Evaluation of Static Time Analysis for Volcano Communications Technologies AB

Susanna Byhlin

7 December 2004

Department of Computer Science and Engineering

Mälardalen University, Västerås

And

Volcano Communications Technologies AB, Göteborg

Supervisor at Mälardalen University: Andreas Ermedahl

Supervisor at Volcano Communications Technologies AB: Anders Kallerdahl

Examiner at Mälardalen University: Björn Lisper

MÄLARDALENS HÖGSKOLA

VOLCANO AUTOMOTIVE GROUP

Evaluation of Static Time Analysis for Volcano Communications Technologies AB

Abstract

Today, small computers are commonly embedded as a part of many different products like mobile telephones, toys and vehicles and the number of embedded computers, used in modern cars, has increased dramatically during the past decade. This evolution puts higher requirements on the communication networks that connect these computerized parts. It is among other things necessary to know exactly how long time each individual task in a computer system takes to execute and also important to assure that the system operates correctly in a worst-case scenario. Today, the *worst-case execution time (WCET)* of industrial applications is usually measured in a traditional way, called *dynamic timing analysis*. However, this method requires a lot of time from the system developer, who cannot guarantee that the worst-case has been found. What most of the software developers today are missing is an analysis tool that generates WCET estimates for programs statically. This can be performed by a method called *static timing analysis* that estimates the WCET of a program, without actually running it. Instead a WCET analysis tool is used that rely on models of a specific processor and the possible programs executions. *Volcano Communications Technologies AB (VCT)* is a company with a current tool chain for development of real-time communication solutions for embedded network systems, principally used within the car industry. This report gives an account of a study that has been done to examine if a commercial WCET analysis tool could be integrated into VCT's current tool chain. If that would be possible, shorter development times and overall production costs could be fulfilled. The results show that an integration of a static analysis tool would be possible but requires a lot of workload and detailed knowledge about the analysed system from the user. The analysed parts have also been measured in a traditionally way with help of an oscilloscope, and the results have been compared to see how similar the generated WCETs are. These dynamic timing analyses were hard to perform and good results were difficult to achieve. On account of insufficient knowledge about the code and lack of time, the WCETs generated by the two methods were quite unreliable and different for some complex and static code snippets. The WCETs generated for some simpler and static code snippets were on the other hand more reliable and similar. The purpose with this study was however not to compare these estimates, but rather to investigate the possibility of applying a static WCET analysis method on VCT's code and observe the difficulties that arise.

Keywords: WCET, Real-Time Systems, CAN, LIN, Static Timing Analysis, Dynamic Timing Analysis

Acknowledgement

This master thesis was performed at the Department of Computer Engineering at Mälardalen University, Västerås but also at Volcano Communications Technologies AB, Göteborg during the period between February and September 2004. The work has been performed by Susanna Byhlin and is the final part of the education at Mälardalen University that leads to a Master of Science degree in Computer Engineering.

First of all I want to thank my main supervisor Andreas Ermedahl at the Department of Computer Engineering at Mälardalen University, Västerås for all his support during this thesis. Then I want to thank my second supervisor Anders Kallerdahl at Volcano Communications Technologies AB for all his support during my time in Göteborg. I also want to express my special gratitude to rest of the people at Volcano Communication Technologies AB who helped me in many ways in Göteborg. Especially Anna Bengtsson for our nice and significant lunch walks, Monica Josgård for all practical help and the other guys on the company, for all the hard and sweaty floorball matches. Many thanks goes to my examiner Björn Lisper at Mälardalen University for his valuable comments and corrections at previous drafts of this report and to Daniel Sandell for his answers to some of my thoughts and questions. Thanks also to the company *AbsInt Angewandet Informatik GmbH* for giving me access to the *aiT Worst-Case Execution Time Analyzer (aiT)* tool both in Västerås and Göteborg. Especially thanks to Christian Ferdinand and Martin Sicks at the company for their valuable answers about the aiT WCET tool. Finally I want to thank Lotta Aldén for the accommodation in Göteborg, which made this work possible at all.

Västerås
7 December 2004
Susanna Byhlin

MÄLARDALENS HÖGSKOLA

VOLCANO AUTOMOTIVE GROUP

ABSTRACT 2

ACKNOWLEDGEMENT 3

1 INTRODUCTION 6

1.1 BACKGROUND 6
1.2 PURPOSE 6
1.3 WORST-CASE EXECUTION TIME 7
1.4 DELIMITATIONS 8
1.5 DEFINITIONS AND ABBREVIATIONS 8
1.5.1 Definitions 8
1.5.2 Abbreviations 9
1.6 RELATED WORKS 9
1.7 THESIS OUTLINE 10

2 RELEVANT TECHNOLOGIES 11

2.1 NETWORK COMMUNICATION PROTOCOLS 11
2.2 CAN COMMUNICATION PROTOCOL 12
2.2.1 The CAN Architecture 13
2.3 LIN COMMUNICATION PROTOCOL 13
2.3.1 The LIN Architecture 14
2.3.2 The LIN Frame Format 14
2.3.3 Signals in LIN 15
2.4 LIN VERSUS CAN 16
2.5 THE VOLCANO CONCEPT 16
2.5.1 The Complete Volcano CAN Tool Chain 17
2.5.2 The Complete Volcano LIN Tool Chain 18
2.6 THE TARGET HARDWARE 19
2.6.1 The Microcontroller 19
2.6.2 Pipelines 22
2.7 STATIC WORST-CASE EXECUTION TIME ANALYSIS 23
2.7.1 Tools for Static Analysis of Worst-Case Execution Time 24
2.8 DYNAMIC WORST CASE EXECUTION ANALYSIS 25

3 PROBLEM DESCRIPTION AND METHOD 25

3.1 THE PROBLEM 25
3.2 METHOD 26
3.2.1 The Hiware HC12 Compiler and SmartLinker 26
3.2.2 aiT Worst-Case Execution Time Analyzer 28

4 SOLUTION 32

4.1 ANALYSE EXECUTABLES WITH AIT 32
4.1.1 User Annotations 33
4.1.2 Performance 39

5 MEASUREMENTS WITH AIT 39

5.1. THE MASTER 41
5.2 THE SLAVE 50

6 MEASUREMENTS WITH OSCILLOSCOPE 57

6.1 THE HARDWARE CONFIGURATION 57

7 RESULT 59

7.1 COMPARISON OF THE TWO ANALYSIS METHODS 59

8 CONCLUSIONS 61

9 FUTURE WORK 63

BIBLIOGRAPHY 65

APPENDIX A 68

APPENDIX B 69

APPENDIX C 70

APPENDIX D 73

1 Introduction

1.1 Background

Nowadays, small computers are commonly embedded as a part of different products like mobile telephones, toys and vehicles. The number of embedded computers, used in modern cars, has increased dramatically during the past decade. This evolution puts higher requirements on the communication networks that connect these computerized parts and make them able to communicate with each other. Today, the systems used in modern cars are distributed real-time systems [CRTM99]. Distributed means that the user interprets the many interconnected computers as one component, while real-time means that the system always must generate correct results in exact right time, not too soon and not too late. In account of this, the time is one of the most important properties of these types of systems, together with the performance and the cost. It is therefore necessary to know exactly how long time each individual task in a real-system takes to execute and also important to assure that the system operates correctly even in a worst-case scenario. Today, the *worst-case execution time (WCET)* of industrial applications is usually measured in a traditional way, called *dynamic timing analysis*. Tools such as oscilloscopes, emulators or logical analyzers generally perform these types of measurements. However, methods like these require a lot of time from the system developer and cannot guarantee safely results, i.e. that the worst-case has been found. That depends on the complexity of modern software and hardware, which makes it difficult to force the worst-case scenario by testing the program with all possible input values in all possible conditions. What most of the software developers today are missing is an analysis tool that could generate WCET for a program without actually running it. This method is called *static timing analysis* and a working method like this may lead to shorter development times of the applications and simultaneous guarantees safely results.

Up to 30 percent of the manufacturing cost of a modern car is today represented by its electronics [Vct04], so there is much to do within this subject field to reduce this cost. For example, an integration of a static analysis tool in a current tool chain could lead to shorter development times, but also a decrease in cost. The earlier in the development process the WCET can be estimated; the earlier essential decisions about the future development can be decided.

Volcano Communications Technologies AB (VCT) is a company with a current tool chain for development of real-time communication solutions for embedded network systems, principally used within the car industry [CRTM99]. The company was founded 1998 and is now owned by Antal Rajnak, Volvo Technology Transfer AB, Motorola, Inc. and the 6th Swedish Pension Fund. VCT has about 55 professionals located in Göteborg; Sweden, Tägerwilen; Switzerland, Budapest; Hungary, Detroit; USA and Cologne; Germany. Their customers are both OEMs and suppliers to the automotive industry and include companies such as Ford/PAG, Daimler Chrysler, BMW, Siemens Bosch and several others. Integration of a static analysis tool in VCTs current tool chain is of great interest for improved development of their applications.

1.2 Purpose

This study is concerned with the improvement of measuring the WCET for applications developed by VCT. The work is carried out within the framework of the competence centre Advanced Software TEChnologys (ASTECs)

WCET-project in Uppsala, Sweden, which is sponsored by Vinnova. For more information about the ASTEC-group and their projects, we refer to [Ast04]. The purpose of this study was to examine an integration of a static analysis tool into VCTs development environment. By using a static analysis method to obtain the WCETs, the development time, as well as the development cost of VCT's products may be reduced. That would also enable VCT to deliver more exact WCETs on request by their customers, who want to use the Volcano concept within their systems. The Volcano concept is a data communications approach that is delivered by VCT. Section 2.4 describes further information about the Volcano concept.

To reach this purpose, several case studies have been performed and WCETs have been generated by a static analysis tool. By estimating the precision and quality of the generated WCETs, it will be possible to see if it would be profitable for VCT to use a static analysis tool in their development environment. If that is the case, such a tool could be integrated into the VCT's current tool chain and replace or complement their present measuring methods.

1.3 Worst-Case Execution Time

The *worst-case execution time (WCET)* is defined as the longest possible execution time a program ever can achieve [Erm03]. This definition assumes that the program is running in isolation and executing undisturbed from interfering activities such as the operating system and concurrently running tasks. The WCET is used to describe the time behaviour of a program and it is essential to bind this time for all applications within a real-time system, to guarantee that the system works correct in all possible scenarios. Real-time systems are systems where the correct result is as important as the time of which the result is produced. When analysing the execution time of this type of systems, *safe* and *tight* values are desirable. Safe means that no underestimation is allowed while tight means that the overestimation must be acceptable, i.e. not too high. For assuring that the execution time is safe and tight, it must be guaranteed that the used tools or models produces safe and tight results [EES+03].

In the same manner as the WCET, the *best-case execution time (BCET)* is defined as the shortest possible execution time a program ever can achieve while the *average-case execution time (ACET)* lies somewhere in-between the BCET and the WCET. The conditions between the different execution times of applications are illustrated in Figure 1.1.

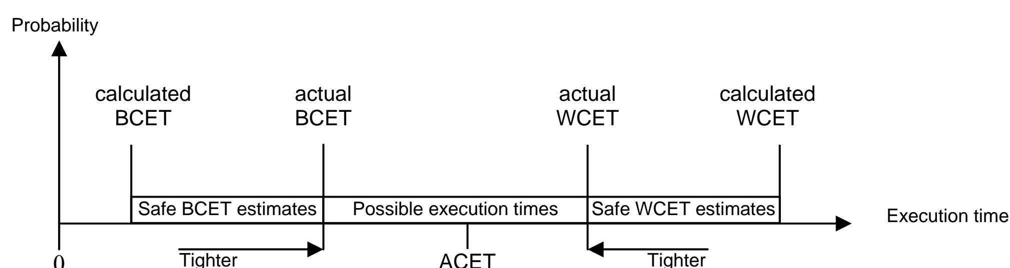


Figure 1.1 *Different estimations of execution times.*

The *actual WCET* is the real WCET, while the *calculated WCET* is the time generated by static timing analysis. This value is an overestimation of the actual WCET and therefore always a safe result. Dynamical WCET analysis, on the other hand, leads to some of the possible execution times in the middle part of Figure 1.1. These values are typically underestimations of the actual WCET and margins must be added to these values to get safely results. We always strive to achieve a tight WCET, with as low overestimation as possible.

$$\text{Measured WCET} \leq \text{Actual WCET} \leq \text{Calculated WCET}$$

In the same manner, the actual BCET is the real BCET, while the calculated BCET is the result from a static timing analysis. This value of the BCET is always safe but underestimated. A dynamical timing analysis of the BCET in a traditional way is on the contrary not safe and a margin has to be subtracted. In this case we always strive to achieve a tight BCET, with as low underestimation as possible.

$$\text{Calculated BCET} \leq \text{Actual BCET} \leq \text{Measured BCET}$$

The WCET of a program depends on the program flow such as loop iterations, recursion depths and function calls, together with machine dependent factors such as processor rate, pipelines and caches. Cache memories are not managed in this study but pipelines are used and described more detailed in Section 2.5.2 below. Components such as caches and pipelines complicate the task of determining WCET. However, analysis methods that do not consider the behaviour of caches and pipelines, typically overestimate the WCET, which further lead to waste of resources. A detailed description of the current hardware used in this work is present in Section 2.5.

1.4 Delimitations

This master thesis includes 20 credits and corresponds to the final part of my education at Mälardalen University, which leads to a Master of Science degree in Computer Engineering. The work has been performed both at the company Volcano Communications Technologies AB in Göteborg and at the Department of Computer Engineering at Mälardalen University in Västerås. Due to lack of time, it was not possible to evaluate the whole of VCT's system within this limited time period and only a part of their system was therefore included in this study. Since *LIN Target Package (LTP)* is smaller and less complex than *Volcano Target Package (VTP)*, LTP was decided to be analysed after consultation with VCT.

1.5 Definitions and Abbreviations

1.5.1 Definitions

- *Bus* – The wire in a network used to transmit data between the interconnected nodes, usually a screened or unscreened twisted pair cable.
- *Central Processing Unit (CPU)* - The primary unit in a computer system that controls the execution of the instructions.
- *Cluster* – Several nodes interconnected by a bus.
- *Electronic Control Unit (ECU)* – Microcontroller connected to the network bus, among others things it contains of a CPU, several memories and an external bus.

- *Frame* - A packet that is sent over the physical bus as a message. Including identifiers, signals, checksums etc.
- *Master* – The node in the LIN cluster that initiates and controls the communication over the LIN network.
- *Network* – Several clusters interconnected by a gateway-node.
- *Node* - Unit connected to the network bus able to communicate with the other nodes, usually an Electronic Control Unit (ECU), a microcontroller including a CPU and memories.
- *Signal* - A unit with an assigned value that is used for communication within the most embedded systems.
- *Slave* – The nodes that are connected to a network but not act as a master.

1.5.2 Abbreviations

- *CAN* – Controller Area Network
- *CFG* – Control Flow Graph
- *CPU* – Central Processing Unit
- *ECU* – Electronic Control Unit
- *EEPROM* – Electrically Erasable Programmable Read Only Memory
- *kbps* – kilo bits/second ⇔ 1000 bits per second
- *LCFG* – LIN Configuration Generator
- *LIN* – Local Interconnect Network
- *LNA* – LIN Network Architect
- *LTP* – LIN Target Package
- *Mbps* – Mega bits/second ⇔ 1000 000 bits per second
- *NVRAM* – Non Volatile Random Access Memory
- *RAM* – Random Access Memory
- *ROM* – Read Only Memory
- *SCI* - Serial Communication Interface
- *STP* – Screened Twisted Pair
- *UART* – Universal Asynchronous Receiver Transmitter
- *UTP* – Unscreened Twisted Pair
- *VCFG* – Volcano Configuration Generator
- *VCT AB* – Volcano Communications Technologies AB
- *VNA* – Volcano Network Architect
- *VTP* – Volcano Target Package
- *WCET* – Worst-Case Execution Time

1.6 Related Works

A similar study by Daniel Sandell has been reported [San04]. His work was done for the company Enea Data that delivers advanced real time technology, system development and IT-solutions [Ene04]. He comes to the

result that it was possible to use a static WCET analysis tool on industrial real-time operating system code with more or less intervention by the user.

Another study by Martin Carlsson [Car01] for OSE, that is a subsidiary to Enea Data, has been reported. He developed an existing prototype of a WCET tool further, to enable transformation of binary executable files for ARM microprocessors into control flow graphs. He also analysed enable-disable interrupt regions within the OSE real-time operating system.

An additional study has been done by the German company *AbsInt Angewandet Informatik GmbH t* [Abs04]. Thesing *et al.* have in [TSH+03] tested out their self developed tool, called *aiT Worst-Case Execution Time Analyzer,* at Airbus France flight-critical software.

Space Systems Finland (SSF) [Ssf04] has further developed an analysis tool, called *Bound-T* [Bou04], for estimating WCET bounds of programs. It was first developed for the *European Space Agency (ESA)* and has successfully been used in some of their space projects which are described more detailed in [HLS00a and HLS00b].

Colin and Puaut used static timing analysis in [CP99] to predict the WCET of the kernel in the real-time operating system (RTEMS). This work was performed by using their self developed WCET analyser, named *Heptane* [CP01].

1.7 Thesis Outline

Chapter 2 presents general background theory to most of the subjects touched in this report. The chapter starts with *Section 2.1* that describes real-time systems and network protocols used in modern cars in general. It is followed by *Section 2.2* that gives a detailed description of the frequent used CAN communications protocol, while *Section 2.3* present a more detailed description of the less-safety LIN communications protocol. *Section 2.4* describes the Volcano concept in brief; including the complete Volcano CAN and LIN tool chains and their workflows. *Section 2.5* present the current target hardware and its including peripherals and pipelines. Finally *Section 2.6* explains measurements of WCET, statically, including a presentation of available WCET tools while *Section 2.7* explains measurements of WCET, dynamically. *Chapter 3* gives a general description of the current problem and the method used to solve it. It further presents a brief description of the *Metrowerks Hiware Compiler* and *SmartLinker* and the *aiT WCET Analyzer*. *Chapter 4* describes the solution in more detail, including a detailed description of the aiT workflow. *Chapter 5* presents some examples of executables analysed by the aiT tool, together with some conclusion about how the entire WCET is influenced by the different input parameters. *Chapter 6* illustrates some examples of WCETs estimated by measurements with an oscilloscope. *Chapter 7* presents the result of the work performed in Chapter 5 and Chapter 6 together with some comparisons of the two methods. *Chapter 8* presents some conclusions that have been drawn from the current work together with the emergence of problems and their solutions. *Chapter 9* finally presents some examples of future work within the WCET subject in general but also together with VCT.

2 Relevant technologies

This chapter presents general background theory to most of the subjects touched in this report such as real-time systems, network protocols and WCET analyses.

2.1 Network Communication Protocols

A modern car contains a number of electronic control units (ECUs) interconnected to several communication networks, as illustrated in Figure 2.1 [LH00]. These ECUs are programmed to perform different functions and to communicate with each other by sending frames over the network bus.

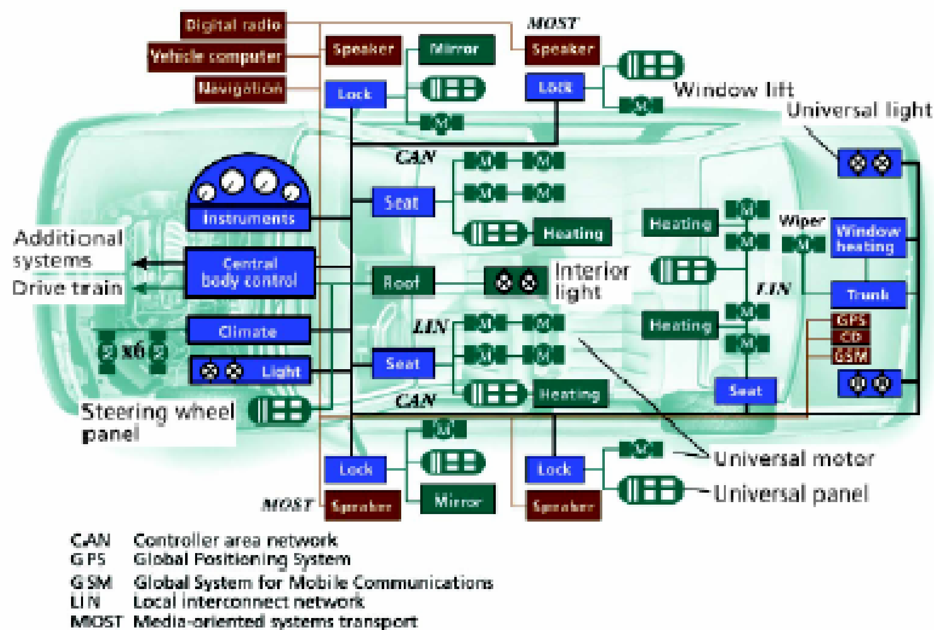


Figure 2.1 *Example of the network architecture in a modern car [LH00].*

Some of the functions implemented in the ECUs have *hard* real-time requirements while other functions have *soft* real-time requirements [Erm03]. A hard real-time requirement means that a missed *deadline* of a function could have catastrophic consequences. Missing a deadline in a hard real-time system could even lead to loss of human life of the system users, which is not accepted under any circumstances. A missed deadline means that a function does not generate correct result within a specific time. One example of a system with hard real-time requirements is the anti brake system (ABS) in a car. If such a system does not react within a specific time, it is worthless and can give terrible consequences. The functions with soft real-time requirements are on the other hand less sensitive to deadline misses and do not lead to catastrophes. The user may not even observe a deadline miss within a soft real-time system, but they could have disturbing consequences. A function in a real-time system with soft requirements is for example the window-control application in a car, i.e. no one gets killed if it does not work, but it could be annoying if the window cannot be pulled up or down.

The communication networks that are used in modern cars are distributed real-time systems, based on some network protocols. A modern car consists of several interconnected network segments using different protocols, often a low-speed and a high-speed *Controller Area Network (CAN)*, a *Local Interconnection Network (LIN)* and maybe a *Media Oriented Systems Transport (MOST)*. CAN is the most common concept used in communication systems and interconnects microprocessors with a serial broadcast bus. (Section 2.2 presents further information about the CAN concept). LIN is a low-cost network used to interconnect nodes that not require the bandwidth and safety properties that CAN offer. (In Section 2.3 a more detailed description about the LIN concept is present). Finally, MOST is a multimedia fibre optic network optimised for multimedia applications. In the future even a *FlexRay* network will be used in modern cars. FlexRay is a high-speed communications protocol that supports the needs of future in-car control, such as X-by-wire systems like drive-by-wire and brake-by-wire [RH03]. Information in form of signals and frames may be transmitted between these different network segments through a specific gateway node. Properties such as speed and communication cost per node for several network protocols are illustrated in Figure 2.2.

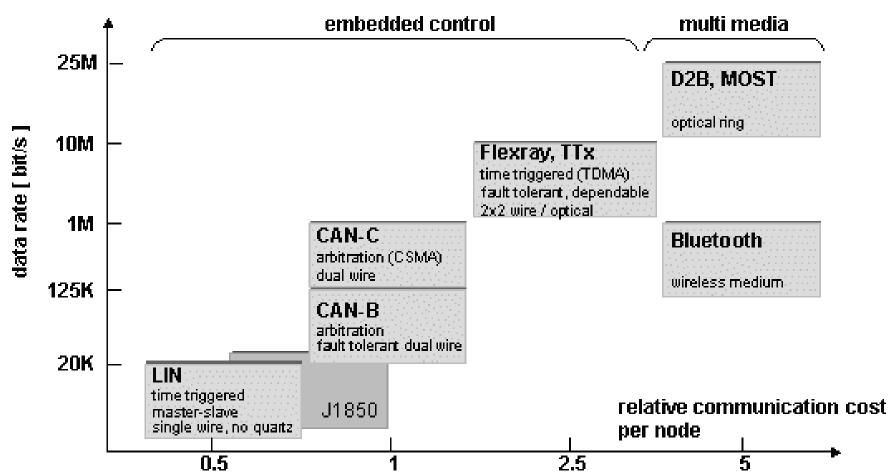


Figure 2.2 *Different properties of some network protocols [Vct04].*

2.2 CAN Communication Protocol

Controller Area Network (CAN) is the most common communication protocol used in the automotive industrial today [JBO+00] and is described more detailed in [Can91, Can02]. In CAN networks, peer stations such as sensors and actuators are connected to a bus topology that provides broadcast communication. Broadcast means that all nodes connected to the bus perceive everything that is transmitted over it. The purpose of using CAN in vehicles is that distributed real-time control with high level of security is supported. When one station in a CAN network wishes to communicate with another station connected to the same network, it sends out a message with a unique identifier on the serial bus. Every other station connected to the bus receives this message and determines if it is relevant or not. If it is, the station processes the message, otherwise it ignores it. Only messages that are of interest for the stations are processed, which leads to less workload at each station. CAN is an event triggered communication protocol using the *Carrier Sense Multiple Access/Collision Resolution*

(CSMA/CR) mechanism to arbitrate access to the bus and avoid collisions. According to the ISO 11898 standard, the used cable is a twisted par, shielded (STP) or unshielded (UTP) [Can]. The bus is designed to operate at a speed up to 1 Mbps as long as the distance of the wire is less then 50 meters. Not all network components require this bandwidth capacity and security that CAN offer. Therefore, CAN is a too expensive solution to interconnect devices with simple functions such as control of indoor-lights, seats and window-drivers. In these cases the *Local Interconnection Network (LIN)* communication protocol can be used instead, to reduce the total cost. The purpose of using LIN is not to replace CAN but to be a complement and make it possible to create a hierarchical bus structure. Section 2.3 presents further information about the LIN communication protocol.

2.2.1 The CAN Architecture

The ECUs connected to the CAN bus contains several applications that perform different kind of functions, as illustrated in Figure 2.3. Functions may also be divided between several nodes within the network, which for example is the case for the alarm function. About 20 to 30 CAN nodes are in general interconnected by a CAN network in a modern car and can perform over hundred of functions [Mel99].

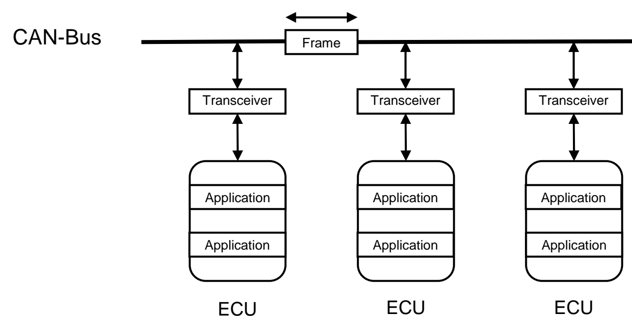


Figure 2.3 *The CAN architecture.*

The ECUs can communicate with each other by sending frames with signals over the CAN bus. In VTP that is using CAN; signals are packed into frames in an efficient way and transmitted as messages over the CAN bus to utilize the bandwidth in a suitable way. To be able to transmit and receive frames over the network each node in the CAN network uses a transceiver, which is a combined transmitter and receiver.

2.3 LIN Communication Protocol

LIN (Local Interconnect Network) is a low-cost communication protocol used in networks where the bandwidth and safety properties of CAN are not required. The cost of a LIN node is inexpensive in comparison with a CAN node [Lin]. The LIN concept was developed by the LIN consortium in 1998, which includes some major car vendors (Audi, BMW, Daimler Chrysler, Volkswagen, Volvo), Motorola and VCT. [Lin04] presents further information about this LIN consortium. LIN is for example used for communication between less safety critical units such as rain sensors, sun roofs and air-conditionings. A LIN network forms a cluster that is composed of one master node and one or more slave nodes connected to a LIN bus. The master node is the gateway node that is connected to the CAN bus and the one who manage the LIN communication. The nodes communicate over the LIN bus by transferring frames with information as illustrated in Figure 2.4. The communication over the LIN bus is always initiated of the master sending out a message header on the bus. One of the slaves is then activated

and starts the transmission of a response, which contains the data to transmit. The LIN bus is designed to operate at a lower speed than the CAN bus, up to 20 kbps, which is sufficient for the units within the LIN cluster.
One key characteristic of the LIN protocol is the use of its schedule table(s), that assure that the LIN bus never will be overloaded, i.e. that no frames are lost. The master controls the communication on the bus by following a predefined schedule table that defines which specific frame to send at each time. This makes LIN to a time triggered network using a *Time Division Multiple Access (TDMA)* mechanism, unlike the CAN network that mentioned above is event triggered.

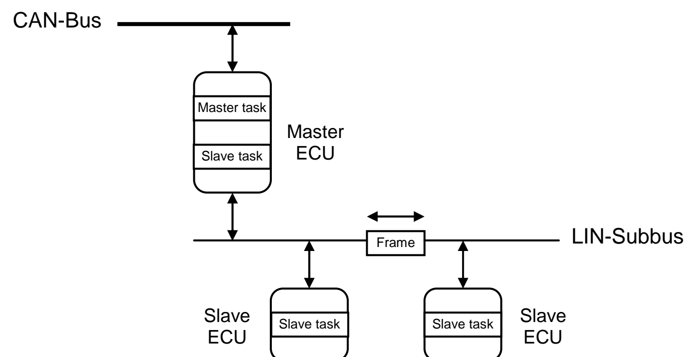


***Figure 2.4** The LIN architecture.*

The first specification of LIN was released in July 1999 and was started in the car production in 2001 (in a Mercedes SL Roadster). Today, about 5-15 LIN nodes, are commonly embedded per vehicle using the LIN concept. The current version of LIN is 2.0. It was released in September 2003, which makes it a relatively new standard and also becoming a *SAE (Society of Automotive Engineers) J2602* standard, which is a standard within the American automotive industry.

2.3.1 The LIN Architecture

Like in the CAN network, nodes in the LIN cluster, are electronic control units (ECUs). The master node in the cluster contains of a slave task and a master task responsible for the communication over the network while the slave node(s) in the cluster only contain a slave task [Lsp03] as illustrated in Figure 2.4. Each node in the LIN cluster has its own set of frames to transmit over the bus on initiative of the master task in the master node. The frames follow a specific standard format illustrated in Figure 2.5 and are described more detailed in the following section.

2.3.2 The LIN Frame Format

The frames used for communication within the LIN network consist of a header and a response part. The header of a frame is composed of a *break field*, a *synchronization field* and an *identifier field*. The first byte in the header, the break field, is used to identify the beginning of the frame and contains 13 or more *dominant bits* followed by a *delimiter* of one or more *recessive bits*. In the LIN protocol a dominant bit has the value zero and a recessive bit has the value one. The next byte is a synchronization field, a byte field that contains the hexadecimal value 0x55 used for clock synchronization. The last byte in the header is an identifier field that

contains an *identifier*, which denotes the content of the message and the *identifier parity*. Each slave in the LIN-cluster looks at the identifier field in the header to determine if it should respond on it. If this is the case, the slave node or nodes sends out a response part on the bus. The response part is composed of between one to eight *data fields* plus a *checksum field*. The data fields are used to carry two types of data within the transmitted frame, either signals or diagnostic messages. The last byte in the response part, the checksum field, is used to detect if the frame is correctly transmitted or not. Finally, each frame is followed by an *inter-frame space*, which is the time from the end of the current transmitted frame to the start of the next transmitted frame. The arrangement of bits in a standard LIN frame is depicted in Figure 2.5.

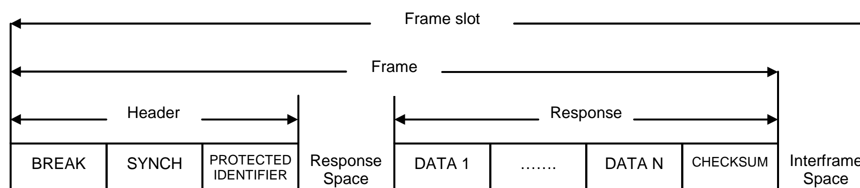


Figure 2.5 *The LIN frame standard format.*

There are four different main types of frames used to transmit signals within a LIN-cluster, separated by different identifiers. *Unconditional frames* is the standard frame type for carrying signals, *event triggered frames* are used to allow multiple slaves to provide response on a header; *sporadic frames* enable dynamic behaviour in the otherwise deterministic signal transmission and *diagnostic frames*, either a master request frame or a slave response frame, are always used to carry diagnostic or configuration data. Signals are bounded to frames of these types and are in that way transmitted over the network. There are additional two types of frames possible called *user defined-* and *reserved frames* but these are less important. For a detailed presentation of how frames are defined and transmitted within the LIN concept, refer to [Pro03].

2.3.3 Signals in LIN

A signal is a unit that consists of small data items and is used for communication within the most embedded control systems. They may be visualised as a "virtual wire" and is either an input to or an output from an ECU [CRTM99]. Signals are assigned values which in the LIN networks are represented by scalar values (1-16 bits) or array bytes (1-8 bytes). These signals are packed into the frames and in this way transmitted between the different nodes within the LIN cluster. Each signal is described in the *LIN description file (LDF-file)*[1] by their name, size, initial value, which node that sends out the signal and which node(s) that receive the signal. Each signal has only one producer but can have one or more receivers.

Hundreds of signals may be transmitted over the communication networks but there are some restrictions:

- The size of one signal is not allowed to be larger than 16 bits.
- If the size is less then eight bits it may not be split between different bytes in the frame.
- Each byte in a byte array should map a byte in a frame.

[1] The LIN description file (LDF-file) describes the complete LIN network and contains all required communication parameters. The LDF-file is used as input both for tools and application programs.

For further information about signal management within the LIN cluster, refer to [Pro03].

2.4 LIN versus CAN

CAN is a serial communications protocol that specifies how frames are transmitted between the different nodes. LIN is on the other hand not only a protocol, but a holistic communication concept. The LIN specification covers in addition to the definition of the protocol and the physical layer also the definition of interfaces for development tools and application software.

The main saving factor of LIN versus CAN is the single-wire transmission which leads to a lower communication cost per node compared to CAN. But the advantage in cost is compromised by a lower bandwidth and the restrictive bus-access scheme. The main features of the LIN and CAN protocol are compared in Table 2.1.

Table 2.1 *Comparison of the main features of LIN and CAN protocol*

	LIN	***CAN***
Bus speed	< 20 kbit/s	< 1 Mbit/s
Cable length	40 m	50 m
Medium access control	Single master/ Multiple slave	Multiple master
Topology	Bus	Bus
Typical size of network	2..16 nodes	4..20 nodes
Media	Single wire (copper)	Dual wire (twisted pair)
Data bytes per frame	1-8 bytes	0-8 bytes
Security level	Low	High
Access mechanism	TDMA	CSMA/CR
Type	Time-triggered	Event-triggered
Typical applications	Seat, door and light functions	Airbag and anti-brake system

2.5 The Volcano Concept

Volcano is a holistic data communications concept for design and implementation of in-vehicle networks that use the CAN, LIN, MOST and FlexRay communications protocols. The concept was developed by Volcano Communications Technologies AB (VCT AB) and is an integration of a chain of tools and engineered software. The Volcano concept was first developed for Volvo Car Corporation when they in 1994 decided to initiate the development of the new P2 car platform, which later was introduced in Volvo S80 [Vct04]. It is a system engineering approach that manages software development from many different suppliers.

Volcano uses a timing model that specifies the time between the generation and the consumption of each signal. For example when the window lift button is pushed down a signal is generated. This signal is then consumed when the window is moving up or down. During the process of the signals different kind of *jitters* [2] may occur, when for example many signals are sending over the network at the same time. Other type of jitters are used as input to the VNA and LNA tools (see Section 2.5.1 and Section 2.5.2) so times for the jitters would be of interest for VCT.

[2] Jitter is a form of delay variation which may depend on different sources such as variation in execution time and bus contention.

2.5.1 The Complete Volcano CAN Tool Chain

Rajnak and Ramnefors [RR00] and Casparsson *et al.* [CRTM99] describe the complete Volcano CAN tool chain that is illustrated in Figure 2.6.

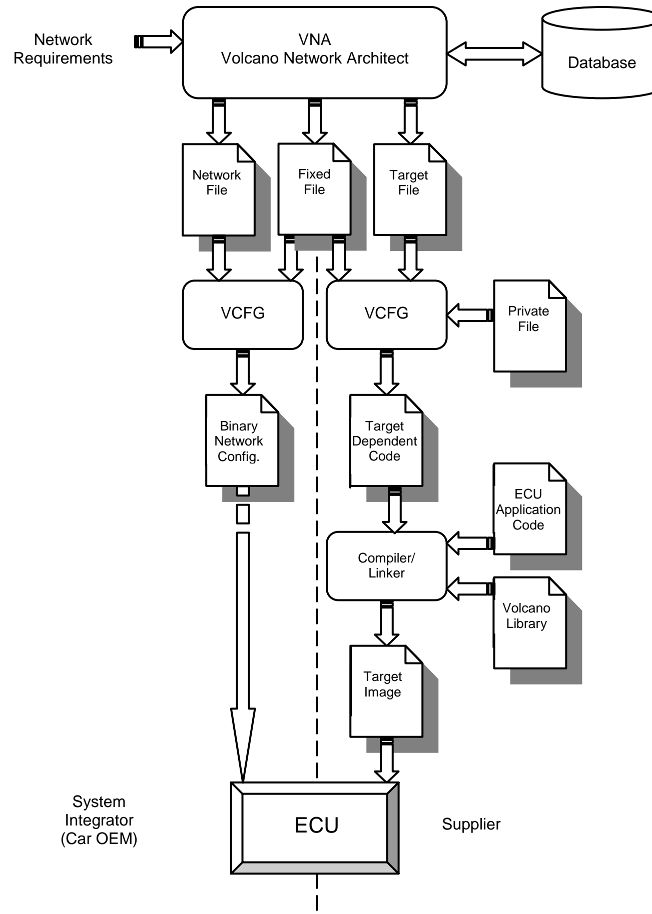


Figure 2.6 *The complete volcano CAN tool chain.*

The *Volcano Network Architect (VNA)* is the top-level design tool in the chain and supports both CAN and LIN based networks. This design tool is used to capture information about the network such as requirements of signals and nodes to design a solution for an electronic vehicle system. VNA is connected to a database where this relevant information about the network is saved. Furthermore, a frame compiler is used to pack signals into frames, which utilizes the available network bandwidth in an efficient way. VNA generates configuration files (for LIN and/or CAN), which are later used as input to the other tools in the tool chain. These configuration files contain of a *network file*, a *fixed file* and a *target file*. The network file describes the network interface configuration, the frames with mapped signals plus generated schedule tables. This configuration information is saved in NVRAM. The fixed file includes available network interfaces and signals for every ECU. Finally, the target file includes hardware specific details about the ECUs, such as type of CPU and compiler, and the size and addresses of the memory reserved to store data, such as RAM and NVRAM pools.

Next tool in the tool chain is the *Volcano Target Package (VTP)* that focuses on the ECUs in the vehicle. VTP consists of two main parts, the *Volcano Configuration Generator (VCFG)*, which is a configuration tool used by both the ECU developer and the system integrator, and the volcano library. The volcano library is a pre-compiled object library, necessary for CAN and LIN communication. VCFG collects the network file and the fixed file, generated as output from VNA, together with a private file as input. The private file is delivered by the supplier of the ECU and contains private information about the specific node. VCFG then creates target dependent code that is linked together with the volcano library; maybe some other libraries and the application source file for a specific ECU. The compiler and linker generate a *target image* that includes the communication behaviour for the ECU. Finally, VTP implements this target into the ECU by help of the software download tool called *Volcano Boot-Loader*. The Volcano software that is downloaded into each ECU must be compatible with the already existing application software.

2.5.2 The Complete Volcano LIN Tool Chain

Olsén [Ols04] and Rajnak and Engler [RE03] present the complete Volcano LIN tool chain that is illustrated in Figure 2.7.

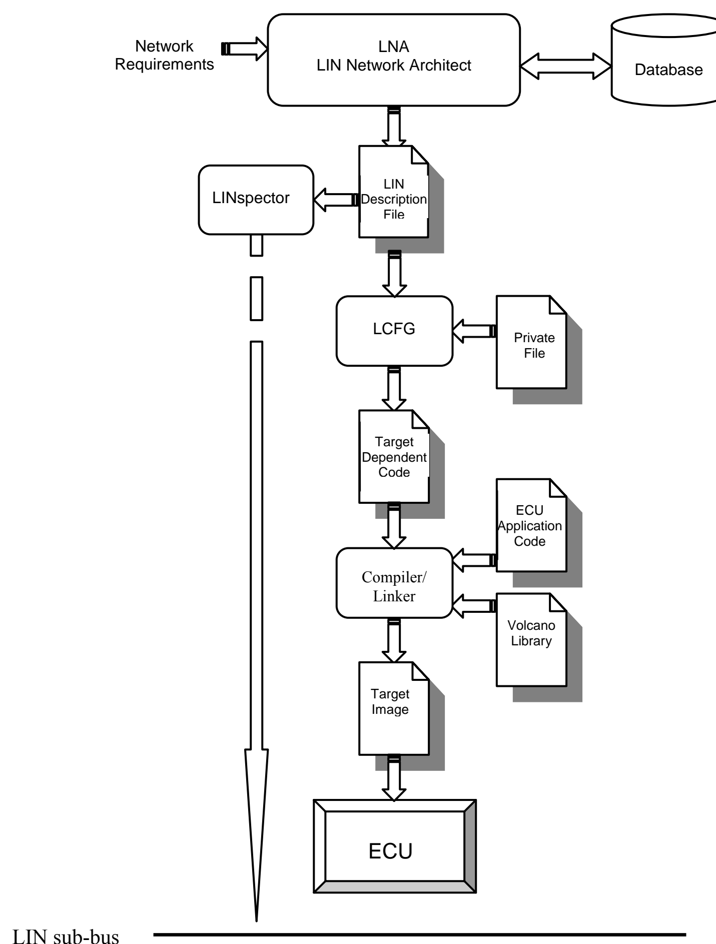


Figure 2.7 *The complete volcano LIN tool chain.*

The *LIN Network Architect (LNA)* is the top-level design tool in the LIN tool chain. It captures information about the network objects like requirements of signals, nodes and frames and automatically configures a LIN network.

LNA packs the signals into frames, assigns identifiers and generates schedule tables. Like VNA, LNA is connected to a database where all relevant information about the network is saved. LNA generates a *LIN Configuration Description File (LDF)* with the extension .ldf that describes the complete communication behaviour of the network with all nodes, i.e. describes the complete LIN cluster. One example of a LDF-file for a simple LIN-cluster is illustrated in Appendix A.

The next tool in the LIN tool chain is *LIN Target Package (LTP),* which consists of two main parts; the *LIN configuration generator (LCFG)*, which is a node configuration tool that sets up single nodes and the *LIN Target Package Library* (lin.lib), which is pre-compiled object libraries necessary for communication within the LIN network. LCFG uses the generated LNA output configuration file (LIN Description File) as input, together with a *private file,* to generate a *LIN target dependent code*. The private file is delivered by the supplier and includes all hardware details for one of the nodes defined in the LDF-file plus other information such as used flags. The *target dependent file* is compiled and linked together with the LIN Target Package Library and the ECU application code to generate a *target image*. In this case, an HIWARE *absolute file* with the extension .abs. (Generation of the HIWARE absolute file format is described more detailed in section 3.2.1.1). The last tool in the LIN tool chain is the *LINspector,* which is Volcano's verification tool. The LINspector is used for developing, testing, verifying and emulating the communication on the LIN network by taking the LIN Description File as input. An additional tool called *LIN graphical object (LINgo)* can further be added to the LINspector to enable a graphical view of the process.

2.6 The Target Hardware

Embedded systems are usually based on a microcontroller which is an integrated microprocessor and a set of peripherals, including an external bus and memories. There are common examples of processor families such as ARM, StrongARM, PowerPC, Motorola 68k and MIPS. The best-selling 32-bit microcontroller of these is the ARM family from Advanced Risc Machines. All ARM variants have a single, simple pipeline, and very few have caches. The second-best selling architecture is the Motorola 68k family to which the current used microcontroller MC9S12DP256 belongs.

2.6.1 The Microcontroller

The microcontroller used in this study for generations of WCET estimations was a MC9S12DP256 from Motorola [Mic00]. This is a 16-bit, well-designed device that among other things is composed of:

- 16-bit CPU (STAR12 CPU) of the family MC68HC12
- Multiplexed external bus, which is 16-bit data path throughout the microcontroller
- 256K bytes of Flash EEPROM
- 4.0K bytes of EEPROM
- 12.0K bytes of RAM
- 2 asynchronous serial communications interface (SCI)
- Five 1Mbps CAN 2.0 A, B software compatible modules (MSCAN12)

STAR12 CPU

The STAR12 CPU is a high-speed unit with a 16-bit data path and several registers. The CPU or central processing unit is the base unit in the processor, which controls the execution of the instructions. The CPU buffers program information by using an instruction queue. This buffer always gives the CPU immediate access to at least three bytes of machine code at the start of every instruction, which further leads to increased execution speed. The CPU has different addressing modes (indexed, inherent, immediate and so on), which determine how the CPU accesses the different memory locations, but it does not support cache memories.

The registers are an integral part of the CPU and contains of:

- Accumulator A
- Accumulator B
- Accumulator D

  A and B are 8-bit accumulators used to hold operands and results of arithmetic calculations. These two accumulators can be treated as one 16-bit double accumulator and is in this case called D.
- Index register X
- Index register Y

  X and Y are 16-bit index registers used in indexed addressing mode.
- Stack Pointer (SP)

  The Stack Pointer is a 16-bit register that points to the last used stack location and is for example used to hold information during subroutine calls.
- Program Counter (PC)

  The Program Counter is a 16-bit register holding the address to the next instruction to be executed.
- Condition Code Register (CCR)

  The Condition Code Register contains of five status flags and three other bits.

Flash EEPROM 256K

Flash EEPROM stands for Flash Electrically, Erasable, Programmable, Read-Only Memory and is a non-volatile[3] data memory. This memory is ideal for program and data storage. The 256K bytes are divided into four flash blocks of each 64K bytes, which further are arranged into 32K parts. The minimum erase sector is one block, 512 bytes (64K bytes) but all the four flash blocks can be programmed or erased at the same time. Different types of memory areas in embedded systems, called banks, usually got different access times. The access time for the banks in the Flash EEPROM is one bus cycle for reading bytes and aligned words and two bus cycles for reading misaligned words. The architecture of the Flash EEPROM in the MC9S12DP256 microcontroller is illustrated in Figure 2.8.

[3] A non-volatile data memory is a type of memory that retains its content when the power is turned off.

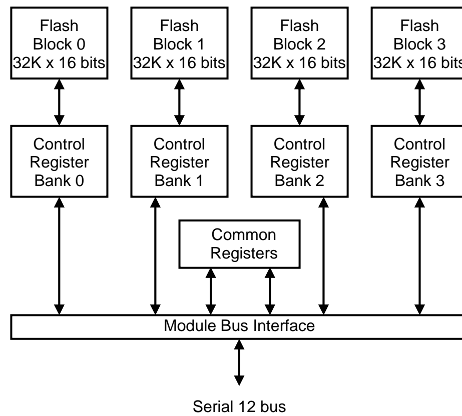


Figure 2.8 *Flash EEPROM 256K block diagram.*

EEPROM 4.0K

EEPROM stands for Electrically, Erasable, Programmable, Read-Only Memory. It is a non-volatile data memory with the size of 4.0K further organized in 2048 rows of two bytes. EEPROM is like the Flash EEPROM memory used to store data but its minimum erase sector is two rows or four bytes. The access time of the EEPROM memory in the microcontroller, MC9S12DP256, is one bus cycle for reading bytes and aligned words and two bus cycles for misaligned words. The architecture of the EEPROM in the MC9S12DP256 microcontroller is illustrated in Figure 2.9.

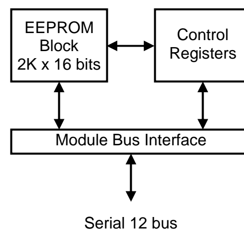


Figure 2.9 *EEPROM 4K Block Diagram.*

RAM

RAM stands for Random Access Memory and is a 12K, volatile[4] memory used for storing instructions, variables and temporary data during the program execution. The version used in the current microcontroller is static (SRAM) and its content can be read or written by the CPU.

SCI

SCI or serial communications interface is used for serial communication with peripheral devices and other CPUs. It supports full duplex, which enable full communication in both directions simultaneously. LIN is based on the SCI (UART) protocol and the physical bus is connected to the SCI.

[4] A volatile memory is a type of memory that requires power to retain stored data.

Motorola Scalable Controller Area Network (MSCAN)

Motorola Scalable Controller Area Network (MSCAN) is a communications control module that is used for implementation of the CAN 2.0 protocol but is not used for LIN communication. It is an embedded part in the CAN nodes, connected to the CAN bus by a TxCAN output pin and a RxCAN input pin. These pins are used for communication over the CAN bus by a transceiver. A CAN node with an integrated MSCAN module is illustrated in Figure 2.10.

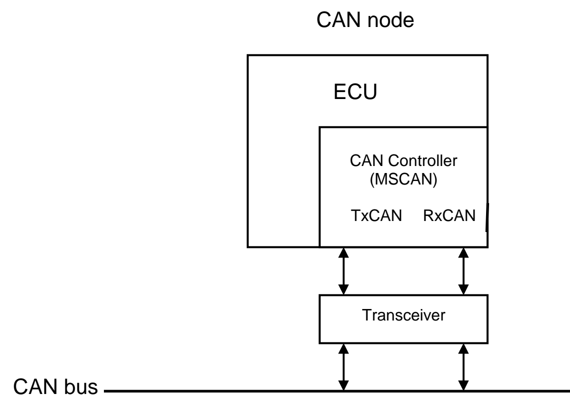


Figure 2.10 *MSCAN module connected to the CAN bus.*

2.6.2 Pipelines

Engblom presents in [Eng02] a detailed description of pipelines used in modern CPUs. A pipeline is divided into a number of stages that the instructions go through during execution. Every instruction must not go through all the pipeline stages and only one instruction can use each stage at each time. Pipelines can be used in many different levels to achieve higher performance but these also lead to more complex pipelines. One example of using a simple pipeline is described below.

Assume a pipeline with five stages and three instructions, A, B and C executing after each other. In the *IF* stage the instructions are fetched from memory, in the *ID* stage the instructions are decoded, in the *EX* stage the arithmetic operations are performed, in the *MEM* stage memory are accessed for data and in the last *WB* stage the computed values are written back to the registers. Instructions A takes four cycles, instruction B takes six cycles and instruction C takes five cycles to execute, as illustrated in Figure 2.11.

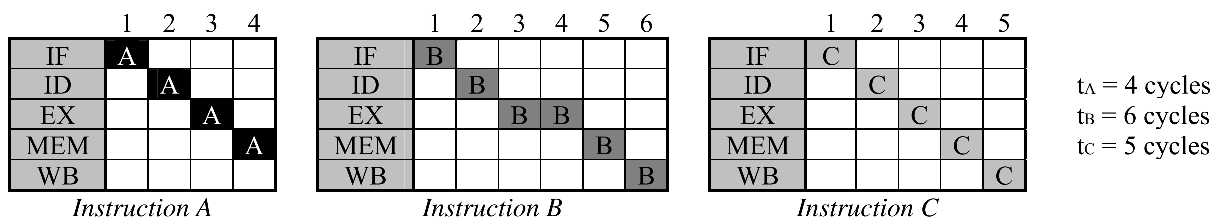
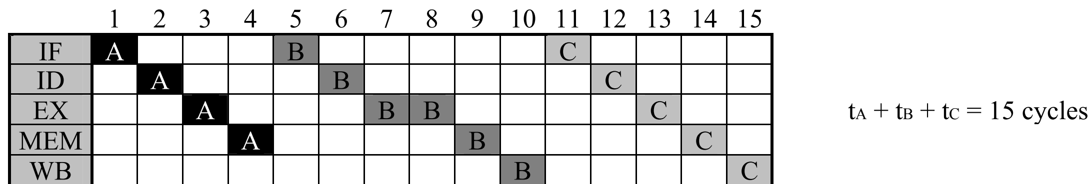


Figure 2.11 *Execution of instruction A, B and C in isolation.*

If these three instructions are executed in a processor without pipelining, each instruction has to finish its execution before the execution of the next instruction can start. In this case we add A's four cycles to B's six cycles and C's five cycles to get the entire execution time of the sequence of the three instructions. This is called non-pipelined execution and takes 15 cycles to execute, as illustrated in Figure 2.12.

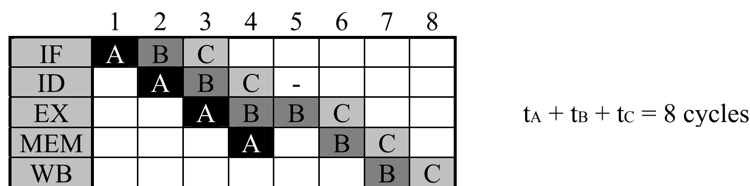


	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
IF	A				B						C				
ID		A				B						C			
EX			A				B	B					C		
MEM				A					B					C	
WB										B					C

$t_A + t_B + t_C = 15$ cycles

Figure 2.12 *Non-pipelining execution of instruction A, B respective C.*

On the other hand, if the three instructions are executed in a processor with pipelining, the instructions are overlapped and only take a total execution time of eight cycles, as illustrated in Figure 2.13. In this example, instruction C must wait in the stage ID because instruction B execute two clock-cycles in the EX stage. When an instruction waits in a stage without performing any work a *pipeline stall* occurs. The stall can depend on that an instruction requires data that is generated by a preceding instruction which is not available, that a preceding instruction reads data from a slow memory or that a stage is occupied by another instruction (see Figure 2.13). The execution time of an instruction is therefore depending on the execution time of its neighbour instructions.



	1	2	3	4	5	6	7	8
IF	A	B	C					
ID		A	B	C	-			
EX			A	B	B	C		
MEM				A		B	C	
WB							B	C

$t_A + t_B + t_C = 8$ cycles

Figure 2.13 *Pipelined execution of instruction A, B respective C.*

The example above shows that a processor with a pipeline enables the execution of different instructions to overlap, which lead to increase in processor performance. The STAR12/HCS12 processor used in this thesis (see Section 2.6.1) uses an instruction queue to buffer program information and increase the execution speed. The mechanism is called a queue rather than a pipeline because a typical pipelined CPU executes more than one instruction at the same time, while the STAR12/HCS12 always finishes executing an instruction before beginning to execute another. Because of the queue, program information is fetched a few cycles before it is used by the CPU. At least three bytes of program information are available to the CPU when instruction execution begins which increase the execution speed [Mic03].

2.7 Static Worst-Case Execution Time Analysis

There are two basic methods to obtain the WCET of a program, either by traditional measurements (*dynamic timing analysis)* or by calculations based on *static timing analysis.* To enable calculation of WCETs for applications statically, among other things software properties such as iteration bounds for loops, recursion depth

and function calls have to be considered. Information about *infeasible paths*, which are paths that are not feasible during the execution and conditions with specific values, generates even tighter WCETs. Furthermore, hardware properties such as caches, pipelines, access time of memories and rate of used microprocessor must be known. When performing WCET analysis for complex processors, the effect of these hardware properties are important. However, with simpler processors, like the STAR12, the software properties like upper bounds of loop iterations are more important.

Static WCET analysis is performed in the following three main steps:

- **Flow analysis**: This first step of the WCET analysis calculates all the possible flows (execution paths) through the program and further produces information about iteration bounds of loops, infeasible paths, function calls, etc.
- **Low-level analysis:** This second step determines the execution time for each separate unit in the program and considers the time effects of instructions and their neighbours. These effects can be pipeline overlaps and speed of memory access.
- **Calculation**: This last step of the WCET analysis combines the results from the flow- and the low-level analysis, to produce the final WCET of the analysed program. The calculation may be done by one of three different main methods, *path based, tree based* or *IPET (Implicit Path Enumeration Technique) based* calculation [EES+03].

The low-level analysis may further be divided into a local and a global part. The local low-level analysis determines the effects of pipelining while the global low-level analysis determines the effects of caches etc. [Eng02]. Since the HCS12/STAR12 processor used in this study does not support cache memories, only the local part is managed in the low-level analysis. In more complex processors it can be hard to distinguish between the local and global low-level analysis.

2.7.1 Tools for Static Analysis of Worst-Case Execution Time

Today, there exist few commercial WCET tools on the market but several research prototypes. They differ by supporting different CPUs, using different calculation methods and interacting with different compilers. Two existing commercial WCET tools are *aiT Worst-Case Execution Analyzer (aiT)* from the German company *AbsInt Angewandet Informatik GmbH* and *Bound-T* from the Finnish company *TidoRum*. aiT supports the ARM7, HCS12/STAR12, PowerPC555, 565, 755 and ColdFire 5307 processors while Bound-T supports some processors in the Intel-8051 series, ERC32 in the SPARC V7 series and the digital signal processor ADSP-21020.

Furthermore, there are several tool prototypes developed within the WCET area. *Heptane (Hades Embedded Processor Timing ANalyzEr)* is a WCET analysis tool developed by Puaut *et al.* at *IRISA (Institut de recherche en informatique et systèmes aléatoires)* and available for free download from [Hep04]. It analyses programs written in ANSI-C with some restrictions and supports the Pentium 90 MHz, MIPS and Hitachi H8/300 processors.

Engblom [Eng02] and Ermedahl [Erm03] present a modular architecture as a starting point for the development of a WCET tool prototype, called *SWEET (SWEdish Execution time Tool)*. Their approach is based on both the IPET- and the path based calculation methods. Target processors that are supported by this prototype are ARM9 and NEC V850E.

Zhao *et al.* present in [ZKW+] an overview of the *VISTA (Vpo Interactive System for Tuning Applications)* framework used for tuning the WCET of applications. VISTA consists of a WCET timing analyzer and an integrated compilation system. Processors supported by VISTA are, the SC100 from StarCore and the MicroSPARC I in the SPARC V7 series.

Bernat *et al.* present in [BCP03] a tool for probabilistic WCET analysis named *pWCET*. This tool combines both measurement and analytical approaches and computes probabilistic bounds of the WCET.

In this thesis the first mentioned tool, aiT, has been used for the static WCET analyses.

2.8 Dynamic Worst Case Execution Analysis

Testing the timing behaviour by repeatedly measure the execution time of the program with varying input values in different conditions is uncertain [Ste02]. This is because it is almost impossible to prove that the conditions and input values, which lead to the worst-case scenario, have been taken into account. It is also hard to measure one program in isolation without influence of interrupts, other programs and the operating system. Therefore, the WCET can be missed using this method. Because of that, a safety margin always must be added to the measured results, to assure of getting safe values. To get dynamically measured times to compare with the statically analysed execution times, different tools such as emulators, oscilloscope and logical analyzers can be used. In this work an oscilloscope was chosen because VCT were already familiar with this type of equipment. The oscilloscope measurements do not generate the worst case directly so several executions of the investigated application had to be done. The worst-case of these executions were saved for comparisons with the statically analysed WCETs.

3 Problem Description and Method

The question to answer with this study was if a static analysis tool for estimations of WCETs could be integrated into the development environment of Volcano Communications Technologies AB. This chapter gives a general description of the current problem and the method used to solve it. It further presents a brief description of the Hiware Compiler and SmartLinker and a more detailed description of the aiT Worst-Case Execution Time Analyzer.

3.1 The Problem

As mentioned above the aim was to examine if a static analysis tool could be used to estimate the WCET of VCT's applications statically. To solve this problem the Volcano LIN Target package (LTP) was chosen as the

part to be analysed. It was first necessary to obtain knowledge about how the LIN system works, including tools and software in the LIN Target Package and knowledge about the generated functions and their tasks. Some function calls are more important then others and more interesting to perform measurements on. The execution times of these functions are not fixed but depend on properties such as size and type of transmitted frames and number of frames and flags. Information about the structure of the analysed source code was also essential to obtain, including occurrence of pointers, function pointers, nested loops, and recursion. Getting a view of the system and understand how everything works is time consuming for outsiders. However, when enough knowledge about the system and its functions has been obtained, the analysis can start.

3.2 Method

As mentioned in section 2.6.1 there are only a few commercial WCET tools available on the market and for this reason traditional measurements are still the most common methods used in industry. The aiT Worst-Case Time Analyzer (aiT) was chosen in this work since it supported some of the processors used by VCT, and because AbsInt work within the ASTEC project together with Mälardalen University.

Some of the target processors that VCT supports are [Vct04]:

- Fujitsu 16Lx
- Hitachi H8S, SH7055, SH7058
- Motorola HC08, HC12, Star12, PowerPC
- NEC V85x
- Texas Instruments TMS470

Among these, the implementation of LIN 2.0 is available for Motorola's HC08 and HC12 processors. Evaluation versions of aiT are available for Motorola's HCS12/STAR12 and PowerPC processors. These factors together with the interest from VCT lead to that the HCS12/STAR12 processor was chosen to perform analyses on.

3.2.1 The Hiware HC12 Compiler and SmartLinker

Different compilers may be used to translate the source code into instructions that a particular processor understands and may execute after a linking process. A linker takes the object files generated by a compiler as input, links them together, and finally generates the executable programs [Lev99, ASU86]. Different compilers generate different type of object file formats and aiT must have knowledge about which compiler and linker to integrate with. aiT for the Star12/HCS12 processor is enabled in two versions, either supporting the Hiware Compiler from Metrowerks, Inc. or the Cosmic compiler from Cosmic. In this work the Metrowerks Hiware Compiler version was chosen because this compiler is supported and more actively used by VCT. So in summary, all the analysed applications in this study were compiled and linked with the Hiware HC12 Compiler and SmartLinker from Metrowerks, Inc. generating an executable absolute file (with the extension .abs). Figure 3.1 gives an illustration of the compiling and linking process.

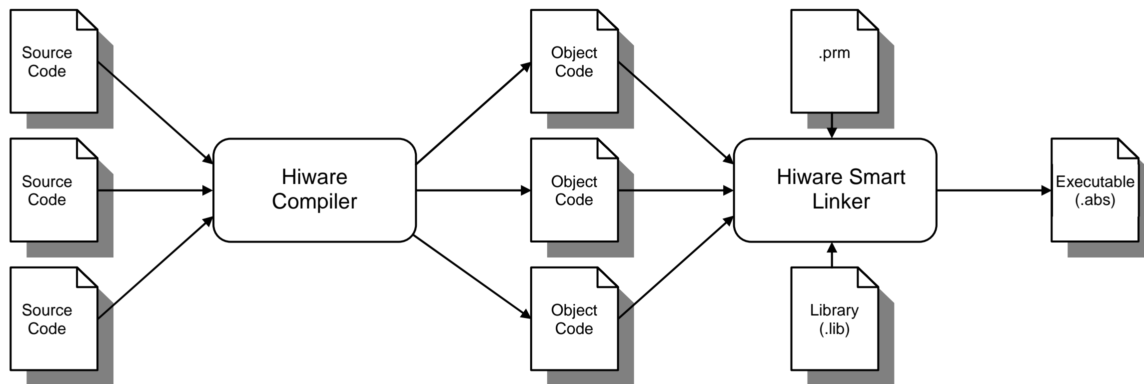


Figure 3.1 *The Hiware Compiler and SmartLinker.*

3.2.1.1 The Hiware HC12 Compiler

The Hiware HC12 Compiler takes the application source files (.c) and header files (.h) as input and generates corresponding object files (.o) containing the target code as well as some debugging information. Further, the Hiware Compiler offers a number of options or flags to control the compiler's operation. Compiler optimization may have advantages or disadvantages and it is in general not easy to provide option settings that generate the best result. Because of that different optimization options must be tested out for specific applications. By default, most optimizations are enabled in the compiler and the analysed applications are compiled with the following options.

-*Cc*: Allocates constant objects into ROM (Read Only Memory).
-*Lasm*: Generate an assembler listing file to which all generated assembler instructions are printed.
-*Lp*: Generate a text file with all preprocessor commands resolved.
-*Ms*: Define that a small memory model is used which contains a 64kb code-address space.
-*Or*: Allocate local variables (char and int) in registers.

Some optimizations may reduce the code size for some functions, but may increase the code size for other functions in the same compilation unit. The analysed applications have been compiled with the different compiler flags below without any influence on the WCET.

-*Cu*: Enable unrolling of loops.
-*I*: Allows including file paths in the settings.
-*OdocF*: Enable varies options to be evaluated over each single function to find out the best combination.
-*Oi:* Enable inline expansion.

The object file format specifies the format of the object files, library files and absolute files. The Hiware Compiler supports two different object file formats, ELF/DWARF and HIWARE. By default the HIWARE format is generated but this can easily be changed to the ELF/DWARF format by setting a specific flag.

3.2.1.2 The Hiware SmartLinker

The Hiware SmartLinker takes the object files, generated by the Hiware Compiler, as input and links them together with a parameter file (.prm) and predefined libraries (.lib). Similarly to the Hiware Compiler, the SmartLinker offers a number of options that are used to control its operations. The analysed applications are linked with the following options:

-*M*: Generates a map file containing information about the linking process.
-*O*: Defines the name of the .abs file that is generated.
-*Add*: Add object and library file to the project without modifying the parameter file.

The linker is a smart linker which only links the objects that actually are used by the current application. It merges the object files into one file called absolute file (.abs) which contains absolute, not relocatable code that could be loaded into the target with help of a debugger. This absolute file is either of the ELF/DWARF or the HIWARE object format. When the application has been compiled and linked successfully and the executable absolute file has been generated, the WCET analysis with aiT can begin.

3.2.2 aiT Worst-Case Execution Time Analyzer

The Company – AbsInt Angewandet Informatik

AbsInt Angewandet Informatik GmbH was developed at the Department of Compiler Construction and Programming Languages at Saarland University, Germany. The comapny was founded in February 1998 and provides advanced development tools for embedded systems, and for validation, verification and certification of safety-critical software. For example flight-critical software from Airbus France [TSH+03], as was mentioned in section 1.6. AbsInt provide creative solutions to their customers, for example tools for analyse and verify the safety and reliability requirements of software, such as WCET. This tool is called aiT Worst-Case Execution Time Analyzer and is the analysis tool that further will be used in this study, to estimate WCET of VCT's applications statically.

AbsInt received 2004 European IST price for the development of their aiT WCET Analyzer (see [Abs04] for further information about the company AbsInt and their tools).

The tool - aiT Worst-Case Execution Time Analyzer

aiT Worst-Case Execution Time Analyzer (aiT) is a WCET tool that analyse an executable piece of code and computes a upper bound of the WCET. The WCET analysis is performed by aiT under assumption that the analysed program is running in isolation and executed undisturbed without influence of outside effects such as interrupts, exceptions, task switches etc. aiT provides a solution to the problems with cache and pipeline behaviour. It uses IPET-based calculation that was mentioned in Section 2.6, using *ILP (Integer Linear Programming)* to compute upper bounds WCETs for selected code pieces. aiT provides support for calculating the WCET of an application and given that all inputs are correct, the upper bound is valid for all inputs and executions of the task. The analyzer takes the program in the form of the absolute file (.abs) that should be

analysed, user annotations like targets of indirect function calls, upper bounds of loop iterations and the start address as inputs, see Section 4.1. Further aiT uses a graph viewer tool, *AiSee* [Abs04], to generate a graphical visualization of the WCET together with the worst-case path that leads to this upper bound. This visualization is a box that displays the WCET and a compounded call-graph and control-flow graph (CFG) that shows the included routines of the program.

The Upper Bound of the WCET

Figure 3.2 illustrates a visualization of the upper bound of the WCET generated by aiT.

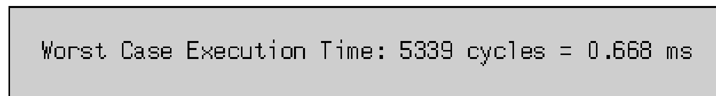


Figure 3.2 *Example of a worst-case execution time generated by aiT.*

Call Graph

The call graph visualizes the routines, which are functions and loops in the analysed program, together with their relationships. aiT performs a loop transformation, which convert loops to separate routines. The arrows between the routines denote that one of the routines is calling another. One example of a call graph generated by aiT is illustrated in Figure 3.3.

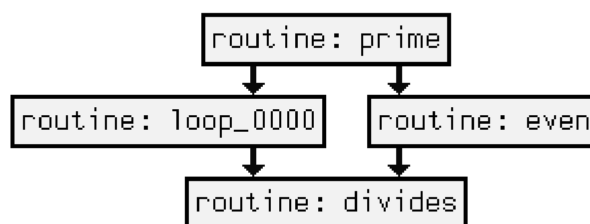


Figure 3.3 *An aiT call graph [Ait04].*

Routines

The nodes in the call graphs are called routines and may be either functions or loops. The routines are visualized by yellow boxes, usually enclosed by a black border and labelled with its routine name. A routine that contain calls to targets that cannot be determined by aiT is visualized by a yellow box with red dotted border. Non-analysed routines, or routines that aiT never analyse are coloured orange and are usually not containing any code, which the yellow boxes do. The routines have an information field that shows the full path name of the file where the current routine is located. Each routine can be opened to visualize its including basic block graph, which is composed of basic blocks and control-flow edges.

Basic Block Graphs

The basic blocks graphs are composed of nodes that are basic blocks and arrows that represent the control-flow between them (see the next section below for more detailed information about the basic blocks). The first basic block in the graph is called *the entry block* and the last block is called *the exit block*. These two blocks are green unlike the other basic blocks in the graph that are blue. *The exit block* is formed as an ellipse labelled *x* while the other basic blocks are rectangles labelled with their hexadecimal start address at the form 0x0:0x4536. The basic block can also be labelled with the corresponding source code, which in those cases is extracted from the debug information in the executable. This makes it easy to see the corresponding source code to each basic block in the graph. In Figure 3.4a and 3.4b two examples of graphs with basic blocks are illustrated, one labelled with addresses and one labelled with source code.

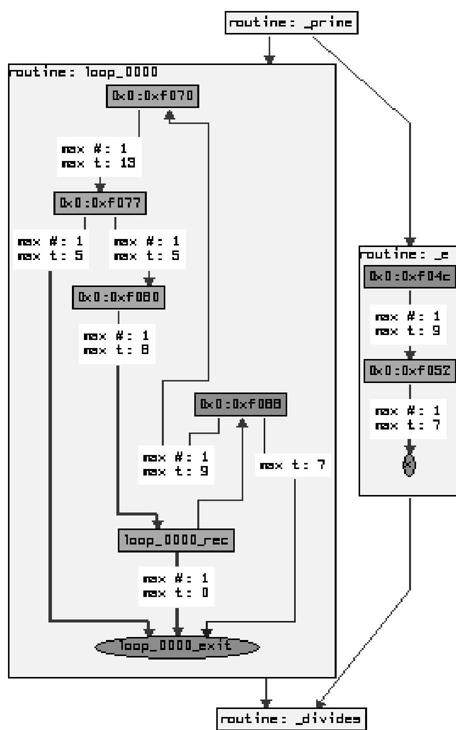


***Figure 3.4a** aiT basic blocks labelled by addresses.*

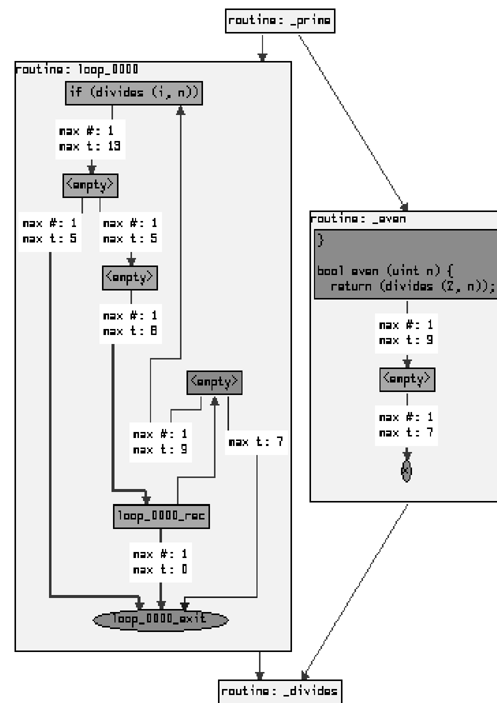


***Figure 3.4b** aiT basic blocks labelled by source code.*

Basic Blocks

A basic block is a sequence of instructions that are entered in the first of them and exit in the last of them. The instructions within a basic block are executed sequentially and jumps, calls and return instructions only occurs in the last instruction of a block, while their targets always are the first instruction in another block [ASU86]. The basic blocks are used to build up basic block graphs which show the connection between them by means of arrows. The basic blocks have information fields that may show the internal name (bx) of the block, where b stands for basic block and *x* is its number and the start address of the block at the form (P:B), where P is the page number in the memory and B is the base address in hexadecimal form. For example in the address 0x0:0x4536,

0x0 is the page number and 0x4536 the base address. Each basic block can further be opened to visualize its contained sequence of assembler instructions.

Instructions

The nodes within the basic blocks are instructions that are visualized as white boxes labelled with the mnemonic and operands of each instruction. Each instruction has an information field that shows the hexadecimal address of the instruction. The graph in Figure 3.5 illustrates an opened basic block with instructions.

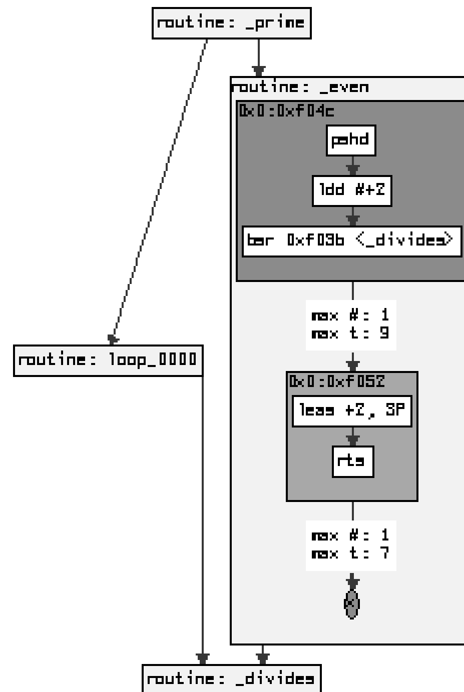


Figure 3.5 *An aiT basic block at instruction level.*

Edges

The edges or actually arrows that describe the control-flow within the call graphs, the control-graphs and the basic-block graphs, have different purposes and therefore different colours. The different colour possibilities for the edges are listed below.

- *Black* – The black edges are normal edges that represent textual succession.
- *Red* – The red edges in the graphs are ”false edges” and represent the flow from a condition instruction that is false.
- *Green* – The green edges are “*true edges*” and represent the flow from a condition instruction that is true.
- *Violet* – If a green and a red edge go to the same target they can be replaced with one violet edge, called *“collapsed edge”*.
- *Dark blue* – These edges represent the call from one routine to another.

Pipeline states

Pipeline states for a specific basic block for a specific context can be visualized by AiSee. Each pipeline state lists the instructions of the basic block it belongs to together with a list of operations performed while executing each instruction. Each operation takes one cycle so the entire numbers of cycles for each instruction depends on the number of operations. Figure 3.6 illustrates pipeline states for the basic block with the address 0x0:0xf04c from Figure 3.5.

```
0x0:0xf04c : pshd
Fetch (optional) : no access, Write (word to stack) [eeprom: 0xff9], Write (word continued) [eeprom: 0xffa]

0x0:0xf04d : ldd #+2
Fetch [0x0:0xf052], Fetch (optional) [0x0:0xf054]

0x0:0xf050 : bsr 0xf03b <_divides>
Write (word to stack) [eeprom: 0xff6], Fetch [0x0:0xf03a], Fetch [0x0:0xf03c], Fetch [0x0:0xf03e]
```

Figure 3.6 *Pipeline states for a basic block.*

4 Solution

This chapter describes the solution in more detail, including a presentation of the aiT workflow.

4.1 Analyse Executables with aiT

aiT is used to compute WCET for a executable piece of code, called a task. To analyse a specific task with aiT you have to give the program the following inputs:

- The executable to analyse, i.e. the absolute file (.abs) that has been generated by the Hiware Compiler and SmartLinker (see Section 3.2.1).
- User annotations like upper loop bounds, infeasible paths, unconditional branches etc.
- A list of memory areas.
- The start address of the task that should be analysed.

exe2crl is a tool that aiT uses to read the executable that should be analysed and deconstructs a control flow graph of the application.

The WCET analysis is performed by aiT in twelve sub-steps, where the four main steps are:

- **Loop bounds analysis** – This step tries to automatically determine the loop bounds in the program, which is the maximal number of iterations each loop may take. If aiT fails to determine some loop bounds the user has to set them manually before the WCET can be estimated.
- **Value analysis** – This step tries to determine the contents of all registers at each program points, which further is used to find infeasible paths.
- **Pipeline analysis** – This step estimate the WCET of each separate basic block.

- **Path analysis** – This step use the WCETs from the previous step together with the flow information to compute the total WCET of the analysed program with IPET-based calculation.

The other states in the analysis are *Creating IR, Transforming loops, Adding start attribute, Applying Results, Solving LP, Creating Visualization* and *Starting aiSee*. For HCS12/STAR12 aiT does not perform any cache analysis since the microprocessor does not contain any cache memories.

4.1.1 User Annotations

The aiT tool has to be supplied with various specifications and annotations to get as precise results as possible. These specifications and annotations are defined in a specific format in an AIS-file or as comments in the source code. If this information is not specified correctly, an incorrect result may be produced in the end. The correction of these annotations cannot be checked by aiT so a careful specification should be performed to assure correct and tight WCET estimates. The specifications in the AIS-file are lined up after each other and each directive is terminated by a semicolon (;).

Among others, following annotations can be set in the AIS-file to generate tight WCET estimates.

- Clock rate

This annotation must be set to enable aiT to generate the WCET in real time. It informs aiT about the clock rate of the microprocessor in *Hz*, *kHz* or *MHz* and can thereby compute the real time of the analysed task. The following example informs aiT that the clock frequency of the internal bus in the used microprocessor is 8 MHz.

clock exactly 8 MHz;

If the clock rate of the microprocessor is not specified, aiT only generates the computed WCET results in cycles and not in real-time.

- Naming the Compiler

The control flow reconstruction module, **exec2crl,** which reads the executable, works better if it knows what type of compiler that has generated the executable that is analysed. That is because different compilers generate different types of object code. The C Compiler for HC12/STAR12 of Metrowerks, Inc. that is used in this work is defined by the following annotation.

compiler "hc12-hiware";

- Declaring Routine Entries

exec2crl usually finds the entry points of the different routines automatically and their names are fetched from the symbol table in the executable file. Other specific addresses can be specified as routine entries by annotation of the following format.

entry 0x0:0x414c; or
entry “foo”;

These annotations generate a new routine that starts at the address *0x0:0x414c* or at the not automatically found routine *“foo*”.

- Stop Decoding
exec2crl may be informed to stop decoding the executable at a specific address. For example when an interrupt routine does not return or when the analysed code snippet ends by entering a non-terminating self-loop. A self-loop is a loop that does not have any path leaving it, which makes it impossible to find the WCET path. For example, when entering a `while(1)` loops that never returns, an end annotation can be used to leave it. End specifications may be specified with of the following annotation formats.

end 0x0:0x414c; or
end ”foo”;

The WCET analysis treats these annotations as leaving the current analysed code snippet when it reaches one of these points. In the case with a `while(1)` loop it treats the loop as only performing one iteration.

- **Control-Flow Specification**
exec2crl usually finds target addresses of calls and branches automatically, but if some non-trivial calls and branches cannot be resolved, these have to be specified manually in the AIS-file. It is easy to see which basic block that contains unresolved calls because these blocks have a red dotted border in the control-flow graph. These are also listed in aiT’s message window and in the report file. The following two lines show examples of annotations that can be used to specify unresolved targets of branches and calls.

instruction 0x0:0x414c branches to ”foo”;
instruction 0x0:0x414c calls 0x0:0x8500;

exec2crl can also by informed that a called routine does not return by means of following annotation form.

snippet “foo” never returns;

This informs **exec2crl** not to read the code that follows the routine *“foo”*.

- Addresses of Memory Access
The value analysis tries to obtain the exact addresses of memory accesses. Otherwise, specifications of the following form can be set manually.

instruction 0x0:0x8500 accesses 0x3a:0x8100 .. 0x3a:81FF; or
instruction 0x0:0x8500 accesses "TAB";

Or if a HCS12 instruction requires more than one memory access it can be specified with the following annotations.

instruction 0x0:0x8500 accesses 0x3a:0x8100 in step 1;
instruction 0x0:0x8500 accesses 0x3a:0x8101 in step 2;
And so on.

- Known Register Values

If some register values at a specific program point is known these can be specified by annotations in the AIS-file in the following format.

instruction 0x3a:0x9110 is entered with X= 0:0, Y = 0:0x100..0:0x1FF;

This leads to that instruction 0x3a:0x9110 always is entered with the value 0 in register X and a value between 0x100 and 0x1FF in register Y. The registers can be the A, B, D, X, Y, or SP registers those were present in Section 2.5.1 and may be defined to have a specific value or a range of values.

- Address Mapping

Memory management is an important part in real-time systems and therefore, this part of the annotations is mandatory. The microprocessor STAR12/HCS12 used in this work and by the current version of aiT uses a memory that is addressed with a 16 bit base address and a 6 bit page number in the form P:B. P is the page number and B is the base address. The base addresses should be mapped to special memory areas and these mappings can be configured by annotations of the following form.

map base 0x0 .. 0x3ff to registers;
map base 0xff00 .. 0xffff to vectors;
map base 0x1000 .. 0x3fff to ram;
map base 0x0 .. 0xfff to eeprom;

The four lines above specify the default memory mapping of MC9S12DP256B, which is the STAR12/HCS12 microprocessor used in this work. These mappings depend of the memory architecture of the current processor which is specified more detailed in Section 2.5.1. The following kinds of memories exist and can be mapped to.

- Registers
- Exception vectors – This 16-bit vector pointing to the memory location where the routine that handles an exception is located.
- On-chip RAM

- On-chip EEPROM
- Non-paged on-chip flash memory
- Non-paged external memory

Correct specification of memory mappings is important for aiT to compute correct WCET, this is because access to different memories needs different access time. Other memory properties that are important are the read-only and data properties of the memory areas. Areas that are read-only can safely be used and areas that contain data need to be decoded as instructions. To specify these properties for a specific area, one of the following three annotation forms may be used.

1. **Access Properties**

   Four different possibilities (*is read-only, is write only, is read-write, is inaccessible*)
   Example: *area 0:0x1000 .. 0:0x1fff is read-only;*

2. **Content Properties**

   Two different possibilities (*contains code, contains data*)
   Example: *area 0:0x1000 .. 0:0x1fff contains data;*

   The two properties defined above can be combined in one annotation.
   Example: *area 0:0x1000 .. 0:0x1fff is read-only and contains data;*

3. **Timing Properties**

   The specific access time of a memory area can be specified with an annotation.
   Example: *area from 0x00:0x8000 to 0x2f:0xbfff access time = 3;*
   This line tells aiT that the instructions in the specified memory area have an access time of three cycles.

aiT knows about the timing properties of the default memory areas. For example, it knows that RAM is read-write and that EEPROM is read-only.

- Non-Analysed Code Snippets

exec2crl may be informed to not decode code snippets that are not of interest, to get a faster analysis. This is specified with some of the following annotation formats.

snippet "foo" is not analyzed and takes 12 cycles; or
snippet "foo" is not analyzed and takes 1640 nsec;

The execution time of these snippets has to be specified to determine the WCET of the code that contains the not analysed part. A not analysed routine is visualized as an orange box with black border in the control-flow graph.

- Infeasible Code

If the analysed code contains infeasible snippets, i.e. code that never executes, it can be specified with an annotation of the following format.

snippet 0x0:0x1a0c is never executed;

0x0:0x1a0c is the address of an arbitrary instruction in the basic block that should never execute. The infeasible code snippets are still decoded and visualized in the control-flow graph but its execution time is not included in the entire WCET. A block that is only reachable from an infeasible code snippet is also infeasible and therefore not reachable.

- Values of Conditions

aiT can be informed about conditions in the analysed code snippet that are *always true* or *always false*.
These annotations have the form.

condition 0x0:0452 is always true; or
condition 0x0:0453 is always false;

Always false means that the target of the branch is not decoded if it is not reachable from another basic block, while always true means that the target behind the branch instruction is not decoded.

- Recursion Depth

Indirect recursion is not allowed in code snippets that are analysed by aiT. Indirect recursion means for example that routine R1 calls routine R2 which further calls routine R1. However, direct recursion where a routine calls itself is allowed, but the maximal recursion depth has to be specified in the AIS-file.
Example:

recursion "foo" max 12;

This means that the routine *"foo"* can be called up to twelve times, including the start call of the routine.

- Loop Bounds

The loop bounds analysis tries to determine the number of loop iterations automatically. If this fails for some loops, their upper iteration bounds have to be specified manually in the AIS-file. All loop bounds have to be set before aiT could determine the WCET and the loop bounds have to be correct for achieving a correct result. The message *"This problem is unbounded"* is generated by aiT if there still are unbounded loop bounds that have to be set. Example of loop bound annotations:

loop 0x0:0x4537 begin exactly 12;

loop 0x0:0x4537 end exactly 3;

begin is used when the test is in the beginning of the loop, like for a while-loop in C while *end* is used when the test is in the end of a loop, like for a do-while loop in C.

- Source Code Annotations

Instead of specify the annotation for loop bounds in the AIS-file they can be defined direct in the source code by annotations in a specific format. Example of a recursion depth annotation in the source file:

```
Uint fac (uint n) {                /* ai: recursion here max 6; */
```

The line above is equal with a annotation of the following form in the AIS-file.

```
recursion "fac" max 6;
```

One example of a loop bound annotation in the source file is specified as follows.

```
for ( i=3; i*i <= n; i += 2)       /* ai: loop here max 20; */
```

Additional Files

Apart from the AIS- file, aiT uses three other files called *AIP-, AMF-* and *ETM-file*.

The AIP-file

This file is used to specify parameters that reduce the number of *contexts*. To enable aiT to find loop bounds automatically a necessary parameter in the AIP-file, called INTERPROC must be set to *vivu4 (virtual inlining virtual unrolling)* as follows.

INTERPROC: vivu4

This parameter can also be set to *vivu,* but in that case must all loop bounds be set manually by the user, as annotations in the AIS-file or in the source code.

INTERPROC: vivu

The AMF-file

The *AMF-file* lists all possible instructions and groups them into classes. The AMF-file is an interface to the ETM-file that provides the timing information for all the instructions in the AMF-file.

The ETM-file

The *ETM-file* specifies timing information for all instructions that is defined in the AMF-file. Timing information consists of the number of cycles each instruction needs to execute. The AMF- and ETM-file must be appropriate for the used hardware to compute correct timings.

4.1.2 Performance

When all necessary inputs have been given to aiT three different choices can be performed.

Compute CFG – This task creates a combined call graph and control-flow graph of the executable without performing any WCET calculation.

Analyze – This choice creates a combined call graph and control-flow graph and also performs the calculation of the WCET. The estimated WCET is visualized in a separate pink box and shows information about the execution time in cycles and real-time as illustrated in Figure 3.2. In this case all edges or arrows that belong to the worst-case path are coloured red. Each edge also got a corresponding white box with information about the maximum number of executions (max #: x) and the maximum execution time (max t: x) per edge for all contexts.

Visualize – This choice performs the same tasks as above but provides access to the pipeline states that were illustrated in Figure 3.6.

5 Measurements with aiT

The LTP, described in Section 2.4.2 was chosen as the part of VCTs system to analyse. LTP consist of a configuration tool and a library of files. The LIN API describes the interface between the network and the application program and has a set of functions. Each of these functions is implemented in an own source file and the most important functions have been analysed separately by aiT. This chapter illustrates some examples of executables analysed by the aiT tool, together with some conclusion about how the entire WCET is influenced by the different input parameters.

A simple master-slave example that only contains one master and one slave node had been used to perform the analyses of the most important functions used by these nodes. The LDF-file for this example is illustrated in Appendix A and the private files corresponding to these nodes are illustrated in Appendix B.

The code size

The size of some main functions within the LIN API is illustrated in Table 5.1.

Table 5.1 *Size of and number of loops in some main functions within the LIN API.*

Function	Code size		~Lines	Loops
	Object file (.o)	Source file (.c)		
l_star12sci_ifc_s_connect()	6 kb	3 kb	30	0
l_star12sci_ifc_m_connect()	6 kb	3 kb	30	0
l_star12sci_ifc_s_init()	6 kb	3 kb	50	0
l_star12sci_ifc_m_init()	6 kb	3 kb	30	0
l_star12sci_ifc_s_rx()	10 kb	14 kb	300	3
l_star12sci_ifc_m_rx()	16 kb	8 kb	200	0
l_star12sci_sch_set()	6 kb	3 kb	50	2
l_star12sci_sch_tick()	10 kb	12 kb	300	8
l_star12sci_sys_init()	4 kb	2 kb	10	0

The table shows the size of the object files (.o) generated by the Hiware Compiler and the corresponding source files (.c). It further shows the approximately number of code lines without comments and the number of loops within the source files.

Code properties

Some of the functions within the API contain one or several loops, actually in the range of zero to at most eight. Only one of these is nested, i.e. called within another loop. Iteration bounds for *simple* loops are generally found automatically by aiT and may be distinguished in the report file, generated during the analysis. However, none of the loops in the LTP functions were found automatically by aiT, so all the loop bounds had to be set manually. This was the case even if the INTERPROC parameter in the AIP-file, described in section 4.1.1, was set to vivu4. The reason to this was that the included loops were not simple but instead parametric[5] and often quite complex. Recursion does not occur at all within the LTP functions so recursions depth annotations did not had to be specified. Further, aiT did not complain about any dynamic branches that were not found automatically, so no branches of this type had to be set.

Four different parameters influenced the WCET of the analysed functions. These parameters are:

- Numbers of frames within the current network.
- Types of these frames, mentioned in Section 2.3.2.
- The size of the frames.
- Number of flags latched to the signals within these frames.

The measurements in Section 5.1 below describe how these parameters influence the entire WCET of some of the functions mentioned in Table 5.1. Only generating one WCET per function could lead to overestimations which further lead to waste of processor resources.

Annotations and its workload

The second largest of the LTP functions is the *l_star12sci_sch_tick()* function. An analysis with aiT of this function generated a call-graph that contained totally 19 routines in four different levels. Three of these routines were other functions, nine were loop-routines and finally seven of them were anonymous routines without

[5] The number of iterations for a parametric loop is not constant but depends on parameters outside the loop.

specific names in the symbol table. These anonymous routines are labelled *Anon_xxxx* in the call-graph, where *xxxx* is the start address of the routine.

Depending on that aiT did not found any of the loop bounds in the analysed functions automatically; these had to be set manually. Further, not executing parts had to be specified to eliminate analysis of uninteresting code. These parts can be error routines or other parts that we know in advance are not going to execute during run time. Conditions that we know to be always are true or always are false may also eventually be specified. However, the only mandatory annotations that aiT requires for WCET calculation is the loop bounds and the memory-mappings. The rest of the annotations, such as rate of the processor and type of the compiler are optional, but may be specified for tighter WCET estimations. Using a template of the AIS-file is recommended to reduce the manual workload each time a new function is analysed. In these cases only the addresses of the annotations, the bounds of the loops and the conditions have to be changed.

Table 5.2 *Number of annotations for some different functions.*

Function	***Memory Mapping***	***Loop Bounds***	***Processor Rate***	***Compiler Type***	***Total***
l_star12sci_ifc_s_connect()	4	0	1	1	6
l_star12sci_ifc_m_connect()	4	0	1	1	6
l_star12sci_ifc_s_init()	4	0	1	1	6
l_star12sci_ifc_m_init()	4	0	1	1	6
l_star12sci_ifc_s_rx()	4	3	1	1	9
l_star12sci_ifc_m_rx()	4	0	1	1	6
l_star12sci_sch_set()	4	2	1	1	8
l_star12sci_sch_tick()	4	8	1	1	14
l_star12sci_sys_init()	4	0	1	1	6

Table 5.2 illustrates the number of annotations required to perform a general analysis of the functions. The number of mandatory annotations was always four for the memory mapping and between zero and eight for the loop bounds. Totally, the average number of annotations for analysing one function was about seven, including processor rate and type of compiler annotations. Further, annotations for conditions, never executing parts and so on may be set in special cases but are optional. The format of the annotations is described in Section 4.1.1.

If the application is changed and a re-compilation is made, the instructions might get different memory addresses, which have to be changed in the annotations within the AIS-file. It is recommended to make detailed comments in the AIS-file about which part of the source code the annotations are concerned with. This makes the first analysis of a function more time consuming but reduces the workload if the application is re-compiled.

5.1. The Master

Several function calls in the master node and their corresponding WCETs generated by aiT are illustrated in Table 5.3. These WCETs are according to the worst case of an application, where the maximum frame size is eight and the maximum number of latched frames is eight. The three largest of these are the *l_star12sci_sch_tick()*, the *l_star12sci_set()* and the *l_star12sci_ifc_m_rx()* functions. Different analyses with different assumptions have been performed on these functions which are described in the sections below.

MÄLARDALENS HÖGSKOLA

VOLCANO AUTOMOTIVE GROUP

***Table 5.3** WCET for some function calls in the master.*

Function	WCET	
	Cycles	*Time (µs)*
l_star12sci_ifc_m_connect()	25	3.125
l_star12sci_ifc_m_init()	182	22.75
l_star12sci_ifc_m_rx()	224	28
l_star12sci_sch_set()	3612	452
l_star12sci_sch_tick()	6823	853
l_star12sci_sys_init()	10	1.25
l_sys_irq_restore()	13	1.625

5.1.1 The l_star12sci_sch_tick() function

The *l_star12sci_sch_tick()* function drives the communication in the LIN network and may only be called by the master node within the LIN cluster. *l_star12sci_sch_tick()* is called periodically with a time base specified in the current LDF-file and follows a pre-defined schedule for frame transmission. Different schedules can be used and is in that case set with the *l_star12sci_sch_set()* function.

A detailed analysis of the *l_star12sci_sch_tick()* function has been performed in several cases. WCET of its including loops and different execution scenarios have been estimated. Initially, there are eight different loops within the *l_star12sci_sch_tick()* function. These loops have been analysed separately and their execution time is independent of each other.

The iteration bounds of all loops that were executed in the different cases were set to their absolute maximum, apart for two of them. These loops were tested for iterations in the range of one to eight depending on the maximum frame length of the transmitted frames. The frames and their corresponding sizes are defined in the current LDF-file. Changing the number of iterations for these loops, that mange the frame size, describes how the size of the frames influence the entire WCET of the *l_star12sci_sch_tick()* function. One loop was on the other hand tested for iterations in the range of one to ten depending on the number of flags latched to the signals within the transmitted/received frames. The number of flags latched for each signal can be identified in the corresponding private-file.

In some of the cases below there are parts that were manually eliminated by setting them to never executing parts. Some of the loops are within these parts and their loop bounds had therefore no meaning and were set to zero. The loop annotations for the cases below are defined as follow.

loop_0001 - *loop 0x0:0x456f begin max (0-7);*
loop_0002 - *loop 0x0:0x4590 begin max (0-9);*
loop_0003 - *loop 0x0:0x4656 begin max 6;*
loop_0004 - *loop 0x0:0x46af begin max 7;*
loop_0005 - *loop 0x0:0x4780 begin max (0-7);*
loop_0006 - *loop 0x0:0x4737 begin max 7;*
loop_0007 - *loop 0x0:0x4609 begin max 7;*
loop_0008 - *loop 0x0:0x4537 begin max 7;*

All WCETs generated for the *l_star12sci_sch_tick()* function within an application with frames of length one to eight and with one to ten latched flags are illustrated in Table C.2 in Appendix C.

In Table 5.4, the number of cycles and times per iteration of each loop in the *l_star12sci_sch_tick()* function is illustrated. A complete table of the dependencies of cycles and times per iteration is shown in Table C.1 in Appendix C.

Table 5.4 *Cycles and times per iteration for the loops in the l_star12sci_sch_tick() function.*

Loop	*Cycles/Iteration*	*Time/Iteration (μs)*
loop_0001	49	6.125
loop_0002	23	2.875
loop_0003	19	2.375
loop_0004	52	6.5
loop_0005	71	8.875
loop_0006	46	5.75
loop_0007	73	9.125
loop_0008	88	11

The *l_star12sci_sch_tick()* function has been analyzed in a number of different interesting cases under assumption that some specific conditions and constraints are valid. The different cases depend on parameters such as number of frames and signals and are described in the section below. Since these parameters do not relate to each other it is possible to perform these types of measurements.

CASE 1

First of all, the entire WCET of the *l_star12sci_sch_tick()* function was analysed. In this case it was assumed that sporadic and event triggered frames may occur, sleep requests may be received, errors could appear, and that several flags (one to ten) may be latched to signals within the frames. The necessary loop bounds were the only annotations that had to be specified to perform this task. All loops except *loop_0003* were executed in the worst case and the bounds for the eight loops specified above were set to their maximum.

If for example the maximum frame size of the transmitted/received frames within the network was two, the estimated WCETs for the *l_star12sci_sch_tick()* function with varying number of flags are illustrated in Table 5.5. A complete table with cycles and times per iteration for different frame sizes is shown in Table C.2 in Appendix C.

Table 5.5 *WCET of the l_star12sci_sch_tick() function for frame size two with varying number of latched flags.*

Flags	*Frame size*	*Cycles*	*Time (ms)*
1	2	5690	0.712
2	2	5713	0.715
3	2	5736	0.717
4	2	5759	0.720
5	2	5782	0.723
6	2	5805	0.726
7	2	5828	0.729
8	2	5851	0.732
9	2	5874	0.735

10	2	5897	0.738
Difference/Flag		***23***	***~0.003***

The table corresponds to the maximum frame size of two and each row in the table corresponds to the number of flags latched to a signal within these frames. Each latched flag gave one loop iteration more, which took **23 cycles**. Further, comparing WCET of applications containing frames of different sizes but with same number of latched flags, in this case ten, generated a difference of **162 cycles** or **0.021 ms** as illustrated in Table 5.6.

Table 5.6 *WCET of the l_star12sci_sch_tick() function for varying frame sizes with ten latched flags.*

Frame size	***Flags***	***Cycles***	***Time (ms)***
1	10	5735	0.717
2	10	5897	0.738
3	10	6059	0.758
4	10	6221	0.778
5	10	6383	0.798
6	10	6545	0.819
7	10	6707	0.839
8	10	6869	0.859
Difference /Frame size		**162**	**0.02**

WCET of CASE 1

Following parametrical WCET formula could manually be obtained from this case.

$WCET_{CASE\ 1}(cycles) = \#\ Flags * 23 + Framesize * 162 + 5343$

$WCET_{CASE\ 1}(time) = WCET_{CASE\ 1}(cycles)\ /\ Processor\ Rate$

$\#\ Flags \in N$
$Framesize \in [1..8]$
Processor Rate = 8 Mhz = 8 000 000 Hz

Conclusions of CASE 1

The conclusion of this first case is that each flag latched to a signal within a transmitted/received frame increase the entire WCET of the *l_star12sci_sch_tick()* function with additional **23 cycles** or **0.003 ms** = **3 µs** and that each frame size increase the entire WCET of the function with **162 cycles** or **0.021 ms = 21 µs.** Further, as mentioned in the Section 2.3 about the LIN protocol, the LIN bus operates at a speed up to 20 kbps. This leads to that one bit takes **50 µs** to transmit and that **3 µs** corresponds to a transmission of a **0.06** bit part and that **21 µs** corresponds to a transmission of a **0.42** bit part of a frame.

CASE 2

In the next scenario of the *l_star12sci_sch_tick()* function it was assumed that no event triggered or sporadic frames were transmitted. The parts of the code that managed these types of frames have therefore been eliminated and were not decoded by aiT. Not analysed code snippets may be manually defined in aiT by the annotation "is never executed", as described in Section 4.1.1. The following settings eliminate the parts that manage event triggered and sporadic frames. The specified addresses are generic addresses in the not executed basic blocks.

snippet 0x0:0x4537 is never executed; # manage received event triggered frames
snippet 0x0:0x4609 is never executed; # manage received event triggered frames

snippet 0x0:0x4737 is never executed; # manage received event triggered or a sporadic frames

The generated WCETs for the *l_star12sci_sch_tick()* function, with frames of maximum size of two and varying number of flags between one to ten, are illustrated in Table 5.7. A complete table with cycles and times per iteration for different frame sizes is shown in Table C.3 in Appendix C.

Table 5.7 *WCET of the l_star12sci_sch_tick() function for frame size two with varying number of latched flags.*

Flags	*Frame size*	*Cycles*	*Time (ms)*
1	2	2352	0.294
2	2	2375	0.297
3	2	2398	0.300
4	2	2421	0.303
5	2	2444	0.306
6	2	2467	0.309
7	2	2490	0.312
8	2	2513	0.315
9	2	2536	0.317
10	2	2559	0.320
Difference/Flag		**23**	**0.003**

Further, the WCET of an application that contains different frames sizes, but with the same number of latched flags, in this case ten, is illustrated in Table 5.8.

Table 5.8 *WCET for the l_star12sci_sch_tick() function for varying frame sizes with ten latched flags.*

Frame size	*Flags*	*Cycles*	*Time (ms)*
1	10	2397	0.300
2	10	2559	0.320
3	10	2721	0.341
4	10	2883	0.361
5	10	3045	0.381
6	10	3207	0.401
7	10	3369	0.422
8	10	3531	0.442
Difference/Frame size		**162**	**0.02**

WCET of CASE 2

Following parametrical WCET formula could manually be obtained from this case.

$WCET_{CASE\ 2}(cycles) = \#\ Flags * 23 + Framesize * 162 + 2005$
$WCET_{CASE\ 2}(time) = WCET_{CASE\ 2}(cycles)\ /\ Processor\ Rate$

$\#\ Flags \in N$
$Framesize \in [1..8]$
Processor Rate = 8 Mhz = 8 000 000 Hz

Conclusions of CASE 2

The difference in execution time for latched frames and different frame sizes was the same but the entire WCETs were in this case about half the times in comparison with CASE 1. That depended on that the analysed code part was smaller in this case. The conclusion of this case was that networks that contain event triggered and sporadic frames had longer WCETs then networks without these types of frames. That depended on that bigger code snippets were used for managing frames and signals in these cases.

CASE 3

In the third analysis case it was assumed that no event triggered or sporadic frames were enabled, no sleep requests could be transmitted and that no errors occurred. This lead to an elimination of more code parts then in the previous cases, to getting a smaller part to analyse. The loop bounds were set manually and the seven code parts that managed event triggered and sporadic frames, sleep requests, and errors were eliminated and not decoded by aiT.

```
snippet 0x0:0x44cd is never executed;     # manages sleep requests
snippet 0x0:0x4537 is never executed;     # manages event triggered frame
snippet 0x0:0x4609 is never executed;     # manages event triggered frame
snippet 0x0:0x4656 is never executed      # manages sleep requests
snippet 0x0:0x4737 is never executed;     # manages event triggered or sporadic frame
snippet 0x0:0x44e2 is never executed;     # manages status flags not set frequently enough
snippet 0x0:0x4614 is never executed;     # manages status flags not set frequently enough
```

The generated WCETs for the *l_star12sci_sch_tick()* function within an application with a maximum frame size of two are illustrated in Table 5.9. A complete table with cycles and times per iteration for different frame sizes is shown in Table C.4 in Appendix C.

Table 5.9 *WCET for the l_star12sci_sch_tick() function with varying number of latched flags.*

Flags	*Frame size*	*Cycles*	*Time (ms)*
1	2	1164	0.146
2	2	1164	0.146
3	2	1164	0.146
4	2	1164	0.146
5	2	1164	0.146
6	2	1164	0.146
7	2	1164	0.146
8	2	1164	0.146
9	2	1164	0.146
10	2	1164	0.146
Difference/Flag		**0**	**0**

This table corresponds to the maximum frame size of two and each row in the table corresponds to the number of flags latched to a signal within these frames. Since the loop that managed the flags was in a not executed code snippet and therefore not decoded by aiT, the iterations did not increase the entire WCET. Comparing WCET of the *l_star12sci_sch_tick()* function within an application containing different frames sizes but with the same number of latched flags, generated a differential of **85 cycles** or **0.011 ms** which is illustrated in Table 5.10.

Table 5.10 *WCET for the l_star12sci_sch_tick() function for varying frame sizes with ten latched flags.*

Frame size	*Flags*	*Cycles*	*Time (ms)*
1	10	1079	0.135
2	10	1164	0.146
3	10	1249	0.157
4	10	1334	0.167
5	10	1419	0.178
6	10	1504	0.188
7	10	1589	0.199
8	10	1674	0.210
Difference/Frame size		**85**	**0.011**

WCET for CASE 3

Following parametrical WCET formula could manually be obtained from this case.

$WCET_{CASE\,3}(cycles) = Framesize * 85 + 994$ $WCET_{CASE\,3}(time) = WCET_{CASE\,3}(cycles) / Processor\ Rate$	$Framesize \in [1..8]$ Processor Rate = 8 Mhz = 8 000 000 Hz

Conclusions of CASE 3

In this third case, there was no difference in time between different numbers of latched flags, within a frame. This is because *loop_0002* was in a not executed code snippet and therefore not decoded by aiT. However, the frame size influence the entire WCET of the *l_star12sci_sch_tick()* and each size increased the entire WCET of the function with **85 cycles** or **0.011 ms = 11 µs.** The entire WCETs were in this case about half the times in comparison with CASE 2. This was because more parts were eliminated and not decoded by aiT.

CASE 4

In this case a type of jitter of a subpart of the *l_star12sci_sch_tick()* function was measured. The part was from the call of the function until the master sends out a break character. The jitter refers to the variability in time taken to execute the specific part of the *l_star12sci_sch_tick()* function i.e. the variability between the BCET and the WCET. aiT was not able to calculate BCET itself so the WCET path was forced manually to go the shortest path.

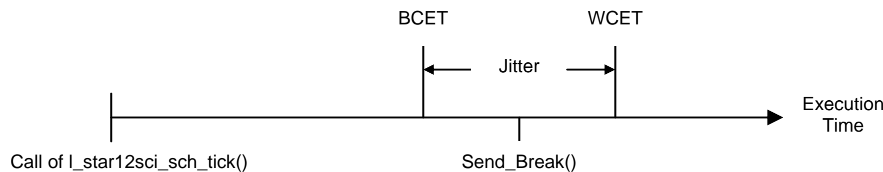


Figure 5.1 *The jitter of a specific part of the l_star12sci_sch_tick() function.*

First the start point of the analysis had to be set to the call of the *l_star12sci_sch_tick* routine which was found automatically by aiT. Then the specific end point of the analysis had to be defined and the parts after this end point set to never execute.

```
end 0x0:0x47f9;                            # ends analysis after a break has been send out
snippet 0x0:0x47fb is never executed;      # the code after send break is not executed
```

After the part of the code to analyse had been defined, the analysis could start.

Example 1 (WCET)

In this example it is assumed that all parts of the code between the specified start and end points may execute. The only limitations were the loop bounds which were set to their absolute maximum. The size of the frames was assumed to be maximum eight and the number of latched flags ten.

WCET of the subpart of the l_star12sci_sch_tick() function.

Frame size	*Flags*	*Cycles*	*Time (ms)*
8	10	6810	0.852

Example 2 (BCET)

In this example it was assumed that event triggered or sporadic frames may not occur. The three parts that manage them were therefore eliminated to decrease the analysed part of the code.

snippet 0x0:0x4537 is never executed; # manage event triggered frames
snippet 0x0:0x4609 is never executed; # manage event triggered frames
snippet 0x0:0x4737 is never executed; # manage event triggered or sporadic frames

The loop bounds were set as in the first example (maximum) which means that the largest possible frame size was eight and that number of latched flags was ten.

BCET of the subpart of the l_star12sci_sch_tick() function.

Frame size	*Flags*	*Cycles*	*Time (ms)*
8	10	3472	0.434

Conclusions of CASE 4

The conclusion of this case was that the jitter which was the difference between WCET and BCET of the executed part of the *l_star12sci_sch_tick()* function with frames size eight and ten latched frames was **0.852 – 0.434 = 0.418 ms.** This result is somewhat unreliable since the execution has been forced to go the shortest way and then been treaded as BCET. aiT may sometimes overestimate the timing values for instructions (which is safe when calculating the WCET), but might lead to an overestimation of the BCET value.

CASE 5

In this case we examine how different combinations of unconditional and diagnostic frames influence the entire WCET. It was assumed that event triggered or sporadic frames may not occur so the parts that managed them were eliminated. The loop bounds are then set to their maximum. When the necessary loop bound had been specified and the necessary parts been eliminated, four different cases were tested out. The frame types were tested by two conditions that each consists of two parts, separated by an OR operator. This condition is set to always true or always false depending on which frame combination to test.

An unconditional frame followed by another unconditional frame

In the first case it was assumed that an unconditional received frame was followed by another unconditional transmitted frame within the master. The first part of the first condition tested if an unconditional frame was received and was set to always true while the second part that tests if a diagnostic frame was received was set to always false.

condition 0x0:0x4547 is always true; # received unconditional frame
condition 0x0:0x4550 is always false; # received diagnostic frame

The first part of the second condition tested if an unconditional frame was transmitted and was set to *always true* while the second part of the condition tests if a diagnostic frame was transmitted and set to *always false*.

condition 0x0:0x47f4 is always true; # transmitted unconditional frames
condition 0x0:0x4757 is always false; # transmitted diagnostic frame

With these settings above aiT generated a WCET of **3315 cycles** equal to **0.415 ms**.

An unconditional frame followed by a diagnostic frame

Then it was assumed that a diagnostic transmitted frame was followed an unconditional received frame. The first part of the first condition was set to always true while the second condition was set to always false.

condition 0x0:0x4547 is always true; # received unconditional frame
condition 0x0:0x4550 is always false; # received diagnostic frame

The first part of the second condition was in this case instead set to always false while the second part was set to always true.

condition 0x0:0x47f4 is always false; # transmitted unconditional frame
condition 0x0:0x4757 is always true; # transmitted diagnostic frame

In this case aiT generated a WCET of **3308 cycles** equal to **0.414 ms**.

A diagnostic frame followed by an unconditional frame

In this third case it was assumed that a received diagnostic frame was followed by an unconditional transmitted frame. The first part of the first condition was set to always false while the second part was set to always true.

condition 0x0:0x4547 is always false; # received unconditional frame
condition 0x0:0x4550 is always true; # received diagnostic frame

The first part of the second condition was set to always true while the second part was set to always false.

condition 0x0:0x47f4 is always true; # transmitted unconditional frame
condition 0x0:0x4757 is always false; # transmitted diagnostic frame

In this case aiT generated a WCET of **2756 cycles** equal to **0.345 ms**.

A diagnostic frame followed by another diagnostic frame

In this forth case it was assumed that a received diagnostic frame was followed by a diagnostic transmitted frame. Here like in the third case the first part of the first condition was set to always false while the second condition was set to always true.

condition 0x0:0x4547 is always false; # received unconditional frame
condition 0x0:0x4550 is always true; # received diagnostic frame

The first part of the second condition is set to always false while the second part is set to always true.

condition 0x0:0x47f4 is always false; # transmitted unconditional frame
condition 0x0:0x4757 is always true; # transmitted diagnostic frame

In this case aiT generated a WCET of **2749 cycles** equal to **0.344 ms**.

Conclusions of CASE 5

The conclusion of this case is that the combination of two diagnostic frames transmitted after each other generates the shortest WCET while the combination of two unconditional frames transmitted after each other generates the largest WCET as illustrated in Table 5.11. That depends on that the master do not process so many diagnostic frames in the *l_star12sci_sch_tick()* function.

Table 5.11 *WCET for different frame type combinations for frame size eight.*

Combination	*WCET*	
	Cycles	*Time (ms)*
unconditional – unconditional	3315	0.415
unconditional – diagnostic	3308	0.414
diagnostic – unconditional	2756	0.345
diagnostic – diagnostic	2749	0.344

5.1.2 The l_star12sci_m_rx() function

The largest function within the master, the *l_star12sci_m_rx()* function, is called when the interface has received one character of data. It is a straight forward function containing no loops. The *l_star12sci_m_rx()* function took **224 cycles** equal to **28 µs** to execute in worst case.

5.2 The Slave

Also the slaves have a set of API functions. Several of the function calls and their corresponding WCETs generated by aiT are illustrated in Table 5.12.

Table 5.12 *WCET of some function calls in the slave.*

Function	*WCET*	
	Cycles	*Time (µs)*
id_received()	210	26.25
init_target()	36	4.5
l_callout_response_slave()	954	120
l_header_to_frame_star12sci()	37	4.625
l_star12sci_ifc_s_connect()	25	3.125
l_star12sci_ifc_s_init()	159	19.875
l_star12sci_ifc_s_rx()	1323	166
l_star12sci_sys_init()	10	1.25

5.2.1 The l_star12sci_s_rx() function

The *l_star12sci_s_rx()* function is the largest and most interesting function within the slave and is called when the slave node receives one character of data. Further, it calls two other functions called *id_received()* and *frame_received()* that contain some loops.

Loops

The *id_received()* function is called when a valid identifier is received in the slave and indicates that it is a normal frame. A correctly received checksum byte is on the other hand a sign that a correct frame has been received and the function *frame_received()* is then called. The *id_received()* function contains one loop, illustrated in Figure 5.2, that executes if an unconditional, diagnostic or event triggered frame will be handled for transmission. This is a for-loop that iterates maximum eight times but actually as many times as the size of the longest frame in the current network, specified in the current LDF-file. The iteration bound for the loop in the *id_received()* function was set with the following annotation.

loop_0000 - *loop 0x0:0x4212 begin max 1;*   # largest frame size

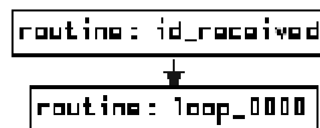


***Figure 5.2** The id_received routine and its including loop.*

The *frame_received()* function on the other hand contains two loops. The first is a for-loop that executes if a complete unconditional or diagnostic frame is successfully received. This loop iterates maximum eight times but should be set to the maximum frame size of the longest frame in the current application. The second loop is a while-loop that iterates as many times as the number of flags, latched to the signals in the received frame. This information may be fetched from the slave's private-file. The frame size and the number of latched flags within one frame may vary and influence the number of iterations of the loops. The iteration bounds of the loops within the *frame_received()* function was set with following annotations.

loop_0000 - *loop 0x0:0x4291 begin max 1;*   # largest frame size
loop_0001 - *loop 0x0:0x42b7 begin max 0-9;*   # maximum number of latched flags

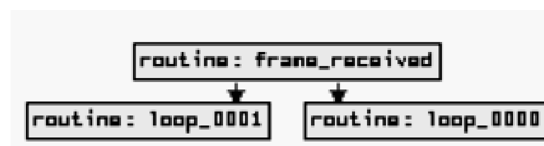


***Figure 5.3** The frame_received routine and its including loops.*

In Table 5.13 cycles and times per iteration of the loops in the *l_star12sci_s_rx()* function are illustrated. A complete table of the dependencies of cycles per iterations and time per iteration is shown in Table D.1 in Appendix D.

***Table 5.13** Cycles and times per iteration for loops in the l_star12sci_s_rx() function.*

Loop	*Cycles/Iteration*	*Time/Iteration (µs)*
id_received - loop_0000	40	5.0
frame_received - loop_0000	36	4.5
frame_received - loop_0001	49	6.125

The *l_star12sci_s_rx()* function has like the *l_star12sci_sch_tick()* been analyzed in a number of different interesting cases under assumption that some specific conditions and constraints is valid. The different cases depend on parameters such as number of frames and signals and are described in the section below. Since these parameters do not relate to each other it is possible to perform these types of measurements.

CASE 1

Running the *l_star12sci_s_rx()* function without any limitation, only loop annotations for the loops in the *id_received()* and the *frame_received()* function mentioned above must be set.

If the maximum frame size of frames within a network was two, the estimated WCETs for varying number of flags are illustrated in Table 5.14. WCET for the *l_star12sci_s_rx()* function for all possible frame sizes and varying number of flags between one and ten are illustrated in Table D.2 in Appendix D.

***Table 5.14** WCET for l_star12sch_s_rx() for frame size two with varying number of latched flags.*

Frame size	*Flags*	*Cycles*	*Time (ms)*
2	1	1323	0.166
2	2	1372	0.172
2	3	1421	0.178
2	4	1470	0.184
2	5	1519	0.190
2	6	1568	0.196
2	7	1617	0.203
2	8	1666	0.209
2	9	1715	0.215
2	10	1764	0.221
Difference/Flag		**49**	**0.006**

The table corresponds to the maximum frame size of two and each row in the table corresponds to the number of flags latched to a signal within this frame. *loop_0001* iterates one more time for each latched flag and each iteration took **49 cycles** or **0.006 ms**. Comparing WCET for the *l_star12sci_s_rx()* function in an application with frames with different size of frames but with the same number of flags, generated a differential of **36 cycles** or **0.004 - 0.005 ms,** which is illustrated in Table 5.15.

***Table 5.15** WCET for the l_star12sci_s_rx() function for varying frame sizes with ten latched flags.*

Frame size	*Flags*	*Cycles*	*Time (ms)*
1	10	1728	0,216
2	10	1764	0,221
3	10	1800	0,225
4	10	1836	0,230
5	10	1872	0,234
6	10	1908	0,239
7	10	1944	0,243
8	10	1980	0,248
Difference/Frame size		**36**	**0.004(5)**

WCET of CASE 1

Following parametrical WCET formula could manually be obtained from this case.

$$WCET_{CASE\ 1}(cycles) = \#\ Flags * 49 + Framesize * 36 + 1202$$

$$WCET_{CASE\ 1}(time) = WCET_{CASE\ 1}(cycles)\ /\ Processor\ Rate$$

$\#\ Flags \in N$
$Framesize \in [1..8]$
Processor Rate = 8 Mhz = 8 000 000 Hz

Conclusion of CASE 1

One conclusion of this first slave case is that each flag latched to a signal within a frame increased the entire WCET of the *l_star12sci_s_rx()* function with additional **49 cycles** or **0.006 ms = 6 µs**. Further each frame size in this case increased the entire WCET of the function with **36 cycles** or ~**0.005 ms = 5 µs**.

CASE 2

In this case the influence on the *l_star12sci_s_rx()* function of number of frames within the current network is analysed. Two loops in functions within the slave iterates as many times as the number of frames. The first loop is *loop_0000* in the *ld2_response_slave* routine. This routine further depends on *loop_0001* in the *ld2_prepare_response* routine and was always set to six and illustrated in Figure 5.4.

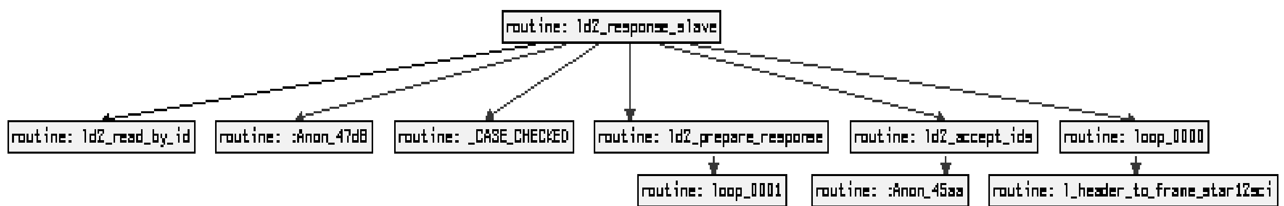


***Figure 5.4** The ld2_response_slave routine and its including routine calls.*

The WCETs for the *ld2_response_slave* routine for different numbers of frame is illustrated in Table 5.16.

Table 5.16 *WCET of the ld2_response_slave routine for different number of frames.*

Number of frames	*Cycles*	*Time (µs)*
1	601	75.125
2	746	93.25
3	919	115
4	1092	137
5	1265	159
6	1438	180
7	1611	202
8	1784	224
Difference/Frame	**173**	**22**

The second loop that depends on the number of frames, *loop_0000*, was in the *ld2_init_slave()* function. The *l_callout_init_slave* routine was set to start point as illustrated in Figure 5.5.

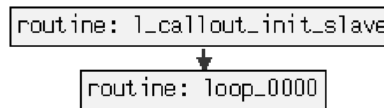


Figure 5.5 *The l_callout_init_slave routine and its loop call.*

The WCETs for the *l_callout_init_slave* routine for different numbers of frame is illustrated in Table 5.17.

Table 5.17 *WCET of the l_callout_init_slave routine for different number of frames.*

Number of frames	*Cycles*	*Time (µs)*
1	25	3.125
2	50	6.25
3	75	9.375
4	100	12.5
5	125	15.625
6	150	18.75
7	175	21.875
8	200	25
Difference/Frame	**25**	**3.125**

WCET of CASE 2

Following parametrical WCET formulas could manually be obtained from this case.

$WCET_{CASE\ 2}(cycles) = \#\ Frames * (173+25) + 400$

$WCET_{CASE\ 2}(time) = WCET_{CASE\ 1}(cycles)\ /\ Processor\ Rate$

Frames ∈ N
Processor Rate = 8 Mhz = 8 000 000 Hz

Conclusion of CASE 2

The conclusion of this case is that increase the frame size by one, increases WCET of the *ld2_response_frame* routine by **173 cycles** equal to **22 µs** and the *l_callout_init_slave* routine with **25 cycles** equal to **3.123 µs**. The entire WCET of the *l_star12sci_s_rx()* by increase the frame size by one was therefore **173 + 25 = 198 cycles** or **22 + 3.123 µs**.

CASE 3

In this case a specific measure on the *l_star12sci_s_rx()* function has been done, the interested thing was which WCET path that was taken from that one identifier was received in the slave until it starts to send out its first byte. When a specific part like this will be analysed, first of all the desirable start and end points must be specified, unless aiT found these points automatically. The start point was defined by writing its address in the textbox at the right of *Start at*: in the program. An annotation could also have been done in the AIS-file, which creates a new routine with this address as start point. For example:

entry 0x0:0x4365;

Then the specific end point where aiT should stop decoding must be specified in the AIS-file. For example with an end annotation of the following format:

end 0x0:0x439a;

After the start and end points to perform measure between had been specified, the analysis can start. However, if a small part of an entire program should be analysed and there is a short path from the entry point to the end point, it is not sure the entire WCET path goes through this desirable sub part. aiT generates the WCET of the entire program but this is not what was desirable in this case. The interested thing was the time from a reception of one specific byte to sending another, when the slave managing normal frames. To do that we specified all conditions that tested the condition of normal frames or not to either "always true" or "always false" depending on if the WCET flow should follow the true or false edges in the call graph. This was simply done by perform the CFG action, look at the call graph and depict which value of the condition that goes the desirable way. The green arrows in the call graph go to the target if the condition is true while the red edges in the graph goes to the target if the condition is false.

Three conditions are specified in this case:

```
condition 0x0:0x4365 is always true;     # test if a normal frame identifier is received
condition 0x0:0x437f is always false;    # test if a frame is correct
condition 0x0:0x438e is always false;    # test if a normal frame identifier is received
```

When an entry point, an end point and the path through the control-flow graph by different condition value have been specified, the size of the analysed code snippet decreased drastically. To see the big difference, first consider the control-flow graph without any restrictions as illustrated in Figure 5.6.

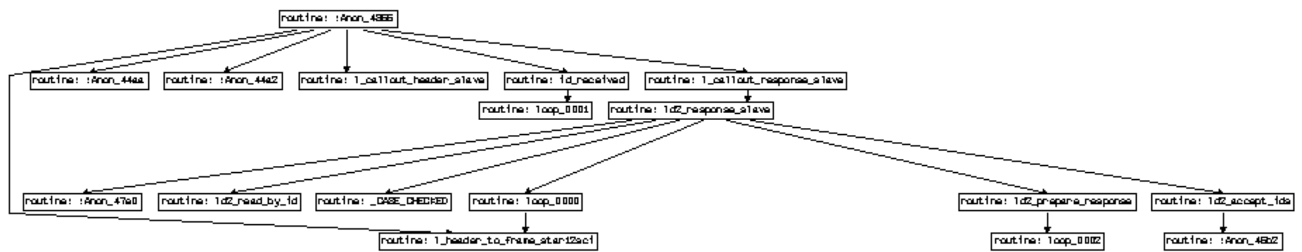


Figure 5.6 *The entire call graph of the analysed part.*

Here only the entry and end points have been specified and aiT generated in this case a WCET of **1174 cycles** equal to **0.147 ms**.

After the conditions had been specified, the call graph above was reduced to the new graph illustrated in Figure 5.7. Here the WCET of interest was generated and reduced to **325 cycles** equal to **40.625 µs**.

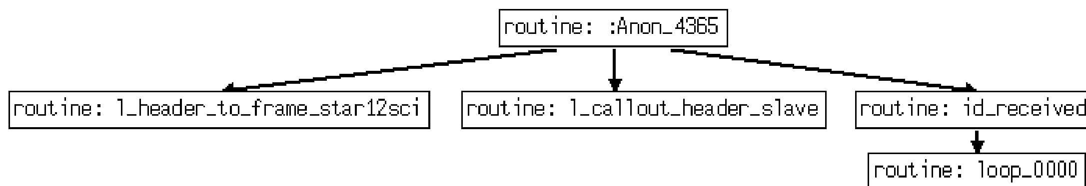


Figure 5.7 *Reduced control-flow graph of the analysed part.*

Conclusion of CASE 3

The conclusion of this case is that WCET for a specific code part may be generated by aiT. But in this case a specific entry point as well as end point of the analysis has to be specified together with extra annotations that force the calculation to go through the particular execution path. If these annotations are not given, the WCET of the code may be calculated from code parts not of interest. Furthermore, it is much simpler to get a view of the analysed part in Figure 5.7 then in Figure 5.6.

CASE 4

In this case we investigated how different number of frames in the application influence WCET of the *l_header_to frame_star12sci()* function which is called from the *l_star12sci_s_rx()* function. The routine does not contain any loops so the only annotations that had to be given in the AIS-file were the mandatory annotations that were described in Section 4.1.1.

To add a frame to the current network that should transmit a specific signal, the signal had to be defined under the signal part and the frame be defined under the frame part in the current LDF-file. The frame must also be defined in the slave part under the node attributes part as a configurable frame [Appendix A]. After the change in the LDF-file a recompilation had to be done to generate a new .abs-file. Since only memory mappings annotations were set, no changes in the AIS-file had to be done.

First the original LTP-file with its signal and frame definitions was used as specified in Appendix A. Then additional frames were added in the LDF-file and the application was re-compiled for each number of frames. The WCETs for the *l_header_to frame_star12sci()* routine for different numbers of frame is illustrated in Table 5.18.

Table 5.18 *WCET of the l_header_to frame_star12sci() function with different number of frames.*

Number of frames	*Cycles*	*Time (μs)*
1	25	3.125
2	31	3.875
3	37	4.625
4	43	5.375
5	49	6.125
6	55	6.875
7	61	7.625
8	67	8.375
9	73	9.125
10	79	9.875
Difference/Flag	**6**	**0.75**

Conclusion of CASE 4

One conclusion of this case is that the each number of frames that are added to the LDF-file, increase the entire WCET of the *l_header_to frame_star12sci()* function with additional **6 cycles** or **0.75 μs.**

6 Measurements with Oscilloscope

To know if the current WCET tool, *aiT,* is useful for VCT or actually useful at all we have to get some traditionally measured times to compare with. VCT does not have any previous performed measurements on the LTP applications so these have to be generated manually. In this work, times for some of the main functions that have been analysed by *aiT* have also been measured with an oscilloscope. This chapter describes the performed oscilloscope measurements and illustrates some examples of the estimated WCETs.

6.1 The hardware configuration

To perform oscilloscope-timing measurements a hardware configuration has to be set up. This contains of a development board containing a Motorola STAR12 microcontroller with a 16 MHz oscillator and a serial debug interface that connects the microcontroller to one COM-port of the computer. Further, a LINspector that controls the communications of the bus and a transformer that regulate the voltage to 12 V is connected to the microcontroller. The obtained hardware configuration is illustrated in Figure 6.1.

The microcontroller has channels connected to the CPU pins, which can be used as either input or output. To perform measurements a pin used for input/output (I/O) at the STAR12 processor must be set. The MC9S12DP256 incorporates eleven ports which may be used for general-purpose I/O. Each of these ports consists of a data register which can be read and written and a data direction register which controls the direction of each pin. In this work, Port P that contains pins PP0-PP7 was decided to be used for the I/O purpose.

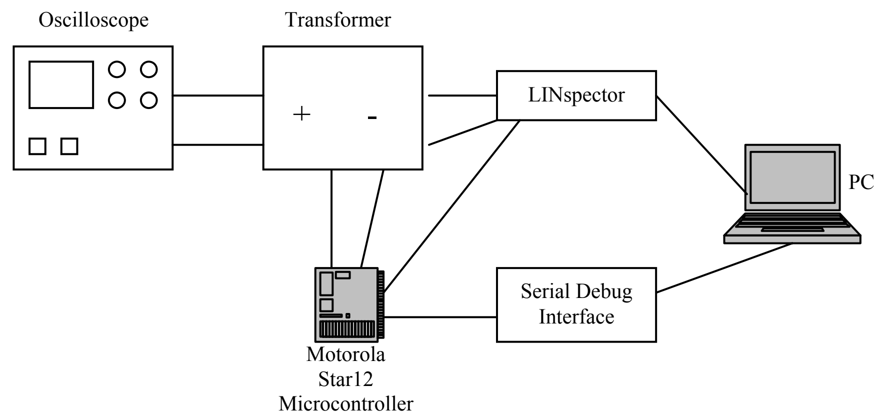


Figure 6.1 *The hardware configuration with a Motorola STAR12 microcontroller.*

The I/O register for port P (PTP) is set to one which means that all associated I/O pins are set to one and that a read returns the value of the port register. Further, the offset address of the register which is 0x018 must be specified in the source code.

The Data Direction Register for port P (DDRP) is also set to one which configures each port P pin as output (if the register is set to zero the associated pins are configured as input). Furthermore, the offset address of this register, which is 0x01A, must be specified. To use the offset addresses for the registers of port P, the base address of the port, which is 0x240, must be set. After these register settings have been specified the oscilloscope measurements can start. The offset and base addresses are found in [Mic00] and are defined in the source code with the following definitions.

```
#define PORT_BASE_ADDRESS 0x240
#define DDRP(n) (*((l_u8*)(PORT_BASE_ADRESS + 0x1a)) = (n))
#define PTP(n) (*((l_u8*)(PORT_BASE_ADRESS + 0x18)) = (n))
```

To measure one specific function in isolation the PTP register was configured as output by setting it to one with PTP(1), before the call of the measured function. After the call of the function the PTP register was then reset to zero with PTP(0).

The Metrowerks development environment was used to create a HCS12 stationary project with the specified settings. First the processor was set to HCS12, then the target interface was set to Motorola Motosil Target Interface and finally the derivative was set to MC9S12DP256B. The current application is then built with the same Hiware HC12 Compiler and SmartLinker from Metrowerks, Inc. that was used in the static analyses in Chapter 5, to generate an executable to analyse.

The next step was to load the current LDF-file into the LINspector and start it. Finally, the *Metrowerks True-time Simulator and Real-time debugger* was opened from the created Metrowerks project and started. Now the

oscilloscope shows the generated variation in voltage as a wave on its display. The time could be obtained by moving the two cursors to the start and end points of the wave.

If the application was re-built the debugger does not discover that automatically so we manually had to erase the programmed flash memories and reload the application into the debugger.

The two most interesting functions within the LIN API are the *l_star12sci_sch_tick()* function and the *l_star12sci_m_rx()* function. These are the two functions that were analysed with aiT and the two functions which were measured with the oscilloscope. Some experiments were done on the other functions but they were too small to generate some visible waves on the oscilloscope.

The *l_star12sci_tick()* function generated an execution time of about **58 µs** if the LINspector was started before the debugger and **74 µs** otherwise**.** The *l_star12sci_m_rx()* function on the other hand generated a execution time of about **24 µs** if the LINspector was started before the debugger and **15 µs** otherwise. That depends on that the LINspector emulate a slave node and that the master stops to process the frames when the slave not answers.

It was hard to measure the exact times that the oscilloscope generated since it took time for the rising edge to go up and the falling edge to go down. That depended on the resistance and the practical capacitance within the system which interfere the voltage to immediately pass through it. But if the generated wave is read correctly the generated time should be correct.

7 Result

In this chapter the result of the work performed in Chapter 5 and Chapter 6 above is present.

7.1 Comparison of the two analysis methods

The WCETs that were generated by aiT in the different scenarios may be very useful for VCT, but only if it could be proven that these times are certain. Because of that, the WCETs generated by the static analyses with aiT were compared with the times generated by the manual measurements with the oscilloscope. Unfortunately, due to shortage of time and insufficient knowledge about the method, the dynamical measurements were not performed thorough enough for detailed comparisons.

The two largest of the functions within LTP, *l_star12sci_sch_tick()* and *l_star12sci_m_rx()*, were analysed by the both methods and the results is illustrated in Table 7.1.

A simple master-slave example that only contains one master and one slave node has been used to perform the dynamic analysis. The LDF-file for this example is illustrated in Appendix A and the private files corresponding to these nodes are illustrated in Appendix B.

A similar configuration with the same number of frames and flags has been used to generate the times with the static analysis tool.

Table 7.2 *Comparison between static and dynamic time analyses.*

Function	*Static analysis (µs)*	*Dynamic analysis (µs)*
l_star12sci_sch_tick()	233	74
l_star12sci_m_rx()	28	24

The table shows that the generated times for the *l_star12sci_sch_tick()* function were quite different, which depends on two main factors. Firstly, it was uncertain that the generated wave on the oscilloscope corresponds to the WCET of the measured function. Secondly, it was hard to set correct annotations in aiT, which further lead to unreliable estimations of the WCET. The *l_star12sci_m_rx()* function on the other hand generates more similar times by the two methods, depending on its more straightforward properties. This function does not require any manual annotations, except the mandatory annotations for memory-mappings that were mentioned in Section 4.1.1. In this case it was certain that the worst-case execution way through the program was generated by aiT.

As we can see, it is difficult to get comparable times by the two methods. There are two possibilities to generate good times for comparison, but both are hard to perform in a satisfactory way. The first possibility is to force aiT, by correct annotations, to execute the same way during the program as it did during the measurement with the oscilloscope. The second possibility is to force the oscilloscope measurement to execute the same way during the program as it did when the worst case was generated by aiT.

To get usable WCETs from aiT, a detailed knowledge about the analysed applications was required from the user, who must know what will happen when running a specific program. This was a time consuming work for a testing person who had not been participated in writing the code or developing the system.

The aiT environment was quite simple to set up depending on that the analysis tool together with the current compiler and linker was the only things that were required. Setting up the environment with the real hardware configuration was harder and took more workload to perform by someone who was unfamiliar with these types of environments.

Performing the WCET estimations for an application with aiT took different amount of time mainly depending on the structure of the analysed code. Straightforward code snippets were faster to analyse while more complex code snippet could be both time consuming and error prone. This is because more annotations are required before a WCET estimation could be generated by aiT. It took a lot of time to achieve enough knowledge about how and which annotations that had to be set to get the best result. The measurements with the oscilloscope were on the other hand simpler, depending on its more straightforward way. It was only to start when the environment had been set up and the required code been added into the source code. When using aiT, hundreds of different annotations may be set, that generate different results. It was therefore hard to assure that the correct annotations in each case had been set and that correct result had been achieved.

However, if the right annotations are set and the WCET is assumed to be certain, aiT is more advantageous compared to the oscilloscope since aiT may show more detailed information about the analysed program. For example, the WCET path that is taken during the execution, how much time it spends in the basic blocks and so on. aiT generates exact times in microseconds or in milliseconds depending on the size of the entire WCET. The measurements with the oscilloscope were simpler. The wave on the display corresponds to the generated time, but this value was on the other hand more uncertain, even if you got the result in microseconds. Furthermore, it was hard to get some visible waves on the oscilloscope for the smaller functions. With aiT the smaller functions were easier to analyse depending on the requirements of less annotations.

8 Conclusions

The task was to examine if static time analysis of WCET could be integrated in VCT's development environment. The first main conclusion of this study is that the current analysis tool, aiT, is possible to use on VCT's applications to estimate WCETs.

This work was performed under a limited time period. Depending on the lack of time, it was impossible to evaluate the whole of VCT's system and therefore only a part of the complete system was included in the study. Since the LIN Target Package (LTP) (see Section 2.4.2) is smaller and less complex than the Volcano Target Package (VTP) (see Section 2.4.1), LTP was chosen to be the part for analysis. It took a lot of workload only to start the analyses depending on that good knowledge about the analysed programs and aiT was required. Manual annotations had to be set by the user who had to have good knowledge about how the annotations should be used and how to get as precise analyses as possible.

Most workload was required for setting the iteration bounds for all the loops that aiT did not found automatically. The workload for the analysis increased with the amount of code, depending on the increment of annotations that were required. The testing part of software development is usually not performed by the same person who has written the code and it is therefore necessary to have a good contact with the programmer during the analyses.

Furthermore, it was quite easy to perform WCET estimations of applications, if some similar measurement already had been done.

It was hard to know if correct annotations were set and if correct results were generated. The environment for the oscilloscope measurements were simpler to set up, maybe because VCT was more familiar with these types of environments for measurements. The aiT environment were on the other hand not familiar to VCT, the only support to get here was with the structure of the code, to set correct loop bounds annotations etc. The occurring problems with the aiT tool, was solved by e-mail support from the company AbsInt. This support was gratefully since the replay times were short and the answers very detailed.

One thing that was missed among the annotations, during the work, was to set one snippet to never execute if and only if one previous code snippet has executed. There were right now, no annotations of that kind.

Nested loops were only found in one of the analysed functions and aiT was enabling to manage this. Single loops were more common, but aiT was not able to found their loop bounds automatically. This was the biggest drawback with the tool depending on that each loop bound needed to be determined and set manually. If the analysed piece of code was large, this was a very time consuming and error prone work. If the loop bounds would be found automatically the workload would be reduced dramatically. The tool was however helpful and easy to use on smaller and uncomplicated code parts which required less annotations.

One other drawback with the tool is its compiler dependency. In this work the Hiware Compiler and SmartLinker from Metrowerks Inc. was used but VCT use many other commercial compilers. Among these are the Cosmic compiler from Cosmic and the IAR compiler from IAR systems. Different versions of aiT would be required by VCT if they would like to integrate the tool in their development environment. This would be an expensive solution, if the desirable versions that are required are available at all. Right now, aiT supports the Hiware compiler from Metrowerks, the Cosmic compiler from Cosmic, the TI compiler from Texas Instruments and the DiabData compiler from WindRiver. Furthermore, VCT is using many new processors and WCET tools for these processors may not be developed in time.

The employees at VCT had a positive attitude towards aiT. They thought it would be very interesting to see how good the results from the tool were and of course if it could be proven that the generated WCETs of their applications were certain. The results may in that case be very valuable for VCT. It would practically be possible to integrate the tool in VCTs current tool chain but a more detailed work would then be required, to prove that the WCETs are certain. A discussion would also be taken to investigate how and for what VCT are going to use the tool. What kind of work in their development that may be automatically done. Many of the desirable times that VCT was interested in were about jitter, which is not the purpose of aiT to measure.

There are some risk factors by using static analyses to estimate WCETs of VCTs applications. That depends on that this area still is a research front with much to do before safe and tight times could be assured. The drawbacks and advantage for the static and dynamic analysis are illustrated in Table 8.1.

Table 8.1 *Advantages and drawbacks for dynamic timing analysis.*

Dynamic analysis	
Advantages	***Drawbacks***
Inexpensive	Time Consuming Error Prone Target machine required Program execution required Undetailed results Difficult for small code snippets Unsafe WCET results

Table 8.2 *Advantages and drawbacks for static timing analysis.*

Static analysis	
Advantages	***Drawbacks***
Theoretically guaranteed safe WCETs	Incorrect annotations ⇒ Incorrect results
No program execution required	Time Consuming in the beginning
Correct annotations ⇒ Correct results	Error prone
User-friendly tools available	Good knowledge about the analysed system required
Detailed results	Expensive, if many different analysis tool required
Simple for small code snippets	Compiler dependent

9 Future work

The future work that can be done within the WCET area has two sides, the academic and the industrial. More detailed analysis may be done both concerning statically- and dynamically timing analyses to get more reliable and comparable times.

The VTP part (Section 2.4.1) may in the same way as LTP be analysed with help of aiT. The HCS12/STAR12 processor has an embedded communication control module, MSCAN (Section 2.5.1), for CAN communication, which makes it suitable for this kind of analyses. Probably, these measurements would take even more time then the LTP part depending on its complexity and increased number of required annotations.

A more detailed analysis may also be done with the oscilloscope to get more precise times. This was not the purpose with this study and depending on lack of time and insufficient knowledge about the technique, only some general measurements were done. It would require much more time to perform a more detailed dynamically analysis. But maybe for someone more familiar with the environment it would be a less time consuming work to generate reliable times for comparison.

Of course, it would be of interest for VCT to get the rest of their applications analysed if it could be proven that aiT generates certain results. More dynamical analyses with help of other tools and methods could be done on the same applications to get results to compare with.

There are some issues that could be considered in the future work with aiT. It would for example be grateful if aiT could found the correct loop bounds automatically that are needed to get the correct WCET estimations. Maybe it would be possible to increase the tool with the missing loop patterns. To get support about this, AbsInt needed the executable that contains the unsolved loops, which was impossible depending on enterprise secrets.

For the simple HCS12/STAR12 processor it was for some WCET estimations of the analysed functions possible to create parametrical formulas. These formulas were influenced of parameters in the LDF-file such as number of frames and signals. Perhaps it would be possible to develop a tool that takes the LDF-file as input and automatically generates these formulas as output. But the hard thing in this work was not to create the formulas but rather to know which parameter that influenced the iteration of each loop. I think that a tool like this would

therefore not be very useful for the user of aiT. Instead the LDF-file perhaps may be used to set the iteration bounds for the loops automatically, which were a time consuming work.

Only a few analyses have been done on the *l_star12sci_m_rx()* function. One further analysis that could be done would be to measure the WCET in each of the states of the master task´s state machine. In this case a switch-statement has to be analysed and its different cases analysed. A first try of this has been done but requires a lot of workload to set correct annotations manually.

From VCTs side it would be of interest to get some WCET times for regions where interrupts are disabled. These regions are usually small and would be relative easy to analyse by aiT. Measurements of different kind of jitters as the one described in CASE 4 in Section 5.1.1.would be useful for VCT. However, generating this kind of times is not the purpose with the aiT tool which is developed for measuring WCET.

MÄLARDALENS HÖGSKOLA

VOLCANO AUTOMOTIVE GROUP

Bibliography

[Abs04] AbsInt Angewandte Informatik WWW Homepage, 2004.
URL: http://www.absint.com/

[Ait04] Worst-Case Execution Time Analyzer aiT for HCS12/STAR12 (Hiware Compiler), User Documentation for Windows – Version 1.5 rl, AbsInt Angewandte Informatik GmbH, February 2004.

[Ast04] WWW Homepage for ASTEC (Advaced Software TEChnology), a competence centre at the Department of Information Technology, Uppsala University, Sweden, 2004.
URL: http://www.astec.uu.se

[ASU86] Aho, A., Sethi, R. and Ullman, J. Compilers: Principles, Techniques and Tools. Addion-Wesley, 1986. Generally known as the "Dragon Book".

[BCP03] Bernat, G., Colin, A., Petters, S. pWCET: a Tool for Probabilistic Worst-Case Execution Time Analysis of Real-Time Systems.

[Bou04] Bound-T tool WWW Homepage, 2004.
URL: http://www.bound-t.com

[Can02] CAN Specification 2.0, Part-A and Part-B. CAN in Automation (CiA), Am Weichselgarten 26, D-91058 Erlangen. http://www.can-cia.de/, 2002.

[Can91] CAN Specification Version 2.0. Robert Bosch GmbH, Postfach 50, D-7000 Stuttgart 1, Germany, 1991.

[Can] Controller Area Network (CAN) – A Serial Bus System – Not Just For Vehicles. Esd gmbh Hannover.

[Car01] Carlsson, M. Worst-Case Execution Time Analysis, Case Study on Interrupt Latency, For the OSE Real-Time Operating System. Master Thesis in Electrical Engineering, Royal Institute of Technology, Stockholm, Mars 2002.

[CRTM99] Casparsson, L., Rajnak, A., Tindell, K. and Malmberg, P. Volcano - A Revolution in On-board Communications. Volvo Technology Report. 99-02-11.

[CP99] Colin, A., Puaut, I. Worst-Case Execution Time Analysis of the RTEMS Real-Time Operating System. Technical Report Publication Interne No 1277, IRISA, 1999.

[CP01] Colin, A., Puaut, I. A Modular and Retargetable Framework for Tree-Based WCET Analysis. In *Proc 13th Euromicro Conference of Real-Time Systems, (ECRTS'01)*, 2001.

[EES+03] Engblom, J., Ermedahl, A., Sjödin, M., Gustafsson, J. and Hansson, H. Worst-Case Execution-Time Analysis for Embedded Real-Time Systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 2003.

[Ene04] Enea WWW Homepage, 2004.
URL:http://www.enea.com

[Eng02] Engblom, J. *Processor Pipelines and Static Worst-Case Execution Time Analysis*, PhD. Dissertation, Uppsala University, Dept. of Information Technology, Uppsala, April 2002.

[Erm03] Ermedahl, A. A Modular Tool Architecture for Worst-Case Execution Time Analysis. PhD thesis, Faculty of Science and Technology, Uppsala University, June 2003.

[Hep04] Heptane Static WCET Analyzer WWW Hompage, 2004.
URL: http://www.irisa.fr/aces/work/heptane-demo/heptane.html.

[HLS00a] Holsti, N., Långbacka, T. and Saarinen, S. Using a Worst-Case Execution Time Tool for Real-Time Verification of the DEBIE software. In *Proceedings of the DASIA 2000 Conference (Data Systems in Aerospace 2000, ESA SP-457)*, 2000.

[HLS00b] Holsti, N., Långbacka, T. and Saarinen, S. Worst-Case Execution Time Analysis for Digital Signal Processors. In *Proceedings of the EUSIPCO 2000 Conference (X European Signal Processing Conference)*, 2000.

[JBO+00] Johansson, L., Broqvist, P., Ohlsson, J. And Torin, J. Efficient CAN based automotive control systems. Chalmers University of Technology, Gothenburg, Sweden, QRTech AB, Gothenburg, Sweden, 2000-04-11.

[Lev99] John R. Levine, Linkers and Loaders. Morgan-Kauffmann, 1999.

[LH00] Leen, G., Heffernan, D. Expanding Automotive Electronic Systems, January 2002.

[Lin04] LIN consortium WWW Homepage, 2004.
URL: http://www.lin-subbus.org/

[Lin] LIN-Consortium Formed to Set New Class-A Communication Standard for Vehicle Electonic Networks, LIN Press Release.

[Lsp03] LIN Specification Package, Revision 2.0, LIN Consortium, September 2003.

[Mel99] Melin, K. Volvo S80: Electric system of the future, Volvo Technology Report, February 1999.

[Mic00] MC9S12DP256 Advance Information, Revision 1.1, December 2000.

[Mic03] S12CPUV2 Reference Manual, Rev 0. 7/2003.

[Ols04] Olsén, H. The Complete LIN Tool Chain. In Elektronik Automotive, Magazine for Design of Automotive Electronics and Telematic, pages 18-19, 2004.

[Pro03] LIN Protocol Specification, Revision 2.0, LIN Consortium, September 2003.

[RE03] Rajnak, A. and Engler, T. LIN- The Open Communication System Enabling Easy System Integration, Datenkommunikation im Automobil, Heidelber, Juny 2003.

[RR00] Rajnak, A. and Ramnefors, M. The Volcano Communication Concept 2002.

[RH03] Ramnefors, M. and Helenelund, S. A Structrured Systems Engineering Approach for Developing Vehicle network Architectures. Baden-baden, September 2003.

[San04] Sandell, D. Evaluating Static Worst-Case Execution-Time Analysis for a Commercial Real-Time Operating System, Master Thesis in Computer Science, Depature of Computer Science, Mälardalen University, Västerås, June 2004.

[Ssf04] Space System Finland (SSF) company WWW Homepage, 2004.
URL: http://www.ssf.fi

[Ste02] Stewart, D. B. Measuring Execution Time and Real-Time Performance. In *Proceedings of the Embedded Systems Conference (ECS SF) 2002*, March 2002.

[TSH+03] Thesing, S., Souyris, J., Heckmann, R., Randimbivololona, Langenbach, M., Wilhelm, R. and Ferdinand, C. An Abstract Interpretetation-Base Timing Validation of Hard Real-Time Avionics

Software. In *Proc. Of the IEEE International Conference on Dependable Systems and Networks (DSN-2003)*, 2003.

[Vct04] Volcano Communic.ations Technologies AB company WWW Homepage, 2004.
URL: http://www.volcanoautomotive.com/

[ZKW+] Zhao, W., Kulkarni, P., Whalley, D., Healy, C., Mueller, F., U, G-R. Tuning the WCET of Embedded Applications.

Appendix A

Example of LIN Description File (LDF)

```
LIN_description_file;
LIN_protocol_version = "2.0";
LIN_language_version = "2.0";
LIN_speed = 10.4 kbps;

Nodes
{
   Master: master, 5ms, 1ms;
   Slaves: slave;
}

Signals
{
   velocity          : 8, 0x00, master, slave;
   lock_door         : 1, 0x00, master, slave;
   button_pressed    : 8, 0x00, slave, master;
   slv_resp_error    : 1, 0x00, slave, master;
}

Diagnostic_signals
{
   MasterReqB0: 8, 0;
   MasterReqB1: 8, 0;
   MasterReqB2: 8, 0;
   MasterReqB3: 8, 0;
   MasterReqB4: 8, 0;
   MasterReqB5: 8, 0;
   MasterReqB6: 8, 0;
   MasterReqB7: 8, 0;
   SlaveRespB0: 8, 0;
   SlaveRespB1: 8, 0;
   SlaveRespB2: 8, 0;
   SlaveRespB3: 8, 0;
   SlaveRespB4: 8, 0;
   SlaveRespB5: 8, 0;
   SlaveRespB6: 8, 0;
   SlaveRespB7: 8, 0;
}

Frames {
   global_info : 0x12, master, 1
   {
      velocity, 0;
   }
   door_control: 0x13, master, 1
   {
      lock_door, 0;
   }
   door_request: 0x22, slave, 2
   {
      button_pressed, 0;
      slv_resp_error, 8;
   }
}

Diagnostic_frames
{
   MasterReq: 60
   {
      MasterReqB0, 0;
      MasterReqB1, 8;
      MasterReqB2, 16;
      MasterReqB3, 24;
      MasterReqB4, 32;
      MasterReqB5, 40;
      MasterReqB6, 48;
      MasterReqB7, 56;
   }
   SlaveResp: 61
   {
      SlaveRespB0, 0;
      SlaveRespB1, 8;
      SlaveRespB2, 16;
      SlaveRespB3, 24;
      SlaveRespB4, 32;
      SlaveRespB5, 40;
      SlaveRespB6, 48;
      SlaveRespB7, 56;
   }
}

Node_attributes
{
   slave
   {
      LIN_protocol = 2.0;
      configured_NAD = 0x45;
      product_id = 0x7ff0, 0xfff0, 1;
      response_error = slv_resp_error;
      P2_min = 20 ms;
      ST_min = 20 ms;
      configurable_frames
      {
         global_info  = 0x4092;
         door_control = 0x4093;
         door_request = 0x40a2;
      }
   }
}

Schedule_tables {
   sch_conflict_resolving
   {
    AssignNAD {0x44, 0x45, 0x7ff0, 0xfff0}  delay 20 ms;
    SlaveResp                               delay 20 ms;
    AssignFrameId {slave, global_info}      delay 20 ms;
    SlaveResp                               delay 20 ms;
    AssignFrameId {slave, door_control}     delay 20 ms;
    SlaveResp                               delay 20 ms;
    AssignFrameId {slave, door_request}     delay 20 ms;
    SlaveResp                               delay 20 ms;
   }
   normal_mode
   {
      global_info  delay 20 ms;
      door_control delay 40 ms;
      door_request delay 40 ms;
   }
   low_power_mode
   {
      door_request delay 1000 ms;
   }
}
```

MÄLARDALENS HÖGSKOLA

VOLCANO AUTOMOTIVE GROUP

Appendix B

Example of Private Files

A.1 Master Private File

```
LIN_private_file;
LIN_protocol_version = "2.0";
LIN_language_version = "2.0";
concurrency_safety   = LTP;


network "door_net" {
                node master;
                file "door_net.Ldf";
}

interface "i1" {
                connects to door_net;
                [include "uart.cfg"]
}
```

A2. Slave Private File

```
LIN_private_file;
LIN_protocol_version = "2.0";
LIN_language_version = "2.0";
concurrency_safety   = LTP;

network "door_net" {
                node slave;
                file "door_net.Ldf";
    original_NAD = 0x44;
}

flag door_lock latches signal lock_door;

interface "i1" {
                connects to door_net;
                [include "uart.cfg"]
}
```

MÄLARDALENS HÖGSKOLA

VOLCANO AUTOMOTIVE GROUP

Appendix C

Master Execution Times

Table C.1 *WCET for the loops in the l_star12sci_sch_tick() function.*

Loop	Iterations	Cycles	Time (µs)
loop_0001	1	12	1,5
	2	61	7,625
	3	110	13,75
	4	159	19,875
	5	208	26
	6	257	32,125
	7	306	38,25
	8	355	44,375
Difference/Iteration		**49**	**6,125**
loop_0002	1	7	0,875
	2	30	3,75
	3	53	6,625
	4	76	9,5
	5	99	12,375
	6	122	15,25
	7	145	18,125
	8	168	21
Difference/Iteration		**23**	**2,875**
loop_0003	1	not in the worst case path	
	2	not in the worst case path	
	3	not in the worst case path	
	4	not in the worst case path	
	5	not in the worst case path	
	6	not in the worst case path	
	7	not in the worst case path	
	8	not in the worst case path	
Difference/Iteration		-	-
loop_0004	1	50	6,25
	2	102	12,75
	3	154	19,25
	4	206	25,75
	5	258	32,25
	6	310	38,75
	7	362	45,25
	8	414	51,75
Difference/Iteration		**52**	**6,5**
loop_0005	1	12	1,5
	2	83	10,375
	3	154	19,25
	4	225	28,125
	5	296	37
	6	367	45,875
	7	438	54,75
	8	509	63,625
Difference/Iteration		**71**	**8,875**
loop_0006	1	41	5,125
	2	87	10,875
	3	133	16,625
	4	179	22,375
	5	225	28,125
	6	271	33,875
	7	317	39,625
	8	363	45,375
Difference/Iteration		**46**	**5,75**
loop_0007	1	not in the worst case path	
	2	not in the worst case path	
	3	not in the worst case path	
	4	not in the worst case path	
	5	305	38,125
	6	378	47,25
	7	451	56,375
	8	524	65,5
Difference/Iteration		**73**	**9,125**
loop_0008	1	83	10,375
	2	171	21,375
	3	259	32,375
	4	347	43,375
	5	435	54,375
	6	523	65,375
	7	611	76,375
	8	699	87,375
Difference/Iteration		**88**	**11**

Table C.2 *WCET for CASE 1 of the l_star12sci_sch_tick() function*

Frame size	Flags	Cycles	Time (ms)
1	1	5528	0,691
1	2	5551	0,694
1	3	5574	0,697
1	4	5597	0,700
1	5	5620	0,703
1	6	5643	0,706
1	7	5666	0,709
1	8	5689	0,712
1	9	5712	0,714
1	10	5735	0,717
Difference/Iteration		**23**	**0,003**
2	1	5690	0,712
2	2	5713	0,715
2	3	5736	0,717
2	4	5759	0,720
2	5	5782	0,723
2	6	5805	0,726
2	7	5828	0,729
2	8	5851	0,732
2	9	5874	0,735
2	10	5897	0,738
Difference/Iteration		**23**	**0,003**
3	1	5852	0,732
3	2	5875	0,735
3	3	5898	0,738
3	4	5921	0,741
3	5	5944	0,743
3	6	5967	0,746
3	7	5990	0,749
3	8	6013	0,752
3	9	6036	0,755
3	10	6059	0,758
Difference/Iteration		**23**	**0,003**
4	1	6014	0,752
4	2	6037	0,755
4	3	6060	0,758
4	4	6083	0,761
4	5	6106	0,764
4	6	6129	0,767
4	7	6152	0,769
4	8	6175	0,772
4	9	6198	0,775
4	10	6221	0,778
Difference/Iteration		**23**	**0,003**
5	1	6176	0,772
5	2	6199	0,775
5	3	6222	0,778
5	4	6245	0,781
5	5	6268	0,784
5	6	6291	0,787
5	7	6314	0,790
5	8	6337	0,793
5	9	6360	0,795
5	10	6383	0,798
Difference/Iteration		**23**	**0,003**
6	1	6338	0,793
6	2	6361	0,796

6	3	6384	0,798
6	4	6407	0,801
6	5	6430	0,804
6	6	6453	0,807
6	7	6476	0,810
6	8	6499	0,813
6	9	6522	0,816
6	10	6545	0,819
Difference/Iteration		**23**	**0,003**
7	1	6500	0,813
7	2	6523	0,816
7	3	6546	0,819
7	4	6569	0,822
7	5	6592	0,824
7	6	6615	0,827
7	7	6638	0,830
7	8	6661	0,833
7	9	6684	0,836
7	10	6707	0,839
Difference/Iteration		**23**	**0,003**
8	1	6662	0,833
8	2	6685	0,836
8	3	6708	0,839
8	4	6731	0,842
8	5	6754	0,845
8	6	6777	0,848
8	7	6800	0,850
8	8	6823	0,853
8	9	6846	0,856
8	10	6869	0,859
Difference/Iteration		**23**	**0,003**

Table C.3 *WCET for CASE 2 of the l_star12sci_sch_tick() function.*

Frame size	*Flags*	*Cycles*	*Time (ms)*
1	1	2190	0,274
1	2	2213	0,277
1	3	2236	0,280
1	4	2259	0,283
1	5	2282	0,286
1	6	2305	0,289
1	7	2328	0,292
1	8	2351	0,294
1	9	2374	0,297
1	10	2397	0,300
Difference/Iteration		**23**	**0,003**
2	1	2352	0,294
2	2	2375	0,297
2	3	2398	0,300
2	4	2421	0,303
2	5	2444	0,306
2	6	2467	0,309
2	7	2490	0,312
2	8	2513	0,315
2	9	2536	0,317
2	10	2559	0,320
Difference/Iteration		**23**	**0,003**
3	1	2514	0,315
3	2	2537	0,318
3	3	2560	0,320
3	4	2583	0,323
3	5	2606	0,326
3	6	2629	0,329
3	7	2652	0,332
3	8	2675	0,335
3	9	2698	0,338
3	10	2721	0,341
Difference/Iteration		**23**	**0,003**
4	1	2676	0,335
4	2	2699	0,338
4	3	2722	0,341
4	4	2745	0,344
4	5	2768	0,347
4	6	2791	0,349
4	7	2814	0,352
4	8	2837	0,355
4	9	2860	0,358
4	10	2883	0,361
Difference/Iteration		**23**	**0,003**
5	1	2838	0,355
5	2	2861	0,358
5	3	2884	0,361
5	4	2907	0,364
5	5	2930	0,367
5	6	2953	0,370
5	7	2976	0,372
5	8	2999	0,375
5	9	3022	0,378
5	10	3045	0,381
Difference/Iteration		**23**	**0,003**
6	1	3000	0,375
6	2	3023	0,378
6	3	3046	0,381
6	4	3069	0,384
6	5	3092	0,387
6	6	3115	0,390
6	7	3138	0,393
6	8	3161	0,396
6	9	3184	0,398
6	10	3207	0,401
Difference/Iteration		**23**	**0,003**
7	1	3162	0,396
7	2	3185	0,399
7	3	3208	0,401
7	4	3231	0,404
7	5	3254	0,407
7	6	3277	0,410
7	7	3300	0,413
7	8	3323	0,416
7	9	3346	0,419
7	10	3369	0,422
Difference/Iteration		**23**	**0,003**
8	1	3324	0,416
8	2	3347	0,419
8	3	3370	0,422
8	4	3393	0,425
8	5	3416	0,428
8	6	3439	0,430
8	7	3462	0,433
8	8	3485	0,436
8	9	3508	0,439
8	10	3531	0,442
Difference/Iteration		**23**	**0,003**

Table C.4 *WCET for CASE 3 of the l_star12sci_sch_tick () function*

Frame size	*Flags*	*Cycles*	*Time (ms)*
1	1	1079	0,135
1	2	1079	0,135
1	3	1079	0,135
1	4	1079	0,135
1	5	1079	0,135
1	6	1079	0,135
1	7	1079	0,135
1	8	1079	0,135
1	9	1079	0,135
1	10	1079	0,135
Difference/Iteration		**0**	**0**
2	1	1164	0,146
2	2	1164	0,146
2	3	1164	0,146
2	4	1164	0,146

MÄLARDALENS HÖGSKOLA

VOLCANO AUTOMOTIVE GROUP

2	5	1164	0,146
2	6	1164	0,146
2	7	1164	0,146
2	8	1164	0,146
2	9	1164	0,146
2	10	1164	0,146
Difference/Iteration		**0**	**0**
3	1	1249	0,157
3	2	1249	0,157
3	3	1249	0,157
3	4	1249	0,157
3	5	1249	0,157
3	6	1249	0,157
3	7	1249	0,157
3	8	1249	0,157
3	9	1249	0,157
3	10	1249	0,157
Difference/Iteration		**0**	**0**
4	1	1334	0,167
4	2	1334	0,167
4	3	1334	0,167
4	4	1334	0,167
4	5	1334	0,167
4	6	1334	0,167
4	7	1334	0,167
4	8	1334	0,167
4	9	1334	0,167
4	10	1334	0,167
Difference/Iteration		**0**	**0**
5	1	1419	0,178
5	2	1419	0,178
5	3	1419	0,178
5	4	1419	0,178
5	5	1419	0,178
5	6	1419	0,178
5	7	1419	0,178
5	8	1419	0,178
5	9	1419	0,178
5	10	1419	0,178
Difference/Iteration		**0**	**0**
6	1	1504	0,188
6	2	1504	0,188
6	3	1504	0,188
6	4	1504	0,188
6	5	1504	0,188
6	6	1504	0,188
6	7	1504	0,188
6	8	1504	0,188
6	9	1504	0,188
6	10	1504	0,188
Difference/Iteration		**0**	**0**
7	1	1589	0,199
7	2	1589	0,199
7	3	1589	0,199
7	4	1589	0,199
7	5	1589	0,199
7	6	1589	0,199
7	7	1589	0,199
7	8	1589	0,199
7	9	1589	0,199
7	10	1589	0,199
Difference/Iteration		**0**	**0**
8	1	1674	0,21
8	2	1674	0,21
8	3	1674	0,21
8	4	1674	0,21
8	5	1674	0,21
8	6	1674	0,21
8	7	1674	0,21
8	8	1674	0,21
8	9	1674	0,21
8	10	1674	0,21
Difference/Iteration		**0**	**0**

Appendix D

Slave Execution Times

Table D.1 *WCET for the loops in l_star12sci_s_rx() function.*

Loop	Iterations	Cycles	Time (µs)
loop_0001	1	Not in the worst-case path	
	2	Not in the worst-case path	
	3	Not in the worst-case path	
	4	Not in the worst-case path	
	5	Not in the worst-case path	
	6	Not in the worst-case path	
	7	Not in the worst-case path	
	8	Not in the worst-case path	
Difference/Iteration		**-**	**-**
loop_0002	1		
	2	48	6
	3	84	10,5
	4	120	15
	5	156	19,5
	6	192	24
	7	228	28,5
	8	264	33
Difference/Iteration		**36**	**4,5**
loop_0003	1	13	1,625
	2	62	7,75
	3	111	13,875
	4	160	20
	5	209	26,125
	6	258	32,25
	7	307	38,375
	8	356	44,5
Difference/Iteration		**49**	**6,125**

Table D.2 *WCET for CASE 1 of the l_star12sci_s_rx() function.*

Frame size	Flags	Cycles	Time (ms)
1	1	1293	0,162
1	2	1336	0,167
1	3	1385	0,174
1	4	1434	0,180
1	5	1483	0,186
1	6	1532	0,192
1	7	1581	0,198
1	8	1630	0,204
1	9	1679	0,210
1	10	1728	0,216
Difference/Iteration		**49**	**0,006**
2	1	1323	0,166
2	2	1372	0,172
2	3	1421	0,178
2	4	1470	0,184
2	5	1519	0,190
2	6	1568	0,196
2	7	1617	0,203
2	8	1666	0,209
2	9	1715	0,215
2	10	1764	0,221
Difference/Iteration		**49**	**0,006**
3	1	1359	0,170
3	2	1408	0,176
3	3	1457	0,183
3	4	1506	0,189
3	5	1555	0,195
3	6	1604	0,201
3	7	1653	0,207
3	8	1702	0,213
3	9	1751	0,219
3	10	1800	0,225
Difference/Iteration		**49**	**0,006**
4	1	1395	0,175
4	2	1444	0,181
4	3	1493	0,187
4	4	1542	0,193
4	5	1591	0,199
4	6	1640	0,205
4	7	1689	0,212
4	8	1738	0,218
4	9	1787	0,224
4	10	1836	0,230
Difference/Iteration		**49**	**0,006**
5	1	1431	0,179
5	2	1480	0,185
5	3	1529	0,192
5	4	1578	0,198
5	5	1627	0,204
5	6	1676	0,210
5	7	1725	0,216
5	8	1774	0,222
5	9	1823	0,228
5	10	1872	0,234
Difference/Iteration		**49**	**0,006**
6	1	1467	0,184
6	2	1516	0,190
6	3	1565	0,196
6	4	1614	0,202
6	5	1663	0,208
6	6	1712	0,214
6	7	1761	0,221
6	8	1810	0,227
6	9	1859	0,233
6	10	1908	0,239
Difference/Iteration		**49**	**0,006**
7	1	1503	0,188
7	2	1552	0,194
7	3	1601	0,201
7	4	1650	0,207
7	5	1699	0,213
7	6	1748	0,219
7	7	1797	0,225
7	8	1846	0,231
7	9	1895	0,237
7	10	1944	0,243
Difference/Iteration		**49**	**0,006**
8	1	1539	0,193
8	2	1588	0,199
8	3	1637	0,205
8	4	1686	0,211
8	5	1735	0,217
8	6	1784	0,223
8	7	1833	0,230
8	8	1882	0,236
8	9	1931	0,242
8	10	1980	0,248
Difference/Iteration		**49**	**0,006**