Hard Realtime Rapid Prototyping Development Platform

Master's thesis
performed in **Vehicular Systems**

by
Christer Rosenquist

Reg nr: LiTH-ISY-EX-3377-2003

15th September 2003

Hard Realtime Rapid Prototyping Development Platform

Master's thesis

performed in **Vehicular Systems**,
Dept. of Electrical Engineering
at **Linköpings universitet**

by **Christer Rosenquist**

Reg nr: LiTH-ISY-EX-3377-2003

Supervisor: **Ph.D. student Per Andersson**
Linköpings Universitet

Examiner: **Associate professor Lars Eriksson**
Linköpings Universitet

Linköping, 15th September 2003

Avdelning, Institution
Division, Department

Vehicular Systems,
Dept. of Electrical Engineering
581 83 Linköping

Datum
Date

15th September 2003

Språk
Language
☐ Svenska/Swedish
☒ Engelska/English
☐ _____

Rapporttyp
Report category
☐ Licentiatavhandling
☒ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport
☐ _____

ISBN
—

ISRN
LITH-ISY-EX-3377-2003

Serietitel och serienummer **ISSN**
Title of series, numbering —

URL för elektronisk version
http://www.vehicular.isy.liu.se
http://www.ep.liu.se/exjobb/isy/2003/3377/

Titel Utvecklingsplattform för snabb framtagning av prototyper för hård realtidsexekvering

Title Hard Realtime Rapid Prototyping Development Platform

Författare Christer Rosenquist
Author

Sammanfattning
Abstract

Matlab Simulink is a commonly used tool in the design process of control systems. To further take advantage of the Matlab Simulink models it is desirable to translate them for realtime use together with the possibility to read/write physical signals.

Real-Time Workshop is an extension to Simulink that automatically generates code from a model to a variety of target platforms. RTAI and RTLinux are hard realtime operating systems, making use of Linux.

To make automatically generated code run on RTAI and RTLinux an adaptation of the generation of code is necessary.

To control, for example, an automotive engine a data acquisition card with an appropriate device driver is required. Comedi, an open source project, provides a number of device drivers for data acquisition cards.

The developed system makes use of Simulink, Real-Time Workshop, RTAI or RTLinux, and the standard data acquisition card NI 6035E using a Comedi device driver. The Simulink models may be executed at frequencies up to 50 kHz on ordinary PC hardware.

The evaluation of the system consisted of measuring the interrupt latency of the used motherboard's bus, measuring computation times running Simulink models with known complexity, running models developed at Vehicular Systems and a comparison of interfacing Simulink/Real-Time Workshop between RTAI and RTLinux.

The recommended realtime operating system is RTAI due to the open source community support of it as a target platform for Real-Time Workshop.

Nyckelord
Keywords rapid prototyping, hard realtime, RTLinux, RTAI, Simulink, Real-Time Workshop

Abstract

Matlab Simulink is a commonly used tool in the design process of control systems. To further take advantage of the Matlab Simulink models it is desirable to translate them for realtime use together with the possibility to read/write physical signals.

Real-Time Workshop is an extension to Simulink that automatically generates code from a model to a variety of target platforms. RTAI and RTLinux are hard realtime operating systems, making use of Linux.

To make automatically generated code run on RTAI and RTLinux an adaptation of the generation of code is necessary.

To control, for example, an automotive engine a data acquisition card with an appropriate device driver is required. Comedi, an open source project, provides a number of device drivers for data acquisition cards.

The developed system makes use of Simulink, Real-Time Workshop, RTAI or RTLinux, and the standard data acquisition card NI 6035E using a Comedi device driver. The Simulink models may be executed at frequencies up to 50 kHz on ordinary PC hardware.

The evaluation of the system consisted of measuring the interrupt latency of the used motherboard's bus, measuring computation times running Simulink models with known complexity, running models developed at Vehicular Systems and a comparison of interfacing Simulink/Real-Time Workshop between RTAI and RTLinux.

The recommended realtime operating system is RTAI due to the open source community support of it as a target platform for Real-Time Workshop.

Keywords: rapid prototyping, hard realtime, RTLinux, RTAI, Simulink, Real-Time Workshop

Preface

This master's thesis has been performed in Vehicular Systems, Department of Electrical Engineering at Linköpings Universitet, winter/spring 2002/2003.

Acknowledgment

I would like to thank my supervisor Per Andersson for his support during the work and the enjoying times in Vehicular Systems automotive engine laboratory. I would also like to thank Erik Sunnegårdh and Andreas Bergström for their interest in making the necessary changes to their models to make them run on the developed system, and Martin Gunnarsson for helping out in the laboratory.

Last, but not least, a thank to the master of science students Emma Strömberg, Andreas Bergström, Carl-Adam Torbjörnsson and Johan Gill and the research engineer Erik Sunnegårdh for interesting discussions about most from feminism to second Gulf war.

The very last thank goes to Wei Hing Ip for delaying my work in a most pleasant way.

Contents

Chapter 1

Introduction

Since some years back Vehicular Systems, Department of Electrical Engineering at Linköpings Universitet (from now on simplified to Vehicular Systems), have sought for the possibility to evaluate models and control systems in hard realtime at higher sample rates than 20 Hz; their former, R.I.P.[1], DOS founded hard realtime system degraded since the creator and maintainer of it disappeared and there was a change of hardware in the laboratory.

This master's thesis is an attempt to overcome this shortage, to develop a system with enhanced execution frequency, faster turnaround time, and more maintainable and extendable properties.

1.1 Objectives

The objectives are:

- Evaluate the means to realize a hard realtime rapid prototyping development platform, to support Vehicular Systems in their research on modeling and control of the automotive engine.
- Implement a rapid prototyping hard realtime system.

1.2 Methods

Internet will be used as the main source of information. To reuse as much work already done in suitable areas will be a leading motive.

[1] Rest In Peace.

1.3 Specifications

The system will consist of or be able to:

- an interface to Matlab/Simulink
- execution of Simulink models in hard realtime
- high execution frequency, i.e. at least 5 kHz
- ordinary PC hardware
- standard data acquisition card, e.g. NI 6035E
- extending the system with more hardware, in particular to use a CAN bus

1.4 Reader's Guide

For a user of the system Chapter 3 and the introduction of Chapter 2 would be enough reading, together with some of the Appendices. The reader interested in more details of the system is recommended to read all chapters.

1.5 Thesis outline

A summary of the chapters:

Chapter 1: Introduction Gives a brief introduction to why, how and what will be achieved.

Chapter 2: System Analysis for Rapid Prototyping An analysis of the components needed is covered.

Chapter 3: Development of Prototyping System The structure of the developed systems is explained; software used, dependencies, etc.

Chapter 4: Evaluation of Prototyping System Using performance metrics, the system is evaluated with respect to interrupt latencies and execution times.

Chapter 5: Conclusions Conclusions and future advice.

Notation Descriptions of acronyms used and some definitions.

Chapter 2

System Analysis for Rapid Prototyping

In developing models and control systems it is a tedious work converting the model and/or control system into source code. A way to hasten this process would be to automatically generate source code, see Figure 2.1.

A development platform for rapid prototyping may consist of:

- a modeling tool
- a simulation tool
- an automatic code generator
- a hard realtime platform
  - computer hardware
  - data acquisition hardware
  - hard realtime operating system
  - device drivers

For modeling and simulation the Matlab toolbox Simulink is well known and used. Real-Time Workshop, an additional toolbox for Simulink, is a general automatic code generator of Simulink block diagrams. The number of targets[1] supported by Real-Time Workshop is limited, so an adaptation of the automatically generated source code must be done.

There are limitations to Simulink models proposed by Real-Time Workshop:

[1] A *target* is an environment – hardware or operating system – on which the generated code will run.

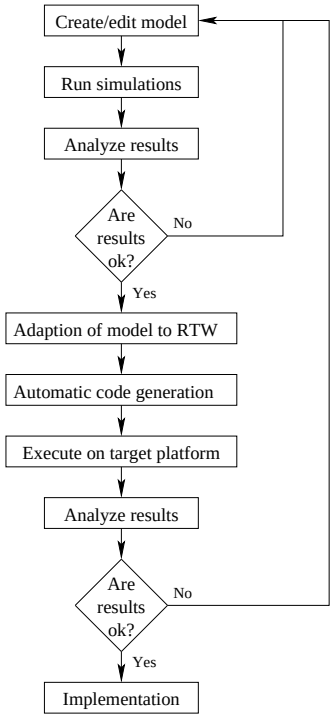

Figure 2.1: Rapid prototyping development process.

Operators Not Allowed : ^ .^ {} . \ .\ ' .' ; ,

Unsupported Blocks The S-function format does not support the following built-in-blocks:

- Matlab Fcn Block
- S-function blocks containing any of the following:
  - M-file S-functions
  - Fortran S-functions
  - C MEX S-functions that call into Matlab
- Scope block
- To Workspace block

Limitations of External Mode External mode[2] does not support

[2]In external mode, Real-Time Workshop establishes a communications link between a model running in Simulink and code executing on a target system. The system Simulink exists on and the target system may be different systems.

changing a parameter that results in a change in the structure of a model. For example, the following is not changeable:

- the number of states, inputs or outputs of any block
- the sample time or the number of sample times
- the integration algorithm for continuous systems
- the name of the model or of any block
- the parameters to the Fcn block

If any of these changes were made to the block diagram, new code must be generated and compiled.

As computer hardware, an ordinary PC will be used, see Appendix A. For the hard realtime execution two operating systems will be evaluated, RTLinux and RTAI. In addition, means to measure and control physical signals will be needed.

Beside executing code in hard realtime, storing data in files is a necessity to be able to analyze results after execution.

To use Microsoft Windows NT as a hard realtime operating system is not possible due to that the latencies imposed by the operating system is not deterministic [1].

2.1 Hard Realtime Linux

Linux is a full-featured UNIX implementation. The main design criterion of the Linux kernel is throughput, while realtime and predictability is not an issue. The main thing making Linux not realtime is the none preemptible kernel.

From the very first version of Linux the scheduler was realtime POSIX[3] compatible, e.g. it supports fixed priority (SCHED_FIFO) policy, which is the base feature to build a realtime system. POSIX offer several advantages: POSIX is a real standard, not an effort to lock customers into a proprietary API; and POSIX is widely known and well documented

On systems using virtual memory it is not possible to build realtime applications due to the random and long delays when RAM is swapped in and out from the hard drive. To circumvent this problem Linux provides the `mlock()` and `mlockall()` functions to disable paging for a specified range of memory, or for an entire process [2].

[3]Portable Operating System Interface for Unix. POSIX is an IEEE standard. Current Linux implementation of POSIX threads (POSIX 1003.1c) is based on the work of Xavaier Leroy, known as LinuxThreads. See `http://pauillac.inria.fr/~xleroy/linuxthreads/`

To provide realtime services within Linux systems[4], there are two main approaches:

Preemption Improvement Preemption improvement makes modifications to the Linux kernel code to reduce the amount of time the kernel spends in none preemptible sections of code. This approach can only be used for soft realtime, and is used e.g. by TimeSys and KURT.

Interrupt Abstraction Interrupt abstraction uses a separate scheduler and makes the entire kernel preemptible by having a hardware abstraction layer with complete control over the hardware interrupts, and simulate the interrupts to the Linux kernel, allowing the kernel to run unmodified on the realtime scheduler, and as the lowest priority task, alongside the realtime tasks [4]. This is the approach taken by e.g. RTAI and RTLinux.

2.1.1 RTLinux

RTLinux is a hard realtime operating system. The design is based on the concept of the *virtual machine* [5]. By modifying a standard operating system to act as a *base* kernel in a system where control is shared with a realtime kernel, both realtime and the richness of the standard operating system is achieved, see Figure 2.2 and Figure 2.3.

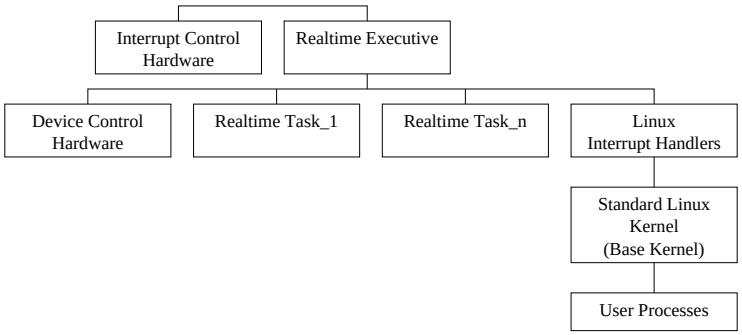

Figure 2.2: The structure of RTLinux. The realtime executive has no superior control of the interrupt control hardware implying the hardware latencies solely depend on the hardware. Linux itself is executing as a thread with the lowest priority, and all interrupts in the Linux interrupt handlers are soft interrupts, making Linux completely preemptible

[4]There are a number of ongoing realtime operating system projects making use of Linux [3].

The modification consists of emulation code that intercepts commands to enable and disable interrupts. The emulation prevents the base kernel from delaying hardware interrupts. Interrupts to the base kernel are passed through the emulation software after all realtime work is carried out. This makes the modifications of the base kernel small[5] [6].

RTLinux treats the base kernel as the task with the lowest priority, executing only when there are no realtime tasks to run.

> RTLinux decouples the mechanisms of the realtime kernel from the mechanisms of the general purpose kernel so that each can be optimized independently and so that the realtime kernel can be kept small and simple [7].

The means for tasks to communicate, either between realtime tasks or between a realtime task and a Linux process, is through special FIFOs or via shared memory. The RTLinux provided schedulers are an earliest deadline first [8] and a rate-monotonic scheduler[6]. Other Scheduling schemes have been implemented [9].

The RTLinux API conforms to the POSIX 1003.13 "Minimal Realtime System Profile" (PSE51) [10].

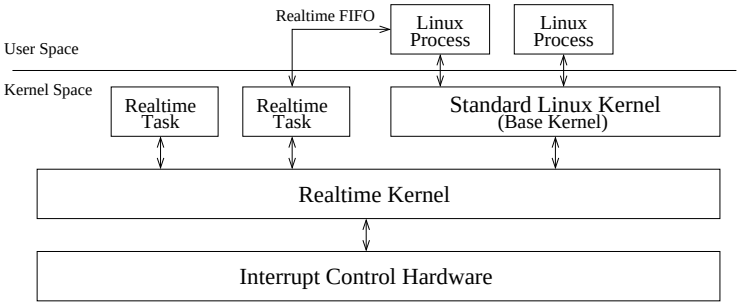

Figure 2.3: Flow of data and control. Ordinary Linux processes executes in user space, a protected environment, while the Linux kernel and the realtime tasks executes in kernel space, without protection against programming errors. Linux processes and realtime tasks communicates, e.g., via realtime FIFOs.

[5] There is primarily three modifications to the base kernel: the `cli` routine to *disable* interrupts, the `sti` routine to *enable* interrupts, and third the low level "wrapper" routines, which save and restore state around calls to handlers.

[6] The rate-monotonic scheduler is implemented by Oleg Subbotin (see the rtlinux.org homepage).

2.1.2 RTAI

RTAI is founded on RTLinux, but its development is detached from RTLinux and has taken a different direction [11]. The main difference is RTAI's much broader API, trying to serve the programmers need in as many cases as possible; where RTLinux like to keep their API as small and clean as possible. The fundamental workings are, however, the same, so the overall description of RTLinux applies to RTAI as well.

2.2 Real-Time Workshop

Real-Time Workshop is capable of generating customizable ANSI C code directly from Simulink models using point-and-click interactions. Generated code can run on PC hardware, DSPs, microcontrollers on bare-board environments, and with commercial, proprietary or open source realtime operating systems [12].

To targeting[7] Real-Time Workshop generated code for a specific platform some work is usually needed. A common way to achieve this is to modify an already existing targeting system.

2.2.1 Targeting RTLinux

A previous work targeting RTLinux has already been performed, using an older version of Real-Time Workshop [13]. The work is an outcome of a PhD research project undertaken at Glasgow Caledonian University. The project was initiated to allow remote monitoring of realtime control experiments. The main goal was to set up a realtime platform for students to test control algorithms.

The work lacks device driver support of the data acquisition card used in this project, see section Device Drivers. It does not support the latest, at the moment of writing, Matlab version. An adaptation to the latest Matlab will be investigated.

2.2.2 Targeting RTAI

A work targeting RTAI is made available by Roberto Bucher, University of Applied Sciences of Southern Switzerland [14]. The work supports the Matlab version used in this project.

The work lacks device driver support of the data acquisition card used in this project, see section Device drivers.

[7] The process of specifying an environment, a target, is called targeting.

2.3 Device Drivers

To be able to communicate with hardware in a decent way, device drivers are needed.

> A device driver is a piece of software that interfaces a particular piece of hardware: a printer, a sound card, a motor drive, etc. It translates the primitive, device-dependent commands with which the hardware manufacturer want you to configure, read and write the electronics of the hardware interface into more abstract and generic function calls and data structures for the application programmer [15].

There are three possibilities to get a device driver:

- the manufacturer of the device supply one.
- get it elsewhere.
- writing a device driver from scratch.

The best case is the manufacturer supplying a device driver and the worst case is writing a device driver from scratch. With the used data acquisition card, the first possibility is lost.

At the moment, not many manufacturers of cards for measurement and control purposes supply device drivers for Linux. But thanks to the open source community, this is no longer a problem. The Comedi project[8] supply a lot of device drivers in this area, and among them the data acquisition card used here, the National Instruments NI 6035E. See the RTAI homepage, `www.rtai.org`, for supported hardware.

2.3.1 Comedi

The Comedi project develops open source drivers, tools, and libraries for data acquisition in Linux. Comedi supports realtime Linux, using the same interface provided by the user space library.

[8] `www.comedi.org`

Chapter 3

Development of Prototyping System

The main part of developing the system consists of targeting the automatically Real-Time Workshop generated code to the hard realtime operating systems RTLinux and RTAI. In addition, how to make the device driver for the used data acquisition card a building block in Simulink will be explained. See Figure 3.1.

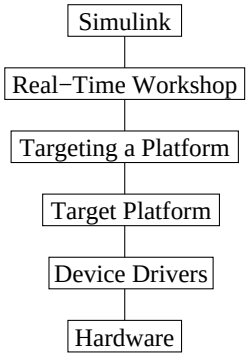

Figure 3.1: A way of viewing the system components and their relations to each other. Target platforms will be RTLinux or RTAI. The targeting of a platform is the conforming of the automatically Real-Time Workshop generated code to a specific platform.

3.1 System Components

The software resides on the computer *bristol.isy.liu.se* and is organized under the directory `/home/Realtime/`, see Figure 3.2. It consists of software packages, configuration files and RPMs[1]. For hardware used see Appendix A.

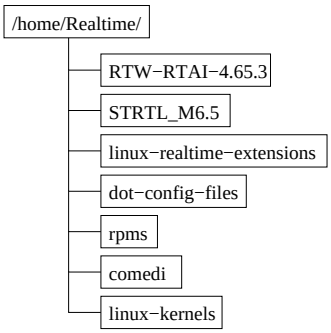

Figure 3.2: The directory structure of the software packages and configuration files.

RTW-RTAI-4.65.3 Source code for targeting the Real-Time Workshop generated code for RTAI.

STRTL_M6.5 Source code for targeting the Real-Time Workshop generated code for RTLinux.

linux-realtime-extensions Source code for RTLinux and RTAI. The directory contains two compressed tarball[2] files:

- `rtlinux-3.2-pre1.tar.bz2`
- `rtai-24.1.10.tgz`.

An example installing RTAI is provided in Appendix D.

dot-config-files Some useful configuration files for Linux kernels 2.4.18 and 2.4.19, RTLinux-3.2-pre1 and RTAI-24.1.10.

rpms Contains the RPMs for *gcc295*, `gcc295-2.95.3-0.i386.rpm`, version 2.95.3 and *g++295*, `gcc295-c++-2.95.3-0.i386.rpm`,

[1] Redhat Package Manager. To install: `rpm -ivh name_of_package`. To query about a specific software package use the flag `-q`.

[2] Tarball is a jargon term for a tar archive – a group of files collected together as one. Tar – Tape ARchive. See `searchSolaris.com`. Tarballs have been the standard way to ship source code distributions since mid-1980s, see GNOME Dictionary 2.0.2.

version 2.95.3. The compilers are needed to compile the Linux kernels and the Linux realtime extensions, i.e. RTLinux and RTAI.

comedi Source code for device drivers. The directory contains two compressed tarball files:

- `comedi-0.7.65.tgz`
- `comedilib-0.7.19.tgz`.

To install see Appendix D.

linux-kernels Linux kernels source code. The directory contains two compressed tarball files:

- `linux-2.4.18.tar.gz`
- `linux-2.4.19.tar.gz`.

In section 3.3 and section 3.4 a more comprehensive description of RTW-RTAI-4.65.3 and STRTL_M6.5 will be given.

3.2 Real-Time Workshop

To better understand the Real-Time Workshop generation of code, the Target Language Compiler and template makefile concepts will be explained. Further, monitoring signals and modifying parameters will be touched. There are two ways of doing this, external mode and a C API The process of generating an executable is viewed in Figure 3.3.

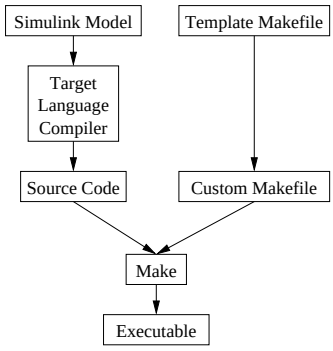

Figure 3.3: In the build process the Target Language Compiler generates source code, and a custom makefile is generated from the template makefile. To create the executable, the `make -f model.mk` command is issued.

3.2.1 Target Language Compiler

Real-Time Workshop generates source code for models and blocks through the Target Language Compiler, which reads script files (or TLC files) that specify the format and content of output source files. Two types of TLC files are used:

1. A *system target file*, which describes how to generate code for a chosen target, is the entry point for the TLC program that creates the executable.
2. *Block target files* define how the code looks for each of the Simulink blocks in the model.

System and block target files have the extension `.tlc` [16].

3.2.2 Template Makefiles

Real-Time Workshop uses template makefiles to build an executable from the generated code. The build process creates a makefile from the template makefile. Each line from the template makefile is copied into the makefile; tokens encountered during this process are expanded into the makefile. The name of the makefile created by the build process is *model.mk*, where *model* is the name of the Simulink model [16].

3.2.3 External Mode

External mode allows two separate systems – a *host* and a *target* – to communicate. The host is the computer where Simulink is executing. The target is the computer where the executable created by Real-Time Workshop runs.

External mode allows modifying, or tuning, block parameters in realtime and to view and log block outputs in many types of blocks and subsystems [17].

3.2.4 C API

Real-Time Workshop includes a C API interface to support development of C application programs for tuning parameters and monitoring signals independent of external mode [18].

3.3 Targeting RTLinux

The starting point of targeting RTLinux is an outcome of a PhD research project at Glasgow Caledonian University [13]. This work, "Simulink Target for RT-Linux" (from now on, referred to as STRTL),

supports Matlab 5.3.1 and Matlab 6.1. It makes use of the external mode for tuning parameters and monitoring, storing signals.

3.3.1 STRTL

The specification of STRTL:

- Monitoring of the realtime experiment using the Simulink's external mode mechanism. The target machine (RTLinux) and the host machine (Simulink) are connected through a TCP/IP connection. Signal values are uploaded and displayed on the Scope blocks.

- A check is performed to ensure that the target platform can handle the requested *Fixed step size* specified in the block diagram. If the *Fixed step size* is too small the application terminates and an error message is displayed.

- File data logging is supported. However, data logging supported by Matlab can not be used. Thus, all blocks used to store data on files must not be included. Instead STRTL permits to create files where data monitored on scopes can be recorded. A file is created for each sample rate used (tid[3]), and in each file all the signals with the same sample time are saved. It is up to the user to discern what data corresponds to which signal. Data is saved in binary format at run time.

- Singeltasking[4] and multitasking[5] mode is supported.

- A watchdog makes sure overrun conditions do not occur.

- A Discarding Algorithm has been implemented to improve realtime monitoring in local area networks. It measures the available bandwidth and discards signal points that do not fit within this bandwidth.

- The last update of STRTL has been developed using Red Hat 6.2 with the Linux kernel version 2.4.4 and RTLinux V 3.1.

[3]Each sample time in a model is assigned a task identifier (tid). The tid is passed to the model output and update routines to decide which portion of the model should be executed at a given time.

[4]Singletasking – A mode in which a model runs in one task.

[5]Multitasking – A process by which a microprocessor schedules the handling of multiple tasks. The number of tasks is equal to the number of sample times in the model.

3.3.2 STRTL_M6.5

The modified STRTL has been renamed to STRTL_M6.5, the M6.5 part indicates the adaptation to Matlab 6.5.

The purpose of STRTL_M6.5 is to make the Real-Time Workshop generated code RTLinux compilable and executable. The emplacement of its functionality can be viewed as in Figure 3.4.

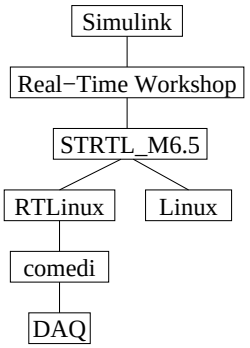

Figure 3.4: System overview with RTLinux as the hard realtime execution component and STRTL_M6.5 as the component conforming the automatically Real-Time Workshop generated code to be runnable on RTLinux.

For the directory structure of STRTL_M6.5 and a brief description of the directories contents, see Figure 3.5.

Changes in Matlab 6.5

A number of new features and enhancements have been added to Real-Time Workshop 5.0 since Real-Time Workshop 4.1. Among the changes, the packaging of generated code into `.c` and `.h` files has changed. And, instead of storing information about the root model in the *SimStruct* data structure, the *rtModel* data structure is preferred. The new rtModel is a lightweight data structure eliminating unused fields in the representation of the root model. For more information about the changes, see [19].

Rewriting of STRTL

Due to the changes in the new release of Real-Time Workshop, a rework of STRTL was needed. To convert STRTL to STRTL_M6.5 [20] was used. In the next two sections the affected files will be listed, and briefly explained.

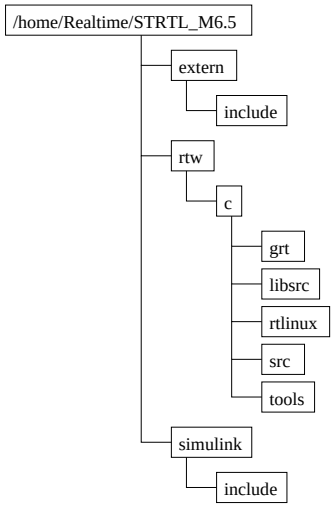

Figure 3.5: The directory structure of STRTL_M6.5. The directory `extern/include` contains header and `.mlib` files. Under the `rtw/c` directory the important directories are `rtlinux` and `src`. The directory `simulink/include` contains Matlab header files.

rtlinux

The `rtlinux` directory contains:

`ext_comm_rtl.c` Host side, transport independent external mode functions. Calls to these functions originate from Simulink and are dispatched through `ext_main.c`.

`ext_comm_rtl.mexglx` How to build the file, see Appendix C.

`rtlin.m` A dummy S-function for the *RTL In* block. During Simulink simulation it returns zeros. When the model is compiled via Real-Time Workshop, the block is replaced by a specific hardware adapter (e.g. a device driver) that allows the hardware to pull in new data from a particular I/O device.

`rtlin.tlc` Script file defining the source code for the *RTL In* block.

`rtlout.m` A dummy S-function for the *RTL Out* block, allowing Simulink to run when the model has not yet been compiled. It functions similarly to the *RTL In* block.

`rtlout.tlc` Script file defining the source code for the *RTL Out* block.

`rtlinux_main.c` Model initialization, execution and termination are controlled within this file.

`rtlinux.tlc` A script file targeting RTLinux.

`rtlinux.tmf` A template makefile for building an RTLinux realtime version of a Simulink model using generated C code.

`rtlinlib.mdl` Defines the Simulink blocks *RTL In* and *RTL Out*.

`slblocks.m` Defines a block library, *Simulink Target for RTLinux* in the Simulink *Blocksets & Toolboxes*, for Simulink; containing the blocks in `rtlinlib.mdl`.

src

The following files, in the `src` directory, are modified according to [20]:

`krnl_main.c` The execution engine in kernel space.

`ext_svr_rtl.h` Header file.

`ext_svr_rtl.c` Writes a signal point to shared memory, from where it is later read in ext_svr_us.c and transmitted to the host machine. It also processes a checksum.

`us_main.c` The execution engine in user space.

`updown.h` Header file.

`updown.c` Handles the details of interacting with the target model.

`ext_svr_us.h` Header file.

`ext_svr_us.c` Carries out tasks such as establishing and terminating connection with the host.

Execution Space

The execution space is divided into two, kernel space and user space. In kernel space, normally, the kernel, device drivers and any kernel extensions run. In RTLinux, realtime modules run in kernel space. In user space, user applications run. How STRTL_M6.5 modules are divided into the two spaces, see Figure 3.6.

Device Driver

To support data acquisition an adapter, `ni_6035e_adapter.c`, in the `src` directory, was developed. The adapter is working as a *glue* between the Simulink blocks *RTL In* and *RTL Out*, and the Comedi device driver. The signals are normalized to the range [-1 1].

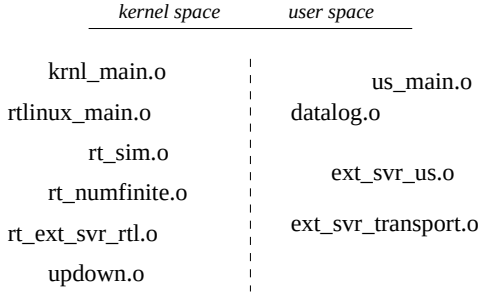

Figure 3.6: The modules are separated into kernel space and user space. The communication between modules is achieved by special realtime FIFOs or special realtime shared memory. The shared memory can not be swapped in and out of the hard drive.

3.4 Targeting RTAI

Roberto Bucher, at University of Applied Sciences of Southern Switzerland, has targeted RTAI, with support for Matlab 6.5 [14]. It makes use of the C API for tuning parameters and viewing, storing signals. The applications making use of this API also uses the Qt platform[6] and Gnuplot[7]. Figure 3.7 shows the emplacement of this component. The software package will be called RTW-RTAI-4.65.3, see Figure 3.8 for the directory structure.

How to install and work with the system, see the PDF-document `rtw_rtai.pdf` in the `Documentation` directory.

Device Driver

The Simulink blocks, *RTL In* and *RTL Out*, in STRTL_M6.5 are used, and, of course, the Comedi device driver too. The difference is the use of `ni_6035e_adapter_rtai.c`, in the `linux_rt/rtai` directory, instead of `ni_6035e_adapter.c` used in STRTL_M6.5.

3.5 Device Driver Blocks in Simulink

The realtime task communicates with external hardware via device drivers, see Figure 3.9. A Simulink device driver block can be added to a model like any other Simulink block.

[6] A multiplatform, C++ application framework. See `www.trolltech.com`.
[7] An interactive plotting program. See `www.gnuplot.info`.

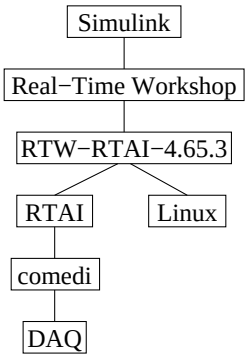

Figure 3.7: System overview with RTAI as the hard realtime execution component and RTW-RTAI-4.65.3 as the component conforming the automatically Real-Time Workshop generated code to be runnable on RTAI.

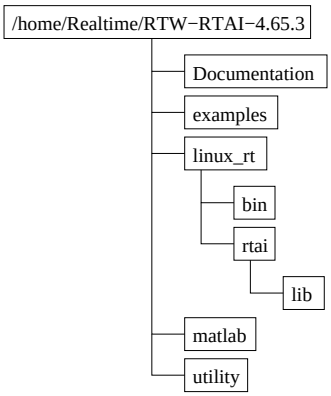

Figure 3.8: The directory structure of RTW-RTAI-4.65.3. The directory `Documentation` contains a PDF-document describing installation and how to use the system. `examples` contains some (useless) examples. `linux_rt` contains the RTAI files necessary to compile and link the realtime module. `matlab` contains the files for the Matlab shell; RTW, MEX etc. `utility` contains the source code of the utilities *scope*, *rtplot*, *rtppar*, *changertpar* and *gengnu*.

Comedi

Some Comedi terminology will be explained:

- Comedi arranges a data acquisition card into subdevices. A sub-

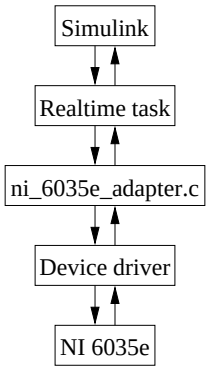

Figure 3.9: Simulink and the realtime task communicates via the external mode. The realtime task and the device driver communicates via `ni_6035e_adapter.c`, the device driver is provided by Comedi. NI 6035E is the data acquisition card.

device may consist of a group of analog input signals, analog output signals, timers, etc. A specific subdevice is referred to as 0,1,2,...

- A 16-bit signal has within Comedi the range [0 FFFF]. Where 0 represents the most negative value and FFFF represents the most positive value, e.g. if the chosen measurement range is [-5 5] the -5 is represented as 0 in Comedi and the 5 is represented as FFFF in Comedi. A value outside the range is silently handled by Comedi.

The Adapter

`ni_6035e_adapter.c`, in STRTL_M6.5, provides three functions:

`ni_6035e_data_write()` The function takes five parameters:

- the subdevice to use
- the channel to write to
- a range
- an analog reference
- the data to write to the channel

The subdevice, channel, range and analog reference parameters are the parameters entered in the Simulink block *RTL Out*, see

Figure 3.12. To convert the normalized signal to Comedi's representation, the formula

$$\frac{(value + 1) \cdot 0xFFF}{2}$$

is used. The resolutions of the analog output signals of the used data acquisition card are 12 bits.

`ni_6035e_data_read()` The function takes five parameters:

- the subdevice to use
- the channel to read from
- a range
- an analog reference
- a pointer to a real_T type

The subdevice, channel, range and analog reference parameters are the parameters entered in the Simulink block *RTL In*, see Figure 3.11. To normalize the signal, the formula

$$\frac{value \cdot 2}{0xFFFF}$$

is used. The resolutions of the analog input signals of the used data acquisition card are 16 bits.

`ni_6035e_open()` The function is used by `krnl_main.c` to initialize the Comedi device driver when loaded into kernel space.

The file must be included in the template makefile `rtlinux.tmf`. In case of RTW_RTAI_4.65.3, the above applies to the file `ni_6035e_adapter_rtai.c`, and must likewise be added to the template makefile `rtai.tmf`.

Simulink Device Driver Blocks

The source code of the Simulink device driver blocks *RTL In* and *RTL Out* are made up of the following files:

- `rtlin.m`
- `rtlin.tlc`
- `rtlout.m`
- `rtlout.tlc`

The `.m`-files are just dummies used for Simulink simulations. The `.tlc`-files are Simulink C MEX S-functions, see [21]. The device driver blocks are contained in the block library file `rtlinlib.mdl`. To display the library, type `rtlinlib` at the Matlab prompt, see Figure 3.10. To configure a device driver block, each block has a dialog box to set configuration parameters, see Figure 3.11 and Figure 3.12.

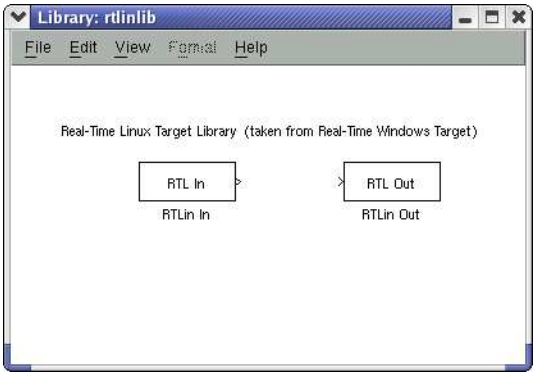

Figure 3.10: The *rtlinlib* block library containing the device driver blocks *RTL In* and *RTL Out*.

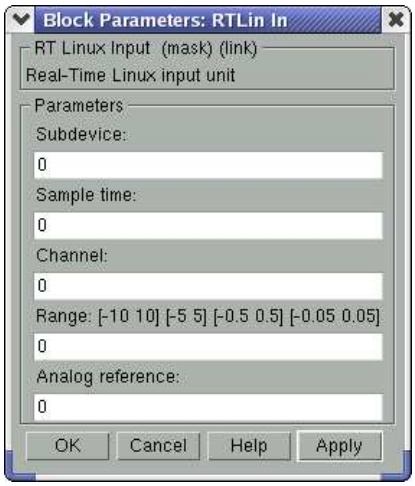

Figure 3.11: The input device driver block interface. Using the NI 6035e card: the *Subdevice* 0 is the group of analog input signals; in *Channel* the wanted input signal is chosen from the range [0 15], i.e. the card can handle 16 analog input signals; in *Range* there are 4 measurement ranges to choose among.

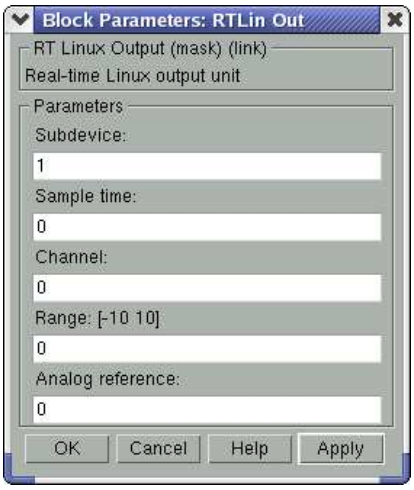

Figure 3.12: The output device driver block interface. Using the NI 6035e card: the *Subdevice* 1 is the group of analog output signals; in *Channel* the wanted output signal is chosen from the range [0 1], i.e. the card can handle two analog output signals.

Chapter 4

Evaluation of Prototyping System

The evaluation of the system consists of:

- Measuring the interrupt latency of the motherboard's bus, running the system under heavy load.
- Measuring computation times running some simple Simulink models with known complexity, to give a brief understanding of the evaluation time needed and of the latencies occurring while running a model.
- Finding the maximum execution frequency of a minimal model in the system.
- Running three different models, developed at Vehicular Systems, in the automotive engine laboratory, to test the system in its targeted environment.
- A comparison of the interfacing to Simulink between RTAI and RTLinux.

To keep the testing simple, most of the testing will be performed using only RTAI. The execution times and the latencies are assumed to be similar between RTAI and RTLinux due to the similarities between them.

All tests were performed running X Windows except when testing the motherboard for optimization at the bus level locking the bus.

4.1 Bus Interrupt Latency

Motherboards may have optimizations at the bus level locking the bus. The optimizations may lock the bus for several milliseconds, making realtime obsolete.

To check the hardware a test application, `buslokchk.c`, provided with the RTAI distribution will be used. The check measures interrupt latencies by using the CPU TSC[1] on the timer interrupt. The test is performed under heavy load and during a period of 20 hours. Latencies of 20/30 micro seconds worst case is considered good. The system load consisted of [22]:

- `ping -f www.sunet.se`
- `ping -f bristol.isy.liu.se`
- `while ‘‘true’’; do ls -aR /; sync; done`
- `while ‘‘true’’; do cp /var/tmp/linux-2.4.19.tar.gz tmp; sync; rm -f tmp; sync; done`
- `top -d 0.1`
- `while ‘‘true’’; do cat /proc/interrupts; cat /proc/rtai/*; done`

4.1.1 Results

Worst case latency during 20 hours test was 18.6 microseconds. The result indicates that the latency in the bus level of the motherboard is good and also the possibility to run models in frequencies up to 50 kHz.

4.2 Computation Times

The computation time for a step in a model will be measured just before and after each step in a Simulink model. This method of measuring introduces latencies from the system into the computation times for the execution steps in the model. Those latencies, which can be regarded as variations, give a good description of how the system performs in the reality. In addition, a test will be performed testing the maximum execution frequency of a model in the used system.

[1] Time Stamp Counter. The counter counts processor clock cycles since reset or since it was programmatically zeroed.

4.2.1 Measuring

To measure the computation time of a time step in a model, two measurement points was introduced in `rt_proc.c`. One before, and one after, the call to update the model. To get the time the function *rt_get_cpu_time_ns()* was used, which returns the CPU time in nanoseconds whatever timer is in use. To log the time it takes to compute a step in the model, the difference between the measuring points is written to a realtime fifo, which then is read by a program and written to a file, see Appendix F and Figure 4.1.

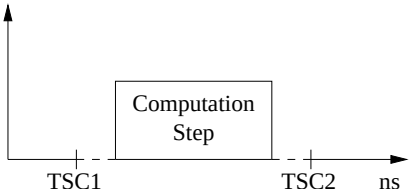

Figure 4.1: To measure the computation time for a given step in a model, the time TSC2 - TSC1 is calculated and stored into a file. The measured time may be extended by interrupt latencies both before and after the computation step of the model.

4.2.2 Algorithms

To get an estimate of the computation time used in a model two different algorithms, see Appendix F, was used:

- Bubble sort[2], see Figure 4.2.
- A bubble sort performing a floating point calculation in each step of the sorting process.

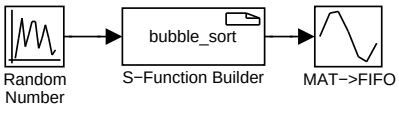

Figure 4.2: A Simulink model using bubble sort.

The complexity of both algorithms are $O(n^2)$. That can be compared to the complexity of the FFT[3] algorithm: $O(n \log n)$.

[2] An algorithm used for sorting.
[3] Fast Fourier Transform.

4.2.3 Results

The tests were conducted with the sample time set to 1 millisecond and run for approximately 2 minutes each. Table 4.2.3 and Table 4.2.3 summarizes the measured computation times, including interrupt latencies.

n	**Min** [μs]	**Max** [μs]	**Variation**[4] [μs]	**Mean** [μs]	**Std dev**[5] [μs]
1	0.6170	8.1400	7.5230	0.8956	0.2717
10	0.9410	8.6040	7.6630	1.2337	0.2925
100	26.8230	38.0810	11.2580	27.3315	0.9131
500	631.4320	642.5030	11.0710	631.8040	0.6840

Table 4.1: The minimum, maximum, variation, mean and standard deviation of computation times for the bubble sort algorithm, using n = 1, 10, 100, and 500. The computation times are quadratic to the size of n, and the variation and the standard deviation of the computation times are *not* related to the size of n. The mean computation times are close to the minimum computation times, indicating quite low frequency of longer latencies.

n	**Min** [μs]	**Max** [μs]	**Variation** [μs]	**Mean** [μs]	**Std dev** [μs]
1	0.6170	7.8310	7.2140	0.8841	0.2780
10	1.1650	11.1290	9.9640	1.4528	0.3495
100	49.2470	58.0280	8.7810	49.5763	0.3663

Table 4.2: The minimum, maximum, variation, mean and standard deviation of computation times for the bubble sort algorithm with floating point operations, using n = 1, 10, and 100. The computation times are quadratic to the size of n, and the variation and the standard deviation of the computation times are *not* related to the size of n. The mean computation times are close to the minimum computation times, indicating quite low frequency of longer latencies.

To show the distribution of computation times histograms are used. As can be seen in Figure 19 to Figure 4.5, n = 1, 10 and 100, the computation times are related to the complexity of the model executed, and the latencies are independent of the complexity. With n = 500, see Figure 4.6, it's only possible to run the bubble sort algorithm, though the used sample time gives < 1000 microseconds of execution time.

Running bubble sort with n = 1 and assuming worst case latency < 20 microseconds, an execution frequency of 50 kHz would be possible. A test, run for 30 minutes, confirmed this. As can be seen, for example in Table 4.2.3, added complexity to a model rapidly decreases the execution frequency.

[5]The difference between the maximum and minimum of computation times.
[5]Standard deviation.

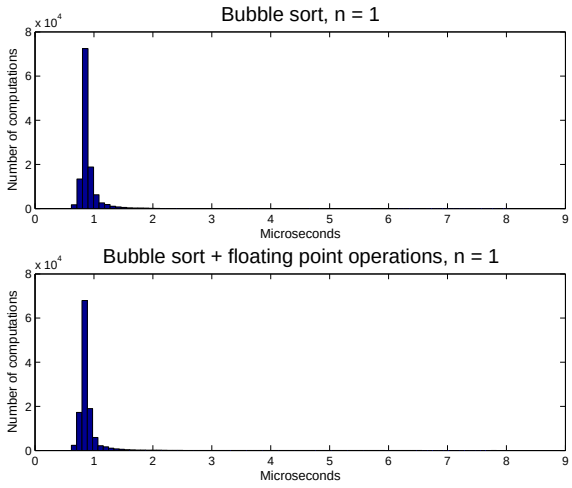

Figure 4.3: Histograms showing the variation of computation times needed for the algorithms bubble sort and bubble sort with floating point operations; where n = 1, sample time = 0.001 seconds, and the duration of the test = ∼2 minutes.

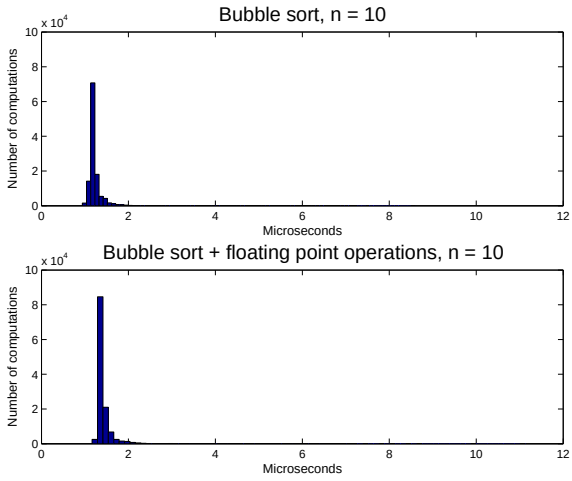

Figure 4.4: Histograms showing the variation of computation times needed for the algorithms bubble sort and bubble sort with floating point operations; where n = 10, sample time = 0.001 seconds, and the duration of the test = ∼2 minutes.

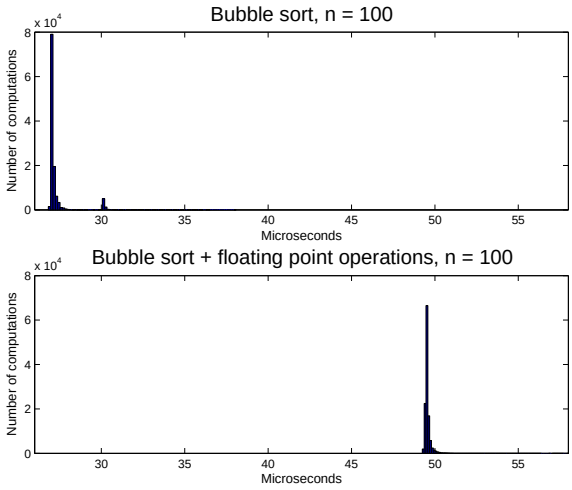

Figure 4.5: Histograms showing the variation of computation times needed for the algorithms bubble sort and bubble sort with floating point operations; where n = 100, sample time = 0.001 seconds, and the duration of the test = ∼2 minutes.

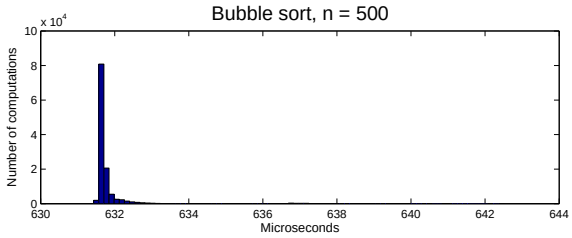

Figure 4.6: Histogram showing the variation of computation times needed for the algorithm bubble sort; where n = 500, sample time = 0.001 seconds, and the duration of the test = ∼2 minutes.

4.3 Simulink Test Applications

To test the system in the automotive engine laboratory three different models have been used:

- a cylinder air-mass flow observer
- an adaptive catalyst model for control
- a modeling and control of torque in an SVC[6] engine

[6]SAAB Variable Compression.

The last model is the only model of the test models making use of an output signal to control one of the engines in Vehicular Systems' automotive engine laboratory.

4.3.1 Cylinder Air-Mass Flow Observer

The model estimates the air-mass flow to a cylinder [23] from the measurable signals, see Figure 4.7:

- air-mass flow
- intake manifold pressure
- intake manifold temperature
- pressure after intercooler
- throttle plate angle
- engine speed

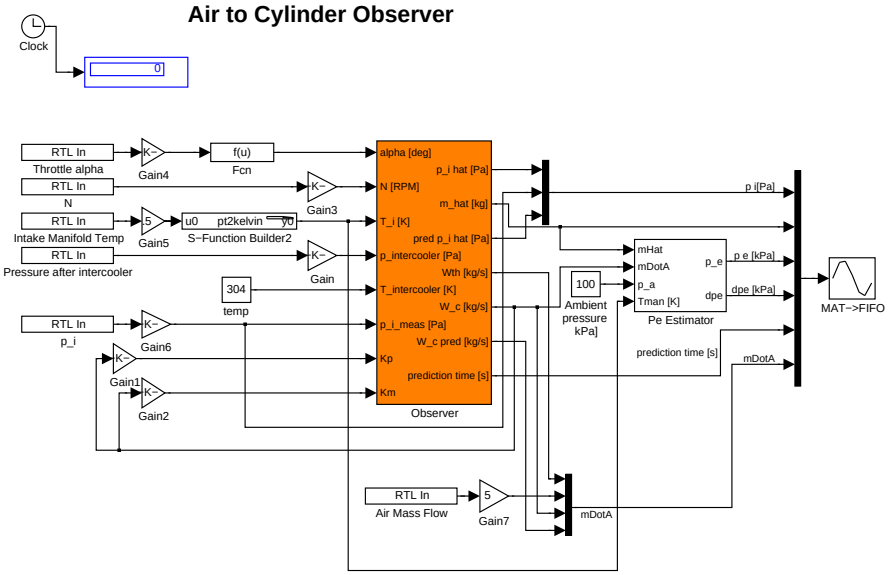

Figure 4.7: A model estimating the air-mass flow to the cylinder.

4.3.2 Adaptive Catalyst Model for Control

A model of the catalyst system aiming at control by an MPC[7] [24], see Figure 4.8. The input signals are:

- *Lambda trail*, from a discrete EGO[8] sond
- *Lambda front*, from a UEGO[9] sond

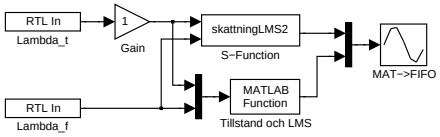

Figure 4.8: A linear adaptive model of a catalyst system.

4.3.3 Modeling and Control of Torque in an SVC Engine

The model is estimating the torque from the measurable signals:

- compression ratio
- engine speed
- intake manifold temperature
- air-mass flow

The controller is fed by the difference between a reference signal and the estimate. The output signal controls a throttle plate angle. See Figure 4.9. The model and the controller is an outcome of a master's thesis [25].

4.3.4 Results

After minor modifications of the models to adapt them to Real-Time Workshop, all models run without problem. The *Cylinder Air-Mass Flow Observer* was also run under RTLinux.

The runnings of those models gave valuable knowledge about how less experienced users were able to handle the system. At the same time the users could, as in the case with the SVC engine, quickly evaluate different solutions.

[7] Model Predictive Control.
[8] Exhaust Gas Oxygen, or binary oxygen sensor.
[9] Universal EGO, or wideband oxygen sensor.

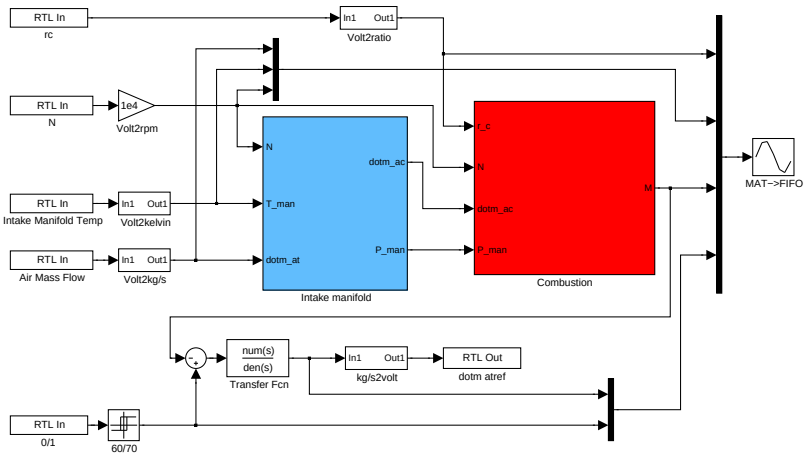

Figure 4.9: A model estimating the torque in an SVC engine and feeding a controller with the estimate.

4.4 Comparison of RTAI and RTLinux Interfaces

The main difference in the user interfaces when using RTAI or RTLinux is that RTAI make use of the C API, see section 3.2.4, while RTLinux uses the external mode, see section 3.2.3. In Table 4.3 differences and similarities are presented. A lack in the use of RTW-RTAI-4.65.3, see section 3.4, is that it is not possible at the moment to change parameters in a model while the model is running, see Appendix E.

Capability	**RTAI**	**RTLinux**
User Interface	Making use of the C API	Making use of the external mode
Changing Parameters	Not possible while the model is running	May be changed while the model is running
Scopes	Only one scope, but many signals, at a time can be used to display, or store, signals	At most seven scopes can be used
Logging	Uses realtime FIFOs for data logging	
I/O	Makes use of a Comedi device driver	
Model Execution	The code is generated in Simulink and compiled and run in a terminal window	

Table 4.3: Differences and similarities between RTAI and RTLinux.

Chapter 5

Conclusions

Both implemented systems, RTLinux and RTAI, have been used in Vehicular Systems automotive engine laboratory with success. Signals have been measured and a control signal has been applied to regulate the amount of throttle plate angle needed in an engine.

Users of the systems have managed to handle the systems by themselves after a shorter introduction, approximately 30 minutes. Also master's students, with less experience of Real-Time Workshop, have successfully used the systems.

The specification part, in Chapter 1, has been fulfilled:

- The system has an interface to Matlab/Simulink, i.e. a model is constructed in Simulink in an ordinary way, with only posing a few restrictions on the model, see Chapter 2.
- From Simulink models C code is automatically generated for a hard realtime operating system, RTAI or RTLinux.
- The system, with the used hardware, is capable of execution frequencies up to 50 kHz, depending of the complexity of the executed code.
- Only ordinary PC hardware have been used, see Appendix A.
- A standard data acquisition card, National Instruments' NI 6035E, has been used.
- The system is easily extendible with most data acquisition cards through the amount of device drivers provided by Comedi. The use of open source software and ordinary PC hardware will add maintainability to the system.
    - For the possibility to use a CAN bus, there is at least two possible ways to do this. First, it is possible to use of the

Comedi concept to develop a device driver for a CAN device. Second, to look at the *rtcan*[1] project. rtcan is a set of functions allowing realtime CAN messages to be sent using a system running RTAI. Message sending and receiving functions can be called from within RTAI threads. rtcan is based on *Ocan*[2], a pure Linux CAN device driver. rtcan uses the TQM8xxL board as the target architecture, and, according to the information on the homepage for the rtcan project, it should be trivial to port the software to any of:

* PC-104 cards built by EuroTech
* PC-ECAN ISA devices
* GEA-Automotive devices
* Applied Data Systems' Graphics Client Plus

5.1 Future advice

The new RTAI distribution, release 24.1.11, includes Real-Time Workshop support with *RTAI-Lab*. The approach taken is similar to RTW-RTAI-4.64.3, it uses the C API mode. The advantage of using RTAI-Lab is the continued support. It also has Simulink library support of the NI 6035e data acquisition card, making use of the Comedi device driver concept. Both STRTL_M6.5 and RTW-RTAI-4.65.3 lacks further support and are therefore less suitable to continue working with.

[1] rtcan – realtime CAN. See `http://www.peak.uklinux.net/gnulin.php`, the homepage will soon move to *SourceForge.net*.

[2] Ocan – Open-CAN – is a device driver for the Intel 82527 CAN controllers.

References

[1] Björn Rudin. *Realtidsegenskaper hos* W*indows* NT. Master's thesis LiTH-ISY-EX-2027-990329, Department of Electrical Engineering, Linköpings Universitet, Linköping, Sweden, March 1999.

[2] Ismael Ripoll, Pavel Pisa, Luca Abeni, Paolo Gai, Agnes Lanusse, and Sergio Saez. *RTOS State of the Art Analysis*. OCERA. `http://www.mnis.fr/opensource/ocera/rtos/book1.html`, March 2003.

[3] Real Time Linux Foundation, Inc. *Variants*. `http://www.realtimelinuxfoundation.org/variants/variants.html`.

[4] Tim Bird. *Comparing two approaches to real-time Linux*. `http://www.linuxdevices.com/articles/AT7005360270.html`, December, 21 2000. CTO of Lineo.

[5] L. H. Seawright and R. A. Mackinnon. *VM/370 – A Study of Multiplicity and Usefulness.* IBM Systems Journal, (18):4–17, 1978.

[6] Victor Yodaiken and Michael Barabanov. *A Real-Time Linux*. New Mexico Institute of Technology.

[7] Victor Yodaiken. *The RTLinux Manifesto*. Department of Computer Science New Mexico Institute of Technology Socorro NM 87801.

[8] Ismael Ripoll, University of Valencia, Spain. *Earliest Deadline First Scheduler*. `http://bernia.disca.upv.es/~iripoll`, 1998. Technical report.

[9] Patricia Balbastre & Ismael Ripoll. *Integrated Dynamic Priority Scheduler for RTLinux*. Department of Computer Engineering (DISCA).

[10] Victor Yodaiken, Finite State Machine Labs Inc. *FSMLabs Lean POSIX for RTLinux*. `http://www.fsmlabs.com/articles/posix/posix.htm`, 2000.

[11] Paolo Mantegazza et al. Lineo Inc. *DIAPM, RTAI Programming Guide 1.0*. `http://www.rtai.org`, September 2000.

[12] The MathWorks, Inc., 3 Apple Hill Drive, Natick, MA. USA. *Real-Time Workshop User's Guide*, updated for version 5.0 (release 13) edition, July 2002. Online only.

[13] Raul Murillo Garcia, Glasgow Caledonian University. *Simulink Target for RT-Linux*.
`http://www.sesd.gcal.ac.uk/raulm/St-rtl.htm`, March 2003.

[14] Roberto Bucher, University of Applied Sciences of Southern Switzerland (SUPSI) Dept. of cs and ee (DIE). *Interfacing Linux RTAI with Matlab/simulink/RTW and Scilab/Scicos*.
`http://a.die.supsi.ch/ bucher/`.

[15] David Schleef, Frank Hess, and Herman Bruyninckx. *Comedi Documentation*.
`http://www.comedi.org/doc/index.html#AEN22`.

[16] The MathWorks, Inc., 3 Apple Hill Drive, Natick, MA. USA. *Getting Started with Real-Time Workshop*, first printing edition, July 2002. Online only.

[17] *Real-Time Workshop User's Guide*, chapter 6. 3 Apple Hill Drive, Natick, MA. USA, July 2002. Online only.

[18] *Real-Time Workshop User's Guide*, chapter 14, pages 77–92. 3 Apple Hill Drive, Natick, MA. USA, July 2002. Online only.

[19] The MathWorks, Inc., 3 Apple Hill Drive, Natick, MA. USA. *Real-Time Workshop Release notes*, updated for version 5.0 (release 13) edition. Online only.

[20] *Real-Time Workshop Release Notes*, chapter 1, Upgrading from an Earlier Release, pages 32–35. 3 Apple Hill Drive, Natick, MA. USA. Online only.

[21] *Real-Time Workshop User's Guide*, page 10. 3 Apple Hill Drive, Natick, MA. USA, July 2002. Appendix C.

[22] E. Bianchi, L. Dozio, Dipartimento di Ingegneria Aerospaziale, Politecnico di Milano, Italy. *Some Experiences In Fast Hard-Heal Time Control In User Space With RTAI-LXRT*.
`http://www.linuxdevices.com/articles/AT9601485147.html`.

[23] Per Andersson. *Intake Air Dynamics on a Turbocharged SI-Engine with Wastegate*. Licentiate thesis, Linköpings Universitet, Linköping, Sweden, 2002.

[24] Erik Sunnegårdh. *Adaptive Catalyst Model for Control.* Master's thesis LiTH-ISY-EX-3249-2002, Department of Electrical Engineering, Linköpings Universitet, Linköping, Sweden, December 2002.

[25] Andreas Bergström. *Modeling and Control of Torque in a SVC Engine.* Master's thesis LiTH-ISY-EX-3421-2003, Department of Electrical Engineering, Linköpings Universitet, Linköping, Sweden, May 2003.

[26] The Mathworks, Inc. *Real-Time Workshop, tutorial 5.* Included in application. Version 6.5.

[27] *Real-Time Workshop User's Guide*, chapter 6, page 28. 3 Apple Hill Drive, Natick, MA. USA, July 2002. Online only.

Notation

Acronyms

API	Application Program Interface
CAN	Controller Area Network
Comedi	Linux control and measurement device interface
DAQ	Data Acquisition
RTAI	RTOS using Linux
RTLinux	RTOS using Linux
RTOS	Realtime Operating System
RTW	Real-Time Workshop
STRTL	Simulink Target for RTLinux
STRTL_M6.5	STRTL for Matlab version 6.5

Definitions

Realtime Throughout the writing of this master's thesis the spelling of *realtime* will be as is, i.e. **not** *real-time*. A Google[3] search with the parameter ``real-time'' gives about 6,360,000 hits[4], filtering out both `real-time` and `real time`. A similar search with the parameter `realtime` gives about 996,000 hits, indicating the use of the chosen spelling is fairly common; even though `realtime` is overrepresented in the digital world; some kind of apartheid against white spaces; or, in an evolutionary mind, not so fit to the environment. The probable cause of the chosen spelling might be tracked down to Swedish heritage[5].

Typeface The typeface `typeface` indicates a `file`, a `directory` or a `command`.

[3] An Internet search engine, `www.google.com`.

[4] Every identified Internet page meeting the search criteria is considered a hit.

[5] In the Swedish language it's preferred many times to write two words, making up something new, into one word, some kind of inventing new words.

Appendix A

Hardware

Listing of hardware used:

Processor Athlon 1900+

Motherboard Asus

Chipset VIA KT333

Memory 256 MB DDR

Graphic card nVidia GeForce 2 MX 32 MB

Hard drive WDC WD400EB-00CPF0 40 GB

Ethernet card VIA VT86c100A Rhine-II PCI

DAQ NI 6035e

Appendix B

Software

Listing of software used:

Matlab version 6.5

Simulink version 5.0

Real-Time Workshop version 5.0

STRTL_M6.5 Simulink Target for RTLinux, using Matlab 6.5

RTW-RTAI-4.65.3 Targeting RTAI, using Matlab 6.5

RTLinux version 3.2pre1, using patched Linux kernel 2.4.18

RTAI version 24.1.10, using patched Linux kernel 2.4.19

Comedi comedi-0.7.65 and comedilib-0.7.19

Workstation Redhat 8.0

Appendix C

External Mode Configuration and Execution

The setup of the model and code generation parameters required for external mode compatible programs will be explained; and further how to generate the code, build the target executable and run the application.

C.1 Setting Up the Model

The information used in this section is mostly an excerpt from [26].

- In the **Simulation Parameters** dialog box, on the Solver pane, set the **Solver options Type** to Fixed-step, set the step time, and select the discrete solver algorithm.

- On the Workspace I/O pane, clear the **Time** and **Output** check boxes. (RTLinux can't log data to the workspace or to a MAT-file.)

- On the Real-Time Workshop pane, select Target configuration from the **Category** menu. If RTLinux is *not* selected, click the **Browse** button and select the RTLinux target from the System Target File Browser.

- Select GRT code generation options from the **Category** menu and select the **External mode** option. This enables generation of external mode support code.

- On the Advanced pane, make sure that the **Inline Parameters** option is not selected. External mode supports inlined parameters, but Simulink Target for RTLinux does not.

The **External Mode Control Panel** lets you configure host and target communications, signal monitoring, and data archiving. It also lets you connect to the target program and start and stop execution of the model code.

- The **Target interface** button opens the **External Target interface** dialog box. This dialog box configures the external mode interface options.
  - The **MEX-file for external interface** field specifies the name of a MEX-file that supports host and target communications on the host side. The default is ext_comm, a MEX-file provided by Real-Time Workshop. When using Simulink Target for RTLinux the ext_comm_rtl will be used.
  - The **MEX-file arguments** field specifies arguments, such as a TCP/IP server port number, to be passed to the external interface program, e.g. 'bristol.isy.liu.se' or '130.236.50.228' (don't forget the single quotes). Note that these arguments are specific to the external interface file used. For more information on the arguments see [27].
- In the **External Mode Control Panel** select the **Signal & Triggering** button and make sure that:
  - **Trigger Source** is set to manual
  - **Trigger Mode** is set to normal
  - **Arm when connect to target** is selected

C.2 Generating Code

Before generating the code, make sure the model is properly setup:

- In the **Simulation Parameter** on the Real-Time Workshop pane, check the **Generate Code Only** option.
- To generate code and create a target program, click the **Build** button on the Real-Time Workshop pane.

The generated code will exist in a subdirectory of the current working directory, i.e. *pwd/<model>_rtl/*.

C.3 Running the Application

The code must be generated before the application can be run. If Simulink is not run on the RTLinux machine, the generated code has to be copied to the RTLinux machine. On the RTLinux machine do the following:

- Change to the directory *.../<model>_rtl* and execute *make -f <model>.mk*

Now Simulink can connect to the application:

- From the simulation mode pull-down menu, select **External**
- Then select **Connect to target** from the same pull-down menu.

Watch window for messages, such as `connect`.

Appendix D

Installation

The software will reside on *bristol.isy.liu.se*.

D.1 System

This example installation will describe an installation of a real time rapid prototyping development platform, except for the installation of a Linux distribution and Matlab (Simulink, Real-Time Workshop), which is assumed to be preinstalled.

The distribution used is Redhat 8.0, and the Matlab version is R13.

The software to install is the Linux kernel, version 2.4.19, RTAI, version 24.1.11 pre release 3, COMEDI, for device driver support, version 0.7.65, and FLTK, for graphical rtailab support, version 1.1.3.

The software is assumed to reside in different subdirectories to the directory */home/Realtime/*.

```
$ cd /usr/src
$ tar xzf /home/RealTime/Misc/fltk-1.1.3-source.tar.gz
$ cd fltk-1.1.3
$ ./configure --enable-threads[1]
$ make
$ make install
$ cd /usr/src
$ tar xzf /home/RealTime/comedi/comedilib-0.7.65.tgz
$ mkdir rtai
$ cd rtai
$ tar xzf /home/RealTime/linux-kernels/linux-2.4.19.tar.gz
$ tar xzf /home/RealTime/linux-real-time-extensions/rtai-24.1.11-pre3.tgz
$ cd linux-2.4.19
```

[1] Other necessary options might be, e.g., CC=gcc295 CXX=g++295.

```
$ patch -p1 < ../rtai-24.1.11-pre3/patches/patch-2.4.19-rthal5g
```

Now might be a good time edit the Makefile, and set the CC to an appropriate compiler[2]

```
$ cp /home/RealTime/dot-config-files/rtai/linux-2.4.19/.config .
$ make oldconfig[3]
$ make dep
$ make bzImage
$ make modules
$ make modules_install[4]
$ make install
```

Now you have to restart the computer with the newly made kernel.

```
$ cd /usr/src/rtai/rtai-24.1.11-pre3
```

Edit the Makefile.modbuild, i.e. set the CC and CXX macros. In *comedi_lxrt/Makefile* and *comedi_lxrt/lib/Makefile* set the path to the comedi directory.

```
$ make menuconfig
$ make dep
$ cd /usr/src/comedi/comedi-0.7.65
$ make
$ make
$ cd /usr/src/rtai/rtai-24.1.11-pre3
$ make
$ ./setsched up
$ make install
$ make dev
$ cd /usr/src/comedi/comedi-0.7.65
$ make install
$ make dev
```

D.2 Installation of Simulink Target for RTLinux

In order to use RTLinux as a rapid prototyping target for Real-Time Workshop, the following must be done:

[2] In */home/RealTime/rpms/* the rpm's for gcc295 and g++295 reside.
[3] You might want to run *make menuconfig* or *make xconfig* after *make oldconfig*.
[4] Not sure this is necessary.

- The directory **STRTL_M6.5** and its subdirectories contains the Real-Time Workshop source files. This directory structure has to be copied to the RTLinux machine.
- The directory **rtlinux** has to be copied to the host machine (Simulink), and the path of the directory added to the Matlab search path. (Make sure *not* to add this directory within the Matlab directory structure, i.e. in the obvious place as a parallell directory to other Real-Time Workshop targets, in a UNIX environment, thus Matlab will not be able to add the new system target file.)
- The definition of the macro MATLAB_ROOT in the template makefile `rtlinux.tmf` must point to the **STRTL_M6.5** directory.
- In order to build the MEX-file, the directory **src** within the directory structure **STRTL_M6.5** has to be present on the host machine (Simulink).

  To build the MEX-file in a UNIX environment:

```
$ cd MATLAB_ROOT/toolbox/rtw
$ mex STRTL_M6.5_ROOT/rtlinux/ext_comm_rtl.c \
    MATLAB_ROOT/rtw/ext_mode/ext_convert.c \
    MATLAB_ROOT/rtw/ext_mode/ext_transport \
    -I/STRTL_M6.5/rtw/c/src \
    -IMATLAB_ROOT/rtw/ext_mode
```

  In a Microsoft environment:

```
$ cd MATLAB_ROOT\toolbox\rtw
$ mex STRTL_M6.5_ROOT\ext_comm_rtl.c \
    MATLAB_ROOT\rtw\ext_mode\ext_convert.c \
    MATLAB_ROOT\rtw\ext_mode\ext_transport.c \
    -I\STRTL_M6.5_ROOT\rtw\c\src \
    -IMATLAB_ROOT\rtw\ext_mode \
    COMPILER_LIBRARY_PATH\wsock32.lib
```

  After the MEX-file been generated, move it to the **rtlinux** directory.

D.3 Comedi

The Comedi software package is assumed to reside on *bristol.isy.liu.se*. The directory to install into is only a suggestion.

```
$ cd /usr/src/
$ mkdir comedi
$ cd comedi
$ tar xzf /home/Realtime/comedi/comedi-0.7.65.tgz
$ cd comedi-0.7.65
```

And continue with reading

```
$ less INSTALL
```

The same steps applies to `/home/Realtime/comedi/comedilib-0-7.19.tgz`.

Appendix E

Frequently Asked Questions

E.1 STRTL_M6.5

How many scopes can be used in a model? At most seven scopes can be used in a model.

What about common.h? There are a few things that might be of interest:

DATA_LOG If set to *TRUE* scope data is stored in files for post processing[1].

DATA_LOG_NAME A user defined name of the logging file(s) can be entered here, the default name is "data". A file will be created for each sample rate in the model, and the file extensions will be .001, .002, ...

DATA_LOG_RES Number of decimals for the data logged.

FINAL_TIME

- 0 – The model runs continuously.
- A positive real – Overrides the final time in Simulink.
- -1 – Keeps the final time defined in Simulink.

E.2 RTW-RTAI-4.65.3

Can I enhance the number of signals displayed on a scope? Yes, the following must be done:

[1] Data logging is only possible in external mode.

- In *rt_proc.h* (*/usr/local/rtai/*) change the value of the constant N_CHANNELS to the new value.
- rtai_lib.mdl (*MATLAB_ROOT/toolbox/rtai/*) has to be modified according to the changes.
- In *scope.c* change the constant N_CHANNEL to the new value. Compile the file and move it to */usr/local/bin/*.
- In *rtplot.h* change the *X* in *pt_old[X]*, *pt_new[X]* and *pen_c[X]* to the new value and update *rtplot.cpp* and *rtppltfrm.ui* according to the changes. Compile rtplot and rtppar and move the executables to */usr/local/bin/*.

Can I change the resolution of the logged signals? Yes, in *scope.c* look for the *fprintf* and make the desired changes. Compile the file and move it to */usr/local/bin/*.

I have problem running *scope*, *rtplot* or *rtppar* This might be due to that gcc 2.9x compiled applications have problems work/link with gcc 3.x compiled Qt. Solution, recompile `scope.c`, `rtplot.cpp` and/or *rtplot.cpp*.

I have problem running *changertpar*. No solution is available to fix this problem at the moment.

E.3 rtailab-24.1.11-pre2

Can I enhance the number of signals displayed on a scope? Yes, in *rtailab.h* change the constant MAX_CHANNELS to the desired number of signals and *make clean; make* in the **rtailab** directory.

Appendix F

Test Code

To measure and log the computation times the file `rt_proc.c` was modified to write the computation times to a realtime fifo, and a C program was written to read from the realtime fifo and store the computation times in a file.

To estimate the computation times a bubble sort and a bubble sort with floating point operations was used.

F.1 Modified `rt_proc.c`

To measure the computation time, for a given time step in a model, some modifications to `rt_proc.c` was made:

A function was added:

```
void write_fifo_long(int fifo_id, long long messg)
{
   rtf_put(fifo_id, &messg, sizeof(long long));
}
```

The function `fun()` was modified:

```
long long tsc1; \* added line *\
long long tsc2; \* added line *\

while(1){
  if(parData->op) checkParam();
  tsc1 = rt_get_cpu_time_ns(); \* added line *\
  update_rtw();
  tsc2 = rt_get_cpu_time_ns(); \* added line *\
  write_fifo_long(63, tsc2 - tsc1); \* added line *\
  rt_task_wait_period();
}
```

Initialization of fifo in `init_module()`**:**

```
ini_fifo(63,1); \* added line *\
```

Cleaning up of fifo in `cleanup_module()`**:**

```
clean_fifo(63); \* added line *\
```

F.2 Logging of Computation Times

To log the computation times, the following C program was used:

```
#include ‘‘stdio.h’’
#include ‘‘unistd.h’’
#include ‘‘sys/types.h’’
#include ‘‘sys/mman.h’’
#include ‘‘sys/stat.h’’
#include ‘‘fcntl.h’’
#include ‘‘signal.h’’

#define DATA_FILE ‘‘computation-times.dat’’

static int end;

static void endme(int dummy) end=1;

int main(int argc, char** argv)
{
   int fifo;
   FILE * fd=NULL;
   int flag=0;
   int i;
   long long msg;

   char device[]=‘‘/dev/rtf63’’;

   if(argc==2) device[8]=argv[1][0];

   if((fifo=open(device,O_RDONLY | O_NONBLOCK)) < 0){
     fprintf(stderr,‘‘Error opening /dev/rtf0\n’’);
     exit(1);
   }

   fd=fopen(DATA_FILE,‘‘w’’);

   signal(SIGINT,endme);
```

```
   while(!end){
     if(read(fifo,&msg,sizeof(msg))!=-1) {
       fprintf(fd,``%d\n'',msg);
     }
   }

   if(fd!=NULL) fclose(fd);
     return 0;
}
```

F.3 Bubble Sort

The source code, for the bubble sort algorithm, used in the S-Function Builder. The `SIZE` is defined in the includes, on the Library pane.

```
static double a[SIZE];
static double b[SIZE];
int i;
int j;
double tmp;

a[0] = u0[0];
for(i = SIZE; i > 0; i--){
   for(j = 1; j < i; j++){
     if(a[j] < a[j-1]){
       tmp = a[j];
       a[j] = a[j-1];
       a[j-1] = tmp;
     }
   }
}
y0[0] = a[SIZE / 2];
```

F.4 Bubble Sort + Floating Point Operations

The source code, for the bubble sort algorithm and the floating point operations, used in the S-Function Builder. The `SIZE` is defined in the includes, on the Library pane.

```
static double a[SIZE];
static double b[SIZE];
```

```
int i;
int j;
double tmp;

a[0] = u0[0];
for(i = SIZE; i > 0; i--){
   for(j = 1; j < i; j++){
     if(a[j] < a[j-1]){
       tmp = a[j];
       a[j] = a[j-1];
       a[j-1] = tmp;
     }
     b[j] = a[j] * a[j];
   }
}
b[0] = a[0] * a[0];
y0[0] = b[SIZE / 2];
```