Evaluation of Methods for Dynamic Time Analysis for CC-Systems AB

Yina Zhang

Augusti 13, 2005

Department of Computer Science and Engineering, Mälardalen University, Västerås
And
CC-Systems AB, Västerås

Supervisor at Mälardalen University: Andreas Ermedahl
Supervisor at CC-Systems AB: Mattias Lång
Examiner at Mälardalen University: Björn Lisper

Abstract

Nowadays, up to 98% of all manufactured computer processors worldwide are used in embedded systems. Most of the embedded systems have time constraints, which means that the computer that controls such a system not only has to compute the correct result, but is also required to produce the correct result in a specific time interval. The time constraints are often expressed by temporal deadlines that must be met by the software tasks in the system. It is therefore important to know how long time the system needs to execute each individual task.

The worst-case execution time (WCET) is an upper bound for all the possible execution times of such a task. The WCET can be obtained by various kinds of methods. Generally, they can be divided into two categories: dynamic methods and static methods. Dynamic methods derive the WCET by measurements. The program code executes, and the execution time is measured by hardware or software, such as a logic analyzer and time-functions provided by the operating system. Static methods obtain WCET without executing the program, instead a WCET analysis tool is used to calculate the execution time for the longest execution path.

In this thesis the WCET was measured by dynamic methods. An oscilloscope and a logic analyzer were used for time measurement. The goal of the thesis was, together with CC-Systems AB (CCS), to compare and evaluates different kind of methods for execution time measurement. The WCET analyses have been performed on interrupt routines used in CCS's welding machine control system. In order to find out the differences in precision and practical difficulties between using measurement and static analysis, the measurement methods and their results were compared with WCET estimates obtained by the static method, the aiT WCET analysis tool. To estimate WCET by a static method was a part of work in another Master Thesis, which was performed by Ola Eriksson at CCS during the same period of time as this thesis.

Through this thesis, we found out that neither the dynamic measurement nor the static WCET analysis is alone perfect. To obtain the best WCET estimates, the recommendation is to combine both kinds of methods.

Acknowledgment

This Master Thesis was performed at the CC-Systems AB in Västerås during the period between January and June 2005. The work is the last part of my education at the Mälardalen University that leads to a Master of Science in Computer Engineering.

I owe a word of thanks to many people who helped me under my performance of this Master Thesis. Several people gave me many valuable advices and support in various aspects of this work. First of all I would like to show my appreciation by many sincere thanks to my supervisor Andreas Ermedahl at Mälardalen University, Västerås for all his help and support for this thesis. Then I want to thank my supervisor at CC-Systems AB Mattias Lång, who helped me by giving me detailed information and explanations of the target systems. I also owe a great thanks to Joakim Adomat at the Department of Computer Engineering at Mälardalen University for his valuable help and advice in the hardware part of this thesis work, and for letting me borrow the logic analyzer from the department. A big thanks also goes to Johan Stärner who helped me with the Linux part in this thesis. My gratitude goes then to my classmate Ola Eriksson who helped me with a lot of questions that I had during the performance of this thesis, and thanks for all the "hostile" debates we had too ☺. I would also like to thank all the nice persons I met during my visit at the AbsInt Angewandte Informatik GmbH in Germany for the introduction course about the Static WCET analysing tool aiT. Furthermore, I want to express my gratitude to rest of the people at CC-Systems AB, it was my pleasure to work with you guys in the passed fem months. Finally, I want to, as always, thank my forever support, my family. My father Gong and mother Bin, my husband Kalle and my daughter Josefin. Without their love and encouragement I could never have finished my education.

Abstract ... 2

Acknowledgment ... 3

1. Introduction ... 6

1.1 Background ... 6

1.1 Purpose ... 7

1.2 Timing behaviour of a program ... 8

1.4 About CC-Systems AB ... 9

1.5 Delimitations ... 9

1.6 Thesis Outline ... 9

2.How can we obtain the WCET? ... 11

2.1 Dynamic Timing analysis ... 11

2.2 Static Timing analysis ... 14

2.3 Overview of WCET analysis techniques ... 16

3. Relevant technologies ... 18

3.1 Different kinds of methods for execution time measurement ... 18
3.1.1 Oscilloscope ... 18
3.1.2 Logic analyzer ... 19
3.1.3 Simulator CrossView Pro in the EDE TASKING ... 20
3.1.4 Emulator ... 21

3.2 Methods used by CC-Systems AB ... 22
3.2.1 Oscilloscope ... 22
3.2.2 Linux ... 23

3.3 The Welding Control System's Layout ... 24

3.4 Introduction of hardware used by the welding machine control system ... 25
3.4.1 The Infineon C167CS-LM Microcontroller ... 27
3.4.2 The Am29F800B flash-memory ... 28

3.5 Code characteristics of the welding machine control system ... 29
3.5.1 System's construction ... 29
3.5.2 WDS ... 30
3.5.3 PSA ... 32

3.6 The methods chosen ... 35

4. Problem Description and Solution ... 37

4.1 Can-interrupt routine in WDS-circuit measured with Oscilloscope ... 37
4.1.1 Preparations ... 37
4.1.2 Measurement with oscilloscope ... 41
4.1.3 Optimisations in WDS ... 46

4.2 PSA-circuit measured by logic analyzer *47*
4.2.1 Preparations *47*
4.2.2 The Can-interrupt routine on the PSA-node *50*
4.2.3 The Regulator-interrupt routine on the PSA-node *52*
4.2.4 Results of Measurement on PSA-node *54*
4.3 WCET Estimations by aiT *54*
4.4 Effort made to make the measurement result and aiT WCET estimation comparable .. *58*
4.4.1 Directs program code according aiT WCET estimation's path *58*
4.4.2 Directing aiT according to the measurement's execution path *59*
5. Result *61*
5.1 Dynamic Methods VS. Static Methods *61*
5.2 Dynamic Methods' Results VS. Static Methods' Results *63*
5.3 Other Usages of Dynamic and Static Measuring Methods *65*
6. Conclusions *67*
7. Designing and Creating Code for Analysability *68*
8. Future Work *69*
Bibliography *70*

1. Introduction

1.1 Background

Today, our everyday lives are very much correlated with computers, not only the desktop computers, but also the computers that are embedded as a part of many different products around us. More than 98% of all manufactured processors worldwide are used in embedded systems [1]. People can find embedded systems in various kinds of areas, such as telecommunications, vehicle industry, automotive systems, consumer electronics, nuclear power plant control, weapon guidance, etc.

Any mechanical or electrical system that is controlled by a computer working as part of an overall system is called an embedded system [2]; in other words, embedded systems are special-purpose computer systems hidden inside other systems or products. They are normally rather small in size and therefore have some design constraints, such as, limited system resources, including memory, processor performance, network bandwidth, and limited display and user interface capabilities. Since they usually have few interfaces to the outside world, the users of such systems are usually not able to modify or maintain the computer systems hidden inside. Furthermore, most embedded systems have to meet real-time constraints, that is to say, the computer that controls the system not only has to give the correct computation result, but it is also required to give the result within a specific time interval. This is often expressed by temporal deadlines that must be met by the software tasks in the system. The deadlines can be either “hard” or “soft”. Hard deadline means that if the task misses its deadline, the consequence can be severe: damaged property or even loss of lives. For example, tasks in the flight-control system in an airplane have hard deadlines. Soft deadline means that if the task misses its deadline, no catastrophe will happen, but the system performance will be deteriorated. For example, the encoding and decoding tasks in a multimedia device have soft deadlines.

To guarantee the safety and the good performance of the systems, it is necessary to prove that all the tasks in the system will meet their real-time constraint even under the most stressful situations. If several tasks are ready to execute at the same time, a schedule must be made. According to the schedule, the scheduler will choose one task to execute on the CPU. A system is schedulable if all the tasks in the system will be executed in such a priority order so that all of them will meet their deadlines. This scheduling can be either static or dynamic, depending on when the decisions for the task selection are made: offline, if it is before the system starts to run; or online, if during execution by the scheduler. There are several parameters that are required for proving the schedulability of the system, including period time, release time, deadline, etc. In [3] there are explanations for all of these factors. Another very important and basic factor for proving the schedulability is the knowledge of an upper bound for the execution time of each task, which is called Worst-Case Execution Time (WCET).

The WCET is defined as the longest execution time of a program that could ever be observed when the program is run on its target hardware [1].

Before computer programs are used in real-time applications, WCET analysis has to be done. This analysis aims to find out the worst possible execution time of program, so that the developer of the system will be able to tell whether the system can handle all kinds of possible scenarios, even under the most stressful situations. It is not easy to find the real WCET. The execution time of the computer program can vary from time to time. This is because it can be different inputs that are given to the functions in the program, which can lead to different execution paths through the program. It is difficult to know whether one has tested all possible scenarios. Modern processors have caches and pipelines to speed up memory access times and to accelerate program execution by overlapping instruction execution. These technologies make it even more difficult to decide the execution time, because in this case the execution time depends on not only the inputs, but also the execution history. For example, whether a data is in the cache or not, depends on which data accesses that have been made before this access. Similarly, the execution time of an instruction in a pipelined processor depends on what instructions that have been executed before. Thus it is very difficult to precisely estimate the WCET.

1.1 Purpose

The goal of this thesis is that, together with CC-Systems AB (CCS) [4], evaluate and compare different kinds of measuring methods for execution time analysis, and eventually give recommendations about which kind/kinds of methods shall be used by CCS in its future work. Dynamic WCET analysis methods are compared with static WCET analysis methods, which aim to find out which methods that are systematically best suited and most easy to use for CCS. CCS is also interested in how they should write their program code to simplify WCET analysis and if they can get more benefit from the method than just WCET values. This thesis also includes a study of the methods that have already been used by CCS.

Two pieces of program code have been measured. Both of them are used on the same welding machine control system that CCS has developed for the welding and cutting machine manufacturer ESAB [5]. An oscilloscope and a logic analyzer are used as the time measuring instruments.

The Advanced Software Technology Centre (ASTEC) in Uppsala [6] has supported this research. ASTEC is a Vinnova (Swedish Agency for Innovation Systems) initiative [7].

1.2 Timing behaviour of a program

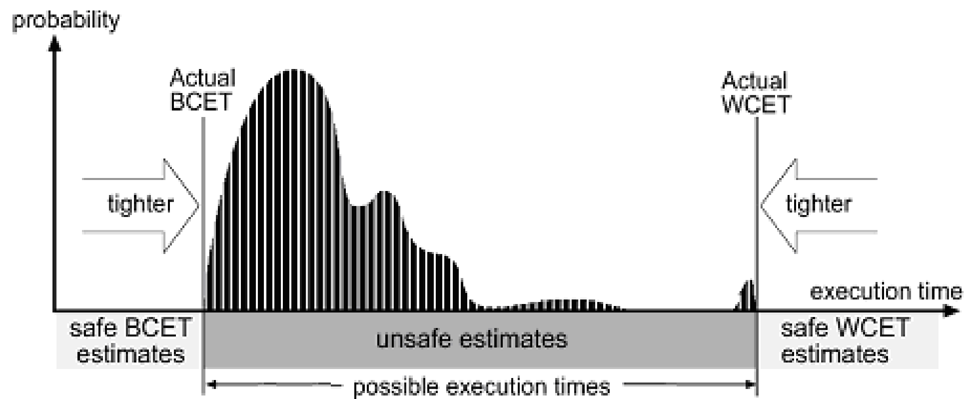


Figure 1.3: Execution time estimates

There are several expressions that are used to describe execution time measures when analysing the timing behaviour of a program.

The worst-case execution time (WCET) is defined as the longest execution time of a program that could ever be observed when the program is run in its target hardware [1]. Observe that this definition is valid only for one program in isolation. This means that interference from background activities such as direct memory access or refresh of DRAM memory are not considered [1]. Likewise the pre-emption or interrupts are also ignored in the WCET analysis. WCET is used to describe the time behaviour of a program and it is essential to bind this for all applications within a real-time system, to guarantee that the system works correct in all possible scenarios.

The best-case execution time (BCET) is defined as the shortest execution time of a program that could ever be observed when the program is run on its target hardware [1]. BCET can be of interest when calculating the start of the response time interval of a real time system. For some kinds applications, the difference between the WCET and BCET is also important to know.

The average-case execution time (ACET) lies somewhere in-between the WCET and the BCET, and depends on the execution time distribution of the program [1].

An execution time analysis should aim to give safe and tight estimates of the WCET and BCET. By safe we mean that the WCET estimates must be guaranteed not to be under the real WCET, and tight means that overestimation of the WCET estimates compared to the real WCET should be acceptable.

Likely, the BCET estimation should not overestimate the BCET and provide acceptable underestimations.

Figure 1.3 shows the relation between the estimated WCET / BCET and the actual WCET / BCET of a program. The x-axis is the time-axis and the curve indicates the possibility distribution of programs execution time. As we can see in this figure, the execution time of the program varies a lot, and the possibility of the actual WCET occurrence is very low, which makes the WCET really difficult to catch.

1.4 About CC-Systems AB

CCS develops and delivers electronic solutions and software for machines and vehicles in tough environments. Some of their main products are: on-board computers, displays, I/O-modules and communication modules. The products are used in many areas, including forestry machines, construction equipments, trucks, marine vessels, industrial automation, railway vehicles and military vehicles. Depending on the customer's needs, they deliver complete system solutions, partial systems, and custom made electronic solutions, products or consulting services. CCS has four offices in Sweden, located in Alfta, Uppsala, Västerås and Örnsköldsvik. The company also has an office in Tammerfors in Finland.

1.5 Delimitations

This Master Thesis corresponds to 20 university credits on D-level. The work was performed in CCS's office in Västerås. Due to time limitation, it was not possible to test and measure the whole control system of ESAB's welding machine, therefore only some interesting parts of the system's program code were chosen to be measured. These were the CAN-interrupt on the WDS-node, the CAN-interrupt and the Regulator-interrupt on the PSA-node.

Another limitation was that for obvious reasons, it was not possible to set up a complete welding system could be set up in the office. The test environment was very much different from the reality, thus it was impossible to get all the possible execution paths of the program code. In order to create the worst-case scenario, certain inputs were sometimes given manually to the system, i.e., the code was forced to go certain executions path. Something that might have affected the accuracy of the measured results.

1.6 Thesis Outline

Chapter 2 presents several kinds of methods that can be used for execution time estimation. The chapter starts with Section 2.1 that describes methods for dynamic measurement, including both measurements by hardware and software. Some concrete examples are shown in this section too. In the following section, Section 2.2, the theory of static WCET estimation is explained, and some examples of tools for static WCET analysis are also given. In Section 2.3, a comparison is made, of all the methods that were mentioned in Chapter 2. Chapter 3 presents information of all subjects concerned in this report. In Section 3.1, more detailed information of some dynamic methods is given, especially of those methods that have been studied for the possibility to be used on CCS's program. Section 3.2 gives brief information about time measurement methods that have been used by CCS. One suggestion of another Linux time function that can be used in CCS time measurement is given in this section as well. In Section 3.3, the welding control system's layout is described. Section 3.4, gives the information about the processor and flash memory used in the welding control circuits. Section 3.5 shows some code characteristics of the program code that is being measured in this thesis. Section 3.6 gives the methods chosen for measurement and the motivation of the choice. Chapter 4 gives descriptions of the measurements and analysis we had done. Section 4.1 illustrates how the measurement of CAN-interrupt in WDS-node is done with the help of oscilloscope. Section 4.2 illustrates how the CAN- and Regulator- interrupts in the PSA-node are measured using a logic analyzer. Section 4.3 gives a general explanation of how to estimate WCET by using the static WCET analysis tool aiT. Section 4.4 describes efforts made to make the results from the dynamic measurement and static estimation comparable. Chapter 5 presents the result of the work performed in Chapter 4, including comparisons of the two methods in Section 5.1 and comparisons of the measured results in Section 5.2. In Section 5.3 some suggestions of other usage of the two types of methods are given. Chapter 6 presents conclusions from the work performed and the results obtained. Chapter 7 gives some suggestions to CCS about how to design and create easy analysable programs. Chapter 8 points out some issues that should have been done better in this thesis, but because of lack of time and other reasons are left as future work.

2.How can we obtain the WCET?

2.1 Dynamic Timing analysis

Traditionally, the WCET analysis is done by measurement, also known as dynamic timing analysis. There are different kinds of time measuring methods in the industry. These methods can be divided into to three different categories: hardware, software and hybrid method. The hardware methods include, for instance, oscilloscopes, logic analyzers, and emulators. The software methods, for example, can be time-functions that different operating systems usually provide, or programs that tool vendors designed specifically for execution time measurement. The hybrid method is a combination of the above two mentioned methods: the system developer puts small code snippets in the program that is being measured, those code snippets work like triggers to the hardware; when the program is executed, the code snippets will start or stop the hardware that is connected to the target system for the execution time measurement. All these three methods have their advantages and disadvantages, which will be discussed later in Section 2.3. By giving "nasty" inputs to the program, people try to find the most difficult and stressful scenarios that the system can ever experience and thereafter catch the WCET. This is a very difficult work and requires a lot of time and effort. However, the result obtained by this way can never be guaranteed to be the actual WCET.

In [8] there are descriptions of some of the dynamic methods, such as:

Stop-watch: This method is only suitable for non-interactive programs, preferably running on single-tasking systems. The method simply uses a digital watch or other equivalent timing device, starts the watch when the program start to execute, and stops the watch when the execution is finished, and reads the time. This method works for big program code that can take minutes to execute and when measurements only need to be approximations.

Date command: is used like a stopwatch except is uses the built-in clock of the computer instead of an external stop-watch. A typical way to use the command is to wrap the program that is being measured in a shell script of alias with the following commands:

```
Date > output
Program > output
Date > output
```

This method is more accurate than a stop-watch, but has the same granularity of only being able to accurately measure non-interactive processes. Both the Stop-watch method and the Date command method will only provide an estimate of how long the full program takes to execute and do not take into consideration pre-emption, interrupts or I/O.

Time command (UNIX): the time command is useful when using a UNIX-based system. Real Time Operating System (RTOS) most often have similar commands. Execution time measurement is activated by prefixing time to a command line. This command not only measures the time between beginning and end of the program, but also consider the execution time used by other interfering activities, such as time for pre-emption, I/O, and other activities that cause the process to give-up the CPU. For example:

```
% time program
8.400u  0.040s  0:18.40  56.1%
```

The output can be little different depending on which version of the time command is used, but it just is the same information in different format. As in the above example, the first item, 8.400u, (u = CPU) is the execution time of the program, which means that it took 8.4 seconds for the CPU to execute the program, time spent on pre-emption or time spent for waiting for I/O, or performing RTOS functions has been removed. The second item, 0.040s (s = system), is the execution time used by the RTOS while running the program. The third item, 0:18.40 is the total execution time of the program, including not only the time for running but also time spent for being pre-empted, blocked, or waiting in the ready-queue, in this case, it was 18.40 seconds. The fourth item is the average percentage of CPU time used when the task was ready or running. This time command method is only suitable for measuring the execution time for a complete program.

Prof and Gprof (UNIX): they are profiling mechanisms available in UNIX and are able to measure execution on a per function basis. Both prof and gprof measure execution time with the granularity of a function, the resolution is usually of the system clock, meaning on the order of 10msec. The difference between Prof and Gprof is that the latter gives much more detailed results. Like the time command, pre-emption is taken in to account. If a process is pre-empted, the clock stops until the process starts to execute again. One thing should be noticed, using these two methods will yield an inaccuracy in the result because the profiling mechanisms will slow down execution of the program. This will make the measured execution time to be longer than the real execution time of the program. The syntax for using these profiling mechanisms can be found in [8].

Clock(): this command can be used to measure even at finer granularity than a function, for example execution time of a code segment or a loop. To measure execution time with the clock() function, you simply add lines of code around the code segment or loop that is being measured so that the clock is read at the beginning and at the end of the code segment. When using the clock() function, there are some issues that should be taken into consideration. Firstly, the measured time can be given in different forms depends on how the function is implemented in the operating system. It can be given in microseconds, seconds, or clock cycles. Normally the format is stated in the reference manual for the

particular operating system. Secondly, it is not sure whether the pre-emption is taken into account when different operating systems implements its own clock() function. Of course, implementations that can handle pre-emption properly are more useful and can give more accurate results. The third issue is that sometimes it is needed to measure the execution time in finer resolution, but the time resolution reported by clock() is usually the same as the system clock, for example, on many UNIX systems, it is 10msec or longer. This issue can be solved in two ways, either create a loop around the code that is to be measured to let the code run for 10, 100 or 1000 times or more, and measure the total execution time that is later divided by the number of times that the code has been executed; or use a hardware-based method.

Software Analyzer: CodeTest [9], TimeTrace[10] and WindView[11] are examples of software analyzers. When using a software analyzer it is very important to determine the resolution and granularity according to the specifications of the product. A good software analyzer can not only provide information on function basis but also contain a means for measuring execution time for smaller program segments, like a loop, a code snippet or even a single statement. Some can even measure the execution time of interrupt handlers and the RTOS overhead. Some soft analyzers provide a timing trace to show precisely what process is executing at what time. This is very helpful when debugging timing and synchronization errors. But software analyzers add overhead and therefore slow down the code. Furthermore, they usually requires a lot of memory to log data, which is a big disadvantage when used together with an embedded system whose memory resources can be limited.

Logic Analyzers: this is one of the best tools for accurately measuring execution time, especially when accurate timing is essential. There are two ways to use the logic analyzer to measure the execution time. One is to hook up the probes to the CPU pins or bus-pins on RAM or flash memory. This method is least obtrusive on the real-time code, but also the most difficult. Another way is to send strategic signals to an output port that is read by the logic analyzer as events. We will discuss more about using the logic analyzer in Section 3.1.2 and Section 4.2.

Other kinds of hardware method: oscilloscope and emulator can also be used to measure the execution time. However, the oscilloscope usually has limited measuring point and is not most suitable for the time measurement. The emulator, on the other hand, is specially built for certain kind of processor and is able to give very nice time and activity trace of the processor. The downside of it is that emulators are usually quite expensive. We are going to talk more about the oscilloscope and emulators in Section 3.1.1, Section 4.1 and Section 3.1.4.

All these above mentioned methods can only measure one execution path at a time, and it is up to the user to find the inputs that will possibly cause the longest execution time.

2.2 Static Timing analysis

In the recent years, a new kind of WCET analysis has came into use, static WCET analysis [1]. Instead of running the program, the static WCET analysis tool estimates the WCET, by statically analysing the program, basically deriving and adding together the execution time of each instruction in the program. In order to make the WCET calculation to be as accurate as possible, many issues will have to be taken into account.

The static WCET analysis is done using a static WCET analysis tool. There are both commercial and research prototype tools available. Before the static analysis can begin, the user must define model of the hardware, like the type of the processor and the type of address- and data-bus used on the target system, so that the timing behaviour of the target system can be imitated. Memory mapping of the target is another decisive information for the WCET calculation.

Static WCET analysis is generally divided into three steps: flow analysis, low-level analysis and calculation. See Figure 2.2 below. Many parts of the normal program development process are also shown in this figure.

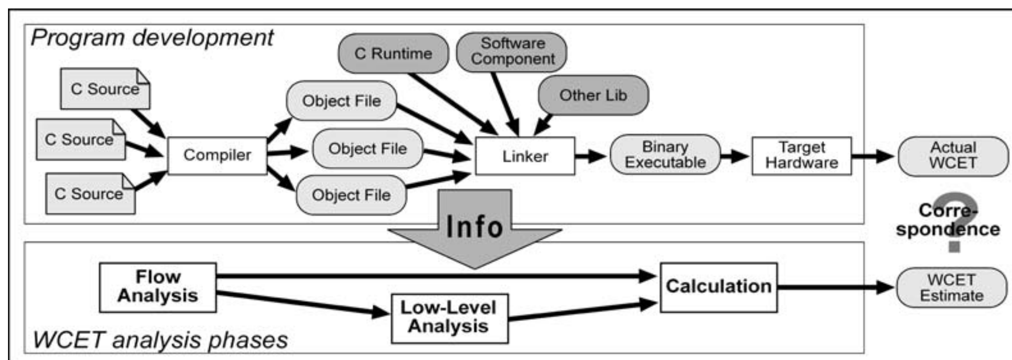


Figure 2.2: Components of WCET analysis

Under the *flow analysis step*, all possible execution paths through the program are found out. These execution paths are expressed as flow constraints, such as upper loop bounds, infeasible paths, function calls, recursion depths etc. These flow constraints are obtained during this step, either calculated by the static tool itself, automatically; or given by the user. The program flow can be extracted from three different types of code; source code (such as C, C++ and Java), intermediate code (code inside the compiler) and object code (such as ELF, COFF). In the source code there is no information about how the code was re-arranged by the compiler at compilation; and this information is important for mapping the flow constraints to the executable code. Intermediate code contains more information from the compiler, but the shortcoming is that it is very compiler dependent. Commercial static tools usually uses object code for their flow analysis since it is less compiler independent, which

makes it is easy to adapt the flow analysis to other compilers and hardware. The downside of using the object code is that it is difficult to see how the program code is designed to work, i.e., from the object code, it's hard to see which part of the code is correspond to the C code. A "reverse engineering" must be done in order to re-obtain the information of program flow.

The second step of static WCET analysis is the *low-level analysis*, which in turn can be divided into global low-level analysis and local low-level analysis. Under this step the execution time for each instruction in the program is calculated. Things like pipeline overlaps and cache effects are also considered in this step. "Global" refers to those issues that can affect the whole program. The global low-level analysis considers how issues like instruction caches, data caches, branch prediction and Translation Lookaside Buffers (TLBs) [12] will affect execution time. "Local" refers to the values of the current instruction and its neighbours. The local low-level analysis takes care of how issues like pipeline overlap and memory access speed will affect the execution time. If the target processor is very advanced, it will be difficult to separate these two kinds of low-level analysis, because some of the above mentioned issues could affect both of them.

The *calculation* step is the final step in static WCET analysis. Based on the execution paths from the flow analysis and the execution time from the low-level analysis, a WCET estimate is calculated. In order to guarantee to find the WCET, the path that generates the longest execution time must be found. The methods to find it can be divided into three main categories: path based calculation, tree based calculation and Implicit Path Enumeration Technique (IPET) [1]. A path-based calculation calculates the execution time for all execution paths and selects the one with the highest execution time as WCET. In the tree-based calculation, The WCET estimate is generated by a bottom-up traversal of a tree corresponding to a syntactical parse tree of the program. The execution times of child nodes are collapsed into one single node and the timing is derived for the new node. IPET calculation is based on a representation of program flow and execution times using algebraic and/or logical constraints. Each basic block[1] and/or edge in the basic block graph is given a time (t_{entity}) holding the time for executing the entity, and a count variable (x_{entity}) representing the number of times this block or edge is executed. The WCET is found by maximising the sum $\Sigma i \in \text{entities } x_i * t_i$, subject to constraints reflecting the structure of the program and possible flows. IPET is most commonly used because it can handle more complex flow constraints than the other two methods.

Nowadays, there are commercial static WCET analysis tools available, such as aiT of AbsInt Angewandte Informatik GmbH [13], RapiTime of RAPITA Systems Ltd. [14] and BoundT of

[1] A basic block is a group of sequential instructions that will always be executed as one atomic element.

Tidorum Ltd. [15]. In [1], [3] and [16], there are more detailed information about different kinds of static WCET analysis tools and how the static WCET analysis is done.

2.3 Overview of WCET analysis techniques

As stated in the above section, there are many kinds of techniques that can be used to measure the execution time, all of them have their own advantages and disadvantages. This means that not a single technique can be taken as the best of all. In this section, a comparison will be made to show how each of the techniques compromises between multiple attributes, such as resolution, accuracy, granularity, intrusiveness, and difficulty. First, we give some explanations of these attributes.

Resolution is used to describe how small time unit the measured time will be given in, such as, in second, in millisecond, in microsecond or in nanosecond.

Accuracy shows how close the measured time is compared to the actual time if an ideal measurement was obtained. The smaller the divergence is, the more accurate is the measuring method.

Granularity represents in what detail the program code can be measured. For example, a method with coarse granularity [8], would generally measure execution time on a per-process, per-procedure, or per-function basis. On the other hand, a method that has fine granularity can be used to measure execution time of a loop, small code snippet, or even a single instruction.

Intrusiveness shows how much the time measuring method influences the program code being measured. If time measurement is done by software, for example: using some kind of software analyzer or inserting a code snippet in program code to trigger a logic analyzer or an oscilloscope, the execution of the overhead of the software analyzer or the triggering code snippet will consume CPU-time. This will affect execution time and possibly the ordering of the execution. Once we remove the code when the time measurement is done, the system will probably be rescheduled and behave differently. This phenomenon is called a Probe-effect [17]. Using software measurement methods, the elimination of probe-effects is almost impossible. The easiest way is leaving the code in the program, but this will cause extra overhead in the program and consume valuable system resources. The less intrusive the method is, the smaller probe-effect it will cause.

Difficulty describes the effort required for obtaining the measurements. For some methods, the user simply needs to run the code; the method will directly show the measurement result in an automatically created file or table. Those methods are considered to be easy. However, some other

methods requires instrumentation and filtering of data to obtain the answers, which are much more difficult. Generally speaking, the easier the method is, the coarser the measurement results will be.

Table 2.3 below summarizes the methods and attributes of each method that presented in Section 2.1 and 2.2. Note that in some cases the attributes are approximated or subjective, not exact values (some values is borrowed from [8]).

Method	Resolution	Accuracy	Granularity	Intrusiveness	Difficulty
Dynamic tools					
Stop-watch	0.01 sec	0.5 sec	Program	None-intrusive	Easy
Date command	0.02 sec	0.2 sec	Program	None-intrusive	Easy
Time command	0.02 sec	0.2 sec	Program	None-intrusive	Easy
Prof and Gprof	10 msec	20 msec	Subroutines	Less-intrusive	Moderate
Clock ()	15 ~ 30 msec	15 ~ 30 msec	Statement	Intrusive	Moderate
Software analyzer	10 µsec	20 µsec	Subroutine	Intrusive	Moderate
Oscilloscope	10 µsec	4 µsec	Statement	Less-intrusive	Moderate
Logic analyzer	8 nsec	40 nsec	Instruction	None-intrusive	Hard
Static tools	1 clock cycle	Varying	Instruction	None-intrusive	Hard

Table 2.3: Summary of methods to measure execution time

3. Relevant technologies

3.1 Different kinds of methods for execution time measurement

3.1.1 Oscilloscope

An oscilloscope [18] is basically a graph-displaying device - it draws a graph of an electrical signal. The appearance of an oscilloscope is quite like a small old-style television set with a rather small screen and more controls buttons and knobs. The front panel of an oscilloscope normally has control sections divided into Vertical, Horizontal, and Trigger sections. There are also display controls and input connectors.

In most applications the graph shows how signals change over time: the vertical (Y) axis represents voltage and the horizontal (X) axis represents time. The intensity or brightness of the display is sometimes called the Z-axis. The Y- and X-axis are marked with equal strides, which depending on the setting by the user can indicate different volts and time unit per stride. It is important to remember that an oscilloscope trades reduced resolution on the signal for the longer time per stride, i.e. for instance, the execution time measured in 1ms time/stride is less precise than the execution time measured in 50µs/stride.

There are oscilloscopes of both analogue and digital types. An analogue oscilloscope works by directly applying a voltage being measured to an electron beam moving across the oscilloscope screen. The voltage deflects the beam up and down proportionally, tracing the waveform on the screen. This gives an immediate picture of the waveform, like a sinus curve. In contrast, a digital oscilloscope samples the waveform and uses an analogue-to-digital converter (or ADC) to convert the voltage being measured into digital information. It then uses this digital information to reconstruct the waveform on the screen. For many applications either an analogue or digital oscilloscope will do. However, each type has some unique characteristics that makes it more or less suitable for some specific tasks. People often prefer analogue oscilloscopes when it is important to display rapidly varying signals in "real time" (or as they occur). Digital oscilloscopes allow you to capture and view events that may happen only once. They can process the digital waveform data or send the data to a computer for processing. Also, they can store the digital waveform data for later viewing and printing.

With the help of an oscilloscope people can do many things. For example, to determine the time and voltage values of a signal, to calculate the frequency of an oscillating signal, to see if there is interference in a signal, in that case how much of the signal that is noise and whether the noise is

changing with time, and to find out how much of a signal is direct current (DC) or alternating current (AC).

3.1.2 Logic analyzer

A logic analyzer [2][19][20] is a display tool that is used to capture the activities of digital systems, similar to an oscilloscope for analogue system measurement. The basic problem that a logic analyzer solves is that a digital circuit is too fast to be observed by a human being, and has too many channels to be examined with an oscilloscope. A logic analyzer is like an array of inexpensive oscilloscopes. The analyzer can sample many different signals simultaneously but can display only 0, 1, or changing values for each. When logic analyzers first came into use, it was common to attach several hundred "clips" to a digital system. Later, specialized connectors came into use.

A logic analyzer would trigger on a complicated sequence of digital events, and then copy a large amount of digital data from the system under test or debugging. There is an internal memory in the logic analyzer, to which the values on the signals will be recorded. When the internal memory is full or the user stops the measuring, the results will be displayed on a screen.

Normally, a logic analyzer obtains data in two modes, which are called “state mode” and “timing mode”. Generally speaking, in the timing mode, the logic analyzer samples the signal more often than in the state mode. Timing mode uses an internal clock that is fast enough to take several samples per clock period in a typical system. On the other hand, in state mode, sampling is controlled by the target system’s own clock, so the signal is sampled only once per clock cycle. Thus, the timing mode gives a better resolution in the signal. However, the timing mode requires more memory to store a larger number of information that is sampled under a single clock cycle of the target system. A logic analyzer does not provide access to the internal state of the processor, but it does give a very good view of the externally visible signals. That information can be used for both functional and timing debugging.

Figure 3.1.2 below shows a typical logic analyzer made by Hewlett-Packard. A set of probes is used to connect the analyzer to hardware pins to monitor and measure logic states. It has totally 8 probe housings [21] with 18 signal lines [21] on each probe housing. It can measure up to 136 logic states of signals and capture time periods for events. The results can be displayed in both a graphical waveform and a list form that is easy and convenient to read. On the front panel there are push buttons for different configurations and display forms.

As shown on the figure below, there is a floppy drive on the logic analyzer. It is a useful tool for transferring data to and from compatible computers or other systems that can read and write MS-DOS

format. The configuration files, measurement results, and even menu and measurement images from the screen, all of these can be saved on a flexible disk. The changes of signals over a period of time can also be stored, which allow a user to study the history of logic states for debugging purposes. Using a logic analyzer, the user is able to supervise and monitor the traffic on data bus, address bus and control bus signals, so that the user will be able to tell, for example, which instruction is executing at a specific time, or which address is accessed at a specific time. Analyzing the trace of the address bus, for example, will help the user to find out which execution path through the code that the program has taken, which is very important to know when measuring the execution time. We will discuss more about using the logic analyzer in Section 4.2.

Figure 3.1.2: a typical logic analyzer made by Hewlett-Packard

3.1.3 Simulator CrossView Pro in the EDE TASKING

In modern computer systems various other tools have made logic analyzers obsolete for many uses. For example, many microprocessors have hardware support for software debuggers. Many digital designs, including those of integrated circuits, are simulated to detect defects before the unit is constructed. The simulation usually provides logic analysis displays. Often, complex discrete logic is verified by simulating inputs and testing outputs using boundary-scan logic. None of these exactly reproduce the high-speed data capture function of a logic analyzer, but they cover most real needs for debugging digital circuits.

The TASKING Embedded Development Environment EDE [22] offers a number of simulators and target hardware debugger. The generic name of the debugger product is CrossView Pro.

CrossView Pro knows four types of CPUs, 166, 167, 167MAC and EXT2MAC. '166' represents the basic C166 architecture. '167' represents the extended architecture, like the C161, C163, C164, C165

and C167 families. '167MAC' represents the extended architecture including the MAC coprocessor, like the ST10x262 and ST10x272 families. 'EXT2MAC' represents the second extended architectures like the C166S v2.0 and Super10. The CPU information can be given by the user in the configuration file [23]. In the “Target” menu in CrossView, the user can set the type of the processor that used on the target system.

Two forms of profiling are implemented in CrossView Pro. Profiling allows the users to perform timing analysis on their software. Both forms of profiling in CrossView Pro are fully implemented in the CrossView Pro debugger.

Function profiling, also called cumulative profiling, gives timing information about a particular function or set of functions. The time spent in functions called by the function being profiled is included in the timing results. To specify the functions to be profiled: Select the Tools | Cumulative Profiling Setup... menu item, there the user can select one or more functions to be profiled. The gathered profile is shown in the Cumulative Profiling Report dialog. To view the profiling results: Select the Tools | Cumulative Profiling Report... menu item. For each function the number of calls, the minimum, maximum, average and total time spent in the function are shown in term of clock cycels. Also, the relative amount of time consumed by a function in respect to the time consumed by the application is shown.

Code range profiling presents timing information about a consecutive range of program instructions. CrossView Pro displays the time consumed by each statement, C or assembly, in the source window. The timing data can be displayed in three different formats: absolute, relative to program, and relative to function. In “Tools” menu the profile report dialog shows the time spend in each function. The time consumed by functions called from the function being profiled is not included in the displayed time. Select the Profiling button in the Source Window to display profile data in the Source Window. If profiling is not enabled, this button also starts gathering of profiling data.

Normally both function and code range profiling will slow down the execution speed of the application being debugged. Therefore, switch off profiling whenever the timing information is not required. (Compare this with the Prof and Gprof (UNIX) in Section 2.1)

3.1.4 Emulator

An emulator [2] is a software package that imitates the hardware operations of a controller. Companies like Nohau and Hitex Development Tools produce and sell C167CS emulators. For the micro controller C167CS-LM, there are two kinds of emulators that support it. They are Dprobe167 [24] and

Dbox167 [25]. Dprobe167 is the compact base modular of Hitex's in-circuit emulation system. It is a fully functional in-circuit emulator (ICE) of handheld size. The Dbox167 features a trace-buffer of up to 512K frames. Each frame includes external inputs that supplied via logic probes and a timestamp with a resolution of 10 ns. Dbox167 is a module to extend the DProbe 167 system into a high-end in-circuit emulator system, i.e. Dbox167 helps the DProbe167 to be upgraded. Figure 3.1.4 below shows how the DProbe167 and Dbox167 are connected to a target system.

Figure 3.1.4 DProbe 167 and Dbox167

Using the emulators, the instructions can be traced under the program run-time, and the execution path and the execution time will be monitored on a PC.

The prices of those two emulators are rather high. It costs over 9,000 euros for a total package that include DProbe167 and adapter that connects the emulator with the target circuit. There are extra module options that can be added on the Dprob167: Trace and Dual Ported Memory. With Trace-memory, it will be able to save the whole execution together with the measured time. Thereby, people can actually see what the program code was doing at a specific time point. It helps to keep watch on the execution process and save a lot of when debugging the system. Those extra modules cost extra too.

3.2 Methods used by CC-Systems AB

As far as this thesis is concerned, CCS measures the execution time of their program code using two methods: an oscilloscope and time-functions in Linux.

3.2.1 Oscilloscope

Unfortunately, there is no detailed documentation saved to describe what and how CCS has performed measurement using the oscilloscope. There are only comments written in the program code, showing the results measured for some functions. For example, they have measured the conversion time on an AD conversion, in order to find out if it is done within 100000 cycles.

3.2.2 Linux

In another CCS's project, Linux was used as the operating system kernel. This project aimed to upgrade and porting the Common Controls CCN01 with CCN02 for Rolls Royce Marine. CCS used a time function provided by the Linux to measure the execution time. The function is called *gettimeofday().* This function retrieves the system time in microseconds, where the system time is the time elapsed since Epoch (00:00:00 UTC, January 1, 1970). The aim was to use the time measurement to continuously check that the execution time for a process-loop was within the permitted time interval. If the execution time exceeded the limit, some compensation would be done. This kind of measurement occurred constantly, and the program code measured including both very big code segment and small statements, depending on which part of the process that was executed.

CCS is also interested in finding out other options of Linux time-function that can be used for this project, as well as any advantages and disadvantages of time analysis by the help of the operating system.

However, the choices of time measurement functions can vary depending on the hardware and which distribution of Linux that is used for the target system. For the Rolls-Royce Marine project, the hardware is a PC 104 X86 platform with a 300Mhz AMD Duron processor from Hectronics. Regarding the distribution of Linux, CCS uses a self-made solution that is combined of a Linux-kernel 2.6.6, which is a standard kernel, and some device drivers for the special hardware.

Linux-kernel 2.6.6 actually provides another time function that can be used by CCS, it is called ***rdtsc()*** (read time-stamp counter)[26]**.** This function is available in all of the 2.6 kernels on X86 platform. *Rdtsc()* gives the time in term of clock cycles, which is more accurate and causes less overhead compared to the *gettimeofday()* which CCS is currently using. The *rdtsc()* function is accessing a time-stamp counter which keeps an accurate count of every cycle that occurs on the processor. The following macros, *rdtscl(low)* and *rdtsc(low,high) [27]*, where low and high are of the type unsigned long, i.e. 32-bit long. The time stamp counter register on X86 platforms is 64-bit long. *rdtscl(low*) reads the low half of the register into a 32-bit variable and the latter reads a 64-bit value into two 32-bit variables. The overhead difference between *gettimeofday()* and *rdtscl()* is stated in [27], for example, the overhead of *gettimeofday()* is 1µs; while the overhead for two consecutive calls to

rdtscl() was measured to 0.0711µs. The reason for the difference is that *gettimeofday()* is a system call that requires two memory operations: one for reading the timer, another one for storing the read value. In some systems, reading the internal timer needs to call the system kernel, which can take relatively long time compares to the actual execution time of the code being measured. Though, *rdtscl()* and *rdtsc()* is not a system call, instead they read time directly from the register (the Time Stamp Counter) in the hardware, which saves a lot of time. *rdtscl()* and *rdtsc()* can be useful when a very fine cycle measurement is needed for a section of code. For example, it can give a good idea of how many cycles a specific set of instructions might take compared to another set of instructions. Another use is to get an average time estimate for a function or a section of code.

The advantages of using an OS to measure the execution time are that it is simple to use, give comparatively good time resolutions and can measure both big code snippets and small statement. A disadvantage of using an OS is that the interference of the OS on the measured execution time can vary. This is because the OS also is responsible for other work, such as file management, network communications, and scheduling. Depending on what the OS is doing when the time-function is called, the time function call can sometimes be delayed and it might take a long time before the OS handles it. In the real time extension of Linux, there is a limit of how much the OS may have interfered with the measured execution time.

3.3 The Welding Control System's Layout

The Welding Control System whose program code is being measured is delivered to ESAB by CCS. ESAB is one of the world's largest producers of welding consumables and equipment. It was established by Oscar Kjellberg in 1904. Since then, the company has constantly improved on existing methods and materials, and developed new methods to meet the requirements of the technological progress. Today the company produces consumables and equipment for all kinds of welding and cutting process and application.

The ESAB welding control system is a modular system consisting of up to four different types of permanent nodes (MMC, Powersource, Wirefeeder and RemoteControl) and one service node (PC ECAT), which are all connected by a CAN bus. See figure 3.3 below.

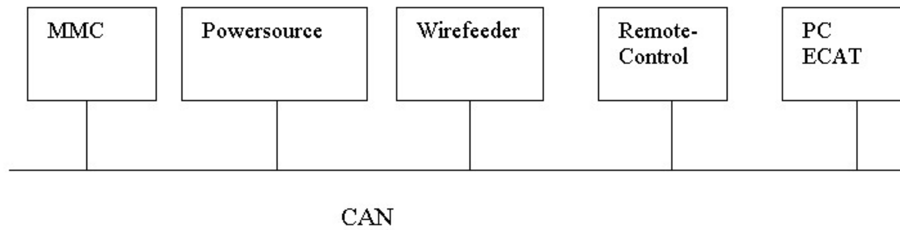


Figure 3.3a: the ESAB welding control system

Among those four permanent nodes, the MMC (Man Machine Communication) node and the PSA (PowerSource A) node exist in all kinds of welding systems; the rest of the nodes, however, are optional depending on the type of the welding system. If needed, several nodes of the same type can be connected in the system.

The MMC node is the interface between the user and the system; being responsible for handling the display and buttons. It is also the master node in the system and has information of the entire system. It regularly checks what nodes that are connected to the system by sending “hello-messages” via the CAN bus and supervises all the other nodes, except the PC node. The PowerSource_versionA (PSA) node controls the power supply to the welding process. The Wire Feeder (WF) node is responsible for feeding wire under the welding process if the current welding method uses wire. The RemoteControl (RC) node is used to replace the MMC panel when the welder is far away from the actual welding unit, which means when the remote control is responding, REMOTE state is entered and the MMC control panel is disabled. The PC node does not take part in the welding process. It is a service tool and is connected to the system only when a service engineer wants specific service information from the system or when the program in one of the nodes is upgraded. The node isn’t a separate piece of hardware, but actually software running on a PC and communicating with the other nodes through a CAN-card in the computer. The MMC node supervises the powersource and the wirefeeder by sending them supervision requests. If they do not respond, an error alert is shown to the user, and the welding process is stopped.

As mentioned before, there can be several nodes of the same type present in the system simultaneously. However, only certain type of nodes might exist in several versions. If there are several nodes of the same type in the system, then it is usually several WF nodes that are connected, responsible for controlling the feeding of different kind of wire in different materials or dimensions.

3.4 Introduction of hardware used by the welding machine control system

For this thesis, the system measured consists of two circuits: one WDS circuit and one PSA circuit. It took quite some time to obtain both circuits and get them connect together and work properly.

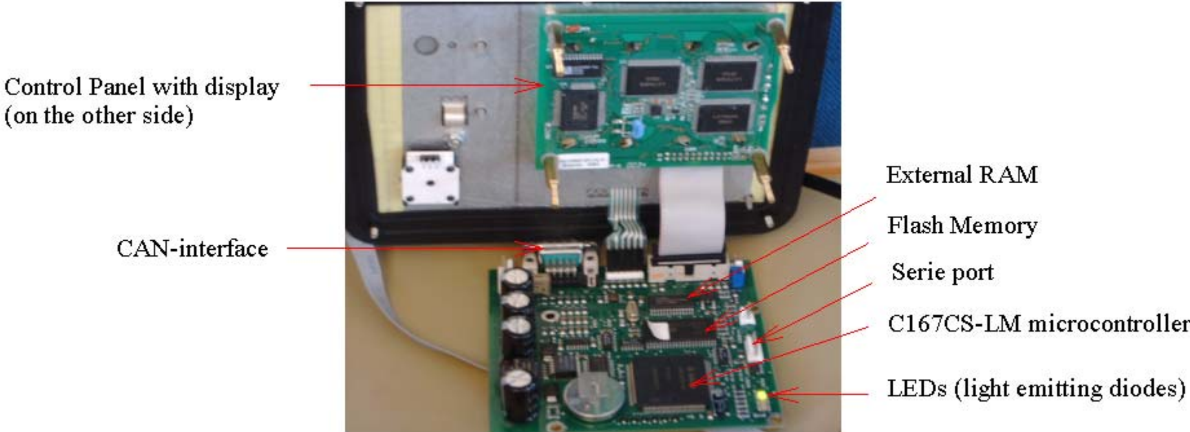


Figure 3.4a: WDS Circuit

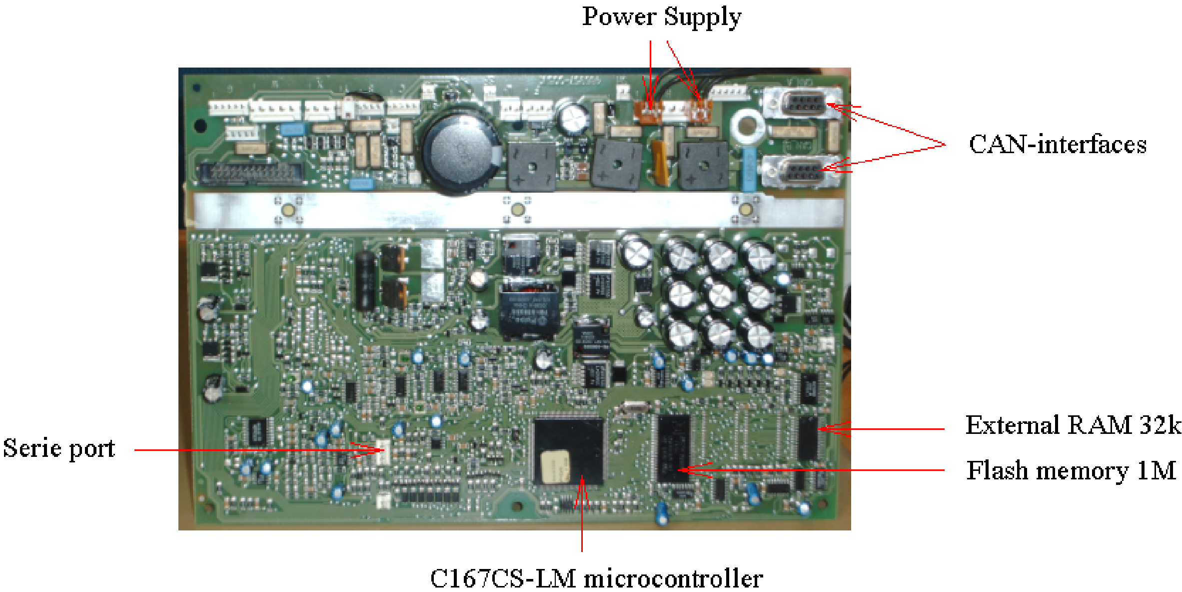


Figure 3.4b: PSA Circuit

The WDS circuit is sitting together with the control panel with a display. The PSA circuit is connected with the WDS circuit using CAN-interface and communicate with each other via CAN-messages. The WDS circuit gets its power supply at 12V via the CAN-interface, and the PSA circuit has special welding power sources at 26V and 9V. See figure 3.4a and 3.4b.

Both of the circuits are designed by ESAB. ESAB added both extra RAM (32k) and Flash-memory (1M) for program code and data. Each circuit includes an Infineon SAK-C167CS-LM microcontroller and neither of them has any operating system (OS).

3.4.1 The Infineon C167CS-LM Microcontroller

The Infineon SAK-C167CS-LM [28] belongs to the C166 microcontroller family. Below is a listing of some of characters of the microcontroller:

- The main core of the CPU consists of a 4-stage instruction pipeline
- 16-bit arithmetic and logic unit (ALU)
- CPU runs at a frequency of 20MHz
- Dedicated SFRs (Special Function Registers)
- Two on-chip CAN modules version 2.0B active
- Peripheral Event Controller (PEC)
- Capture Compare Unit (2x16 channels)
- 4-channel PWM (Pulse Width Modulation) unit
- 24-channel 10Bit A/D Converter
- Idle, Sleep and Power Down Mode with Flexible Power Management
- Two Multi-Functional general purpose timer units with five 16-bit timers
- Watchdog Timer and Oscillator Watchdog
- Up to 111 General purpose I/O lines
- Full Automotive Temperature Range: -40°C to +125°C
- Up to 16 MB of external RAM and/or ROM can be connected to the microcontroller.

Most of the C167CS's instructions can be executed in just one machine cycle that requires 100ns at 20 MHz CPU clock. For example, shift and rotate instructions are always processed during one machine cycle independent of the number of bits to be shifted. All multiple-cycle instructions have been optimised so that they can be executed very fast as well: branches in 2 cycles, a 16∗16-bit multiplication in 5 cycles and a 32-/16-bit stride in 10 cycles. Another pipeline optimisation, the so-called "Jump Cache", allows reducing the execution time of repeatedly performed jumps in a loop from 2 cycles to 1 cycle.

In order to get an idea of where we could possibly measure on the processor, the functions of all the ports on the microcontroller were firstly carefully studied. Following are some of the ports that were eventually considered interesting for bus sniffing.

- *Port 0 and Port 1* were used as address and data lines when accessing external memory if they were configured as external buses.
- *Port 4* outputs could be configured as push/pull or open drain drivers and could be used to output the segment address lines and for serial interface lines.
- *Port 2, Port 8 and Port 7* (and parts of *Port 1*) were associated with the capture inputs or compare outputs of the CAPCOM units and/or with the outputs of the PWM module.
- *Port 3* included alternate functions of timers, serial interfaces, the optional bus controlled signal BHE, and the system clock output CLKOUT (or the programmable frequency output FOUT).

One thing that should be mentioned here was that the Infineon C167CS-LM originally runs at 25Mhz. When ESAB constructed the circuits, they put an external crystal that controls how fast the processor would work. The crystal was connected to the pin number 138 on the processor and made the processor run at 20Mhz. In the manual, it was shown how to clock the device from an external source: drive XTAL1 (pin 138), while leaving XTAL2 (pin 137) unconnected, which was the case shown on the ESAB circuit's map.

3.4.2 The Am29F800B flash-memory

On both the WDS and the PSA circuits, there was a 1M byte flash memory. The function of it was to store the program code. The flash memory was of type Am29F800B and manufactured by Advanced Micro Devices (AMD) and Fujitsu [29].

The Am29F800B was an 8M bit, 5.0 volt-only Flash memory organized as 1,048,576 bytes or 524,288 words. The word-wide data (x16) appeared on DQ15–DQ0; the byte-wide (x8) data appeared on DQ7–DQ0. This device was designed to be programmed in-system. Below is the connection diagram of Am29F800B flash memory.

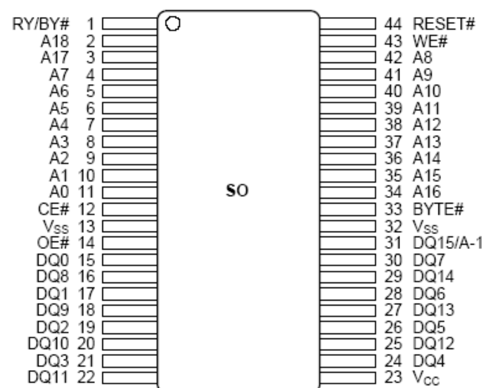


Figure 3.4.2 the connection diagram of Am29F800B flash memory

As shown on Figure 3.4.2, there were 19 pins for the address-bus (A0 ~ A18) and 16 pins for the data-bus (DQ0 ~ DQ). Besides the bus pins, there were also some other pins that were useful and interesting to know for the measurement's sake.

- Pin number 12 (CE#). When listening to the bus, the address on the address-bus was sometimes not valid because of interference from other devices or because of bit-flip. It was therefore necessary and desirable to filter those "trash addresses" away. The CE# pin made it possible. The CE#-pin was "active high", meaning that the address was available only when the CE#-pin was set to 0. By checking the CE#-pin at the same time as when listening to the address-bus, it was possible to easily identify only valid addresses, which saved a lot of time.
- Pin number 33 (BYTE#). The BYTE# pin controlled whether the device data I/O pine DQ15 ~ DQ0 operated in the byte or word configuration. If the BYTE# pin was logic '1', the device was in work configuration, and DQ15 ~ DQ0 were active and controlled by CE# and OE#. If the BYTE# pin was set at logic '0', the device was in byte configuration, and only data I/O pins DQ0 ~ DQ7 were active and controlled by CE# and OE#. The data I/O pins DQ8 ~ DQ15 were tri-stated, and the DQ15 pin was used as an input for the LSB (A-1) address function. Before the bus sniffing might begin, it must therefore be clear if the memory access was done in byte-mode or in word-mode, otherwise the address from the address bus might differ from the one shown on the development environment or the debugging software.

3.5 Code characteristics of the welding machine control system

In this section we will give descriptions of how the welding machine control system's program code is structured, as well as some main characters of the two parts of the code that have been measured.

3.5.1 System's construction

All of the program code was written in the object-oriented programming language C++. As introduced in Section 3.3, the welding system consisted of five kinds of nodes: MMC, PowerSource, Wirefeeder, RemoteControl and PC-ECAT. Images of these nodes could be found in the software too. They were classes PowerSource, WireFeeder, RemoteControl and ESAT, which implemented the 'interface' functionality that each node had to the MMC-node. These classes were aggregates to class WeldDataUnit that represented the MMC-node. Supervision was also handled by these classes. In method *ProcessMessage*, messages to a specific node were erected, as well as decoding of incoming messages. The supervised nodes sent error information to the MMC-node. This error mask was stored in respective object of the class. Every class had timer in the main loop, to handle timer events.

The program-design and code-structures of WDS-node and PSA-node were quite similar. The whole welding process was constructed as a big endless-loop, in which the system were designed to do its routine work, such as, reading from keyboard, interpreting if there had been a press on the keyboard, checking both in- and out-buffer of the CAN-message, etc. Besides this loop, there were seven kinds of interrupt that took care of different events that may happen during the welding process. They were:

1. NMITRAP: Used by undervoltage signal from reset circuit.
2. STOTRAP: Stack overflow trap.
3. STUTRAP: Stack underflow trap.
4. BTRAP: Hardware traps.
5. T5INT: Interrupt for software timer.
6. T6INT: Interrupt for welding regulator.
7. CAN1INT: Interrupt for CAN controller.

Those seven kinds of interrupt had the same name in WDS-node as well as in PSA-node, but they took different actions for handling the interrupts in each node.

3.5.2 WDS

The MMC-node was represented by the WDS-node in the program code, so the WDS-node had two main roles in the system: being the master node in the system, and being the interface between the system and the user.

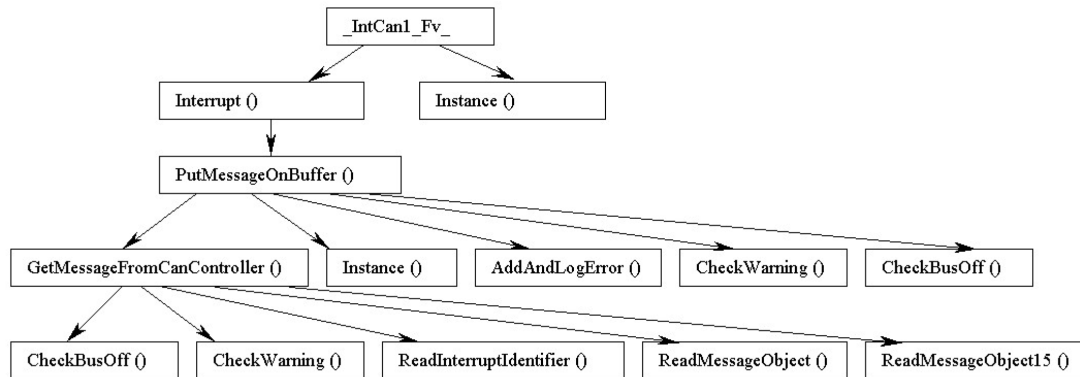
The following code characteristics were found for the WDS-node by Ola Eriksson, see [16]:

- 126 C++ source-code files. Between 30 and 2400 line of code per file including comments. There is a file with 13000 lines but it contains only icons written as char-arrays.
- There were 133 header-files with between 20 and 600 lines each.
- There are about 1100 functions
- The code itself doesn't contain assembler routines but there are some library routines written in assembler that are used, for example, floating-point additions and type casts.
- Switch-statements are commonly used since many parts of the node can be considered as state-machines and the switch-statements are there to make sure that the correct state is executed.
- Recursion seems no to occur, or are seldom used.
- There are 119 for-loops in the node, among which ten are nested. All of them are two-levels deep. 15 for-loop condition-tests are dependent on functions calls. There are no triangular for-loops or for-loops dependent on pointers. At least three for-loops contain a switch-statement.
- There were 119 while-loops, none of them are nested. Ten while-loop conditions-tests are depending on functions calls. Seven pointer-depending while-loop conditions-tests. 57 of the

while-loops are none-terminating loops, designed to halt the execution of the system, if a serious error occurs. The system will be rebooted when a timer expired.

CAN-interrupt in WDS

The Can interrupt in WDS was one of the code snippet that was measured under this thesis. Therefore it was necessary to take a more closer look.



Functions in WDS's CAN-interrupt

In the above figure, each arrow represents a function-call. There were totally three types of message that were valid at the WDS-node. By valid we mean that the message would be received/transmitted and handled by the WDS-node. The three types are:

- Normal message: an ordinary CAN-message.
- Message15: used for service messages from the ESAT node.
- STATUS message: message from the actual CAN-controller informing of an error in the circuit.

Every time a CAN interrupt occurred, *Interrupt()* was called from an interrupt vector. Up to six messages could be received and put into the receive-buffer during one single interrupt. The incoming message was put into the receive-buffer by *Interrupt()* calling *GetMessageFromCanController().* The CAN-message was then read from the CANcontroller and the type of the message was checked, and according to the message type, the corresponding actions were taken. If it was a STATUS message, the cause of the interrupt were checked; afterwards an error log was put on the display to alarm the user. If it was a Normal message or a Message15 message, the message was put into the buffer and if there had been a buffer overflow, the user would be alarmed too.

3.5.3 PSA

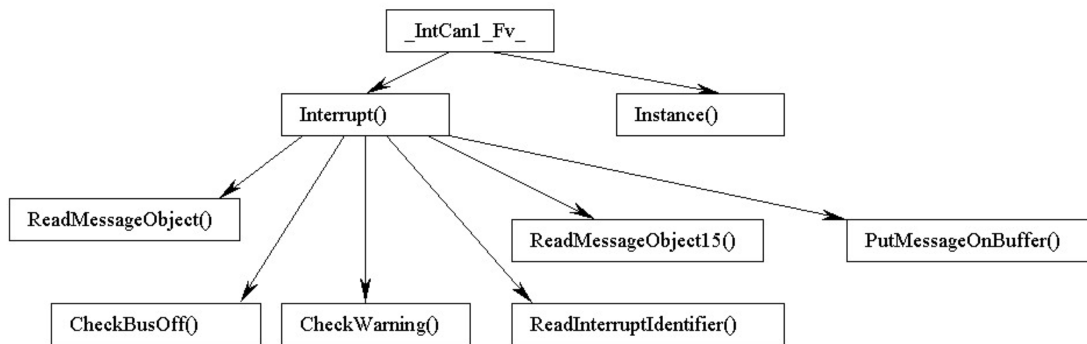
The PSA-node controlled the power supply to the welding process. According to the welding method present, it would increase or decrease the current at the weld-point to get the best welding result. Generally speaking, the program code for the PSA-node was less complicated than for the WDS-node.

The following code characteristics have been found for the PSA node [16]:

- 41 C++ source-code files. Between 40 and 2400 line of code per file including comments. Most of the files have less than 400 lines of code.
- 40 header-files up to 400 lines each. Most of them have less then 100 lines.
- There are about 500 functions.
- The code itself doesn't contain assembler routines but there are some library routines written in assembler that are used for example floating-point additions and typecasts.
- Switch-statements are commonly used, since many parts of the node can be considered as state-machines, and the switch-statements are there to make sure that the correct state is executed.
- Recursion seems not to occur very often. There is recursion in the library function fflush() and the recursion depth in that function is up to five.
- There are 64 for-loops in the node. Many for-loops are identical and placed in several cases in the same switch-statement. Three of them are nested and all of them are two-levels deep. Basically all for-loop condition-tests are simple integer comparisons.
- 58 while-loops. No nested while-loops. Basically all while-loop condition-tests are simple integer comparisons but some while-loop condition-tests are, however, dependent on pointers. About half of the while-loops are designed to wait for something to be ready, for instance, "while(busy)".

Two code parts at the PSA-node were measured under this thesis, which were code for CAN-interrupt and Regulator-interrupt. The following will give more details on how these two interrupts were constructed.

CAN-interrupt in PSA



Functions in PSA's CAN-interrupt

The CAN-interrupt in the PSA-node had almost the same actions as the one in the WDS-node, but the code was considerably different. Simply speaking, the CAN-interrupt code in the PSA-node was much more clearer and easier to understand than the one in the WDS-node. For example, the function Interrupt() checked the message-type and took actions accordingly to the message type directly, instead of, as in WDS-node, calling PutMessageOnBuffer() that in its turn called GetMessageFromCanController() to do it. Another example, CheckWarning() and CheckBusOff() were used to find the reason that cause a STATUS-message. In the WDS-node, this work was done twice, both in GetMessageFromCanController() and in PutMessageOnBuffer(), which made it a unnecessary double-work. However, in the PSA-node this work was done only once by the function Interrupt() itself, directly after it acknowledged a STATUS-message. The code differences made the CAN-interrupt for PSA-node, compared with CAN-interrupt for WDS-node, more effective, since it saved a lot of function calls; and was also easier to understand.

Regulator-interrupt in PSA

There were two different welding methods on the ESAB welding machine that we looked at: MMA and TIG. Either of them could be chosen as the present welding method. The MMA-method is also called as "pin-weld". The welding machine was holding a coated electrode, as soon as the pin got close enough to the work piece, there would automatically be an arc between them so that the pin melts. The pin would be getting shorter and shorter under the welding process and eventually needed to be replaced by a new one. The TIG-method, on the other hand, the welder would hold a TIG rod (like solder) that was melted on the work piece. The welder needed to press on a button of the handle

to start the welding process. See the Figures 3.5.3a and 3.5.3b below, for illustrations of the two methods.

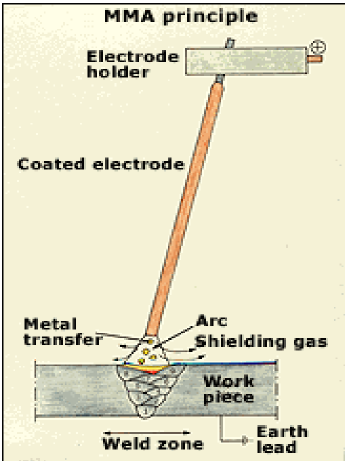


Figure 3.5.3a Welding method MMA

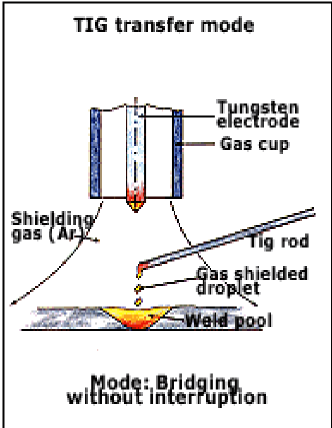
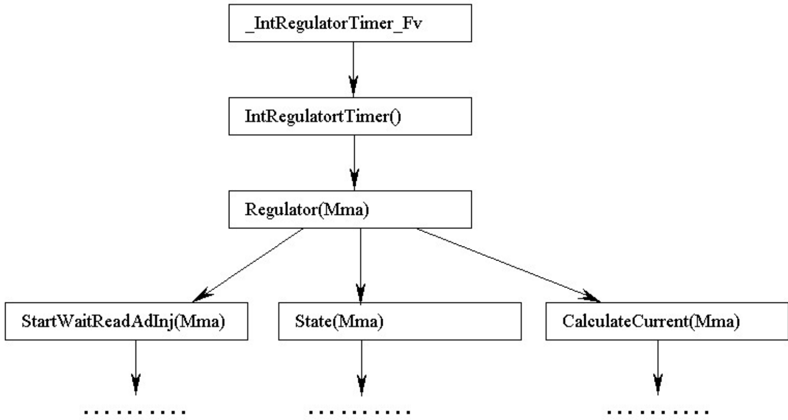


Figure 3.5.3b Welding method TIG

The whole welding process could be divided into different states. For different methods, the states will also be different. There were six states in the MMA-method and 20 states in the TIG-method.

The Regulator-interrupt was responsible for regularly checking the state of the welding process and adjusting the current consequently. When a regulator-interrupt occurred, first IntRegulatorTimer() will get called to find out what kind of welding method that was used, then the Regulator-interrupt function in the corresponding method will be called.

Function-calls Regulator-interrupt in MMA-method

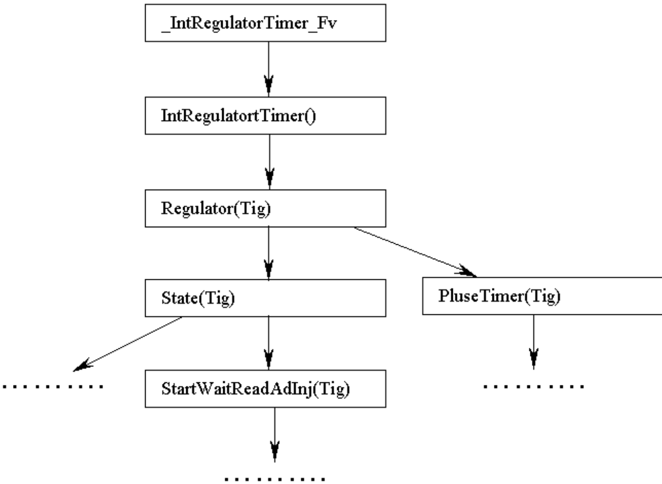


Functions in PSA's Regulator-Interrupt for method Mma

The Regulator-interrupt in the MMA-method worked like this: First the voltage and current of the arc was measured by StartWaitReadAdInj(); then according to the state of the process, the voltage was changed in the State() function, for example, if it was under state IDLE the voltage control would be set to off; later in the CalculateCurrent() an appropriate current would be counted and given to the machine.

There was a switch-statement with six cases (six states of the welding process) in both State() and CalculateCurrent().

Function-calls Regulator-interrupt in MMA-method



Functions in PSA's Regulator-Interrupt for method Tig

Compared to the regulator-interrupt in MMA-method, the regulator-interrupt in Tig-method was little more complicated. Partly because there were 20 different states for the welding process, which was much more than the states in MMA-method; partly because almost all of the work were done in the State() function, which in some cases, had to call many other functions to get the job done.

3.6 The methods chosen

There was no OS in the system; therefore we could not get any help from the time-functions that an OS could provide.

The C167CS-LM processor run at 20Mhz on both circuits, which means that one clock cycle was 50 nanoseconds. We do not need to analyze the system behaviour at every clock cycle. However, in order to measure the execution time for a small function, which may take only a couple of microseconds, the measuring method should be able to provide a resolution finer than millisecond. Furthermore most of

the timers in the program, such as the timer for the current sampling, and the timer for reading from the keyboard, are set at a millisecond resolution. Hence a measuring method with at least a resolution at microseconds should be chosen.

Since we wanted our measurements to be as little intrusive as possible (see Section 2.3), we decided to use hardware-based measurements instead of software-based ones. An oscilloscope was chosen to measure the CAN-interrupt in WDS, while a logic analyzer was used to measure the CAN-interrupt and Regulator-interrupt. Although CC-Systems AB had measured their code with an oscilloscope before, there was not any documentation available to describe how the measurements were done. Besides, they had only measured very small code segments, such as a for-loop or one single statement. Our measurements was therefore aimed to give a more detailed documentation so that other program developers may have it as a reference.

4. Problem Description and Solution

In this chapter we will discus how the measurements were done, both on the WDS-node and the PSA-node, and what results that we got. For instance, how the oscilloscope and the logic analyzer were connected to the target circuit; which configuration or format that were chosen on the equipment, etc.

4.1 Can-interrupt routine in WDS-circuit measured with Oscilloscope

4.1.1 Preparations

We had to solve several issues before the measurements could start:

1. The WDS circuit had to be provided with a power supply.
2. Since we did not have the possibility to set up a full real-life welding system connected to the WDS-node in the CCS office, a tool that could send CAN-messages to the WDS-node was needed. In other words, a tool that could pretend to be other nodes that in reality would be connect to the WDS-node via the CAN-bus.
3. We had to download the executable file to the circuit.
4. We had to find suitable measuring points on the circuit, i.e. we had to decide where on the circuit we could put our probes to read the signals of the code under investigation, the start and finishing points of an interrupt routine.
5. We had to decide what settings the oscilloscope should have in order to get the best measurement results.

According to the above five issues, we will now step by step, describe how the measurement set-up was done before the actual measurements were performed.

Provide a power Supply

The WDS-circuit needed 12V DC from the CAN-interface. In real working environment, the WDS-circuit is connected with the PSC-circuit via a CAN-interface that also supplies it with power. However in the office test environment, the WDS-node was not connected to the PSA-node, and therefore we had to make some changes on the CAN-cable so that it got directly connected to a power supply unit. Thereby the power supply problem was solved. The same CAN cable was also used to connect the circuit with a PC, from which the CAN-messages were sent to the WDS circuit. The next paragraph gives the preparations required for setting up the CAN-message tool. When the circuit got its power supply, it did a kind of self-test. There were three LEDs (Light Emitting Diodes) on the circuit. Each diode had two small lights, one green and one red. If the self-test were passed, the green light in LED number 3 would light.

CAN-message Simulator

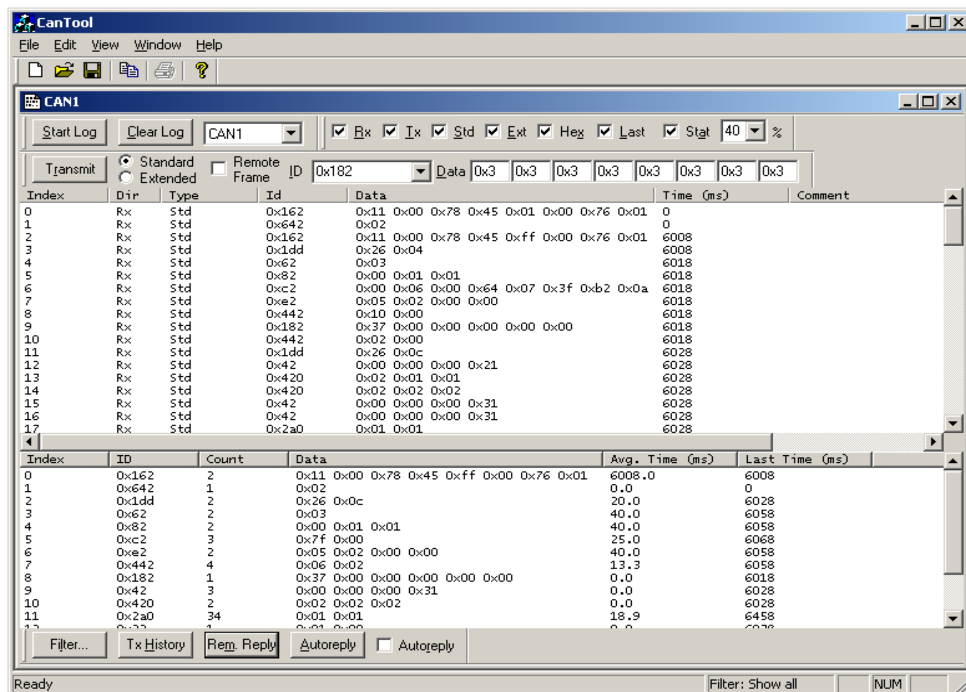


Figure 4.1.1 Parts of the graphical interface of the CanTool

At CCS, they had implemented a program called “CanTool” to simulate data communication by CAN-messages. They had also manufactured the CAN-card themselves to meet their own requirements. The CanTool was used to both send/receive messages to/from all circuit that use the CAN-message for communication. In the graphic interface of the CanTool, the user simply gives the messageID and the appropriate data of the message and then press “send”, the message is then sent to the connect circuit. If there was a reply or ACK-message from the receiver, it would be also shown to the view-window. From the view-window, the user could see all the traffic that happens on the CAN-bus. Information such as message’s index, direction, type, ID, data, and time since the CanTool was started were presented to the user. Figure 4.1.1 above depicts parts of the graphical interface of the CanTool program.

Downloading the executable file

First the program code was compiled into Motorola S records (.sre) format, which was done by the TASKING developing environment. The sre-format was the format for EPROM programming. Then the sre-file was downloaded to the target circuit via the serie-port by a program called “Launcher”. Here, one thing had to be noticed: every time, before we downloaded the sre-file, the target system had to first be rebooted, otherwise Launcher would complain that it could not find the target system.

Selecting a measuring point on the circuit

To select a good place on the circuit, where we could attach our measurement equipment, turned out to be the most difficult and import decision.

The idea was to insert a row of code before and after the CAN-interrupt function, so that one out-signal would be set “high” before the execution and “low” after the execution of the interrupt function. By measuring time spent between the changes of the signal, we could find out the execution time of the CAN-interrupt function.

The oscilloscope had only one probe that we could connect somewhere on the circuit to get the signal. The question was: where? As mentioned before in Section 3.3.1, there were several ports on the processor that could be used for this out-signal. In order to affect the system as little as possible, one pin that was very seldom used by the program code should be chosen.

In many books, such as [20], it states that LEDs are very useful to indicate the system state, for instance, error conditions, entering of a certain routine or idle time activity. As we could see, there were actually three LEDs designed on the WDS-circuit, so it was necessary to find out what functions they have. After studying the program code carefully, we found that pins number 0 to 5 on port 8 are connected to the LEDs and were only used when there was a warning or an error on the circuit. This even made the measurement easier, because the pins on the processor were sitting very tightly together and there was a risk for getting a short circuit if the probe of the oscilloscope was wrongly attached to them. Since the pins were connected to the LEDs, we could simply put the probe on one of the LEDs to catch the signal. In the program code there were functions used to turn on and off the light in the LEDs: *ELed_Green(unsigned short a), ELed_Red(unsigned short a), ELed_Both(unsigned short a)* and *ELed_None(unsigned short a)*, there “a” indicates the diode’s number. In those functions *_putbit()* was used to set low or high on the corresponding pins on the processor in order to turn on or off the lights. For example, *_putbit(0,P8,0)* set “low” on pin number 0 on the port8, which would turn on the green light in diode number 3; and *_putbit(1,P8,0)* turned the same light off. Among those three diodes, number 2 was the least used one, and it was turned off in the original code. Therefore, we decided to insert “*ELed_Green(2);”* before the CAN-interrupt function and “*ELed_Red(2);”* after it. During the measurements the oscilloscope probe was attached to the connect point of the green light on diode number 2; so the probe could feel the signal changes when the light was turned on and off.

One point should be mentioned here, actually we could use *_putbit(0,P8,2)* and *_putbit(1,P8,2)* directly to turn on and off the green light in diode number 2, but the execution time for the CAN-interrupt function was so short that the light could never manage to show that it had been turned on. Therefore *ELed_Green(2)* and *ELed_Red(2)* were used instead. Furthermore, these codes made the

diode2 light red after the interrupt, so that we could know directly and clearly see that a CAN-interrupt had occurred.

Although we only inserted two small functions calls in the code, it still caused a little probe-effect on the system, which made the measured result somewhat inaccurate. Hence the execution time for these two functions was subtracted from the total execution time measured for the interrupt routine. How we did it will be explained in Section 4.2.2.

Specifying the settings on the oscilloscope

The oscilloscope used was called HAMEG Digital Storage Scope. We used the oscilloscope's "trigger" function that works as follows: once the green light was turned on, the probe attached to the LED felt the signal change, which triggered the oscilloscope to start to store the signal. This storing lasted until next time the probe felt another signal change, which was at the time the green light was turned off, and then the storing was stopped. Afterwards, the signal stored during these two signal changes was shown on the screen. The screen of the oscilloscope was divided into 10 parts along the X-axis and 8 parts along the Y-axis, which indicated different time/stride and voltage/stride respectively, see Section 3.1.1 for details. When the measured signal was shown on the screen, we could read how many time-strides the signal covered. By multiplying the number of strides with the time/stride that was set on the oscilloscope, we got the time spent from when the code entered the CAN-interrupt function until it exit it, i.e., the execution time for the CAN-interrupt function.

The smallest resolution on this oscilloscope was 10μs per time-stride on the screen, which made the whole x-axis on the screen showing a time-span of 100μs. The smallest time resolution that the oscilloscope could save and store was 20μs per time-stride on the screen, which made the whole x-axis covering a time-span of 200μs. This was the time resolution that we used. With this setting we were able to measure all kinds of execution time that was shorter than 200μs.

For the voltage/stride, we chose 1V/stride. The actual setting of the voltage/stride was not so important in this case, because we were not interested in how much the voltage-level was, but only in when the voltage-level changed.

There were two channels on the oscilloscope; we decided to use channel 1 for the measurement and attached the probe to its socket.

In order to trigger the storage of the signal, the "STOR" button and the "DOTJ" button were pressed down. Before every new measurement, the trigger had to be reset by pressing the "reset" button.

The figure 4.1.1 below shows all the settings of the oscilloscope.

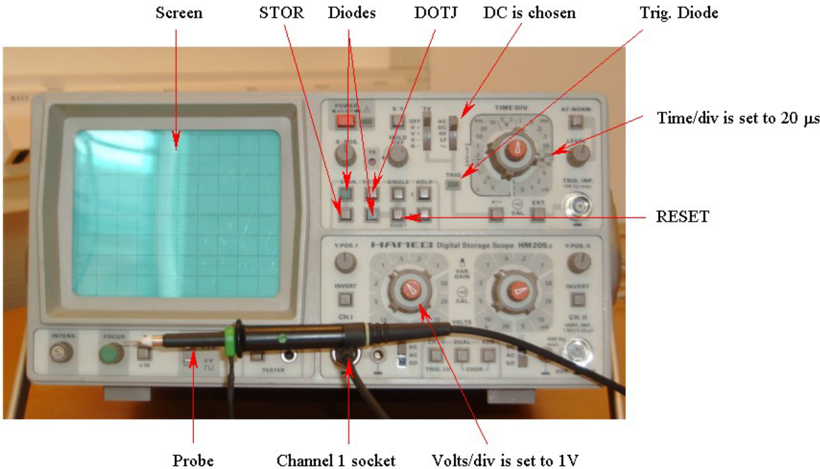


Figure 4.1.1 the Hameg Digital Storage Scope

4.1.2 Measurement with oscilloscope

Once all preparations were done, the next work was to figure out the message IDs of all of the messages sent on the CAN-bus. As mentioned before, there were three kinds of message: Normal messages, Message 15, and STATUS message. The normal message included 14 different message-IDs. Every message ID consisted of 11 bits. The first six bits indicated the message type, such as IDENTIFICATION_MSG, TRIGGER_MSG, ERROR_MSG, etc.; bit 7 and 8 indicated the unit_number; and the last three bits indicated the unit_type. Depending on the message type, the unit_type and the unit_number could together be used to identify the receiver or sender of the message. For IDENTIFICATION_MSG and ERROR_MSG, the last five bits indicated the sender of the message. For example: message ID: 162_{HEX}, which is 00101100010_{BIN}, this was a power source identification message, for the PSA-node with unit_type 2.

For to the different message type, unit_number and unit_type, we calculated all message ID from message 1 to message 15. STATUS message was defined directly to have the Can Interrupt ID 1. Table 4.1.2a lists some information derived for the fifteen messages.

Message box	Message ID	Receive/ Transmit	Function
1	0x164	Receive	Identification, remote control
2	0x204	Receive	Status, remote control
3	0x162	Receive	Identification, powersource
4	0x163	Receive	Identification, wirefeeder
5	0x1a2	Receive	Measure, powersource
6	0x042	Receive	Error, powersource
7	0x043	Receive	Error, wirefeeder
8	0x1dc	Receive	Service, esat
9	0x2a0	Receive	Start/stop acknowledge, powersource
10	0x466	Receive	Start/stop, wirefeeder
11	0x000	Transmit	All sent messages, all units
12	0x422	Receive	Generic status, powersource
13	0x3a2	Receive	Trigger, powersource/wirefeeder
14	0x044	Receive	Error, remote control
15	0x180	-	-

Table 4.1.2a CAN-Messages

We first filled in message ID 0x180 (message 15) with 1 byte arbitrary data in the CanTool program; the oscilloscope probe was attached to the connection-point of the green light on LED2; the “STOR” and “DOTJ” buttons on the oscilloscope were pressed down so that the “STOR” “DOTJ” and “TRIG” diodes got lit on the oscilloscope. The message was sent from the CanTool to the WDS-circuit and the oscilloscope caught one signal at 104μs. We reset the trigger and same message with the same data was sent to the WDS-node. This procedure was repeated several times, every time we got the same time of 104μs. This was expected, since the program should have the same system behaviour for the same input.

In the next step we tried to find out the execution path that corresponded to the 104μs measured value. This was very difficult to do, because the only help we had were the three LEDs on the circuit. There were many if-else-statements and switch-statements in the CAN-interrupt function, and to figure out all the possible paths through those statements was almost impossible. Therefore, we just tried to find out what was the execution path for the measured time. This is one of the disadvantages of dynamic measurement methods: it is hard to figure out what execution path that was executed and it is hard to guarantee that the time measured was the time for the worst case.

We lit different lights in the different cases in switch-statements and if-else-statements in order to see which part of the code that was executed. Since there were only 3 LEDs with 2 lights each, we could only test a small part of the code each time. We were finally able to figure out what execution path that corresponded to the measured value of the 104µs. Unfortunately this execution path was surely not the worst case, because there was one if-else-statement where the if-part would obviously took longer execution time, but the execution path of the 104µs took the else-part.

In order to make the code to go in the if-part, the code was changed so that the if-condition got true. The changed code was recompiled, and the executable file was downloaded to the WDS-circuit. Exactly same message was sent to the node, and this time the oscilloscope reported a time at 115µs. Afterwards, the number of data byte was changed to 8 bytes while the message ID remained unchanged. For this message configuration the interrupt took 152µs.

Later on the message ID was changed to normal messages. The execution time for each message with one respective eight bytes data was measured. For every execution, the path was re-examined after the time measurement. It showed that it was the same problem as for the message 15 type: there was a similar if-else statement in one of the interrupt's functions, where the execution would always take the shorter path, i.e. the else-part. Therefore the code was changed like in the message 15-case, and then the measurement was remade. The results showed that execution time of the interrupt routine are the same for all the normal messages: it took 121µs for 1 byte data and 145µs for 8 bytes data. The results were shown in the Table 4.1.2b below:

Msg typ	Msg ID	Time in µs for 1 byte data	Time in µs for 8 byte data
Normal message, including message 1 - 9, 13 and 14:		121	145
15	0x180	115	152

Table 4.1.2b Measured results for CAN-interrupt for the messages with manipulated code

You might have noticed that message type 10, 11 and 12 are missing in the table 4.2.1b. The reason for this was that message type 10 and 12 needed unit_nr and unit_type as inputs. With the measurements made by oscilloscope, we tried to give unit_nr = 1 and unit_type = 2 that was the unit_type for PSA, but the message were not accepted by the WDS-circuit, thus no signal was triggered and no time could be measured. Message type 11 is a special TRANSMIT message that could not cause the CAN-interrupt on the WDS-node.

Obviously, after the program code was changed and directed into a longer path, a longer execution time was obtained. However, the code modification also caused some strange and remarkable system behaviours. For example, before we changed the code, there had always been three reply messages received after certain normal messages were sent, such as message 3 and 4. But after the code was directed into the longer path, no reply message was ever received. The reason for this was possibly the statements executed in the if-statement had affected the system in a wrong way, so that it could not run as normal. Another strange thing that should be noticed was that: for one byte data, the execution time for the normal message was longer than the message 15, but for eight byte data, it was shorter than message 15. However, we could not find any good explanation for it. Therefore, changing program code to direct it into a certain execution path was not the best choice when trying to measure the WCET. On the other hand, in our case, we were forced to do so, because under the conditions that we measured we did not have all other necessary nodes connected to our test environment, and it was thus impossible to get the WCET in the natural way. The next question was: should we take these “faked” results as possible execution times? After considering the benefit and drawback of it, we decided not to do so. Instead, we let the program execute in its natural way and measured the execution times of the CAN-interrupt for all kinds of the messages again. The results were shown in the Table 4.1.2c:

Msg typ	Msg ID	Time in µs for 1 byte data	Time in µs for 8 byte data
Normal message, including message 1 ~ 9, 13 and 14:		115	139
15	0x180	104	132

Table 4.1.2c Measured results for CAN-interrupt for message 1~15 with original code

In the next step, the execution time of the STATUS message was measured. The STATUS message was sent to the WDS-node when some error occurred on the circuit, such as bus-off or warning. Since such an error could never happen in the test environment that we had at CCS, we were once again forced to modify the program code. This was made by still sending normal messages to the WDS-node, and just before the messageID was checked, it was changed into STATUS. Thereby the execution was directed to go into the part of code when there was a warning or there was a bus-off, when both warning and bus-off occurred on the circuit.

During the measurement, the set-up of the oscilloscope had to be changed, because the execution time for the interrupt caused by a STATUS message was much longer than the other messages. The setting on time per stride on the oscilloscope was changed to 0.5ms/stride when we measured the execution time for an interrupt routine handling STATUS message that caused by either a WARNING or a BUS-OFF. We used a 1ms/stride when we measured a STATUS message that caused by both a WARNING

and a BUS-OFF. Other settings remained unchanged. The measured execution times are listed in Table 4.1.2d:

Reason caused STATUS message	Execution time for the Interrupt-routine
WARNING	4.8 ms
BUS-OFF	4.8 ms
WARNING & BUS-OFF	9.75 ms

Table 4.1.2d Measured results for CAN-interrupt caused by STATUS message

There were two ways to direct the program code, one way was to let functions *CheckWarning()* and *CheckBusOff()* in file "Ecanmess.cpp" to directly return true; another way was to change the code in ECAN.cpp when the WARNING and BUS-OFF flags in STATUS_REGISTER were checked. Both ways were however tested, with no noticeable difference of the measured execution.

As previously mentioned, the lights on the three LEDs were used to trigger the measurement and also to show the execution paths. Functions called to turn on and turn off the lights also consumed execution time, which must be eliminated from the measured execution time to obain the actual execution time. In order to find the execution time for functions, such as *ELed_Red()* and *ELed_Green(),* both functions were called 20 times each and the execution time for these 40 function-calls were saved. In order to make sure that the execution time of those 40 function-calls was constant, it was measured several times, and each time we got the same execution time. This led to a conclusion that each function-call would take about 2µs to execute. The time for turning on and off the lights were then subtracted from the total execution time measured for the interrupt-routine. For a normal message with 1 byte data, the execution time was 115µs – 2 * 2µs = 111µs and with 8 bytes data the execution time was 139µs –2*2µs = 135µs; for a message 15 with 1 byte data, the execution time was 104µs – 1*2µs = 102µs and with 8 bytes data the execution time was 132µs – 1 * 2µs = 130µs. The code that was altered to direct the execution path for a STATUS message took relatively little extra time, maybe a couple of microseconds, which was negligible compared to the total execution time on milliseconds. Hence the final results of the execution time measured for the CAN-interrupt-routine caused by different CAN-message could be summarized like below:

Msg typ	Msg ID	Execution Time	Execution Time
STATUS	-	4.8ms (WARNING or BUS-OFF)	9.75ms (WARNING and BUS-OFF)
15	0x180	102µs (1 byte data)	130µs(8 byte data)
Normal	-	111µs (1 byte data)	135µs(8 byte data)

Table 4.1.2e Summary for measured results for CAN-interrupt at the WDS-node

The execution time for handling the STATUS message when there was neither a WARNING nor BUS-OFF message was also measured, to be approximately 60µs. This case could however never happen in reality, since a STATUS message was sent out only when there had been a warning or a bus-off. Therefore this 60µs execution time could only be used for comparing the result to the execution time obtained by the static WCET analysis tool aiT (see Section 4.3).

Some similar problem occurred for the code changing, like when we changed the code to direct it into a longer execution path. After the code was modified, the execution time of CAN-interrupt for STATUS message was measured. But we could only “create” the STATUS message once, the second time we tried to fake a STATUS message the CAN-interrupt was never triggered. The reason might be something like: Since a normal message was used to fake the STATUS message, a normal message was actually stored in the buffer of the CAN-controller. Normally the message should be removed from the buffer once it was read, but it was never removed in the STATUS-case, because the program code for STATUS message-handling never read from the CAN-controller (STATUS message has no data), instead it only checked if there had been a WARNING or BUS-OFF on the circuit. Thus the second time a normal message was sent, the buffer of the CAN-controller was still full, which led to the message got ignored or discarded.

4.1.3 Optimisations in WDS

Since CCS was also interested in finding out how different kinds of optimisations would affect the execution time of the program, we measured the execution time of the same program code with different kinds of optimisations too.

The TASKING compiler could do different optimisations when compiling the program code. The program code could be optimised for speed, which made the execution of the code faster; and for size, which made the executable file smaller in size; or, by using some user defined optimisation.

Since changing the program code would affect the system in an unknown and wrong way, which was undesirable, the code was changed back into its original shape. The code was compiled first with default optimisation, then with optimisation for speed and at last with optimisation for size. The execution times for all messages were measured for all three types of optimisations. It was quite easy to measure execution time of the different kinds optimisations of the program code. The optimisation-options were set in TASKING, using the EDE menu, C compiler options, project options, optimization. The settings on the oscilloscope remained unchanged. The results are shown in Table 4.1.3:

Msg type (ID)	1 byte data (µs)			8 byte data (µs)		
Compiled	**Optimization**			**Optimization**		
	Default	**For speed**	**For Size**	**Default**	**For speed**	**For Size**
15(0x180)	102	102	105	130	119	127
Normal message, including message 1 - 9, 13 and 14:	111	108	108	135	130.5	132

Table 4.1.3 Measured results for different kinds of optimisations

4.2 PSA-circuit measured by logic analyzer

The largest difference between the logic analyzer and the oscilloscope was that there were many more measuring channels on the logic analyzer, i.e. more information could be obtained simultaneously. A logic analyzer was therefore chosen to measure the PSA-circuit, we wanted to find out how much more a logic analyzer could do than an oscilloscope and whether it was much more difficult to do the measurements. Execution times for two kinds of interrupts were measured: CAN-interrupt and Regulator-interrupt.

4.2.1 Preparations

Same as in Section 4.1.1, we had to do some reparations before the measurements can begin.

Power Supply

The PSA-circuit used a special power supply for welding. The circuit got 26V AC respective 9V AC from pins H1-2 respective pins H6-7 on the wiring diagram. It must be connected to a WDS-circuit via its CAN-interface, otherwise it would not pass the self-test and all of the three LEDs on the PSA-circuit would light red. That is to say, to make a PSA-circuit work normally, a WDS-circuit must be connected to it. When the two circuits were properly connected, they also had a power supply. After the self-test was passed on both circuits, the green light on LED1 on the WDS was turned on and all the LEDs on PSA were turned off.

Measuring point on the circuit

The measurements were done in a completely non-intrusive way: by listening to the address-bus of the circuit. With help of a logic analyzer, a whole trace of addresses that had been accessed could be observed and stored with their corresponding access times. The time difference between the start and return address of the interrupt-routine was then the execution time for the interrupt.

The first problem that had to be solved was how to connect the logic analyzer to the address-bus. As mentioned before, the Infineon C167CS-LM processor has very narrow spaces between the pins. It

was therefore not possible to directly attach the clips on the address-pins on the processor. The address-bus was connected with both the external RAM and external flash-memory on the PSA-circuit. The spaces between the pins on both of the memories were fortunately big enough to hook up the clips of the logic analyzer without causing any short circuit. Actually it was also possible to purchase adapters for the processor, but we decided to choose this easier and cheaper way. We decided to attach the clips on the address pins on the flash memory.

The pod1 and pod2 probes of the logic analyzer were used. There were 16 channels on each of the pods and every channel could be connecting to one bit of the address-bus. Before attaching the clips, the Pin number 33 on the flash memory, the BYTE#-pin was measured (see Section 3.4.2). This pin was set to high, which indicated that the memory access was done in word-mode. The same byte in memory has half the size address on word-mode as on byte-mode. For example:

<table>
<tr><th></th><th colspan="9">Memory</th></tr>
<tr><td>Address in byte-mode</td><td>0x1</td><td>0x2</td><td>0x3</td><td>0x4</td><td>0x5</td><td>0x6</td><td>0x7</td><td>0x8</td><td>…</td></tr>
<tr><td>Address in word-mode</td><td colspan="2">0x1</td><td colspan="2">0x2</td><td colspan="2">0x3</td><td colspan="2">0x4</td><td>…</td></tr>
</table>

In our case, the address was actually wanted in byte-mode, because the start and return addresses of the interrupt routines we got from the simulator CrossView and the static WCET analysis tool aiT were in byte-mode, and we would like to use them directly without modifying them. Therefore, when attaching the clips to the address-pins on the flash-memory, channel 0 on pod1 was left unconnected, i.e. all the addresses observed by the logic analyzer would be shifted one bit to the left. Then, according to the connection diagram (see Section 3.4.2), the clips were connected bit by bit to the address-bus. Channel 1 on pod 1 was connected to the address-bus bit 0, channel 2 on pod 1 to address-bus bit 1, etc. There were totally 19 bits on the address-bus, so channel 0 to 15 on pod1 and channel 0 to 3 on pod2 were used for listening to the address-bus. As previously mentioned, the channel 0 on pod1 was left unconnected, while the rest of the channels were attached with the address pins of the flash-memory.

When the logic analyzer was attached to the address-bus, pin number12, the CE# pin was connected to channel 7 on pod2 on the logic analyzer. CE# was an active low pin on the flash-memory and could be used to filter out “trash addresses” and make the study of the address trace easier. See Section 3.4.2 for details.

Obtaining start and return addresses of the interrupt routines

In order to get the start and return addresses of the interrupt-routines that should be measured, the program had to be first executed on the TASKING CrossView Pro simulator or analysed by the static WCET analysis tool aiT, because their graphical interfaces could show the start and return addresses

of the interrupt routines. Thus the program was first compiled in .abs format, since aiT and CrossView required this format. Then the code was compiled in .sre format and downloaded to the processor. To guarantee that the addresses showed in Crossview and aiT corresponded to the same addresses accessed on the flash-memory exactly the same code was compiled into both kinds of formats. For our measurements, the addresses obtained by aiT were used because it was easier to see the addresses of the single instructions in aiT's graphical interface aiSee. See Section 4.3 for examples of this graphic interface aiSee. The addresses from CrossView were also tried and they could absolutely also be used, it was easier to see the related source code that matched the addresses from CrossView.

Specifying the settings on the Logic Analyzer

The Logic Analyzer used was a HP 1670 Deep Memory Logic Analyzer. Following are the setting we used for our measurements.

In the "Config" menu, the type of Logic Analyzer was set to "Timing".

In the "Format" menu that the user could select which channels that should be active. Channel number 0 to 15 on pod1 and channel number 0 to 3 on pod2 were selected to be active, since they were the bits for the address-bus. Besides, channel number 7 on pod2 was connected to the CE# pin on the flash-memory and also selected.

Then in the "Waveform" menu, under label "Acq. Control", the mode was set to "manual", the memory length was set to the largest, the trigger position was set to "center", and the sample period was set to "40ns". The C167CS-LM processor on PSA-node was running on 20Mhz, so every clock cycle was 50ns. If the logic analyzer sampled with a 40ns period, we should be able to see the address accessed in every clock cycle.

In the "trigger" menu, we chose "Label a" and defined it to be the start address of the interrupt routine. In the field "Timing Sequence Levels", it showed "trigger on "a" > 40 ns". During system running, once this start address was accessed, the logic analyzer would be triggered and started to store all the activities that were occurring on the address-bus. Afterwards, the label "Marker off" was selected and changed to "Pattern". Then label "Specify patterns" was shown on the screen. The user could specify the addresses to be searched. In our case, we set the 0-pattern to be the "entering" the start address of the interrupt routine; and set the X-pattern to be "leaving" the end-address of the interrupt routine.

Every time when the measurement was done, we simply chose the "List" menu, there the triggering 0-pattern (start address of the interrupt routine) was always shown with time 0ns; and we searched the

first X-pattern (return address of the interrupt routine) that was caught after the trigger, the time of the X-pattern was then the execution time for the interrupt routine being measured.

Downloading the executable file

We compiled the original codes for both WDS-node and PSA-node into the .sre format. Then the executable files were downloaded in the circuits via the serie-ports.

4.2.2 The Can-interrupt routine on the PSA-node

Since the PSA-node was connected to the WDS-node, it was not necessary to use the CanTool to send the message to the PSA-node, the two nodes were automatically sending messages to each other. We only measure the execution times of the CAN-interrupt using the logic analyzer.

From the map file we could see that the address to the CAN-interrupt vector was 000100h, and the CAN-interrupt routine had start address 0007A6Eh, which agreed with the start address we got from aiT. We first set 007A6Eh as the trigger address for our logic analyzer, but after consideration, the trigger address was changed to the address of the CAN-interrupt vector, because the execution time for calling the interrupt routine should also be included in the execution time for the CAN-interrupt handling. Shown on the control-flow graph we got from aiT, 007AA0h was the return address of the interrupt-routine in the main-loop, thus we set 007AA0h as the return address from the CAN-interrupt.

The “run” button was pressed and we caught a time at 56.32µs. The measurement was done repeatedly by pressing “shift” + “run”. There were three different times that were caught by the logic analyzer: 56.3±0.5µs, 169.8±0.5µs and 184.8±0.5µs. Among them, time around 56µs was most frequently occurring, about 85% of the totally number of measurements. Why did the execution times vary so much? Was the 184µs the longest execution time the CAN-interrupt could take? Our next step was to find out the execution paths of all these three times.

One execution trace was saved for each of those three times. The execution traces were than copied from the logic analyzer to a PC for analysing. A trace from the logic analyzer looked like the following:

```
ADRS_LIST_56a.txt - Anteckningar
Arkiv  Redigera  Format  Hjälp
MACHINE 1    -  State Listing

Label      >    Lab1      Time
Base       >     Hex    Absolute
___________    _______  __________

         0    000100      0  s
         1    000100     40  ns
         2    000100     80  ns
         3    000102    120  ns
         4    000102    160  ns
         5    000102    200  ns
         6    000102    240  ns
         7    000104    280  ns
         8    000104    320  ns
         9    000104    360  ns
        10    003000    400  ns
        11    00786E    440  ns
        12    007A6E    480  ns
        13    007A6E    520  ns
        14    007A70    560  ns
        15    007A70    600  ns
        16    007A70    640  ns
        17    007A70    680  ns
        18    007A72    720  ns
        19    007A72    760  ns
        20    007A72    800  ns
        21    007A72    840  ns
        22    007A74    880  ns
        23    007A74    920  ns
        24    007A74    960  ns
        25    007A76  1.000  us
        26    007A76  1.040  us
        27    007A76  1.080  us
        28    007A76  1.120  us
```

Figure 4.2.2a address trace taken by a logic analyzer

In order to find out what really happened during the execution, we had to match all these addresses with the program code, i.e. to find out what path through the program the execution had taken. For example, as shown on the picture, in column "Lab1 Hex", 000100 was the address to the CAN-interrupt vector, which triggered this measurement; 480ns after the measurement started the CAN-interrupt routine was called at address 007A6E. As mentioned before, the address of each instruction could be obtained either by aiT or CrossView. Often, the program code was analyzed, for example, after the address of an if-statement, the instruction subsequent was always analyzed to see whether the code went into the if-statement or the else-statement. For each address the corresponding instruction and statement in the program code was extracted. Finally the whole execution trace was interpreted, and the complete execution path was discovered. An interpreted execution path could look like the following:

```
ADRS_LIST_56a.txt - Anteckningar
Arkiv  Redigera  Format  Hjälp
Label      >    Lab1      Time
Base       >     Hex    Absolute
___________    _______  __________

         0    000100      0  s//interrupVektorn för Can-interruptet.
        12    007A6E    480 ns//start adressen för Can-interrupt rutinen.
        63    007A8A  2.520 us//call instance__13CommunicationSFv
        74    000CDA  2.960 us//från den adressen börjar "COMMUNICATION_3_PR". står i map-file
       107    007A8E  4.280 us//tillbaka till basbloket IntCan1
       121    001010  4.840 us//in i Interrupt
       158    00101E  6.320 us//static ECAnMessage mess;
       162    001034  6.480 us//första gången In for(i=6;i;--i)_loopen
       177    0010CA  7.080 us//in i Interrup(loop)
       188    00103A  7.520 us//3A - 3E kör koden:
       189    00103A  7.560 us//"interrupID = m_can.ReadInterruptIdentifier()"
       195    00103E  7.800 us//anropa m_can.ReadInterruptIdentifier()
       203    001042  8.120 us//switch (interruptID), men gick inte in där dirkt
       207    0154D4  8.280 us//in i ReadInterruptIdentifier() koden
       243    0154E2  9.720 us//return från ReadInterruptIdentifier() koden
       244    001042  9.760 us//In i switch(interruptID)
       281    001094  11.24 us//default (vanligt msg)
       289    001098  11.56 us//inför if(interruptID <= 16)
       293    00109A  11.72 us//if-vilkoren är true
       347    0010B4  13.88 us//Call ReadMessageObject()
       358    0059DC  14.32 us//in i koden för ReadMessageObject()
       395    0159EC  15.80 us//inför if ( ucMessageObject && (ucMessageObject < 15) )
       425    0159FA  17.00 us//if vilkoren är true, gick in.
       491    015A16  19.64 us//kolla om en bit är satt
       492    015A16  19.68 us//motsvara koden if(*pucMCFG[ ucMessageObject ] & 0x04)
       507    015A5C  20.28 us//biten är inte satt, därför jump till else-satsen
       508    015A5C  20.32 us//calculate STD Identifier
       563    015A76  22.52 us//ut else-satsen
       590    015A80  23.60 us//jump if bit not satt
       591    015A80  23.64 us//motsvara koden if ( *pusMCR[ ucMessageObject ] & 0x0800 )
       598    015A82  23.92 us//pre-hämtade adress den skall hoppa till om if-satsen är sann
```

Figure 4.2.2b An interpreted address trace

When all the three execution traces were interpreted, we found out that only the execution time at 56μs was a pure execution of a CAN-interrupt routine. In the other two executions the CAN-interrupt had been disturbed by a Regulator-interrupt. For example, for the execution of 184μs: 16.34μs after the measurement was triggered, a Regulator-interrupt occurred, and the handling of the Regulator-interrupt lasted until 144.5μs after the trigger, which mean that the Regulator-interrupt took 144.5 – 16.34 = 128.16μs to process. When we subtracted 128.16μs from the total execution time 184.8μs, the pure execution time for the CAN-interrupt was 56.6μs. Similarly for the execution of 169.8μs, the Regulator-interrupt took 112.92μs, so the CAN-interrupt took actually 169.8 – 112.92 = 56.88μs. Therefore, all three executions of the CAN-interrupt had in fact very similar execution times and were all between 56μs and 57μs.

From these execution scenarios we also discovered that the different kinds of interrupts had different priorities. Obviously, the Regulator-interrupt had higher priority than the CAN-interrupt, and whenever a Regulator-interrupt occurred it could break in directly in the middle of a CAN-interrupt to be handled first.

Was the longest execution time for a CAN-interrupt actually 57μs? So far all the measurements were done quite long time after the circuits began to run. Could the CAN-interrupt take longer time at the system start-up phase, when WDS and PSA communicated for the first time? With this question, new measurement was done right after the power supply was given to the circuits. A new execution time was caught and the result showed a significant difference: an execution time of 75.1μs. The next step was to make a new interpretation was done for this new time, with the goal to find out why the times differed about 18μs. The traces of the execution paths were compared, and the difference was because of a static variable was created. At the first time the PSA-node received a CAN message from the WDS-node, a static variable “EcanMessage mess” was created, which took about 18μs extra. But this only happened at the start-up phase for the communication between the two nodes. The same measurement was repeated several times and every time we got the more or less the same time of 75.1μs.

4.2.3 The Regulator-interrupt routine on the PSA-node

For the measured PSA-circuit, two different kinds of welding methods could be chosen: MMA and Tig, using the menu on the display that was connected with the WDS-circuit. For different welding methods, the Regulator-interrupt routine would be different. We started by looking at the Regulator-

interrupt routine for Tig, because the two Regulator-interrupts that were wrapped in CAN-interrupt were of this kind.

TIG

The execution times of these two Regulator-interrupts were at 128.16µs and 112.92µs; there was a 15.24µs time difference between them. After studying the trace of these two executions, the reason for these extra 15µs was discovered: There was an if-statement in function SetInputValue(), where the execution with the longer time went in, but the other execution did not. Under the whole execution this function were called twice, and each time it took 7.5µs extra to go through the if-statement. Therefore, the total cost for the longer execution path was 15µs extra.

Afterwards, we looked at the Regulator-interrupt in isolation. The trigger address was changed to 000098h, which was the address to the Regulator-interrupt vector. The return address (X-pattern) was changed to 007A6C that we got from the aiT tool. This return address was checked up in CrossView too, which gave the same address. The measurements were done repeatedly, there were two times mostly caught: 128.1µs ± 0.5µs in about 95% of the total number of measurement made, and 113µs ± 0.5µs. The rest of times caught were all shorter than 128.1µs, so it was not so important to find out their execution paths since we were only interested in the longest one.

MMA

When we had finished the measurement of the TIG-method, we changed the welding method to MMA by alter the option on the display. The trigger and return address remained unchanged. Three different execution times were obtained: 99.5µs, 107.2µs and 113.5µs. Among them 98% of the measurements were around 99.5µs and 107.2µs. Actually we only got one measurement at 113.5µs.

All the execution paths were, once again, found out. The difference between 99.5µs and 107.2µs depended on an if-statement in function SetinputValue() which calculating the current, where the execution at 107.2µs went in to the if-statement, but not the execution at 99.5µs.

When we studied the execution path more in detail we discovered that another type of interrupt had occurred in the middle of the Regulator-interrupt, which took about 6µs extra to handle. From the map file, we found the address on the interrupt vector; it was an “ADCInterrupt” whose function was unknown to us. The only thing we know about the ADCInterrupt was that it had a higher priority than the Regulator-interrupt.

Another interesting thing that happened when we measured the execution time for the MMA welding method, was that sometimes we got very large time variations, such as, 820µs, 672µs, 533µs, 188µs etc. But after the path was found out, it showed that for all of these "strange" times, there had been a DisableInterruptTimer6() call occurring. That is to say, the Regulator-interrupt was turned off directly when it occurred. The system first finished execution of the code in main-loop that was running before the Regulator-interrupt, and then it took care of the interrupt. For example, for the execution time at 820µs: the Regulator-interrupt was turned off in 721.4µs, thus actually the Regulator-interrupt took only about 99µs. Therefore, if we ignore the time spent when the interrupt was disabled, all the execution time was below 108µs.

4.2.4 Results of Measurement on PSA-node

All the measurements on PSA-node reported on so far were totally non-intrusive. There was not a single line of program code changed or inserted for probing. The results of measurements are summarised in Table 4.2.4:

Type of interrupt	Time measured	Conditions
CAN-interrupt	56.5µs±0.5µs	Under normal execution.
	75.0µs±0.5µs	Under the start-up phase.
Regulator-interrupt:		
Method TIG	113µs±0.5µs	Under normal execution.
	128.1µs±0.5µs	Under normal execution.
Method MMA	99.5µs±0.5µs	Under normal execution.
	107.2µs±0.5µs	Under normal execution.

Table 4.2.4 Summary for measured results for CAN-interrupt and Regulator-interrupt at the PSA-node

There was large variations in the measured times. The reason for this was that it could take different time to access memory from execution to execution, and bit-flip occurred with slightly different frequency. Hence the execution time could vary with several hundreds nanoseconds.

4.3 WCET Estimations by aiT

One of the purposes of this thesis was to find out which measuring methods CC-System could use in their future work, how difficult it would be to use the method and how accurate the result could be. In this section, some general description will therefore be given to the readers about how the WCET was estimated by using the static time analysis tool aiT. The results measured by the oscilloscope, were also compared with the WCET estimations made by aiT, see Section 5.2. The WCET estimation by aiT was another Master Thesis [16] that was done by Ola Eriksson at CCS in conjunction with this thesis.

The aiT Worst-Case Execution Time Analyse is designed and developed by the AbsInt Angewandte Informatik GmbH in Germany. It is a commercial WCET analysis tool, and belongs to the category of static timing analysis method.

The WCET analysis in aiT is divided into six steps: first aiT reads the executable file; the executable format supported by this version of aiT is the IEEE-695, the ELF, the COFF and the XCOFF file formats. The format that was used was the IEEE-695 format, which was the only one of the supported formats that could be produced by TASKING. The second step is the construction of call graph and control-flow graph based on the executable file. The third step is called loop bound analysis, which tries to determine bounds on the number of loop iterations. The fourth step was called value analysis, which tried to statically determine possible values of register at different program points. The fifth step is called pipeline analysis; there the WCETs of the basic blocks[2] were computed. The last step is called path analysis, which derives the overall WCET from the block WCET. [30]

Before the WCET analysis could begin, some hardware information and software information, as well as the loop bounds and other flow information were required by aiT. Hardware inputs included, for example: which processor core that was used, which X-Bus features that were enabled/disabled and the memory configuration. The software information, which is also called user annotations, included: compiler used, context specification, memory accesses, known register values, known loop bounds, etc. The latter information was used to for example control, restrict or direct the flow of the program, making the calculated WCET bounds more accurate. The file that the user used to give annotations is called an ais-file. The annotations given by the user are very important, since most of the cases the estimation results depend very much on how the annotations are given. For example: if aiT could not find the loop bound automatically and the user did not give the loop bound in the ais-file either, then the WCET analysis would fail. Too large loop bound would cause overestimation of WCET, and too small loop bound would cause underestimation of WCET. For more information about the aiT tool

[2] A basic block is a sequence of instructions that have consecutive addresses and are executed sequentially, without any branching of control flow.

please see [16]. Regarding the annotations for the CAN-interrupt on the WDS-node, Ola gave different annotations for those three types of messages. See his report [16] for more detailed information.

The aiT tool could not only give the WCET estimation, but also provided a graphical interface, which could be used to see the code's structure from a call-graph. The call-graph could be shown in different levels of detail: just functions, basic blocks in function, or instructions in basic blocks. The execution time of each basic block could also be viewed either in clock cycles or the corresponding absolute time. The call-graph was very useful; it can simply help the user to have a complete view of the whole program code structure, and even give a trace of the worst-case execution path. The program that used to show the call-graph was called aiSee, also developed by AbsInt Angewandte Informatik GmbH. Following are examples of figures provided by the aiT Tool.

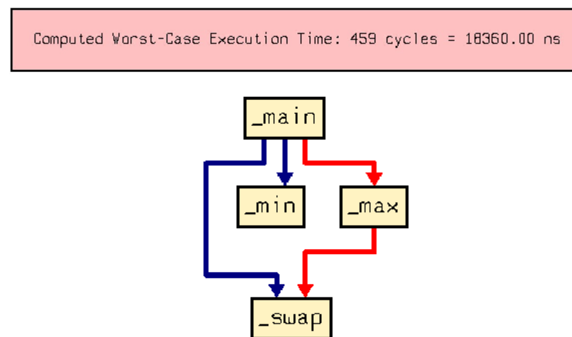


Figure 4.3a Call-graph with WCET provided by aiT. (The red arrows show the worst-case execution path)

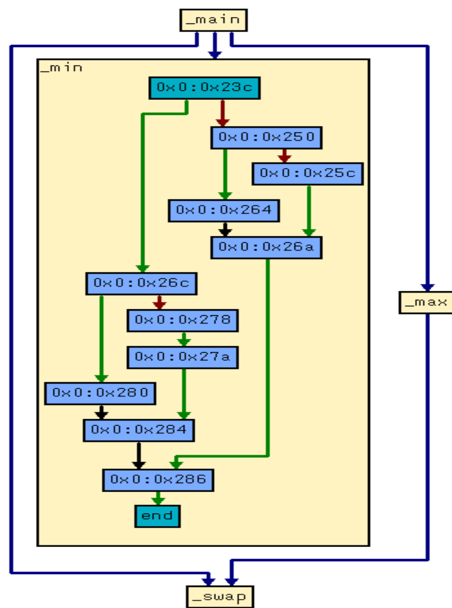


Figure 4.3b Basic blocks with their start addresses

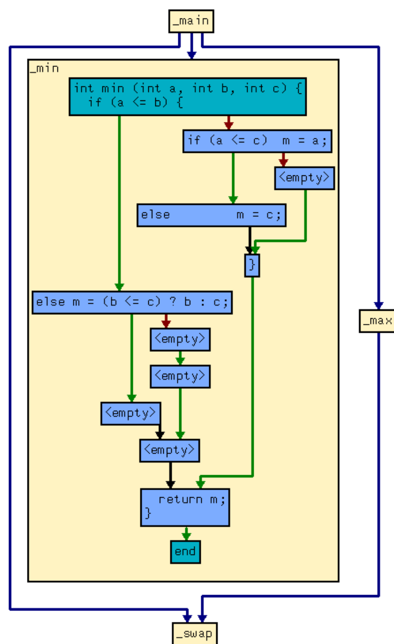


Figure 4.3c Basic blocks with source codes shown

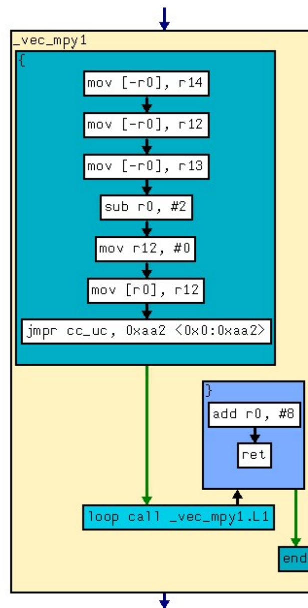


Figure 4.3d Basic blocks with instructions shown

4.4 Effort made to make the measurement result and aiT WCET estimation comparable

The results in Section 4.2.4 had been obtained in a non-intrusive way, but after studying their execution paths and comparing them to the worst-case execution path that we got from aiT, we discovered that the code had not taken the worth-case path reported by aiT for any of the execution times obtained. This was because that under the test environment, it was not possible to make the system behave in the worst way or to end up in the most complicate state. However, in order to be able to compare the measured time with the aiT's time estimations, we must make them have more or less the same execution path. To direct the programs to go as similar paths as possible, changes were made both in the program code (Section 4.4.1) and extra annotations were added to aiT's ais file (Section 4.4.2).

4.4.1 Directs program code according aiT WCET estimation's path

It was difficult to direct the execution for the measurement, every time the code got changed, new executable files had to be generated, both for the target hardware and for aiT. Furthermore, we did not know whether the changes could affect other parts of the system and the kind of consequences it could get. Therefore, only the code for Regulator-interrupt in the MMA method was changed according to the worst-case execution path obtained by aiT.

The WCET estimation using aiT for the Regulator-interrupt for MMA was 321.2µs without memory annotations and 314.7µs with memory annotations. First we compared the execution paths without

changing anything in the MMA code. The main difference between the measurements and aiT estimation was the paths taken in different case-statement in functions State() and CalculateCurrent(): the measurements took “case: OVERTEMP” in State() and “case: STOP” in CalculateCurrent(); while aiT took “case: START2 ” in State() and “case: WELDING” in CalculateCurrent(). The code was therefore modified so that the new execution went in the same branches as in aiT’s estimation. A new measurement was done; and a time of 216.8µs was obtained. The execution path of this new time was analysed, and then compared with the path in aiT, but the paths still differed in four places, all of them being if-statements. For example, there was an if-statement where the “m_bShortCircuitControl” was checked, and aiT estimation assumed that the short circuit was not true, but under the execution it was in fact a short circuit, and because of this the executions differed from the estimation. Therefore this 216.8µs was still not the longest time either. Then the program code was manipulated further so that in all those four if-statements the execution made exactly same decisions as aiT. The new code was compiled and downloaded to the node and aiT program. Measurements were once again repeatedly made, and two kinds of execution times were obtained: either around 260.2 µs or about 285µs. First the execution path of 285µs was compared with aiT’s estimation, there were still a few places, where the paths differed, and all of them were nested if-statement. Later on, the execution path for 260.2µs was compared with the one for 285µs; the difference was also a nested if-statement. However, for that if-statement, the 285µs measurement had the same selection as aiT, but not the 260.2µs measurement.

In fact, if we continued manipulating the program code, so that the execution could go exactly the same path as the aiT estimation, the execution time could become more and more closer to the estimation, but it was still not possible to get the same execution time as the estimation. This is because aiT sometimes makes some overestimations. Moreover, it was absolutely not sure that the execution path that we got from the aiT estimation was feasible, because it depended very much on the annotations given in the ais file. That is to say, if incorrect annotation was given to aiT, the calculated worst-case execution path it found could not be really trustable.

4.4.2 Directing aiT according to the measurement’s execution path

As mentioned above, it was very difficult to direct the measurement to follow the same execution path that we got from aiT. Fortunately, on the other hand, it was relatively easy for aiT to estimate the execution times according the execution paths we had in our measurement. Although they were not the WCET, it could give an idea roughly how much overestimation that was made in the estimated execution times.

There were several ways to direct the execution path in aiT. Some examples are: flow-annotations could be used to adjust the execution path in the for-loop or a switch-statement; value of conditions

could be used to decide the execution path for an if-else-statement. For detailed information on how to give annotations to direct the aiT execution, please see [16].

During the measurements, we also discovered that the address trace that we got from the logic analyzer could provide valuable help to the aiT annotations. For example, we could see from the address trace the addresses that were accessed when reading or writing data in certain instructions. Later on we used the map file to see which memory interval these addresses belonged to, and gave the interval in the annotation for the instruction. Thereby, the estimated execution time for the instruction could be improved, i.e. get closer to the measured time. This is because when aiT doesn't know which parts of memory to access, it will use the worst memory accessing time.

Some settings in the aiT program, such as addresses to the register pointer and the stack pointer were also added to improve the estimation.

5. Result

In this chapter we will give some comparisons between the dynamic methods and static methods tested in this thesis. The advantages and disadvantages of each kind of methods will be discussed in aspects of difficulty and accuracy. The measurement results will also be compared to each other.

5.1 Dynamic Methods VS. Static Methods

Dynamic Methods

Advantages:

1. If there have been only changes in the program code and the test environment remains unchanged, it is easy to make new measurements.
2. Dynamic measurements can also give other timing behaviours of the system, such as ACET, and people can get a distribution of the execution times.
3. Easy to measure execution time for different kind of optimisations of the same code.
4. Do not require any detailed knowledge of the program code.
5. The execution path of every measured execution time is feasible.

Disadvantages:

1. Many preparations have to be done before the measurement and the quality of the measured result relies very much on having the test environment correctly set up. In order to get tight and believable result, all the relevant hardware components must be connected and work properly, which was not always possible. For example, if the components are developed and manufactured in different places.
2. With some methods, it is difficult to see if there has been interference included in the measured result. For example, if there has been pre-emption or delayed time function calls happening during the measurement. Only those methods that provide time traces can show the execution path.
3. It is never guaranteed that the longest execution time measured is the actual WCET. The WCET happens very rarely and the conditions to make it happen are normally unknown.
4. Can cause probe effects.

Static Methods

Advantages:

1. The only thing needed to begin the WCET estimation is the static WCET analysis tool and executable program.
2. No probe effect.

3. User can get a complete view of the program by the help of the call- and control-flow graphs that are provided by many of the static timing analysis tools.
4. Guaranteed safe WCET estimates, if correct annotations are given.

Disadvantages:

1. Requires good knowledge of both program code and the static tool. Wrong annotations can generate a non-feasible WCET estimate. Tight and safe WCET estimates require accurate settings both in the static tool and in the annotation file.
2. New annotations have to be made for every change in the program code or every kind of optimisation of the program code.
3. The types of processors and compilers supported by the static tools are limited. Static tools cannot analyse all kinds of systems.
4. There will always be overestimation. Too large overestimations will make the WCET estimate non-usable. The overestimations also make it hard to see if the obtained estimate corresponds to any actually possible execution that could be obtained by measurements.

5.2 Dynamic Methods' Results VS. Static Methods' Results

- *CAN-interrupt in WDS-node*

Message Type	***Conditions***	***Dynamic Measurement***	***Static Measurement***	***Difference***
MESSAGE	1 byte data	111 µs	116.15 µs	4.6 %
	8 byte data	135 µs	143.80 µs	6.5 %
MESSAGE15	1 byte data	102 µs	109.25 µs	7.1 %
	8 byte data	130 µs	136.90 µs	5.3 %
STATUS	Warning and Bus-Off are both false	60 µs	71.75µs	19.6 %
	One of Warning or Bus-Off is true	4.8ms	10.223ms	113.0%
	Warning and Bus-Off are both true	9.75ms	21.089ms	116.3%

Table 5.2a Comparison of the measured execution times and the aiT estimates of the CAN-interrupt at the WDS-node

Table 5.2a shows the comparison of the measured execution times and the aiT execution time estimates of the CAN-interrupt at the WDS-node. The differences between the measured execution time and the aiT execution time estimates are given in percentages. For the normal messages, the differences are between 4.6% and 7.1%, which is acceptable, because aiT always does overestimation when there is uncertainty at for example, memory access. For the STATUS message, on the other hand, the differences are quite big. The reason for this is that, when there was a STATUS message sent out, some error-handling functions were called, it was quite allot codes for those error-handling functions, and a lot of possible execution paths through the codes, aiT took the worst execution time. But since we used an oscilloscope as the execution time measuring method, we could not see which execution path the measured execution time has taken, so we could not direct the aiT to go the same execution path as the measured execution time. Therefore, the measured execution times and the aiT estimates had probably taken different execution paths.

- *CAN-interrupt and Regulator-interrupt in PSA-node*

Interrupt type	*Dynamic measurement*	*Without memory annotations*		*With memory annotations*	
		Static measurement	*Difference*	*Static measurement*	*Difference*
Can	56.5μs±0.5μs	58.60 μs	2.8 %	—	—
	75.0μs±0.5μs	79.00 μs	4.6 %	—	—
Regulator MMA	99.5μs±0.5μs	104.00 μs	4.0 %	102.90 μs	2.9 %
	107.2μs±0.5μs	113.45 μs	5.3 %	110.60 μs	2.7 %
	285μs±0.5μs	321.20 μs	12.5%	314.70 μs	10.2%
Regulator TIG	113μs±0.5μs	119.30 μs	5.1 %	117.10 μs	3.2 %
	128.1μs±0.5μs	138.20 μs	7.5 %	132.50 μs	3.0 %

Table 5.2b Comparison of the measured execution times and the aiT estimates of the interrupts at the PSA-node

Table 5.2b shows the comparison of the measured execution times and the aiT estimates of the CAN-interrupt and Regulator –interrupt at the PSA-node. The differences are also given in percentages. As show in the table, the differences are between 2.7% and 12.5%, which is acceptable. One thing should be noticed here, the measured execution time of 285μs of the Regulator MMA interrupt was obtained by modifying the program codes so that the execution would go the same path as the aiT estimate, and rest of the execution times had the aiT tool direct the execution time estimates to go the same execution paths as the measured execution times by altering the annotations in the ais files. As we can see in the Table 5.2b, even after we had tried to let the aiT execution time estimates and the measured execution times go the same execution path, with additionally memory annotations in the ais files, there were still differences between the measured execution times and the aiT estimates. The reasons for this are probably due to that, there were still small differences on the execution paths, and we were not able to give all the memory accessing annotations to the aiT tool.

5.3 Other Usages of Dynamic and Static Measuring Methods

The dynamic and static time analysis methods could be used in other areas. In this section we focus on the methods that have been used in this thesis and discuss other possible usages.

Both the logic analyzer and aiT could be used to make the “Time Accurate Simulation” simulator (TAS) to work better. Time Accurate Simulation was another Master thesis performed at CCS 2001 [31]. It is based on the existing simulation technology that has been developed by CCS. During the developing-phase of a program code, CCS uses TAS to simulate the functional behavior of the system. A PC is used to simulate the program that should be run on a target node, and normally the simulated program always works faster than the processor on the target circuit. In order to make the execution behavior of the program code more like on a real target processor, timing accurate management has been added to this simulator. Basically, TAS will provide timing information that tells the PC how long a certain piece of simulated code should take to run. The thesis [31] focused on how to add the time accurate behavior to the system but not on how to obtain the proper execution times needed to make the simulation time accurate. Both the dynamic and static time analysis methods could be used to obtain the proper execution time. However, there is a difference in what type of timing each method could provide. From the aiT tool, only a possible overestimated WCET could be obtained. If the WCET is given as the interrupt execution time, it will mean that every time the interrupt will execute for the longest possible time, which is not true in reality. Therefore it is better to use ACET (average-case execution time) obtained by the measurements, which will give a normal and more likely behavior of the simulation. Moreover, this will not cause much overhead. The easiest way to get ACET is to use a logic analyzer that is connected to the address bus, to measure the execution time of the interrupt over and over again. Exactly like what we did in this thesis. Each execution time should be saved, and then divide the total execution time by the number of measurements made. The more measurements that are done, the better the ACET will be.

The address traces produced by the logic analyzer are very helpful because they can give a processor activity list sorted by time. From this list people can easily see how and when the program code is executed, and whether the functions are executed exactly in the way that they are designed or planned to be. If there has been an error, the trace can also help us to find out why it happened and even how the error has affected other parts of the system. There are many other advantages by using a logic analyzer, such as a way to observe the number of loop iterations or the depth of the recursive function calls, which are important information when developing the software. We can also supervise the memory accesses made using the logic analyzer. For example, we can see whether there has been an illegal access in a protected memory area.

One advantage of aiT is that it has a well-developed graphical interface, and can provide a very clear and easy handled control flow graph of the whole program. This can help the user of the software developer to have a complete view of the whole system. For instance, from the graph we can see which function that is calling which function; all the possible execution paths in the program and the input's influence on the path choices. All this information is very valuable for the programmers and software developers. AiT also enables us to do temporal testing in parallel with the programming. Once one task in the system is implemented, we can do the WCET analysis directly for this task. Then the WCET obtained could be used as a parameter in the scheduling of the system.

The times from the logic analyzer and aiT could also be used to find places in the code, which are time-consuming and suitable for optimization.

6. Conclusions

All three methods tested for timing analysis: the oscilloscope, the logic analyzer and the aiT WCET analysis tool, can be used separately by CC-Systems for its program development. However, none of them is the single best method.

- An oscilloscope has limited amount of measuring points and limited granularity. It is difficult to see what has happened in small functions or if there has been pre-emptions, so it is not easy to find the exact execution path using the oscilloscope. The measured results have rather coarse time resolution and cannot be guaranteed to be the WCET.
- A logic analyzer is a much better option for a detailed timing behaviour analysis. It can observe several measuring points simultaneously, and give results on a level of single instructions. The activity trace of the address-bus can be used in both functional testing and temporal testing. Furthermore, the measurements can be done in a totally non-intrusive way. The shortcoming of the logic analyzer is that the measured result cannot be guaranteed to be the WCET.
- The aiT tool can do the WCET analysis without the target system and the result is guaranteed to be a safe WCET estimate. But it requires that the user to have a very good understanding of both the aiT tool and the program code; otherwise the quality of the estimated results can be seriously affected and resulting in large overestimations.

However, a better way to do the WCET analysis is probably to combine the dynamic measurement with the static analysis so that the two kinds of methods are able to compensate each other. From the static method, we may get a complete view of the code executed together with the possible longest execution path. Using dynamic methods we can find out if this path is feasible or not, and the required conditions/inputs for the program for going this longest path. If we can, somehow, make sure that these required conditions are fulfilled, and afterwards measure the execution time of this path by a dynamic measuring tool, e.g. a logic analyzer, then we will be able to say, with high confidence, that we have found the actual WCET of the program. On the other hand, the dynamic measurement can provide useful information, such as addresses for annotations of memory, to get tighter static analysis results.

A dynamic method, such as a logic analyzer can also be used to get other timing behaviours of the system, such as ACET. The ACET can be used by the Time Accurate Simulation in order to get a more reliable simulation for CCS's program development.

7. Designing and Creating Code for Analysability

It is very important is to think about the analysability already in the design-phase of the system and through the whole development of software the developers should have the analysability on their mind. Following are some suggestions:

- Implement different functionality in tasks instead of using a big loop with different kinds of interrupts. Minimize the amount of interprocess communication, synchronization and resource sharing.
- Avoid using recursive function calls, deep-nested if-else-statements and switch-statements, thereby making the code structure as simple as possible.
- Avoid using pointers in branch-conditions and loop-conditions, thereby making the code as predictable as possible.
- Avoid returns in the middle of the code and organize the error-handlers in a simple and clear way, such as throw and catch commands.
- Write comments as detailed as possible, thereby helping other programmers to more easily understand the code.

8. Future Work

Due to the limited amount of time and the limitation of the testing environment there are many things that could have been done better or tested in more detail.

When comparing our results for the WDS-node, there are quite large difference between the measured results and the static estimations of aiT. The reason for this is still unsure. Is it an unacceptable high overestimation by aiT, or are the measured results are really far from the real WCET?

In order to make the measured results comparable to the aiT's estimations, when measuring the Regulator-interrupt of the MMA method on the PSA-node, the program code could be even further directed according to the aiT's execution path. There are still places where the execution trace differs from the static execution path. Since the execution times did not differ too much, we didn't bother to alter the code to make it act exactly like the aiT's execution path.

The same effort should be done for all other codes that were tested. That is to say, all the measured execution paths should be compared with aiT's execution paths to discover the differences, the code should then be directed to go the worse-case path obtained by aiT, in order to see if it is possible to obtain the WCET by measurement and to compare the measured results with the estimations obtained by aiT.

A logic analyzer should be also used to measure the code of the CAN-interrupt on the WDS-node; in order to find out if there has been any other interrupts that pre-empted the CAN-interrupt.

The time analysis using Linux time function rdtsc() should also be tested more throughoutly. In this thesis, only some general descriptions are given to the reader. However, neither real tests nor measurements were done to evaluate the difficulty and accuracy of the method. Moreover, a list of all available time functions should be made to compare their portability, overhead and resolution. Furthermore, comparisons of time functions of different operating systems, or different versions of the same operating system can also be interesting to make. Another thing that should also be interesting to evaluate is how much the program needs to be modified so that these time functions can be applied in CCS' product development.

Bibliography

[1] A. Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. Phd thesis, Uppsala University, Uppsala Sweden, 2003. ISBN 91-554-5671-5

[2] Steven F. Barrett, Daniel J. Pack. *Embedded System Design and Applications with the 68HC12 and HCS12*. 2005 by Pearson Education, Inc. ISBN 0-13-140141-6

[3] Stephan Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. Phd thesis, Universität des Saarlandes, Saarbrücken Germany, 2004. ISBN 3-937436-00-6

[4] Company homepage for CC-Systems AB, URL: http://www.cc-systems.com

[5] Company homepage for ESAB, URL: http://www.esab.se

[6] Homepage for The Advanced Software Technology Centre (ASTEC) in Uppsala, URL: http://www.astec.uu.se

[7] Homepage for Vinnova (Swedish Agency for Innovation Systems), URL: http://www.vinnova.se

[8] David B. Stewart, *Measuring Execution Time and Real-Time Performance*, Embedded Systems Conference, San Francisco, CA. April 2001

[9] Company homepage for Metrowerks, URL: http://www.metrowerks.com/MW/Develop/AMC/CodeTEST/default.htm

[10] Course Description of TimeSys TimeStorm/TimeTrace Training, URL: http://www.timesys.com/_content/media/docs/courses/5.pdf

[11] Homepage for *WindView user's guide 2.2,* URL:http://www.eelab.usyd.edu.au/tornado/docs/windview/wvug/

[12] Homepage for explanation of item TLB, URL: http://ciips.ee.uwa.edu.au/~morris/CA406/TLB.html

[13] Company homepage for AbsInt, URL: http://www.absint.com

[14] Company homepage for Rapitasystems, URL: http://www.rapitasystems.com

[15] Company homepage for Tidorum Ltd., URL: http://www.tidorum.fi

[16] Ola Eriksson, *Evaluation of Static Time Analysis for CC Systems*, Master Thesis, Mälardalen University, Västerås Sweden, 2005

[17] Henrik Thane. *Design for Deterministic Monitoring of Distributed Real-Time Systems.* Technical Report, November 1999.

[18] Homepage for Oscilloscope Tutorial, URL: https://www.cs.tcd.ie/courses/baict/bac/jf/labs/scope

[19] Explanations for item "Logic analyzer" at online dictionary www.answers.com, URL: http://www.answers.com/topic/logic-analyzer

[20] Wayne Wolf, *Computers as Components Principles of Embedded Computing System Design*, 2001 by Academic Press. ISBN 1-55860-541-X

[21] HP 1670D-series *Logic Analyzer, User's Guide*, Publication Number 0167-97004, First Edition, August 1996.

[22] The TASKING Embedded Development Environment EDE Manual.

[23] The C166/ST10 CrossView Debugger Pro user's guide.

[24] Company homepage for Hitex, URL: http://www.hitex.de

[25] Company homepage for Spacetools, URL: http://www.spacetools.com/tools4/space/44.htm

[26] *Using the RDTSC Instruction for Performance Monitoring,* URL: http://www.math.uwaterloo.ca/~jamuir/rdtscpm1.pdf

[27] Peter I. Kujanpää, *Measuring Linux Kernel Performance*, Master's Thesis Mälardalen Univeristy. Sweden, 2004

[28] *Infineon C167CS-LM microcontroller Manual*, http://www.infineon.com/cmc_upload/documents/026/503/c167cs_l_4r_datasheet_v20.pdf

[29] *Flash-memory Am29F800B Manual*, URL: http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/21631c2.pdf

[30] *WCET Analyzer aiT for C166/ST10, User documentation for Window-Version 1.5r2*

[31] M Nilsson, *Time Accurate Simulation*, Master's thesis Uppsala University, Sweden (2001)