Czech Technical University in Prague
Faculty of Electrical Engineering

DIPLOMA THESIS

Software for Eurobot Competition and its Timing Analysis

Prague, 2009

Author: Martin Žídek

Prohlášení

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

V Praze dne ______________ ____________________

podpis

Acknowledgements

I would like to thank especially the supervisor of my thesis, Ing. Michal Sojka, who supported me with patient guidance and advices in course of working on the topic. His deep insight into embedded real-time systems was of invaluable help.

Next I want to thank Dr. Ian Broster and other engineers in Rapita Systems, Ltd. for providing me with RapiTime, which can be considered state-of-the-art tool in the dynamical timing analysis field.

Last but not least I want to express my great gratitude to my family and my girlfriend, who supported me during the whole time of my university studies.

Abstrakt

Tato práce se zaobírá tvorbou softwaru pro mobilního autonomního robota pro soutěž Eurobot a především možnostmi časové analýzy takového softwaru. V rámci práce byla vytvořena metoda pro přesné měření času na cílovém vestavěném systému s procesorem architektury PowerPC. Tato metoda byla použita pro časovou analýzu vybraných částí řídicího softwaru pro mobilního robota. Byla také provedena případová studie zabývající se možnostmi časové analýzy zdrojového kódu, vygenerovaného automaticky ze simulačního schématu MATLAB/Simulink pomocí nástroje Real-Time Workshop.

Abstract

The aim of this thesis is to describe control software for an autonomous mobile robot for the Eurobot competition and especially to explore possibilities of timing analysis of the software. A method for precise time measurement on a target embedded system based on PowerPC processor was established. The method was then used to measure selected parts of the mobile robot control software. A case-study was carried out, describing posibilities of timing analysis of an application code generated automatically from a MATLAB/Simulink simulation model using Real-Time Workshop plugin.

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta elektrotechnická
Katedra měření

Akademický rok **2008-2009**

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student:	**Bc. Martin Žídek**
Obor:	**Kybernetika a měření – blok Měřicí a přístrojové systémy**
Název tématu česky:	**Tvorba a časová analýza softwaru pro soutěž Eurobot**
Název tématu anglicky:	**Software for Eurobot Competition and its Timing Analysis**

Zásady pro vypracování:

1. Seznamte se s pravidly soutěže Eurobot pro rok 2008.
2. Ve spolupráci s ostatními členy týmu vytvořte software pro řízení soutěžního robota.
3. Prozkoumejte možnosti přesného měření času na procesoru MPC5200 s OS Linux.
4. Naměřte skutečnou dobu běhu jednotlivých softwarových komponent a vytvořte model systému pomocí nástroje VirtualTime.
5. Vytvořte případovou studii na použití softwaru VirtualTime pro časovou analýzu kódu generovaného nástrojem Matlab Embedded Coder.
6. Vše důkladně zdokumentujte.

Seznam odborné literatury:

[1] Freescale Semiconductors, *Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture*, dostupné on-line

Vedoucí diplomové práce:	Ing. Michal Sojka (katedra řídicí techniky)
Datum zadání diplomové práce:	27. června 2008
Platnost zadání do[1]:	22. ledna 2010

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE Fakulta elektrotechnická L.S. -7-

V.Z. [illegible]
Prof. Ing. Pavel Ripka, CSc.
vedoucí katedry

v.z. [illegible]
Doc. Ing. Boris Šimák, CSc.
děkan

V Praze dne 27.6.2008

[1] Platnost zadání je omezena na dobu tří následujících semestrů.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Timing analysis is an important part of control applications design process. Worst case execution time is a key concern of safety critical applications developers, especially in automotive or aerospace industry. Machinery control is also one of application fields where timing analysis is crucial for propper function of a control application. Modern control theory offers many elaborate and sophisticated control algorithms, but for these to work, time constraints on the execution time of their actual implementation must be met.

Worst case execution-time (WCET) analysis methods are generally divided into two categories, according to the approach taken when doing assumptions on WCET parameters of a code block. An overview of the methods used and tools available for this purpose is given for example in (Wilhelm et al., 2008).

The first category of methods used for WCET analysis is referred to as *statical analysis*. Statical analysis is based on mathematical models of the system (mostly the CPU) the application under analysis is running on and on analysis of all possible control flows in the application. This method, although it can produce very tight WCET estimates, is very demanding, because its results highly depend on the accuracy of the created model. As modern microcontrollers have still more and more complicated architectures, deriving of such models becomes more and more challenging task. It might be event impossible to derive the model, or the time costs related to the process might get unreasonable. Each new processor and even a setup using this processor requires a new model to be created.

Another category of methods for WCET analysis is called *dynamical analysis*. These methods are based on measuring the execution time on the actual target system. These methods give also only estimates of the WCET and how tight these estimates are depends very much on the test vectors used. Only inputs causing the worst-case execution path can produce reliable estimates. It is however very difficult, if not impossible, to generate such testing vectors. There are nevertheless efforts, which aim to provide proof of having observed the WCET on the measurement level. (Schaefer et al., 2006)

Sometimes it is also uselful to do a timing analysis of an application, for which actual worst-case execution time is not that crucial, but having typical execution times might come in handy. This is also the case of the application work described in this thesis. A method for measurement of execution times on a PowerPC-based embedded target was developed. The method was then applied to a mobile robot control software.

The structure of the work is as follows.

Chapter 2 gives a brief overview of the software developed for a mobile autonomous robot, which was built to take part in an European competition of mobile robots called Eurobot. The structure and architecture of the software is described to lay grounds for later discussion of the measured timing behavior of the software.

Chapter 3 describes the research that was done in order to figure out possibilites of precise time measurement on a PowerPC target system with GNU/Linux as an operating system. This section contains discussion of all the possibilities available and their advantages and disadvantages.

Chapter 4 describes the actual implementation of measurement method arising from the results of the research done in previous chapter. All the tricky aspects of time measurement framework implementation on top of an embedded target system are described here.

Chapter 5 represents a short case-study of possibility of application of the timing analysis on a source code generated automatically from a simulation schema in MATLAB/Simulink environment. Automatic C code generation from a simulation schema is an interesting feature offered by the Real-Time Workshop plugin for MATLAB/Simulink, allowing creation of advanced control algorithms without substantial knowledge of the C programming language and pitfalls connected with embedded software development. The timing information acquired thanks to the measurement can then be used to create a simulation of the timing behavior of an application generated from MATLAB/Simulink. This section was originaly created as a part of a deliverable (Žídek et al., 2008) for the FRESCOR project (*Framework for Real-time Embedded Systems based on COntRacts* (FRESCOR project, n.d.)).

The last chapter summarizes results achieved while working on this topic and outlines possibilities of its further application and development.

The presented thesis is a result of three years of work on the mobile robot project, so it is kind of a compilation of the results achieved during this time period.

Chapter 2

Mobile Robot Software

One part of this diploma thesis was to create software to control a mobile autonomous robot for the Eurobot competition.

There is a team at the Department of Control Engineering at the Faculty of Electrical Engineering of CTU, which has taken part in the Eurobot competition for past three years. Eurobot is a competition of mobile autonomous robots that is designed especially for teams of young people, organised either in student projects or in independent clubs. There are national rounds of the competition in most european countries each year, of which the most successful teams can take part in international finals. The competition itself is very challenging, since its rules are redesigned every year, to give newly participating teams same status as the ones, which have already taken part in the competition. Nevertheless the basic concepts of the competition remain the same: matches are organised in rounds lasting 90 seconds, the maximal allowed dimensions of the robots remain more or less the same and the playground for the competition has also fixed dimensions.

The software for the robot was developed continuosly during the past three years by members of the team. Since the beginning we designed the software with its maximum reusability on mind. For this reason the software components are organised in independent layers in a modular manner.

Since the software itself is result of team work, I will present only its basic concepts in this work. Main focus of the thesis lies in the timing analysis of the software. The description of the overall design of the software is important in order to understand the results of the timing analysis presented later in the work.

2.1 Mobile Robot – Structure of the System

The robot was designed in a highly modular manner, in order to allow a high level of reusability of all parts of the system, since the robot has to be redesigned each year for the competition because of changing rules.

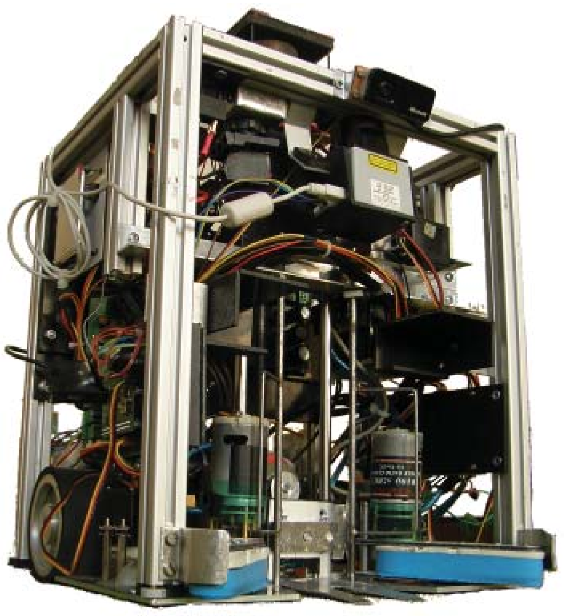


Figure 2.1: Robot for Eurobot 2008

A block diagram describing the system can be seen in Fig. 2.2.

The heart of the system is an embedded board with an MPC5200 processor by Freescale (PowerPC architecture). This board has the function of main control computer. More information on the board can be found further in section 3.1.

The electronics of the robot is divided into submodules (drive control, power supply, sensors, etc.). All modules together with the main control board are interconnected by CAN bus. This bus was chosen especially because of its noise immunity and easy scalability. This bus is widely used in industrial control applications and in automotive applications.

The main controller intercepts data from the submodules on the CAN bus and issues control commands back to the submodules. This functionality is serviced by a standalone application on the controller board. All data are then redirected to the ORTE communication layer. ORTE is an open-source implementation of real-time publish-subscrib protocol. More information on ORTE can be found in section 2.2. The greatest ad-

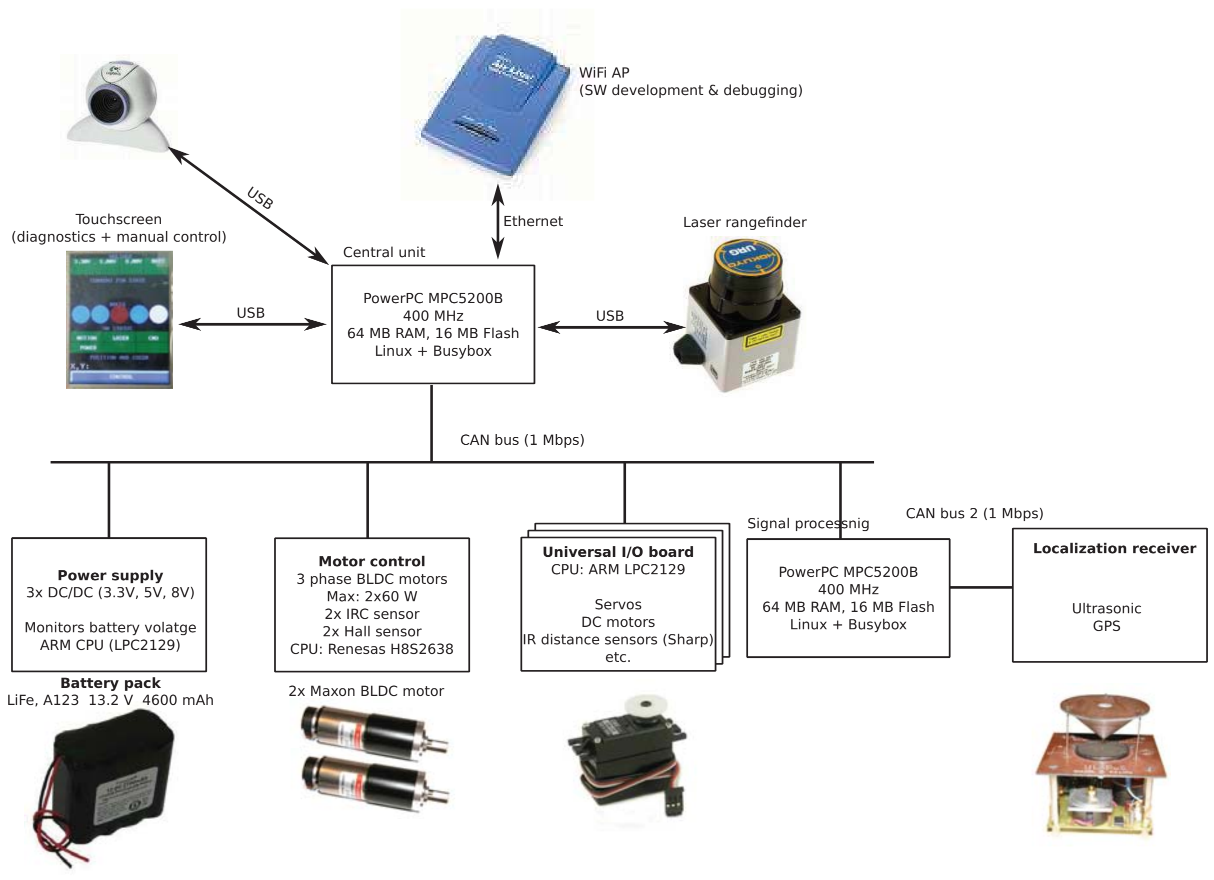


Figure 2.2: Schematic diagram of the robot

vantage of this architecture is, that the data can be read from the ORTE layer in the controlling application on the main robot controller, as well as in applications on remote hosts used for debugging and tuning of the algorithms. This is because the ORTE layer is based on the Ethernet interface, which can be easily connected to remote hosts using wireless network adapter.

The robot uses ultrasonic transmitters placed on defined spots around the playground and an ultrasonic receiver placed on the robot itself to calculate the position of the robot on the playground. The calculation is based on Kallman filtration and uses a principle simillar to GPS system. Since the calculation needs a lot of computing power, there is another MPC5200 based computer dedicated to this task. The results of the calculations are then sent to the main controller over the CAN bus.

2.2 ORTE

ORTE stands for *OCERA Real Time Ethernet* (Smolik et al., 2004) and it was a part of OCERA project. ORTE is an open source implementation of RTPS (*Real-Time Publish-Subscribe*) communication protocol defined by *Real-Time Innovations (RTI)*. RTPS is an application layer protocol, which has two main communication modes. One is the publish-subscribe protocol for transferring the data from publisher to subscribers, and the other one is Composite State Transfer (CST) protocol, which transfers state. RTPS protocol was designed to use an unreliable underlying network protocol, such as *IP/UDP*.

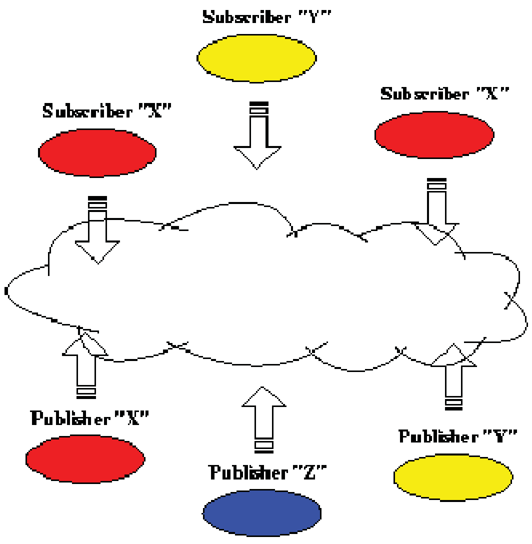


Figure 2.3: Publisher-subscriber model of ORTE communication

The publish-subscribe architecture was designed to simplify data distribution from one source to many recipients. The publisher does not have to have any knowledge of the number or location of subscribers. Also, the subscribers simply receive the data anonymously and thus they don't need to know any information about the publisher. An application can be publisher and subscriber at the same time.

The publish-subscribe architecture is best suited for distributed applications. It is scalable and the data flows can be managed easily regardless of the number of nodes (publishers and subscribers) connected to the system. When subscribing to a data flow, the application specifies only the topic of the data it wants to receive, rather than to any specific publisher.

The publish-subscribe services are typically made available to applications through middle-ware, that sits on top of the operating system network interface and presents an application programming interface.

In our case, we are using ORTE to publish data from an application, which intercepts data from CAN bus. Then the data are subscribed in the main controlling application running on the main controller board, but also in a graphical application running on a standard desktop or portable computer, which we use for visualizing all the data from the robot. Thus we can see on-line all the sensors readings, information about the position of the robot etc. The controlling application publishes all the commands for controlling the motion of the robot to ORTE also, and the data are subscribed in CAN bus communication application, which transforms them into appropriate CAN messages and sends them to the motor driver board. This approach makes it possible to control the robot as from the controlling application as from the graphical application on a separate computer in the same manner.

2.3 Robot Software Structure

A schematical representation of the structure of the software for the mobile robot can be seen in Figure 2.4. Boxes represent individual processes, the arrows show data flow among the processes.

2.3.1 Interface Between CAN-bus and ORTE Layer

As was mentioned earlier, the communication between the main controller of the robot and its sensoric and actuator submodules is carried out over the CAN-bus. There is a stand-alone application running on the main controller (denoted as *cand* in Fig. 2.4), which receives all the data packets from the CAN bus and then publishes the data to the ORTE layer. It also subscribes data containing robot control commands, which are published on ORTE by the main control application, and transforms them into appropriate messages to the CAN bus.

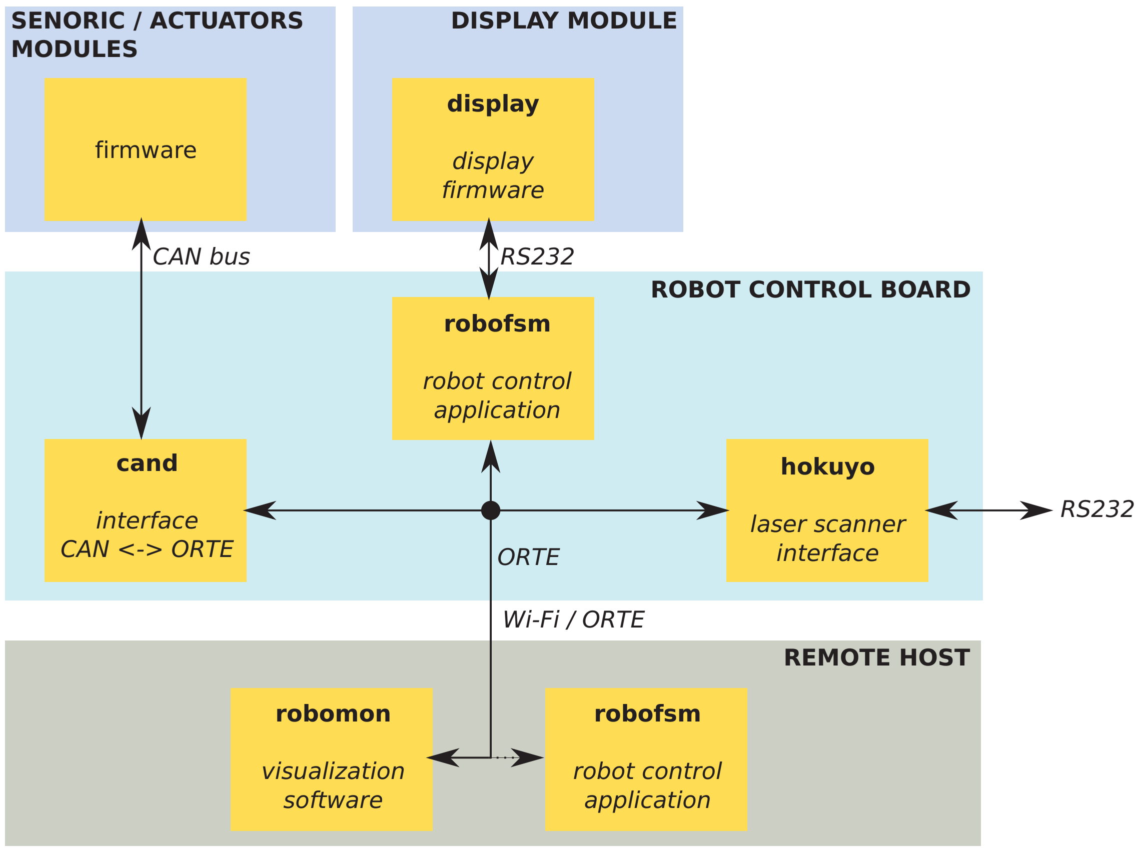


Figure 2.4: Software structure

2.3.2 Robot Control Application

The software controlling the whole system and implementing the actual game strategy has the form of parallel finite state machines (further refered to as *FSM*). One state machine serves for motion control, one for actuators control and another one control the game strategy. The communication between the state machines is implemented as events issued by individual tasks.

The main control application is denoted as *robofsm* in Figure 2.4. It is interesting to note, that thanks to the ORTE communication channel used and modular software archietcture, it is possible to run the control application not only from the main controller placed on the robot, but also on a host computer connected to the robot through a wireless Ethernet connection. This is useful especially during the algorithms development phase.

There was a library created containing many C language preprocessor macros, which eases the use of the FSM framework by hiding the implementation details of the automatons from the programmer writing the robot control application. Sample of the notation used to work with the FSM framework is in Listing 2.1

```
FSM_STATE_DECL(state1);
FSM_STATE_DECL(state2);

FSM_STATE(state1)
{
    switch(FSM_EVENT) {
        case EV_ENTRY:
            FSM_TIMER(100);
            break;
        case EV_TIMER:
            some_action();
            FSM_TRANSITION(state2);
            break;
    }
}

FSM_STATE(state2)
{
    switch(FSM_EVENT) {
        case EV_ENTRY:
            another_action();
            FSM_TIMER(200);
            break;
        case EV_TIMER:
            FSM_SIGNAL(ANOTHER_FSM, EV_SOME_EVENT, *data_pointer);
            break;
    }
}
```

Listing 2.1: Sample FSM code

As can be seen from the sample code, the code controlling the robot based on inputs from its sensors and game strategy algorithms really represents an event-driven finite state machines paradigm. We came out with this architecture, because in our opinion a robot control algorithms can be well represented in this way, since the control usually consists of waiting for some input, like a particular sensor value, which represents a *state* in the FSM terms. When the expected value appears, a *transition* to another state occurs. In this state the robot performs a requested action.

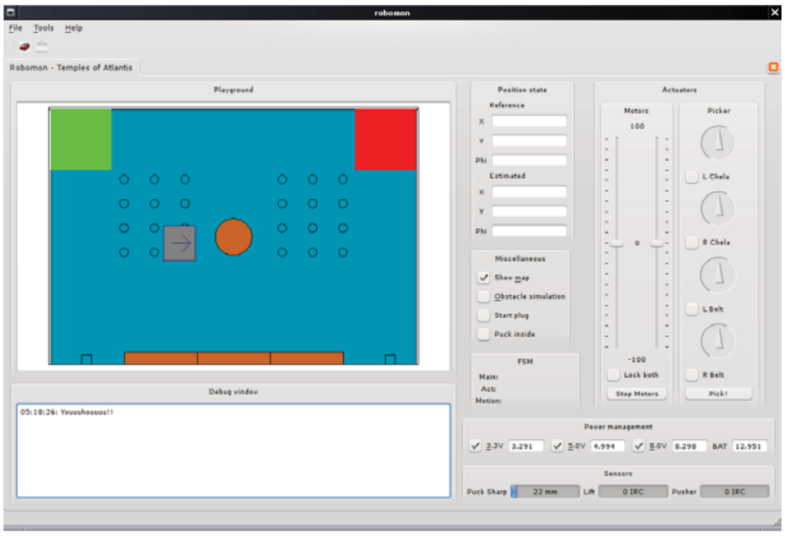
The FSM architecture introduces a higher level of abstraction, when creating control algorithms for the robot. The programmer does not have to take of low-level infrastructure of the code, like timers, inter-process communication and so on, since all these tasks are solved by the FSM library.

2.3.3 Visualization Application

The fact, that all the data from the sensoric subsystems of the robot are published to the ORTE layer allowed us to create a graphical application, which can visualize the robot state information in a convenient graphical representation. This application is denoted as *robomon* in Figure 2.4. The application can be run on a remote host computer, connected to the robot through the Wi-Fi connection.

The application was written in C++ language, using the Qt user interface framework. A screenshot of the application is in Figure 2.5. The application not only allows us to see all the sensors readings from the robot, it also shows the position of the robot on the playground. Moreover, it is possible to fully control the robot using the application.

When developing some algorithms involving movement of the robot on the playground, it is possible to use this application as a simulation platform. Thus we can tune the algorithms without using the actual robot and only then perform testing using the real hardware.



05:18:26: Youuuhouuuu!!

[✓] Show map

[✓] 3.3V 3.291 [✓] 5.0V 4.994 [✓] 8.0V 8.298 BAT 12.951

Puck Sharp 22 mm Lift 0 IRC Pusher 0 IRC

Figure 2.5: Robot visualization application

Chapter 3

Timinig Analysis on GNU/Linux and MPC5200 Target

This chapter gives a detailed analysis of possibilities of execution time measurements on a PowerPC based target system with GNU/Linux as an operating system.

Timing analysis is a process, which has to be always tailored for the given application to be measured. Each target platform has its own specifics, to which the process has to be adapted.

This thesis deals with measuring execution times on a PowerPC processor-based embedded board, which is running GNU/Linux operating system. The execution time measurement process developed in this work can however be generalized to certain degree to be used on a different target platform with Linux operating system.

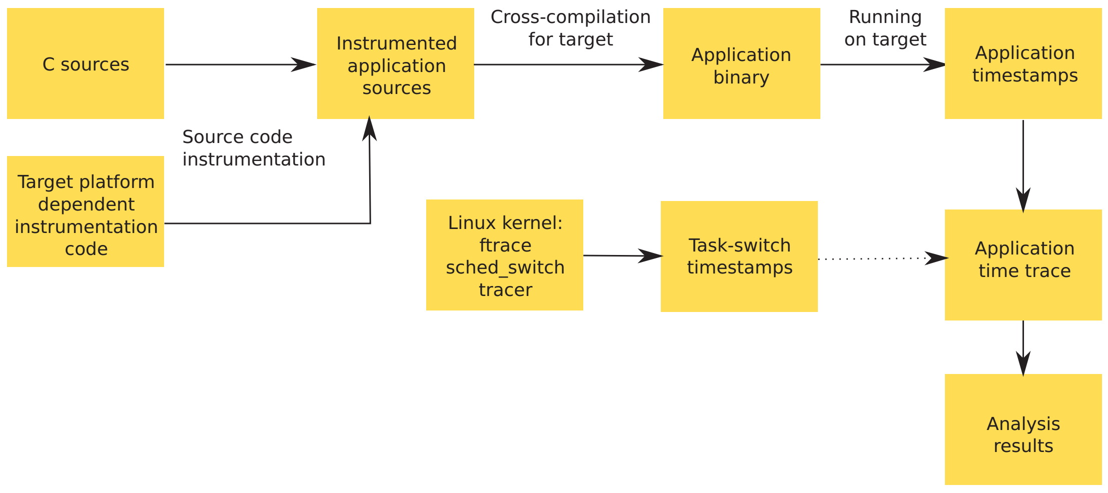

C sources
Instrumented application sources
Cross-compilation for target
Application binary
Running on target
Application timestamps
Source code instrumentation
Target platform dependent instrumentation code
Linux kernel: ftrace sched_switch tracer
Task-switch timestamps
Application time trace
Analysis results

Figure 3.1: Timing analysis process

3.1 Target Board

The target board used for practical application during work on this thesis was based on a processor MPC5200B by Freescale, running at 396 MHz. This processor has 32-bit PowerPC architecture and can be considered as a widely accepted standard on market with embedded system nowadays. This is probably caused by its high calculating power, low power consumption and effective architecture of the processors. The processor module used for this work can be seen in Figure 3.2. It is a SHARK module from MIDAM Control System series manufactured by the czech company Mikroklima, s.r.o.

Figure 3.2: Processor module MIDAM Shark

The carrier board had 128 MB of SDRAM memory and 64 MB of Flash memory. As a bootloader was used U-Boot. Custom GNU/Linux kernel version 2.6.29 (vanilla) was compiled for the board.

3.2 Execution-Time Profiles

When describing a timing behaviour of an application, it it also necessary to establish some common way of representing the results of the analysis. Execution-time profiles are mostly used for this purpose.

Execution-time profiles represent a mean to describe the probability distribution of different execution times of the code measured. This is usually achieved by taking histograms of execution time observed and dividing those by the overall number of measurements observed, effectively norming the distribution to 1 (Petters, 2007).

The representation of ETPs used in this work has observed execution times on the X-axis and complementary cumulative probability mass function on the Y-axis. A sample of such representation is in Figure 3.3.

This representation provides best visibility of the measured values for visual inspection. The interpretation of the graph can be as follows: what is the probability, that the execution time of the function will exceed a given time? On the sample ETP shown below, we can thus conclude, that the probability of the execution time exceeding 5 μs is almost one. This means, that almost all the measured execution times exceeded this value. On the other hand, we can also read from the ETP, that the longest execution time observed was slightly below 15 μs.

Also note, that a logarithmic scale is used on the Y-axis.

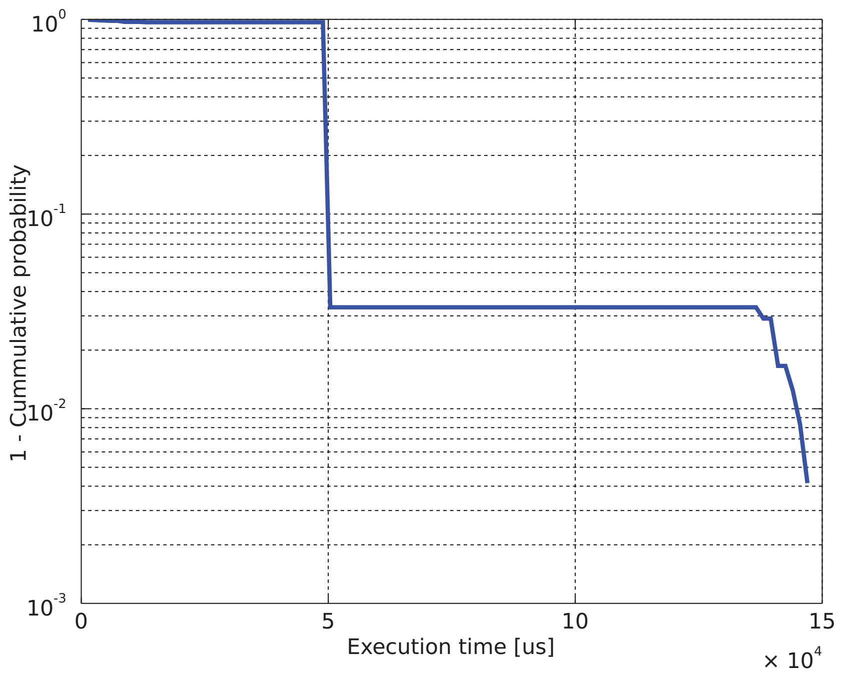


Figure 3.3: Sample execution-time profile

3.3 Measuring Application Execution Times

Since we are concerned in measurement of execution times of individual functions in an application, the only way how to accomplish this is to record timestamps together with unique identifiers on entry and leave points of the measured functions. The data structure containing a timestamp and an identifier will be called *instrumentation point* or *ipont* in further text. There are several possibilities how to do the actual recording of instrumentation points.

3.3.1 On-target buffer

One possibility is to record the timestamps together with identifiers to a buffer directly on the target system. This would be typically operating memory of the system. This method does not impose additional requirements on technical equipment necessary to carry out the measurement. However operating memory is usually limited in embedded systems, which means that only limited number of timestamps can be recorded.

This approach can be further divided into two possible cases. One possibility is to extract a single trace at the end of each mesurement. Another option would be to collect the recorded data periodically in smaller amounts during the measurement on a host computer.

When collecting the recorded data as a single trace, amount of memory available to store the data during the measurement constitutes a fixed constraint on the length of the trace, that can be acquired. This method was chosen for this work.

3.3.2 Direct tracing

More advanced method is direct tracing, which basically means, that the instrumentation point data are sent out directly by setting an output port of the target system. Data can then be acquired using a suitable data logger. The data logger must be able to capture data on high speed and have sufficient storage capacity. Modern fast logical analyzers can fulfill these requirements.

When collecting a trace using this method, the size of the trace is not limited by amount of resources on the measured system. Nevertheless it is more demanding because of the expensive equipment needed to capture the data.

In this case only unique identifiers identifying location in code, from which the instrumentation routine has been called, is sent to the output port. The actual time measurement is done by the external device, using its own clock source. This has also an advantage in the fact, that the data capturing device can utilize more precise and/or stable clock source than the one available on the target board.

3.3.3 Instrumentation code

As was mentioned before, the on-target buffering method was chosen to be used for further work. This decision brings along the necessity to develop instrumentation function, which will do the actual timestamping of the code.

There are several considerations that must be taken into account when developing code for this purpose. The most important one is probably the requirement, that the timestamping function should be as fast as possible, so that it wouldn't affect the execution time of the application to be measured.

Another consideration is the way, in which the measured data are going to be processed to obtain meaningful results. The data can be processed either by a custom application, written to fit the measurement needs, or by an industrial-grade tool. This consideration affects in particular format of the captured trace – in other words the data that has to be actually recorded to the trace (e.g. unique identifiers or instruction pointer).

Since Linux operating system is used on the target system, there are at least two posibilities available of how to actually obtain the timestamps. One possibility is to create a custom *ipoint* function in assembly language for maximum efficiency, another is to exploit some of the standard library functions to read the system time, like `gettimeofday()` or even better `clock_gettime()`. Lets have a closer look at the options now.

The easier of the approaches described above is using standard library functions. When precise time measurements are considered, the only function usable for this purpose is `clock_gettime()`. The function `gettimeofday()` cannot be used, because it does not provide the required accuracy. The resolution of this function is μs and its purpose is to give the best guess at wall time and it can eventually go backwards. On the other hand `clock_gettime()` has nanosecond resolution on recent Linux kernel with high-resolution timers and the returned time is monotonic.

When writing instrumentation code for time measurement, it is important to identify a reliable clock source on the target platform. The PowerPC architecture offers the Time Base (TB) register for this purpose. This is a 64-bit register, which is incremented at a configurable frequency. The frequency of the timer is usually set by bootloader when an operating system is used on the target system. This is also true for the target platform used in this work. The `uboot` bootloader used on the board sets the update frequency of the TB register to 33 MHz. Since the register is 64-bit long, its reading on a 32-bit architecture requires a special approach, which is described later in section 4.1.1.2. This register is also used as a general clock-source by the Linux kernel.

The approach using a custom assembly-based function may be more complicated, but is supposed to return better results in terms of the function execution time. Thus it should introduce less overhead to the measured application.

An evaluation was done to compare the effectivity of the approaches descibed above. In this evaluation, execution times of the instrumentation functions written in C using library functions and written in assembly were compared. To measure the execution time, the algorithm described in listing 3.1 was used. The execution times of the instrumentation point function were calculated as differences of timestamps of two adjacent records in the buffer.

```
timing_library_initialization(); /*allocate buffer for timestamps*/

for(i=0; i<10000; i++) {
    /*save timestamp and id to buffer*/
    timing_instrumentation_point(1);
}

timing_library_finish(); /*print timestamps from buffer*/
```

Listing 3.1: Measuring instrumentation function execution time

The result of this measurement for instrumentation point based on the function `clock_gettime()` can be seen in ETP in Figure 3.4. The mean value is (0.929±0.005) μs.

The results of the execution time measurements for the instrumentation point function hand-written in assembly can be seen in ETP in Figure 3.5. The mean value is (0.344±0.007) μs.

As expected, the hand-written assembly code demonstrated better performance in this comparison. The reason for this fact is that in the assembly function, only actions done are reading of the TB register and storing this value to the buffer together with ID value. The `clock_gettime()` function, on the other hand, performs another operations over the TB value read. Also a userspace to kernel mode switch has to be done. Since the function returns a time value in a structure containing seconds and nanoseconds, conversion from the raw TB value has to be done. Also all the time values returned by the function represent a difference to one time-point in past (mostly boot time of the operating system, but this is not guaranteed on all architectures), an initial value of the

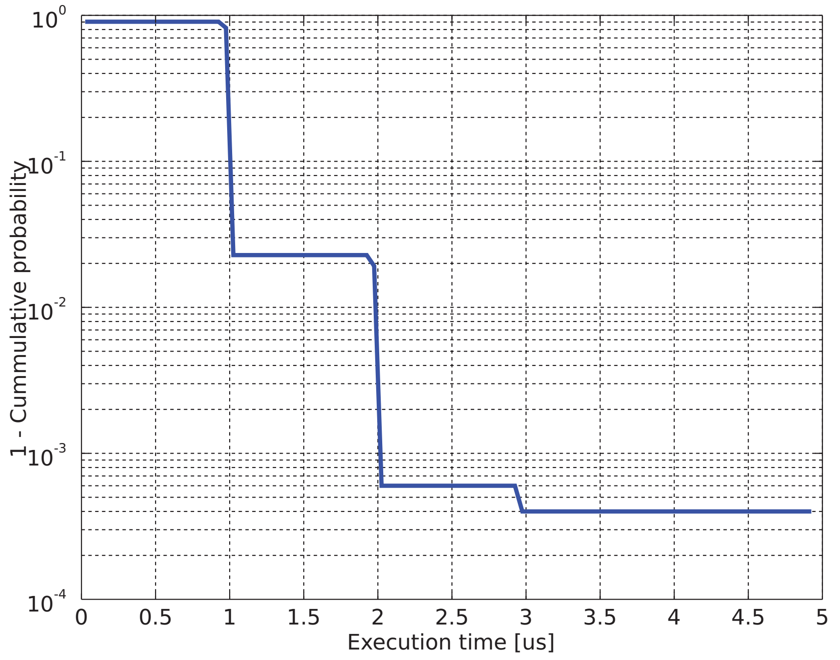


Figure 3.4: ETP for duration of instrumentation point based on function `clock_gettime()`

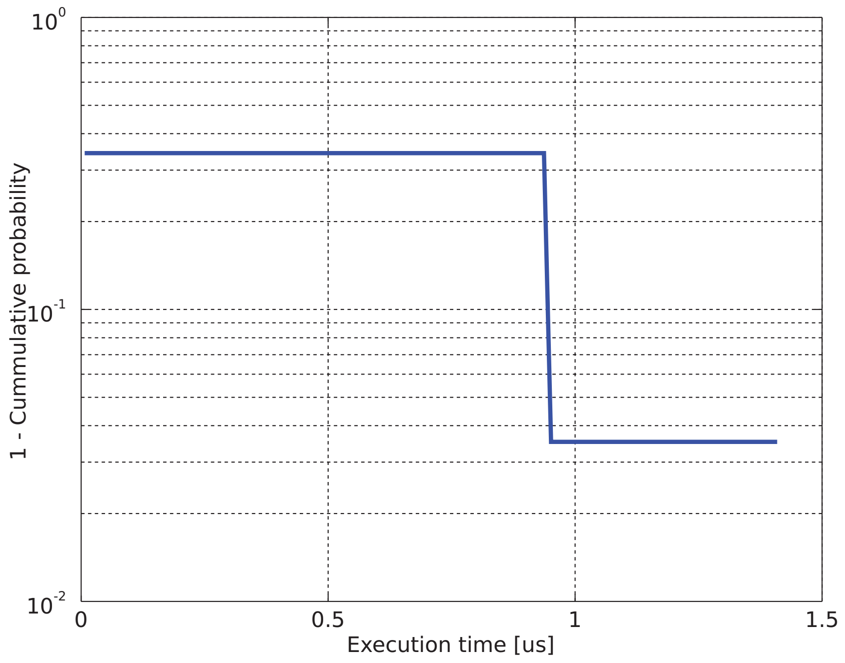


Figure 3.5: ETP for duration of instrumentation point based on ASM code

TB register has to be subtracted from the read value. More information on conversion of the raw TB value to a human-readable time value are presented in section 4.1.1.3. Another factor introducing further overhead to the function is the fact, that a mutex locking has to be performed inside the function, in order to guarantee its atomicity.

The instrumentation function written in assembly language does not perform any transformations of the read TB value. The only actions carried out in the function are reading of the time base register, saving it to the buffer together with identification mark and incrementing the buffer pointer. The transformations of the time values is done only after finish of the measurement, when the measured data are printed to the screen or to a file. Detailed description of the implementation of the assembly function is given in section 4.1.1.2.

3.4 Task Switch Tracing

3.4.1 Significance of Task Switch Tracing

Since the application to be measured is running in a multitasking environment (the operating system), its execution may be preempted in favor of another process, depending on the scheduling policy used by the operating system. Context switches must be taken into account when performing timing analysis on a multitasking system, since context switch can occur during execution of a measured function. This scenarion is ilustrated in Figure 3.6. The measured process (P1) starts its execution in time 0 and its duration is 7 cycles. However, at cycle 4 it gets preempted by another process (P2), which runs for 3 cycles. The first process is scheduled again and finishes its execution at cycle 10. Thus the measured execution time of the process is 10 cycles, although its real execution time is only 7 cycles.

When measuring execution time, we are usually interested in the actual time a particular function was executing, so time that was spent executing other tasks has to be subtracted from the measurement.

.

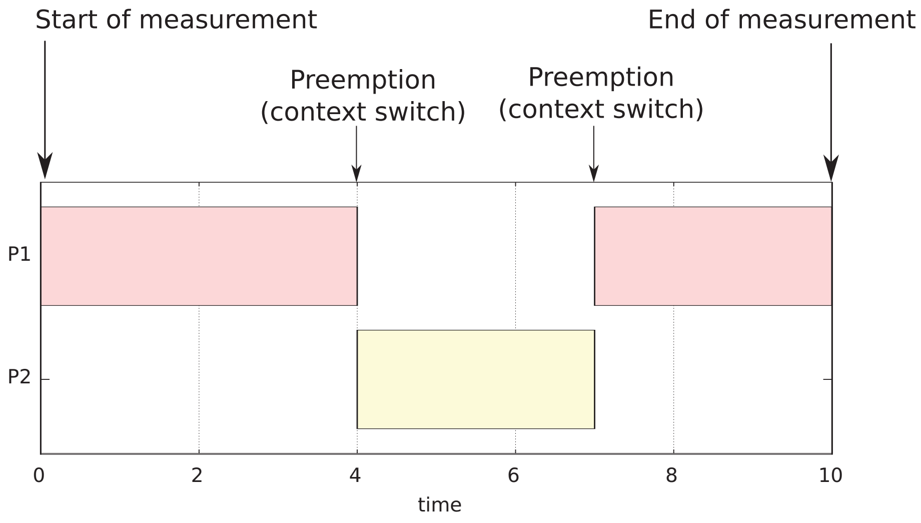


Figure 3.6: Impact of context switch on execution time measurement

3.4.2 Possibilities of Task Switch Tracing on Linux

Task switch tracing in a multitasking environment basically means saving a timestamp whenever the task under measurement is preempted by another task in a multitasking environment. Operating systems have a scheduling function, which takes care about scheduling active tasks in the system.

Scheduling function of the Linux operating system can be found in the source code file `kernel/sched.c`, it is called `schedule()`. A hook has to be added to this function in order to record timestamp everytime a task-switch happens, together with identifiers of the task, between which is being switched.

There are several approaches possible to solve this problem. The most obvious one is to create a custom timestamping function and add a call to it to the scheduler function. This approach is however also the most laborous one, because modifying core function of the operating system kernel has to be carried out with extreme caution.

Another possibility is to exploit some of the tracing functions that are already included in the Linux kernel. Recent version of the kernel includes so called Function Tracer - *ftrace*. The *ftrace* serves not only for tracing kernel function, as the name suggests, but it also contains infrastructure that allows for other types of tracing. Among others a tracer allowing context switch tracing is included.

Very interesting feature offered by the function tracer is *dynamic ftrace*. When this option is enabled in the kernel configuration, the system will run with virtually no overhead when function tracing is disabled. The actual tracing code is replaced by `nop` instructions on system start-up. When tracing is enabled the nops are patched back to calls.

The ftrace functionality does not require any special user-space tools. Special filesystem, called *debugfs*, is used to control the tracer and to obtain the recorded trace from the kernel.

3.4.2.1 Basic usage of ftrace

Once Linux kernel has been compiled with support for *ftrace*, the *debugfs* file system can be mounted using following command:

```
mount -t debugfs nodev /mount_point
```

The mount-point is usually `/debug` or `/sys/kernel/debug`. We will use `/debug` further on for simplicity.

List of tracers available can be retrieved by calling:

```
cat /debug/tracing/available_tracers
```

The output should look similar to the following one:

```
~$ cat /debug/tracing/available_tracers
function sched_switch nop
```

The `sched_switch` tracer should be present in the listing. If it is not, check that the kernel has been compiled with configuration option `CONFIG_CONTEXT_SWITCH_TRACER=y` set.

The tracing process can be controlled using files `/debug/tracing/current_tracer` and `/debug/tracing/tracing_enabled`. Tracing of context switches can be turned on using following commands:

```
echo sched_switch > /debug/tracing/current_tracer
echo 1 > /debug/tracing/tracing_enabled
```

To stop the tracing, simply issue:

```
echo 0 > /debug/tracing/tracing_enabled
```

The recorded trace can printed in the terminal by calling:

```
cat /debug/tracing/trace
```

The resulting trace has following format illustrated in the following example.

```
tracer: sched_switch

      TASK-PID   CPU#    TIMESTAMP  FUNCTION
         | |      |          |         |
     bash-3997  [01]   240.132281:   3997:120:R   +  4055:120:R
     bash-3997  [01]   240.132284:   3997:120:R ==>  4055:120:R
    sleep-4055  [01]   240.132371:   4055:120:S ==>  3997:120:R
     bash-3997  [01]   240.132454:   3997:120:R   +  4055:120:S
     bash-3997  [01]   240.132457:   3997:120:R ==>  4055:120:R
    sleep-4055  [01]   240.132460:   4055:120:D ==>  3997:120:R
[...]
```

The first column of the trace contains the name and PID of the task running, when the record has been made. The following column contains identifier of the CPU the task was running on (this information is meaningful only on SMP systems).

Timestamp of the event is shown in the third column of the trace. The function tracer uses *ring buffer* to record the events. Each record in the ring buffer has a timestamp, which is obtained by calling the function `sched_clock()` from the linux kernel. This function is used as a high-resolution time source by the linux scheduler. It is important to note at this point, that the `sched_clock()` function uses the *Time Base* (TB) register as the clock source.

The last three columns contain information about the tasks it's being switched between. The first of these columns indicates the task running when the context switch happened, the last column indicated the task to which the scheduler switched. The signs between these two columns indicate, whether a context switch ('`==>`') or a wake-up ('`+`') of the task happened. The information about the current and next task have the following format: `PID:KERNEL-PRIO:STATE`.

PID is the process identifier of a task. It's important to realize, that in Linux threads are actually treated as child processes on the kernel level, thus each thread has its own unique PID.

The kernel priority shown in the log is the inverse of the actual priority with zero (0) being the highest priority.

Possible task states are:

```
R - running : wants to run, may not actually be running
S - sleep   : process is waiting to be woken up (handles signals)
D - disk sleep (uninterruptible sleep) : process must be woken up
                                        (ignores signals)
T - stopped : process suspended
t - traced  : process is being traced (with something like gdb)
Z - zombie  : process waiting to be cleaned up
X - unknown
```

Task states `running` and `sleep` are mostly to be encountered when actually performing a measurement.

The *ftrace* tracer offers some more configuration options available through the *debugfs* file system. Of these only one of concern when performing the measurements described in this work is the one for configuration of the ring buffer used to record the events. The ring buffer must be set to be large enough to hold the complete trace since the beginning of a measurement. If the end of the buffer is reached in course of tracing, the records at the beginning of the buffer get overwritten by new ones. The size of the ring buffer can be controled through the file `/debug/tracing/buffer_size_kb`. Desired size of the buffer in kilobytes can be written to this file. The memory for the buffer is statically allocated.

Required size of memory for the ring buffer of the tracer depends on the measured application (how often it causes context switches) and other applications running on the system, as well as on the duration of the measurement.

3.5 RapiTime

There are some industrial-grade tools available in the market. These tools are usually designed for full featured WCET analysis. An example of such tool can be RapiTime by Rapita Systems Ltd. RapiTime was evaluated to be used for evaluation of the data measured in this work, thanks to courtesy of Rapita Systems Ltd.

This tool-suite offers complete WCET analysis of embedded code, combining both static and dynamic analysis. It consists of several applications, covering all steps

RapiTime consists of several tools, which are used in the process of timing analysis. First a tool called `cins` is used for instrumentation of the code to be measured. The instrumentation is done in an automatic manner and the tool offers fine-grained control over the required level of details of measurement. Based on configuration provided by annotations placed to the source code, `cins` places instrumentation points automatically to preprocessed sources. Apart from that, `cins` also prepares information on structure of the code used in another step of the analysis. The source code is then built and linked with library containing the instrumentation code.

Once the data on structure of the code is ready, structural analysis is done by another application in the chain, `xstutils`.

The compiled executable is then run on the target system and a time trace is collected. The measured data can then be transformed into desired form using `traceutils` utility. This tool can also merge more trace files into one resulting trace.

The final step is to generate a report using `traceparser`. The report can then be viewed in RapiTime Viewer, which is a plugin for the Eclipse platform. The report contains detailed information on the timing behavior of the application, resulting from both structural analysis of the code and actual measured values. The timing data are directly realted to individual functions and their contribution to the total WCET is indicated. This helps to identify bottlenecks in performance of the measured code.

More information about RapiTime can be found for example in (Rapita Systems Limited, n.d.).

Although RapiTime is very professional tool offering many possibilites and data in terms of timing behavior of an application, it was not possible to use it for the work in the end. So far the tool has never been used to analyse an application on top of a full-blown operating system, like GNU/Linux. However it was possible to make it analyse some simple sample applications. The trouble nevertheless is, that the source code parser and structural analysis tool are not ready to work with source code, part of which has been

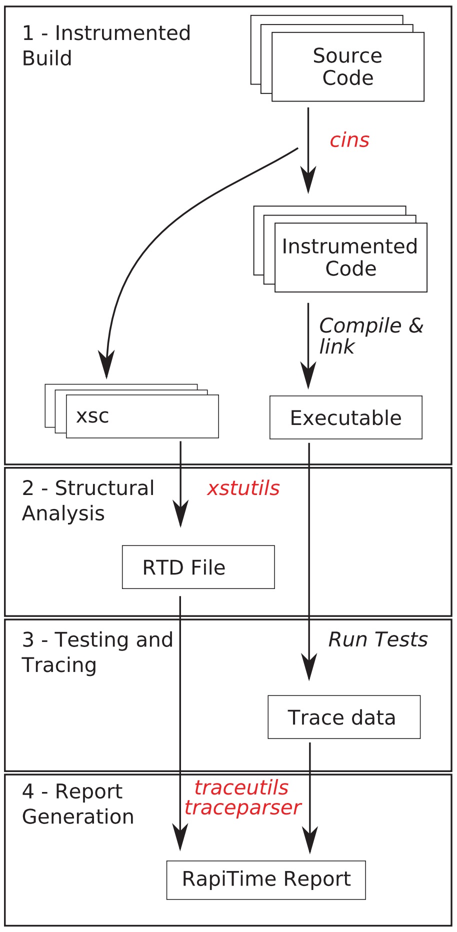


Figure 3.7: RapiTime workflow

written in C++. The tools are designed for analysis of C source codes, since C language is still a widely accepted standard in embedded applications development, especially because of its effectivity on systems with limited resources. After research done by engineers at Rapita Systems it was found out, that adding support for C++ code to their tools would be very complicated, if not impossible. Since part of the robotic software that was to be analysed in this work was written in C++, it was not possible to use RapiTime for the analysis and another approach had to be taken.

Chapter 4

Implementation and measurement results

This chapter demostrates how the measurement methods developed as a part of this work and described in previous sections can be applied to a real-world application. Selected parts of the software for the mobile robot described in chapter 2 were analysed.

4.1 Timing Analysis Library

A software library was created as a result of research on posibilities of timing analysis on the given target system. The library contains functions for recording of timestamp for both userspace applications and kernel task switches. The timestamps can be printed either on screen or to a file for later analysis using scripts developed as a part of the library.

The library is configurable during compile-time. It uses OMK make system (OCERA, OMK, n.d.) to manage the build process, so it is fully compatible with the robotic software being analysed.

The API functions visible to the applications linked with the library is listed in Listing 4.1.

4.1.1 Library configuration and interface

The library is fully configurable using compile-time options specified in Makefiles. Follows a description of the library application interface. Each function is accompanied by a list of configuration parameters it is affected by.

```
timing_init(void);
timing_ipoint(uint16_t id);
timing_set_pid(uint16_t id, pid_t pid);
timing_output_trace(void);
timing_finish(void);
```

Listing 4.1: Timing analysis library API

4.1.1.1 Function timing_init()

This function shall initialize the timestamping functionality in both userspace and kernel (task switch tracing). For the userspace applications tracing, a propper instrumentation point type is set based on the configuration given at compile time. Available ipoint types are listed in table 4.1.

The task-switch tracing is also turned on at this point. First the debugfs filesystem is mounted. Then the tracer `sched_switch` is set as the current tracer and finally the tracing is turned on.

4.1.1.2 Function timing_ipoint()

This is the core function of the timing analysis library. It performs the actual timestamping of the point in an application, where it is called from. As was already described in section 3.3.3, two ways of writing the instrumentation code were tested and evaluated. Although the assembly hand-written code showed better performance and was selected as the main method for recording of timestamps during real measurements, the C code based on the function `clock_gettime()` was left in the library for comparison and reference purposes. User can select, which of the two versions of the instrumentation code will be actually used by the library using build-time configuration parameters listed in the Table 4.1.

Lets have a more detailed look on the actual implementation of the instrumentation code in both versions, as it has some specifics, which should not be missed when designing

Table 4.1: Configuration parameters specifying version of the instrumentation point code

Configuration parameter	*Effect*
`CONFIG_TIMING_IPOINT_ASM`	the hand-written ASM function is used as the instrumentation point
`CONFIG_TIMING_IPOINT_CLOCK_GETTIME`	the function `clock_gettime()` is used to obtain the timestamps

a function for this purpose.

The version using standard library functions and written completely in C language shall be presented first, as it is easier to understand and follows basically the same principles like the assembly version. The code is presented in Listing 4.2.

```
static inline void timing_ipoint_clock_gettime(uint16_t id)
{
        timestamp_t *p;

        if(!timestamp_enabled)
            return;

        pthread_mutex_lock(&mtx);
        p = timestamp_ptr;
        timestamp_ptr++;
        pthread_mutex_unlock(&mtx);

        if (p >= timestamp_buff_end) {
                p = timestamp_handle_overflow();
                if (!p) {
                        return;
                }
        }
        p->id = id;
        clock_gettime(CLOCK_RT, &(p->ts));
}
```

Listing 4.2: Ipoint function using `clock_gettime()`

There are several things to be pointed out about the implementation. There is a global array of type `timestamp_t` in the library, which functions as the buffer to store measured timestamps. Another global variable, `timestamp_t *timestamp_ptr`, is used as an index to the buffer. In the instrumentation function, the address stored in the index variable is first copied to a local variable and the index is incremented immediately. These two actions are done inside a section guarded by a mutex. This construct is used to prevent corruption of the recorded data that could result from concurrent access to the buffer from two points in the measured application – in other words, this construct secures the reentrancy of the instrumentation point function. Since the index is copied to a local variable and incremented, even in case of context switch happening in the middle of execution of the instrumentation function, a valid pointer to the timestamps array is retained after switching back to the original task. If another task calls the instrumentation code during the context switch, already incremented index to the array will be used in this task.

```
static inline void timing_ipoint_asm(uint16_t id)
{
        timestamp_t *p, *pinc;
        uint32_t tbu, tb, tbu2;

        if(!timestamp_enabled)
                return;
        /* Store current TB */
        asm volatile (
                "1:\n"
                "       mftbu %0\n"
                "       mftb  %1\n"
                "       mftbu %2\n"
                "       cmpw  0,%0,%2\n"
                "       bne-  1b\n"
                : "=r" (tbu), "=r" (tb), "=r" (tbu2) /* Outputs */
                : /* No inputs */
                : "cr0" ); /* CR0 changes */

        asm volatile (
                "1:     lwarx   %0,0,%4\n"
                "       addi    %1,%0,%3\n"
                "       stwcx.  %1,0,%4\n"
                "       bne-    1b\n"
                : "=&b" (p), "=&b" (pinc), "=m" (timestamp_ptr)
26                  : "n" (sizeof(timestamp_t)), "b" (&timestamp_ptr)
27                  : "cr0");
28
29          if (p >= timestamp_buff_end) {
30                  p = timestamp_handle_overflow();
31                  if (!p)
32                          return;
33          }
34
35          p->id = id;
36          p->tbu = tbu;
37          p->tbl = tb;
38  }
```

Listing 4.3: Ipoint function handwritten in assembly

The assembly code has basically the same structure, as the code written in C presented earlier in this section. The first block of assembly code (lines 9 - 18) serves for reading of the TB register. Since the register is 64-bit long, it can't be read in one instruction on a 32-bit architecture. The register actually consists of two 32-bit registers: *TBU* and *TBL*, which contain the upper, respectively the lower half of the 64-bit value. On line 11, the *TBU* register is read, on line 12 the *TBL* is read. Because of possibility of a carry from *TBL* to *TBU* occuring between reads of the TBU and *TBL*, the integrity of the read value has to be guaranteed. For this reason, the *TBU* value is read once more and compared with the one stored in the first reading. If the values don't match, the whole procedure is repeated.

The second block of assembly code (starting at line 20) does atomic incrementation of the pointer to the next free position in the buffer. The pair of PowerPC instructions `lwarx` / `stwcx` is used to emulate read-modify-write operation to specified memory location. If the store is performed, the use of the `lwarx` and `stwcx` instructions ensures that no other processor or mechanism has modified the target memory location between the time the `lwarx` instruction is executed and the time the `stwcx` instruction completes. This is a analogous to the mutex mechanism used in the previously described function.

Both implementations of the instrumentation function have the same mechanism to prevent overflow of the buffer, where the timestamps are stored. If the index to the buffer points outside of the buffer after it is incremented, no write operation is further performed to the memory and a flag indicating buffer overrun (`timestamp_enabled`) is set.

4.1.1.3 Function timing_output_trace()

When this function is called, the collected trace data are either printed to the terminal, or saved to a text file.

Since the timestamps recorded from the measured application are represented by a raw timebase register value and the timestamps printed by the *ftrace* context switch tracer are floating point time values with microseconds resolution, it is necessary to convert the TB values to the same format first. Otherwise further processing of the traces would not be possible.

For this reason, all the operations the Linux kernel does with the TB value before it is presented as the *ftrace* output had to be researched from the kernel source code files, because the Linux kernel does not provide any documentation on this issue.

As was mentioned before in the section describing the *ftrace* operation (3.4.2.1), the source of the time information for *ftrace* is the kernel scheduler clock function `sched_clock()`. Lets have a look how this function works with the information read from the timebase register. The implementation of the function can be found in kernel sources in file `arch/powerpc/kernel/time.c` and is listed in Listing 4.4.

```
/*
 * Scheduler clock - returns current time in nanosec units.
 *
 * Note: mulhdu(a, b) (multiply high double unsigned) returns
 * the high 64 bits of a * b, i.e. (a * b) >> 64, where a and b
 * are 64-bit unsigned numbers.
 */
unsigned long long sched_clock(void)
{
        if (__USE_RTC())
                return get_rtc();
        return mulhdu(get_tb() - boot_tb, tb_to_ns_scale) << tb_to_ns_shift;
}
```

Listing 4.4: Linux scheduler clock function

The function `get_tb()` reads the raw value of both parts of the TB register and returns the resulting 64-bit value. Then the value `boot_tb` is subtracted. This value is stored by the Linux kernel during system boot phase, in function `time_init()`. It represents the TB register value at system start. The resulting value after subtraction is multiplied by the timebase scale factor `tb_to_ns_scale`. This factor is found out by the kernel during time subsystem initialization and it reflects the frequency, on which the TB register is updated.

The function `mulhdu()`, which does the actual multiplication, is implemented as an assembly stub in the kernel header files and is not accessible form the userspace. Since there is no straightforward way of multiplication of two 64-bit integers on a 32-bit architecture, a function emulating the calculation had to be written in C. It is listed in Listing 4.5.

```
uint64_t mulhdu(uint64_t a, uint64_t b)
{
        uint64_t ret, ah, al, bh, bl, carry, c1, c2, c3;

        ah = a>>32;
        al = a & 0xffffffff;
        bh = b>>32;
        bl = b & 0xffffffff;

        c1 = al*bl;
        c2 = (ah*bl)<<32;
        c3 = (al*bh)<<32;

        carry = 0;
        if (~c1 < c2) carry++;
        c1 += c2;
        if (~c1 < c3) carry++;

        ret = ah*bh + (ah*bl>>32) + (al*bh>>32) + carry;

        return ret;
}
```

Listing 4.5: Implementation of `mulhdu` function on a 32-bit architecture

The result of the multiplication is then shifted to get time value in nanoseconds.

The timing library has to find out values of the constants used by the `sched_clock()` function in order to be able to reproduce the calculations done by the function. The value `boot_tb` is unique each time a system is started. Since the Linux kernel does not provide any method of reading this value from userspace, a patch had to be created making the kernel function `time_init()` print this value during system boot phase. The value can then be read by the timing library from the system log printed using the `dmesg` command from the Linux shell.

The values `tb_to_ns_scale` and `tb_to_ns_shift` are constant for a given target system. These values are printed to the system log by the kernel and thus can be also read using the `dmesg` command.

Now that we have scheduler clock value calculated from the TB value, we have to convert it to the representation used by the *ftrace* tracer, when printing the context switch trace. As was already said, the tracer prints the timestamp as a floating point value in seconds, with microseconds resolution. The details on how `ftrace` solves this task can be found in kernel source file `kernel/trace/trace.c` in implementation of function `print_trace_fmt()`. The scheduler clock timestamp value is divided by 1000000 to get the number of seconds and the remainder of the division is used as the microseconds part of the final time value.

All the operations described in this section are performed on the TB values recorded during the measurement for the measured application in userspace. The resulting trace is then printed to a terminal or to a file, depending on the configuration of the library specified at compile time.

4.1.1.4 Function timing_setpid()

When this function is called before beginning of the measurement, the PID supplied to it as an argument will be printed in the resulting trace, to enable distinction between timestamps recorded from distinct threads.

4.1.1.5 Function timing_finish()

When this function is called, the tracing process is ended. The kernel context switch tracer is stopped and the instrumentation function is replaced by a function printing warning to the screen, if an instrumentation point is called after this function has been invoked. This function should be called the exit point of the measured application, before the `timing_output_trace()` function is called.

4.2 Processing of the measured values

To process the traces resulting from the measurements done with the timing library, a set of scripts mostly in *awk* language was created. Now follows a brief description of the data processing procedure.

As a result of time measurement using the described library, two data files (if the output of the library was configured so) are created in the working directory of the measured program. One of these files contains the trace collected from the application, while the other one contains the output from the *ftrace* context switch tracer.

First step is to transform the *ftrace* output to the same format, that is used by the application tracer. Once this is done, the two traces are merged and the records are sorted according to timestamps.

When the sorted trace of the whole system is available, the actual execution times of the measured blocks of code are calculated. Possible task switched are taken into account when calculating the total execution time. The time that was spent executing different task, than the one being measured, is subtracted from the execution time.

Another step is to produce a file containing only the execution times measured. This file is then used as an input to a script written in MATLAB, which produces the resulting ETP from the data.

4.3 Measurement Results

This section contains the actual measured execution-time profiles of selected functions from the robot control software.

The selected functions are the ones, that are supposed to consume a large portion of the total execution time of the whole application. Also since the functions are directly involved in controlling the movement of the robot, they have quite stringent requirements regarding the timing behaviour.

Apart from the motion control functions, measurements of delays on ORTE layer were carried out.

4.3.1 Motion Control

The results presented in this section were acquired from approximately 10 seconds long traces.

4.3.1.1 Function `do_control()`

This function is running in the motion control thread of the FSM control application. It is responsible for setting the trajectory of the robot movement while avoiding obsatcles, which might be present in the area in front of the robot.

The measured execution-time profile of the function is in Figure 4.1. The mean value of the measured execution time of the function was $(278.078 \pm 2.046)\mu s$ (the value is represented as $mean_value \pm$standard_devation).

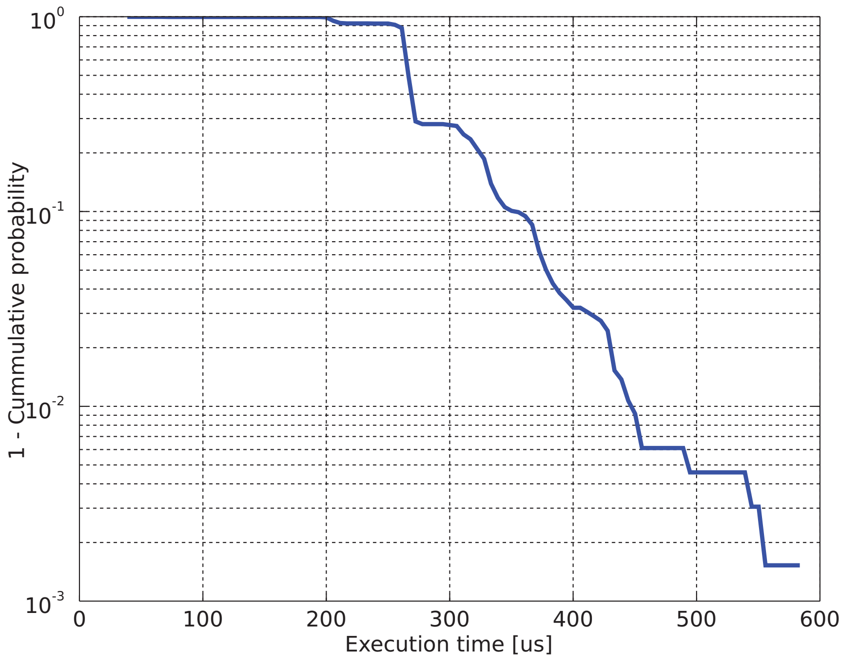


Figure 4.1: ETP of function `do_control()`

4.3.1.2 Function `do_estimation()`

This function calculates the estimation of the robot position on the playground from the data from the ultrasound system and from encoders on the wheels (odometry). The calculation is based on 8^{th}order Kalman filtration. The measured mean value was $(671.478 \pm 3.183)\mu s$.

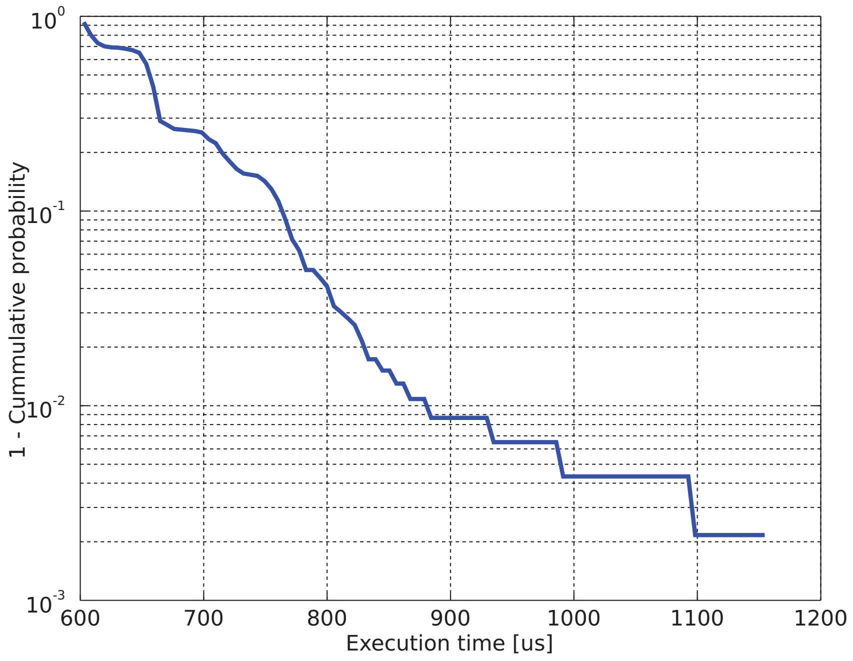


Figure 4.2: ETP of function `do_estimation()`

4.3.2 Function `new_goal()`

The function `new_goal()` calculates and simplifies a path to goal position avoiding obstacles. The measured mean execution time was (2.244±0.940)ms.

4.3.3 Delay Introduced by ORTE

We found it also interesting to measure delays between a data packet reception on the CAN bus by *cand* application (see section 2.3.1) and the time it is actually received by an application from the ORTE layer (see section 2.2).

Since each message transmitted over ORTE contains timestamps saying when it was issued by a publisher and when it was received by a subscriber, it was not necessary to use the timing library for this measurement.

The results of the measurement are presented as an ETP in Figure 4.4. The mean value of delay measured was (1.458±0.01)ms.

During analysis of the results of this measurement, there was found no correlation between the length of the data message transfered throught the ORTE layer and the measured delay.

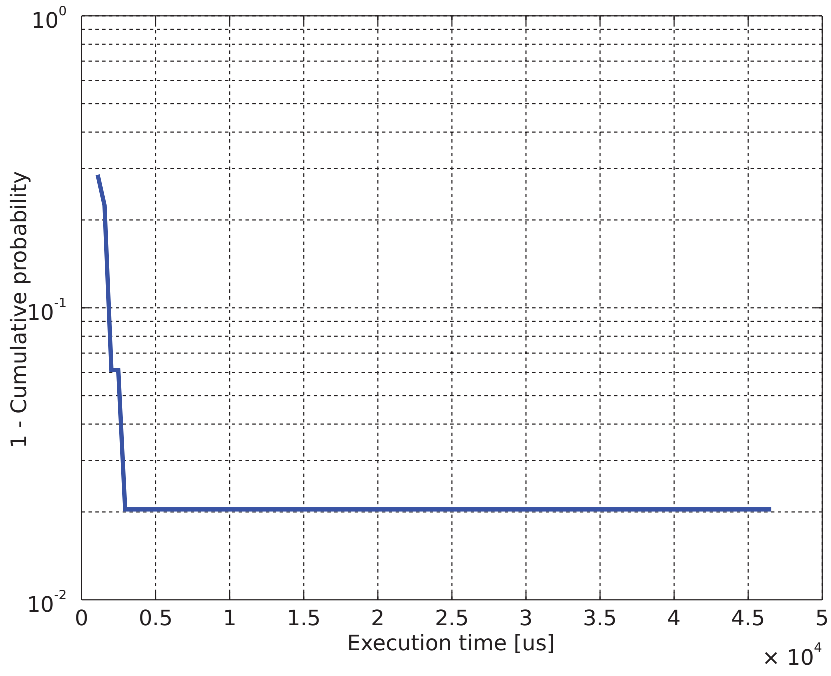


Figure 4.3: ETP of function `new_goal()`

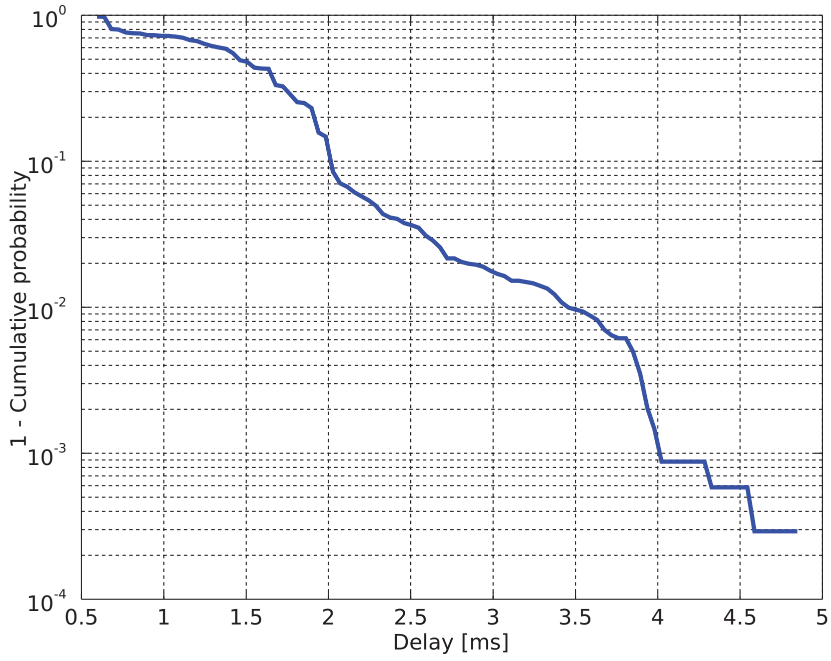


Figure 4.4: ETP of delays introduced by ORTE layer

Chapter 5

Matlab Generated Code Analysis

This section describes work done as a part of deliverable (Žídek et al., 2008) for the FRESCOR project (FRESCOR project, n.d.). The section is included because it is related to the timing analysis problems and might of interest to people involved in the issue.

5.1 Motivation

Matlab Simulink allows engineers to develop a control application algorithm in high level graphical language of data-flow diagrams. It contains a tool called Real Time Workshop (RTW) which can be then used to automatically generate C code of the Simulink diagrams. Together with development tools for the target platform this tool set provides a basis for rapid application development (RAD).

Automatically generated code allows a seamless development process where a non-programmer (e.g. control engineer) or a team designs and implements the entire system. Therefore the designer focuses on the controlled object from the beginning to the end of the development and the implementation issues such as the drivers, the programming language, the scheduling policy and the other nonfunctional aspects remain in background. All the implementation issues are covered by the *code generator target* developed by the hardware and real-time specialists as a support for the control engineers work (Bartosinski et al., 2007).

The quality of the generated code is comparable to the hand-written code, it is readable, the development time is shorter and possible error sources are reduced. The C code for a rapid prototyping is generated by the tool Real-Time Workshop (The MathWorks, Inc., n.d.). The add-on RTW Embedded Coder (The MathWorks, Inc., 2005) can be used for the highly optimized production quality code. The targets for RTW and RTW Embedded Coder differ.

The rapid application development approach does not bring only the automatic code generation. It is a model based development method supported by a tool chain covering entire “V” model development chain. The validation of each development phase is done by the simulation in the Matlab Simulink. First “Model in the Loop” validates the model of the controller. After the code generation, the “Processor in the Loop” simulation can be used to validate the real-time execution of the controller on the target hardware in the loop with the plant model in Simulink. Then the “Hardware in the Loop” simulation can be used to validate the entire control unit. All these phases can be supported by Simulink and the corresponding code generator target. The results of each experiment are used to continuous improvement of the Simulink model. Therefore the model is still synchronized with the code and can also act as documentation. Contrary to the hand-written code, there is no gap between the model and the implementation.

For proper execution of the automatically generated application it is necessary to ensure there is enough resources for the application to meet its deadlines determined by Simulink model’s sampling rate.

This case study was carried out using the RapiTime WCET analysis tool (see 3.5).

5.2 Introduction

Hardware-In-the-Loop (HIL) simulation is one of the testing methods used while designing control systems. It is based on simulation of the system to be controlled. This is realized by computer running the mathematical model of the system. Sensors and actuators similar to the real system are used and connected to the computer so that the control system can be employed in the same way as in the real system.

We have taken a real airplane model which was used for testing of aircraft control systems. Part of this model is used in this case study. It is a transfer function between rudder deflection (radians) and aircraft yaw rate (radians per second). This model is used for testing of the aircraft side vibration dumper. It is 5th order linear time invariant (LTI) discrete system with single input and output (SISO) sampled by 40 ms period. The

Simulink model used for the case study can be seen in Figure 5.1. The output of the system with rectangular pulses on input is in Figure 5.2.

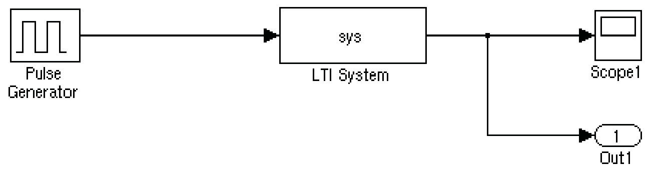


Figure 5.1: The Matlab/Simulink model used for the case study

To generate source code from the model in Simulink, Real-Time Workshop (The MathWorks, Inc., n.d.), a product from The MathWorks company, was used. Real-Time Workshop generates stand-alone C code for developing and testing algorithms modeled in Simulink and Embedded MATLAB code. This allows even users without any deep knowledge of real-time programming aspects to create various kinds of applications. Having some means of checking a solution timing behaviour would be beneficial for users and would allow them to compare timing behaviour of different implementations of the same algorithm.

Each Simulink block is associated with a code template in TLC format (Target Language Compiler). Each block template implements a standard interface, making it possible to integrate blocks in an automated code generation process. Another part used for the final application generation is the template containing the `main()` function of the application, called system target file. This file always has to be customized for the target CPU architecture and operating system. We use a target created for GNU/Linux PowerPC systems.

Since Matlab/Simulink models are, by their very nature, strictly periodic tasks with deadlines equal to their periods they are easy to model.

5.3 Code analysis

After exploring the set of source code generated from the Matlab/Simulink model, it was observed that the most important files are `model.c` and `linux_grt_target_main.c`. These files provides the API illustrated in Figure 5.3.

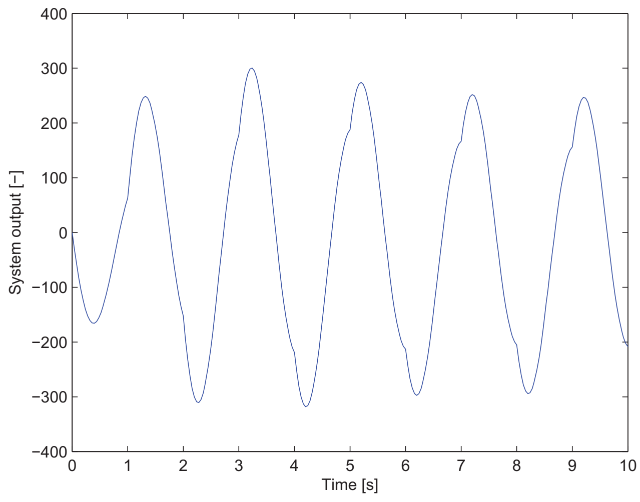
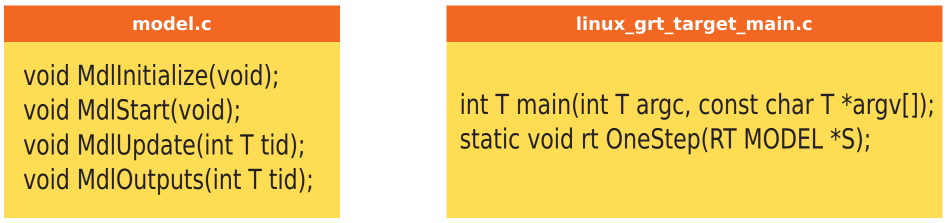


Figure 5.2: Output of the Matlab/Simulink model used for the case study

The `main()` function initializes the whole simulation and then a timer is set to call the `rt_OneStep()` function with period given by the model sampling rate specified in the Simulink model. In the `rt_OneStep()` function update of internal model state is performed by calling the appropriate functions from `model.c` file.

When an automatic generation of source code comes into play, the main concern is always the structure and readability of the resultant code. In case of RealTime Workshop, the quality of the resultant code can be considered to be quite high, because the code structure is logical and can be understood by a reader.

The update function contains all the necessary calculations in each step and the model state is updated with the results. The function comprises of code snippets associated with function blocks placed in the Simulink diagram. As was mentioned before, the code snippets are specified in templates for each block using the TLC language. This makes



```
void MdlInitialize(void);
void MdlStart(void);
void MdlUpdate(int T tid);
void MdlOutputs(int T tid);
```

```
int T main(int T argc, const char T *argv[]);
static void rt OneStep(RT MODEL *S);
```

Figure 5.3: Interface of source files generated from Matlab/Simulink model

the creation of a timing model quite straightforward and painless process and will be demonstrated on the example described at the beginning of this chapter. The description is very brief as it does not deviate from the procedure described with the first case study.

First it is necessary to instrument the generated code with RapiTime after preprocessing them with standard compiler preprocessor. The instrumentation points can be placed to the `MdlUpdate()` function in `model.c`, because this function contains the actual calculations done in each step. It is also possible to instrument the `rt_OneStep()` to see, if the whole model updates at the desired (sampling) rate. file. Then the application is compiled and run on the target system. Once the timestamps are acquired and combined with information about task-switch times, RapiTime can be used to obtain the information about code execution times.

Chapter 6

Conclusions

The thesis described software for mobile autonomous robots, developed and used for the Eurobot robotic contest by members of the team CTU Dragons (CTU Dragons, n.d.) at the Department of Control Engineering, FEE, CTU in Prague.

Main focus of the work described in previous chapters was on timing analysis of software running on an embedded target board with GNU/Linux operating system. A method for precise time measurement on a system running on a microprocessor with PowerPC architecture was developed and implemented. The method consists of tracing execution times of individual functions of an application and tracing context switches between running tasks in a multi-tasking operating system. The measured timestamps from the application and context switch times are then combined together and execution times are calculated. The results are presented in form of exectution-time profiles. The results of the measurement show, that highest influence on the delays in the system has the ORTE communication layer.

The method for application execution time tracing established in this work can be used without modifications on any embedded system running on a PowerPC processor and using GNU/Linux operating system. All aspects of developing of such method were described in detail in the thesis, so its application on a different target system can be derived from the information presented hereinbefore.

A case study was carried out, exploring the possibilities of timing analysis and simulation of timing behavior of an application generated automatically from MATLAB/Simulink model.

The timing analysis library will be used at the Department of Control Engineering for further development of mobile robot control algorithms.

The VirtualTime models of the software were not described in the thesis. The possibility of modeling timing behavior of software using VirtualTime simulation tool from Rapita Systems Ltd. was evaluated in a report for project FRESCOR (Žídek et al., 2008), but the result of the evalution was, that it is not possible to create a reasonable model of such a complex system, as a mobile robot control software.

Bibliography

Bartosinski, R., Hanzálek, Z., Stružka, P. and Waszniowski, L. (2007), Integrated environment for embedded control systems design, *in* 'Proceedings of 21st International Parallel and Distributed Processing Symposium', IEEE, Piscataway, p. 147.

CTU Dragons (n.d.), 'CTU Dragons [online] [cit. 2009-05-20]', Available from `http://rtime.felk.cvut.cz/dragons`.

FRESCOR project (n.d.), 'Frescor project [online] [cit. 2009-05-20]', Available from `http://www.frescor.org`.

OCERA, OMK (n.d.), 'OCERA Make system [online] Last modified 2009-02-13 [cit. 2009-05-20]', Available from `http://rtime.felk.cvut.cz/omk`.

Petters, S. (2007), Execution-time profiles, Technical report, tech. rep., NICTA, NICTA, Sydney 2052, Australia.

Rapita Systems Limited (n.d.), 'RapiTime white paper', Rapita Part#: DOC-060118-1, available from `http://www.rapitasystems.com/system/files/RapiTime-WhitePaper.pdf`.

Schaefer, S., Scholz, B., Petters, S. and Heiser, G. (2006), 'Static analysis support for measurement-based WCET analysis', *Editors: Timothy Bourke and Stefan M. Petters Work-in-Progress-Chair: Liu Xiang, Peking University, China* p. 25.

Smolik, P., Pisa, P., Vacek, F., Sebek, Z., Krakora, J. and Hanzalek, Z. (2004), 'Orte documentation – communication components', Available from `http://www.ocera.org/download/components/WP7/orte-0.3.1.html`.

The MathWorks, Inc. (2005), 'Real-Time workshop embedded coder user's guide', www.mathworks.com.

The MathWorks, Inc. (n.d.), ‘Real-Time workshop data-sheet’, Available from `http://www.mathworks.com/products/rtw/`.

Žídek, M., Sojka, M. and Hanzalek, Z. (2008), ‘Analysis of automatically generated code from Matlab/Simulink models’, Deliverable of the FRESCOR project (D-OP6).

Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T. et al. (2008), ‘The worst-case execution-time problem:overview of methods and survey of tools’, *ACM Transactions on Embedded Computing Systems* **10**, 1347375–1347389.

Appendix A

Content of the enclosed CD

The CD enclosed to this work contains the complete source codes of the timing library described in the work, including scripts used to process the measured values. The source code of the analysed portion of the robot control software is also included. Complete source codes of the robot control software are not publicly available, but they can be requested from Ing. Michal Sojka (sojkam1@fel.cvut.cz), manager of the project.

Text of the thesis is also included in PDF format (`zidek_2009.pdf`).