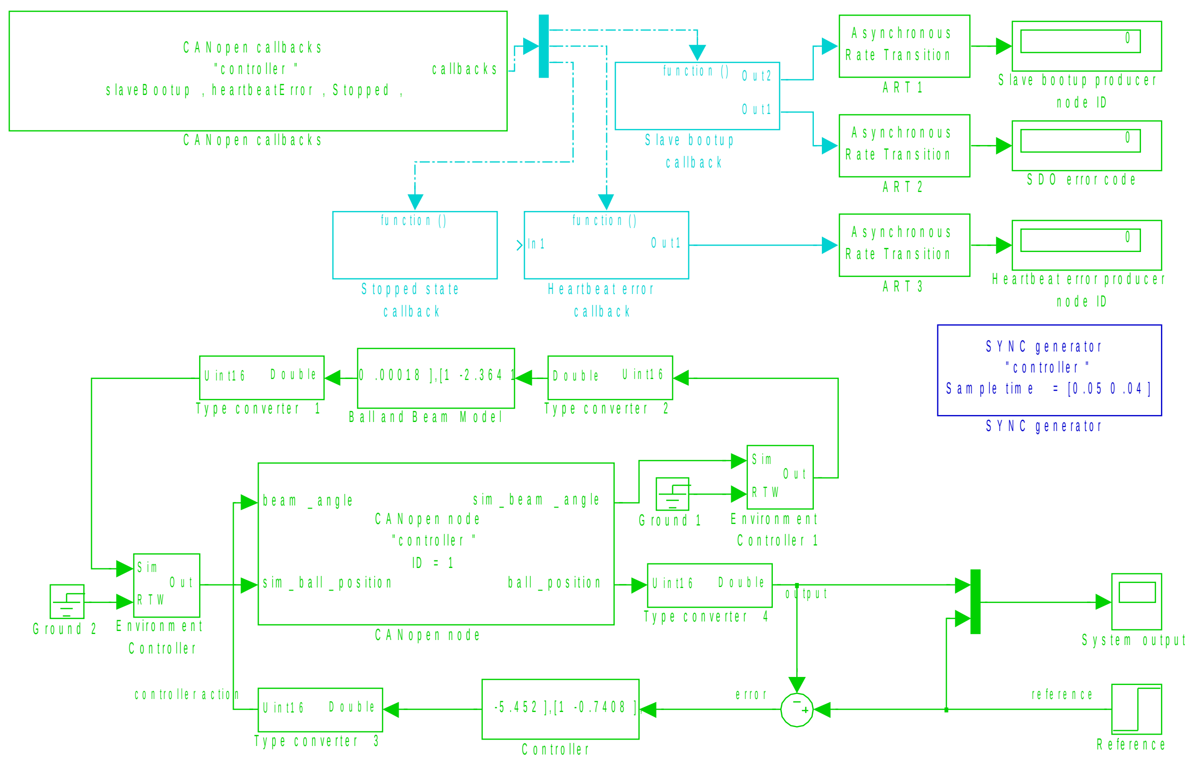


Figure B.9: Final CANopen node model

in the Linux ERT Target documentation. While having the generated program running at the target BOA computer, you can switch to external simulation mode and start the simulation as usual. It will perform the model control process in the real hardware instead of the simulation LTI model as before.

Appendix C

Blockset common use cases

The following section briefly describes a set of simple demo models distributed together with the blockset. The demos show the blockset used for creating the common CANopen node types. Each demo contains the Simulink model prepared for code generation and necessary Object Dictionary files (*.od*, *.eds*, *.c* and *.h* files). The Object Dictionary is created in OD editor (see section 3.3.1) and it can be modified by running the Matlab command *objdictedit('odName.od')*.

It is necessary to customize the target settings in configuration parameters before starting the code generation of the demo model. There are some options in the *Real Time Workshop* pane that have to be filled according to your computer, BOA machine and network configuration. You have to set up the path of *CanFestival* driver libraries in *CANopen support* pane. It should be filled automatically, so disabling and enabling the support option should update the paths of library and include folders. Then, the BOA machine IP address has to be filled in *Target* and *Interface/External mode* panes.

C.1 Synchronous master node

This example shows how to create simple CANopen node with SYNC generator and a single input and output transfered by PDOs synchronized by SYNC. This model is executed synchronously with SYNC, but it inserts a unit delay into the system as it uses the received data for output calculation just in the next step (after the next SYNC). See figure C.1 for the task principle.

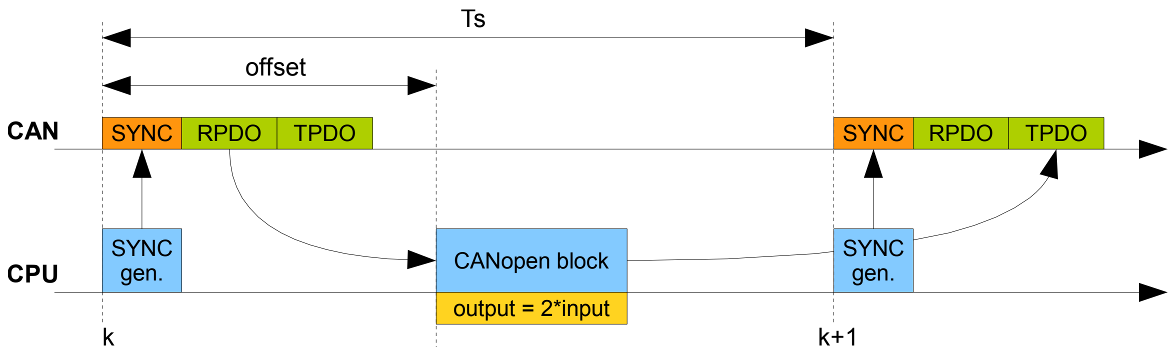


Figure C.1: Gantt diagram of example functionality

Object dictionary name used by all blocks is *sync_master_od*. It is the name of *.eds*, *.c*, *.h* files as well. These files were prepared in the OD editor. The only definition which is necessary to be done in the editor is the registration of a single RPDO and a single TPDO. The transmission type of these messages is 1 (transfered after SYNC). A single input variable of type *8-bit unsigned integer* is mapped into the RPDO and a single output variable of the same type is mapped into the TPDO.

SYNC generator's sample time is set to 1s. It means that it generates SYNC message once per second. The main block uses the same sample time delayed by offset 0.1s. It makes the block to be execute just 0.1s after SYNC which means that synchronous PDOs will have been surely transfered before the block execution.

Callbacks block registers the *PreOperational* callback. It activates the function call subsystem after the local node boot up. The Asynchronous block is placed in this subsystem and switches the local node into the *Operational* mode. See figure C.2 for the Simulink realization of this task using CANopen blockset.

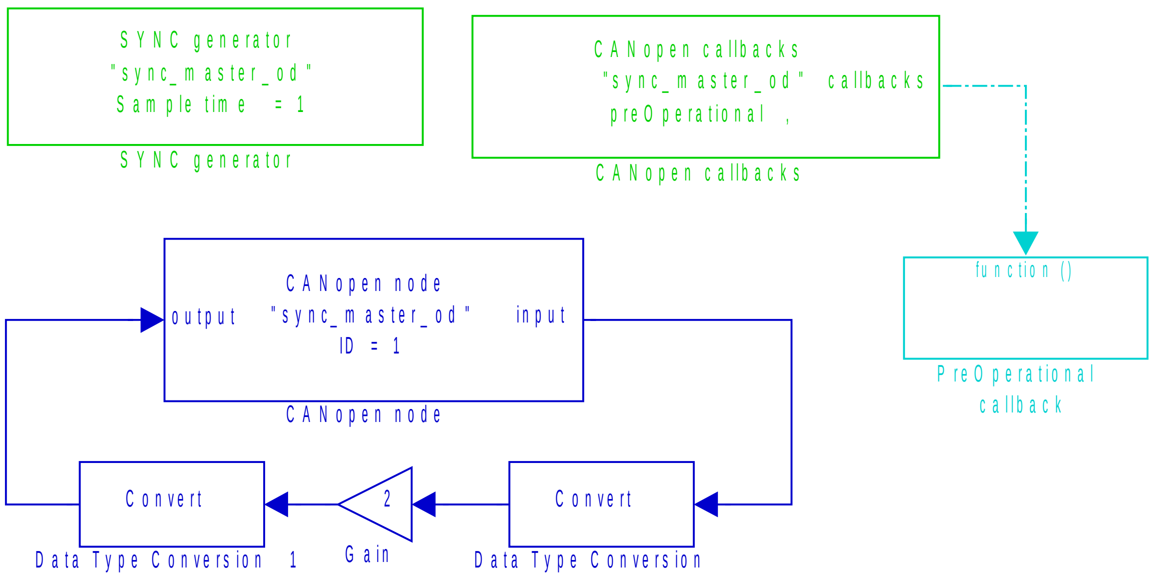


Figure C.2: Model of CANopen node used as SYNC master

C.2 Synchronous slave node

This example shows how to create simple CANopen node having single input and output transfered by PDOs synchronized by SYNC. However, the SYNC messages are generated by another node in the network. The local node is a SYNC slave. This model does not use any periodic sample time as it is synchronized by SYNC. However, the constant sample time should be set in configuration parameters *Solver* pane to control the program loop. This model inserts a unit delay into the system as it uses the received data for output calculation just in the next step (after the next SYNC). See figure C.3 for the task principle.

Object dictionary name used by all blocks is *sync_slave_od*. It is the name of *.eds*, *.c*, *.h* files as well. These files were prepared in the OD editor. The only definition which

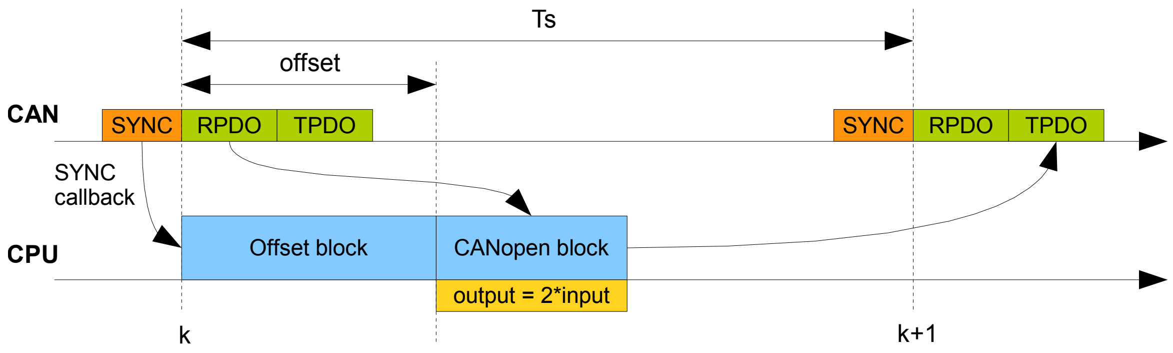


Figure C.3: Gantt diagram of example functionality

is necessary to be done in the editor is the registration of a single RPDO and a single TPDO. The transmission type of these messages is 1 (transfered after SYNC). A single input variable of type *8-bit unsigned integer* is mapped into the RPDO and a single output variable of the same type is mapped into the TPDO.

Callbacks block registers the *PreOperational* and *SYNC* callbacks. The first callback activates the function call subsystem after the local node boot up. The Asynchronous block is placed in this subsystem and switches the local node into the *Operational* mode. The second output of the Callbacks block activates the other function call subsystem after reception of SYNC message. The main block and the node calculations are performed in this subsystem. The CANopen block's sample time is set to -1 (inherited) as it activated by the callback. See figure C.4 for the Simulink realization of this task using CANopen blockset including the SYNC callback and offset subsystems content in figure C.5.

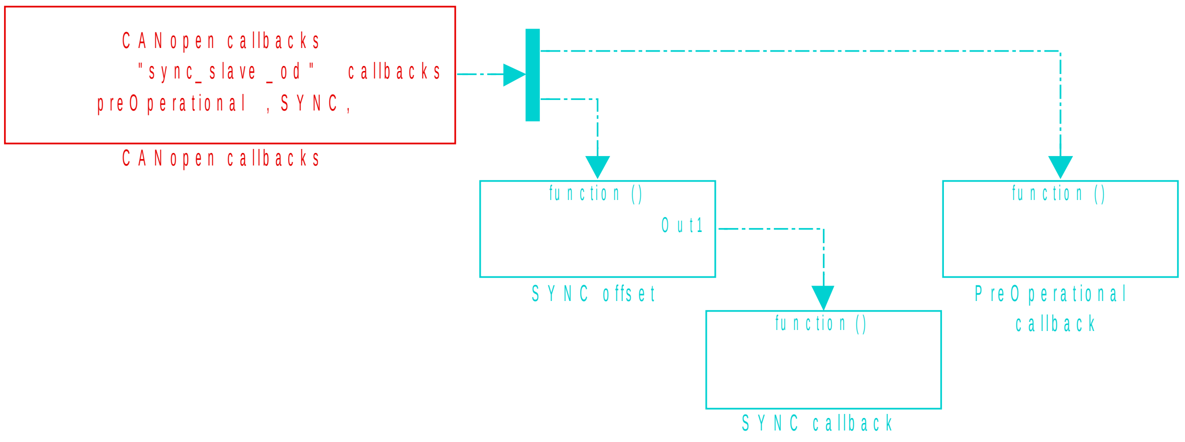


Figure C.4: Model of CANopen node as SYNC slave

C.3 Synchronous CANopen sensor

This example shows how to create CANopen node representing some sensor. This device measures (generates in the example) some value and sends it in the PDO just after reception of SYNC message. This model does not use any periodic sample time as it is

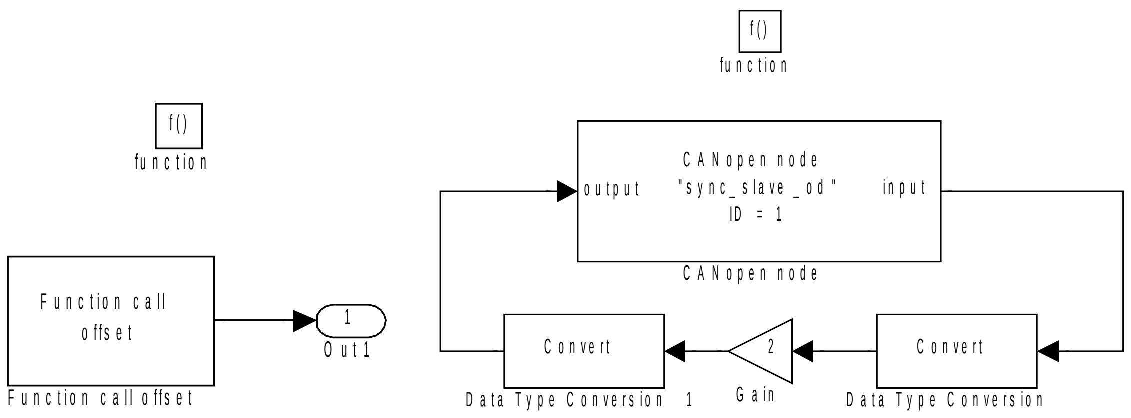


Figure C.5: Function call offset (left) which delays the node execution (right)

synchronized by SYNC. However, the constant sample time should be set in configuration parameters *Solver* pane to control the program loop. The main block does not perform any periodic functionality in this use case. It just loads the EDS file and controls the CANopen driver. Its sample time is set to -1 to inherit the fundamental sample time of the model. See figure C.6 for the task principle.

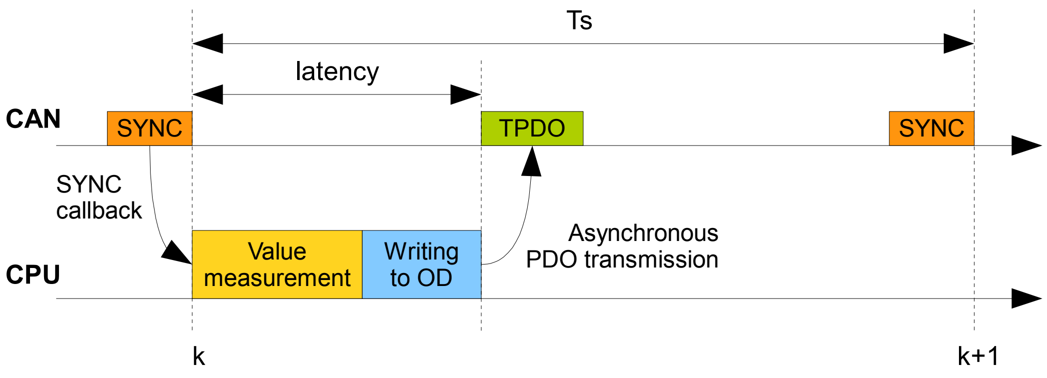


Figure C.6: Gantt diagram of example functionality

Object dictionary name used by all blocks is *sync_sensor_od*. It is the name of *.eds*, *.c*, *.h* files as well. These files were prepared in the OD editor. The only definition which is necessary to be done in the editor is the registration of a single TPDO. Its transmission type is 255 (transfered after change). After the SYNC reception, the measurement is performed first and the PDO is sent just after the new value is stored into Object Dictionary. Mind, that it is necessary to ensure fast enough value calculation between reception of SYNC and sending the PDO to preserve the synchronous behavior. A single output variable of type 8-bit unsigned integer is mapped into the TPDO.

Note After writing the value into OD, the *sendPDOevent* operation has to be used in Asynchronous block to send all asynchronous TPDOs. It is used in this example as well.

Callbacks block registers the *PreOperational* and *SYNC* callbacks. The first callback activates the function call subsystem after the local node boot up. The Asynchronous block is placed in this subsystem and switches the local node into the *Operational* mode.

The second output of the Callbacks block activates the other function call subsystem after reception of SYNC message. The counter simulating the real value measurement is placed there. Its value is then stored into the OD and the PDO is sent. See figure C.7 for the Simulink realization of this task using CANopen blockset. There is the SYNC callback subsystem content in the figure C.8.

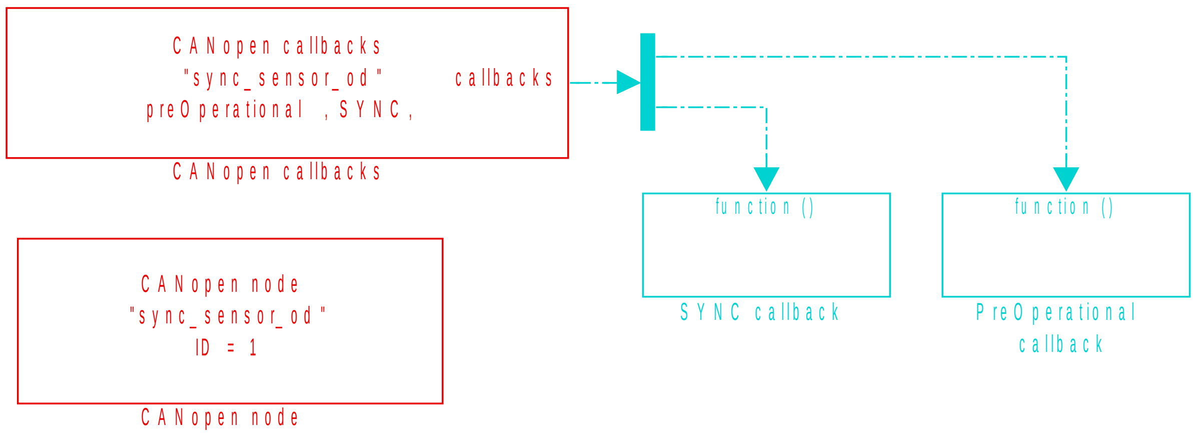


Figure C.7: Model of CANopen sensor

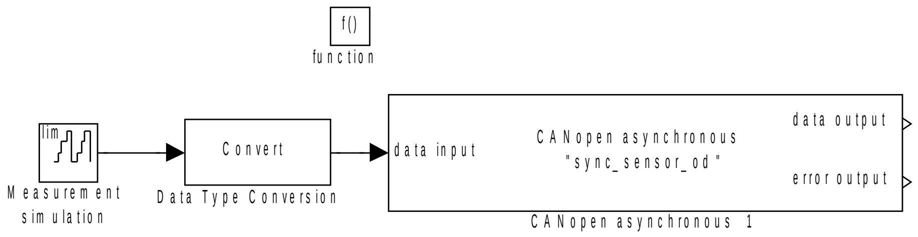


Figure C.8: Subsystem executed by SYNC callback

C.4 Asynchronous CANopen node

This example shows how to create CANopen node which works in asynchronous mode. It just waits for the change of its OD entry by received PDO. Then, it reads the value, performs some operation over it and store it into another OD entry. Just after the calculation, the PDO is transmitted. This model does not use any periodic sample time. The execution is supposed to be aperiodic. However, the constant sample time should be set in configuration parameters *Solver* pane to control the program loop. The main block does not perform any functionality in this use case. It just loads the EDS file and controls the CANopen driver. See figure C.9 for the principle of this example. The basic model of this example is almost the same as in the previous case (see figure C.7). The only change is that the second callback is not triggered by SYNC but by the OD entry change.

Object dictionary name used by all blocks is *async_change_od*. It is the name of *.eds*, *.c*, *.h* files as well. These files were prepared in the OD editor. The only definition which is necessary to be done in the editor is the registration of a single RPDO and a single

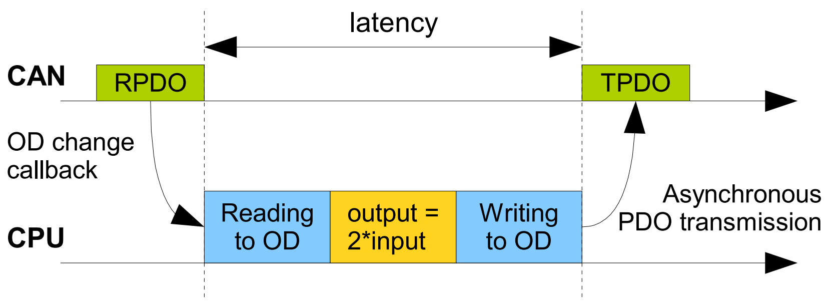


Figure C.9: Gantt diagram of example functionality

TPDO. The transmission type of these messages is 255 (transfered after value change). A single input variable of type 8-bit unsigned integer is mapped into the RPDO and a single output variable of the same type is mapped into the TPDO.

Callbacks block registers the *PreOperational* and *OD change* callbacks. The first callback activates the function call subsystem after the local node boot up. The Asynchronous block is placed in this subsystem and switches the local node into the *Operational* mode. The second output of the Callbacks block activates the other function call subsystem after change of OD index *2000 hex* (the RPDO is mapped there). This entry is read in the callback subsystem, multiplied by 2 and stored into the OD entry with index *2001 hex* (TPDO mapped variable). The TPDO is transfered after the data change (see the subsystem content in the figure C.10).

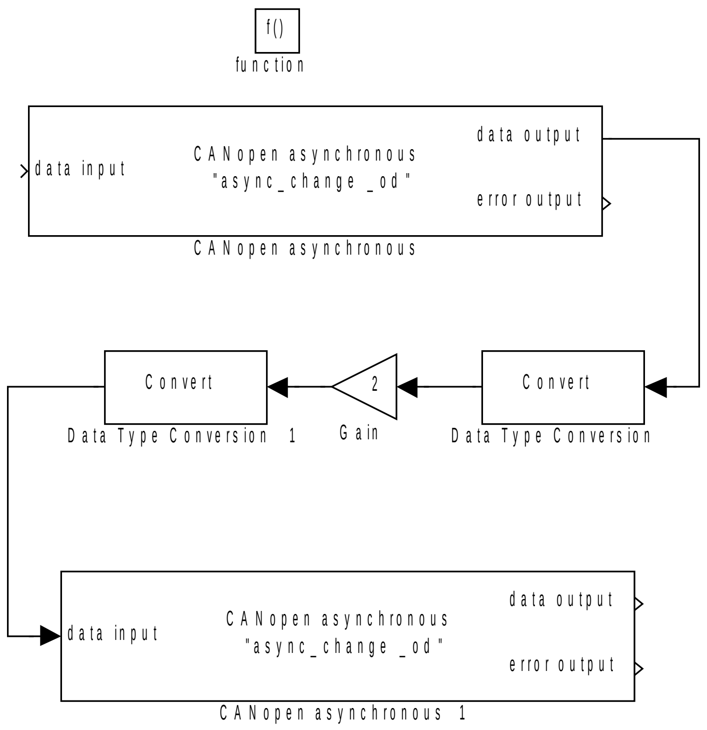


Figure C.10: OD change callback subsystem

C.5 Model with two CANopen nodes

This example (see figure C.11) shows how to create two CANopen nodes in a single model. It can be used for working with two separate CANopen networks at once. Each of them is connected to one of CAN ports provided by BOA computer. This example simulates the following situation. At the first network (CAN port 0) the SYNC master node synchronously receiving and transferring PDOs is created. The node is exactly the same as in *sync_master* example (see section C.1). Data of received RPDO are multiplied by a constant and transmitted in the next SYNC period by TPDO. This simulates the fast periodic control loop. Moreover, the device is connected to some supervisor station by another CANopen network. The second node using the appropriate CAN port is created to operate with this network. Its only function is to wait for asynchronous RPDO and use its value as a multiplication constant in the control loop. Mind that both nodes are working in the separate networks so that they can have the same node IDs and use messages having the same IDs as well.

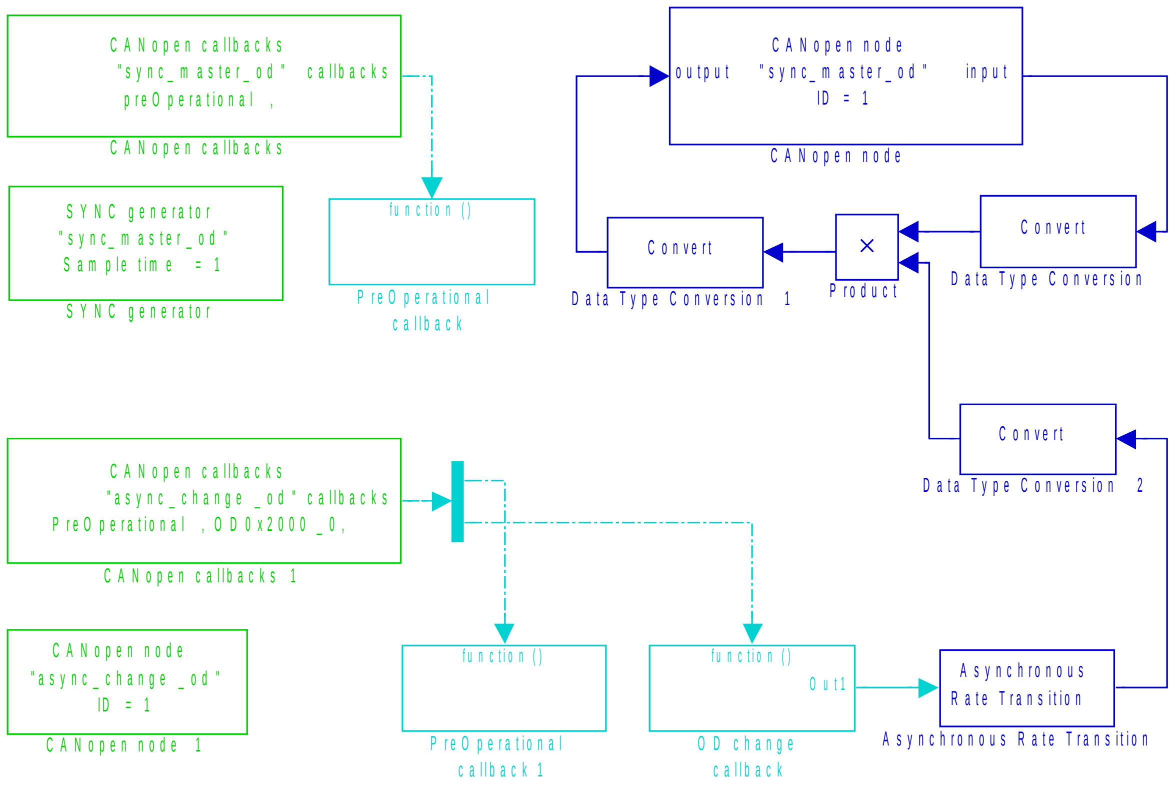

CANopen callbacks "sync_master_od" callbacks preOperational ,
CANopen callbacks
SYNC generator "sync_master_od" Sample time = 1
SYNC generator
function ()
PreOperational callback
CANopen node output "sync_master_od" input ID = 1
CANopen node
Convert
Data Type Conversion 1
×
Product
Convert
Data Type Conversion
Convert
Data Type Conversion 2
CANopen callbacks "async_change_od" callbacks PreOperational , OD 0x2000 _0,
CANopen callbacks 1
CANopen node "async_change_od" ID = 1
CANopen node 1
function ()
PreOperational callback 1
function () Out1
OD change callback
Asynchronous Rate Transition
Asynchronous Rate Transition

Figure C.11: Model with two CANopen nodes

Each node has its own Object dictionary (*.eds*, *.c*, and *.h* files as well). The OD name parameter of each block is used to recognize which OD the block works with. The first node OD is called *sync_master_od* and it is exactly the same as in *sync_master* example. The second node OD is called *async_change_od* and it is almost the same as in the *async_change* example (see section C.4). In contrast to that example, it does not use any TPDO and its RPDO mapped variable is called *async_input* instead of *input* to avoid variable name conflict with the other OD while building. The blocks used in both nodes are the same as in the mentioned examples. Moreover, at the connection link between asynchronous and synchronous part of the model, the Asynchronous Rate Transition block has to be used.

Appendix D

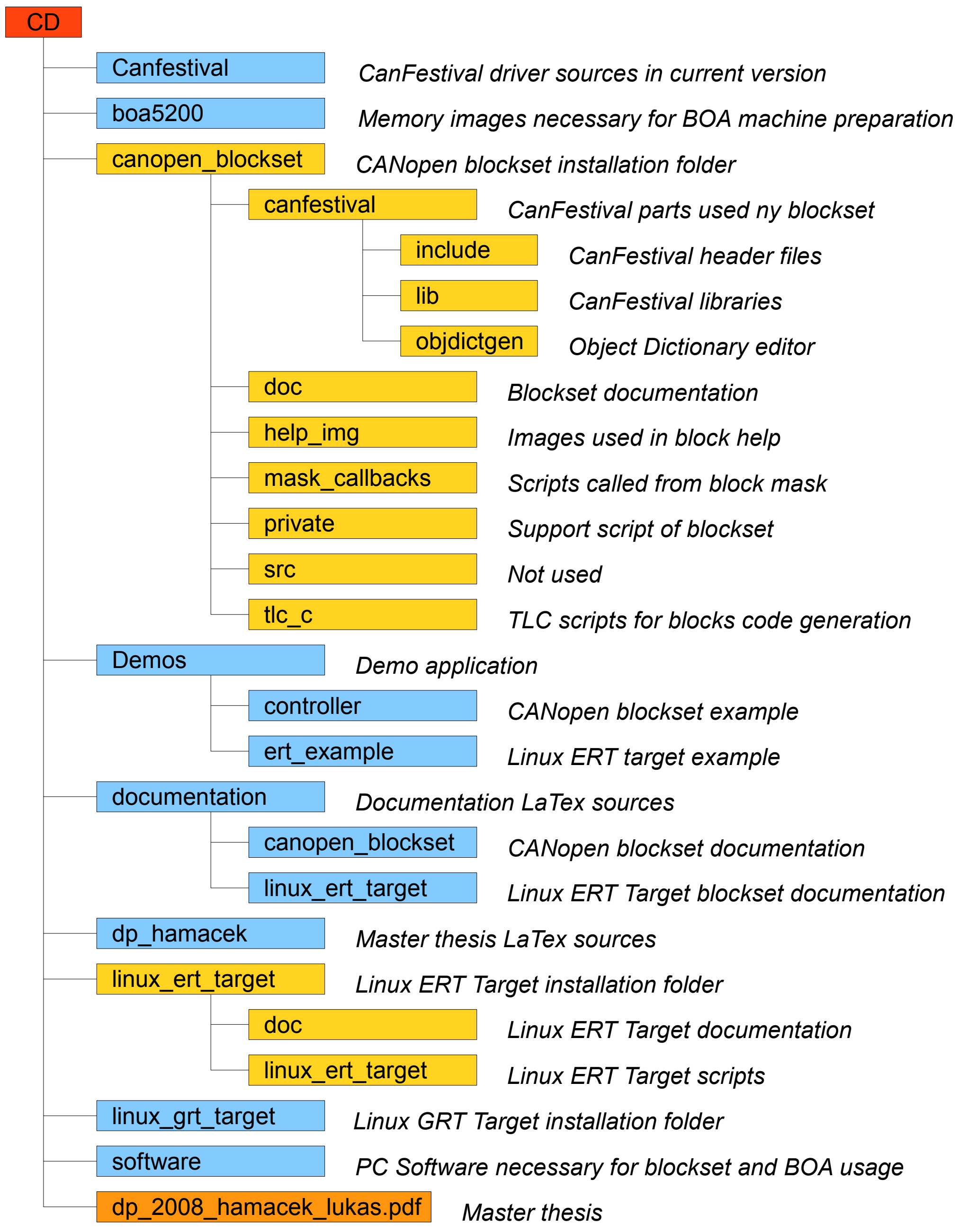
Abbreviations

HIL	Hardware In the Loop
HW	HardWare
IO	Input Output
ISR	Interrupt Service Routine
JFFS2	Journale File System version 2
MinGW	Minimalist GNU for Windows
MSYS	Minimal System
OS	Operating System
RedBoot	Red Hat Embedded Debug and Bootstrap
RT	Real Time
RTW	Real Time Workshop
SCP	SeCure Copy
SSH	Secure SHell
STF	System Target File (TLC)
TFTP	Trivial File Transfer Protocol
TLC	Target Language Compiler
CAN	Controller Area Network
EDS	Electronic Data Sheet
PDO	Process Data Object
SDO	Service Data Object
NMT	Network Management
OD	Object Dictionary

Figure D.1: Abbreviations

Appendix E

Attached CD content



CD
Canfestival
CanFestival driver sources in current version
boa5200
Memory images necessary for BOA machine preparation
canopen_blockset
CANopen blockset installation folder
canfestival
CanFestival parts used ny blockset
include
CanFestival header files
lib
CanFestival libraries
objdictgen
Object Dictionary editor
doc
Blockset documentation
help_img
Images used in block help
mask_callbacks
Scripts called from block mask
private
Support script of blockset
src
Not used
tlc_c
TLC scripts for blocks code generation
Demos
Demo application
controller
CANopen blockset example
ert_example
Linux ERT target example
documentation
Documentation LaTex sources
canopen_blockset
CANopen blockset documentation
linux_ert_target
Linux ERT Target blockset documentation
dp_hamacek
Master thesis LaTex sources
linux_ert_target
Linux ERT Target installation folder
doc
Linux ERT Target documentation
linux_ert_target
Linux ERT Target scripts
linux_grt_target
Linux GRT Target installation folder
software
PC Software necessary for blockset and BOA usage
dp_2008_hamacek_lukas.pdf
Master thesis