A Remotely Configurable, Dynamic Acceptance Filter for the Reliable Logging of CAN Data in a Time-Triggered Embedded System

Submitted by: Christopher Barlow

Project Supervisor: Dr Michael J Pont

THESIS

Submitted in partial fulfilment of the requirements
for the degree of Master of Science
in Reliable Embedded Systems
at University of Leicester, 2013

Acknowledgements

Thanks to Dr Michael J. Pont for his tuition in the MSc Reliable Embedded Systems programme. The intensive taught modules have given me the knowledge and confidence to pursue a career in automotive embedded systems, which I would not have been able to achieve without attending these courses.

I would also like to thank the team at Smith Electric Vehicles, in particular Ross Cooney, for their continued support and encouragement during my work placement at the company. The exposure to real-life challenges and problem-solving opportunities not only gave me an application area to work to in this project, but also gave me the motivation to complete my work to a high standard.

Thanks also go out to my partner, family and friends who continue to offer support and encouragement for all of my endeavours.

Abstract

This thesis presents the concept of dynamically modifying the acceptance filters of off-the-shelf Controller Area Network (CAN) hardware, with the purpose of efficiently discarding unwanted messages on a busy network. The bus topology means that any node has visibility of every messages transmitted by the other nodes on the network. Challenges arise when a node only requires visibility of a subset of these messages; the node must either use software lookup tables to determine which messages to keep and which to discard, or use hardware 'acceptance filters' to prevent the unwanted messages from entering the CAN controller's receive buffer. In a Time-Triggered embedded software environment, the former presents the problem that unwanted messages take up space in the buffer, which could lead to messages being missed if the buffer polling frequency is insufficient. Using acceptance filters is preferred in these circumstances; however, the size of the acceptance subset is then limited by the specification of the CAN controller hardware.

Related research exists surrounding application layer protocols that schedule CAN message transmission, providing control of the message timing. However, in an application area such as the automotive industry, where subsystems are often designed independently, it is not always possible to have such finite control over message transmission. This project focusses on using knowledge of existing network conditions to predict the next message to arrive and schedule CAN controller mailbox configuration at runtime. This flexibility allows the messages for acceptance to be defined by a remote system after the embedded software has been compiled.

A simulation application is written in 'C' to read text-based CAN logs and predict the behaviour of the proposed paradigm. Further to this, an embedded 'C' implementation of the algorithm is developed to operate from a Time Triggered Hybrid (TTH) scheduler running on a Texas Instruments TMS320F28335 microcontroller.

A supporting desktop application, written in 'Processing' allows the acceptance message subset to be configured remotely via an RS232 connection, and monitors performance of the embedded system. Performance comparisons are made between the new embedded system, the simulation application and an existing automotive data-logging device.

Issues discovered during the development process are discussed, including messages arriving out of order, the management of varying message cycle times, and handling of data burst conditions. The timing behaviour of the new system is analysed, and limitations and future scope for development are presented.

Table of Contents

List of Tables and Figures

Chapter 1 Introduction

1.1 Overview

Controller Area Network (CAN) has become a standard method of communication between embedded systems in automotive applications [1]. CAN messages contain data transferred between nodes on a bus network and each message is given a unique identifier (ID) to provide context to the data field of the message. Nodes on the CAN bus use CAN controller hardware to buffer messages that have been transmitted by other nodes.

An electric commercial vehicle company uses a telemetry device to log data from a CAN network on their vehicles. Logged data are transmitted via GPRS to a dedicated server that performs the post-processing necessary to store the information in a database. The company are now exploring new hardware and software options for an updated device, which include the ability to modify remotely the subset of CAN messages logged by the device. Since the data are of high importance to the company, it is imperative that the embedded software operating on the device is reliable and, because of this, a Time-Triggered (TT) scheduler [2] has been proposed to replace the predominantly event-triggered architecture currently in use. It is therefore necessary to investigate software logic that complements the inherently predictable nature of the TT scheduler, without compromising the compression and transmission protocols that are currently in use. Although this architecture should allow performance guarantees to be made to the company [3], using a TT scheduler to log data events is not without its challenges, which will be addressed in this project.

1.2 Problem Statement

The inherent nature of a CAN bus network is that, at the physical level, all nodes have visibility of every message that is being transmitted. A node therefore has to interrogate the identifier of a received message in order to decide whether it needs to read the data content. This interrogation can be carried out in software, by comparing the identifier to a lookup table containing those to be accepted. Alternatively, the hardware can be configured with ‘acceptance filters’, which restrict the

identifiers that are allowed into the CAN controller's receive buffer [4]. Both methods have drawbacks; if acceptance filters are used, the number of unique identifiers that can be accepted by a node is limited by the filters available within the CAN controller. Therefore, if a node has interest in a large subset of messages, the solution has to involve software filtering. When interrogating the identifier in software, each message uses up space in the receive buffer regardless of whether the message is of interest to the node. This is particularly troublesome if the node is on a busy network where it is possible to miss messages if the software allows the buffer to become full.

Further problems arise if the identifiers to be accepted by a node are not known at compile-time. In these circumstances, neither software acceptance tables nor hardware mailbox configurations can be hard-coded in the embedded software. To overcome this, software mechanisms have to be put in place to allow configuration in the field.

1.3 Research Scope and Contributions

This project focuses on the development of a novel approach to the problems stated above, whereby the CAN message identifiers to be accepted by a real-time embedded system are specified post-software compilation, along with the known order and timing properties of each unique message. This acceptance list is transmitted to the embedded system from a remote configuration application and used to produce filter configuration sequences at run-time. Using a time-triggered architecture, these configuration sequences are used to predict the identifiers of the next messages on the CAN bus for any given software 'tick'. A periodic task uses this prediction to modify the CAN controller acceptance filters dynamically for the expected messages.

The performance of this system is tested first through a desktop simulation application and secondly with an implementation on the target microcontroller. A desktop 'remote configuration' application is written to transmit the logging list to the embedded system. Comparisons are made between the embedded implementation and the simulation, and with an existing, polled-buffer data-logging device.

The project draws from the subject of Time triggered scheduling, which is a predominant theme in the MSc Reliable Embedded Systems programme and, in particular, the Time-Triggered Hybrid (TTH) scheduler introduced in module A2. It also involves the topic of Controller Area Networks (CAN), which is presented in module B3, as well as shared resource concepts, which are covered in modules B2 and B3. Some monitoring and instrumentation techniques learned from module B2 are also applied.

1.4 Thesis Organisation

Chapter 2 of this thesis introduces the concepts of Controller Area Network and Time-Triggered software architecture. Consideration is given to the benefits and complications of using the two technologies together in an embedded system.

Chapter 3 discusses work relating to the subjects introduced in Chapter 2. Areas for further research are identified, and the relevance and contribution of this project are highlighted.

An existing, comparable, embedded system is described in Chapter 4, highlighting some areas for improvement in the current software architecture. An 'instrumentation' approach is described, which is used to measure performance of this software for comparison with the proposed system.

Chapter 5 discusses the proposed algorithm and its implementations in this project. A software simulation is introduced, which uses text-based CAN logs to determine the 'hit rate' of the algorithm for various network conditions. An embedded software implementation and a bespoke remote configuration tool are also described.

The above implementations are analysed in Chapter 6. Results from testing both the simulation and embedded solution are presented, along with comparisons with the existing system.

In Chapter 7, various behavioural factors discovered during the development process and are discussed that influenced the final algorithm. Also presented are limitations to the system and areas for future development. The final chapter presents some concluding remarks.

Chapter 2 Background

2.1 Introduction

This chapter will introduce the underlying technology relating to this project; Controller Area Network (CAN) is introduced and the concepts of Event-Triggered (ET) and Time-Triggered (TT) software architectures are discussed, along with the challenges involved in using the two technologies together.

2.2 Controller Area Network

Controller Area Network (CAN) is a serial data communications protocol that uses a two-wire serial data bus. Bosch's CAN Specification 2.0 [5] describes the Physical and Medium Access Control (MAC) layer and part of the Logical Link Control (LLC) layer of the OSI reference model [6]. The majority of the LLC layer and the Application layer have been left open to interpretation, allowing engineers and developers a great amount of flexibility when designing CAN based systems [4]. This project will be working within the scope of the Acceptance Filtering aspect of the LLC layer, and the Application Layer (see Figure 2-1).

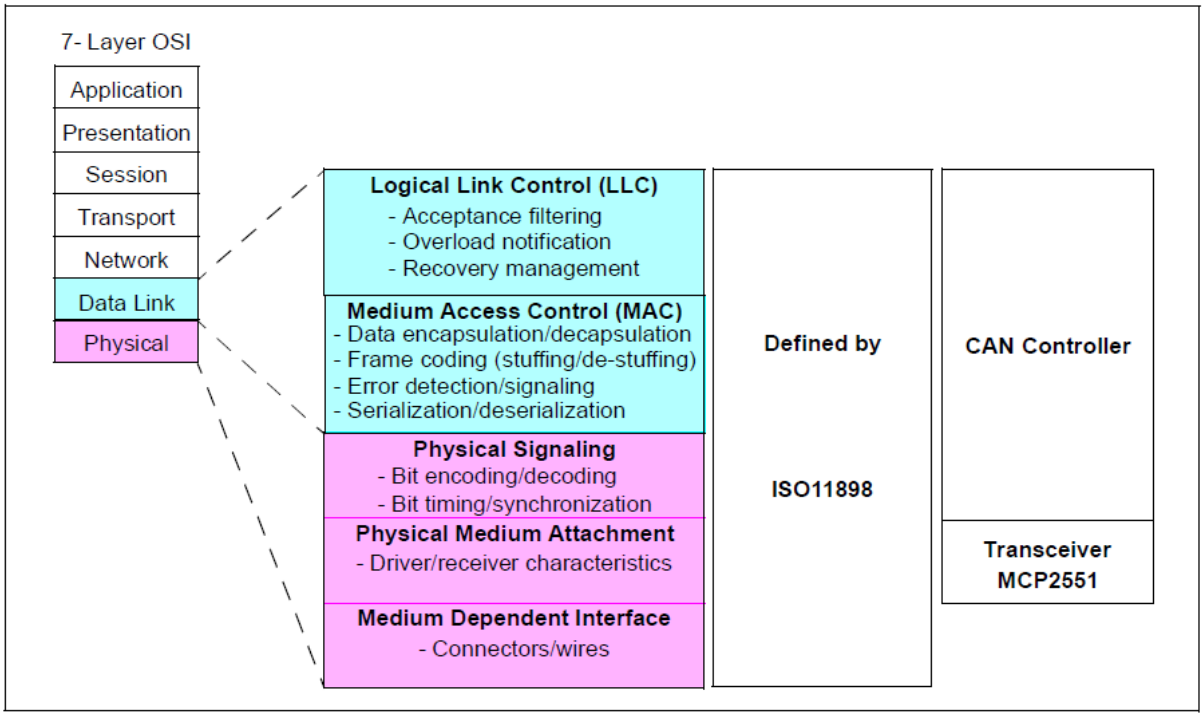


FIGURE 2-1: CAN AND THE OSI REFERENCE MODE [7]

2.2.1 CAN Hardware

CAN communication is achieved through the integration of dedicated hardware into the embedded system. This hardware, called a 'CAN controller', can be either a stand-alone IC, or an integrated block built into a microcontroller [4]. The function of the CAN controller is to employ the CAN specification to transmit data to the other nodes on the CAN bus and to receiving data transmitted by the other nodes, storing it for the retrieval of a host (e.g. a microcontroller). The control of CAN message reception is classified as either 'basic CAN' or 'full CAN' [4]. Figure 2-2 shows a two-node CAN network, utilising both types of CAN controller:

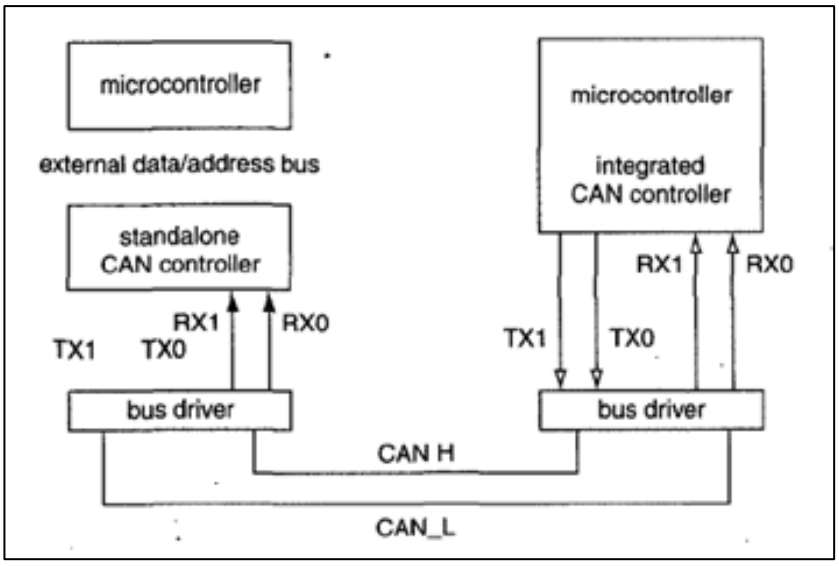


FIGURE 2-2: VARIANTS OF CAN HARDWARE [4]

2.2.2 Basic CAN

In 'basic CAN', a buffer is used to store incoming messages in the controller hardware. This buffer is usually in First In, First Out (FIFO) arrangement, the depth of which varies between hardware manufacturers [1]. This means that the client processor must read all messages in the buffer, and interrogate the identifier in order to ascertain the context of the message data field. In order to avoid wasting processing time, such hardware usually has the option to set several 'acceptance filters' that ensure that only relevant messages are stored in the buffer. The number of acceptance filters, again, varies between manufacturers.

A typical software flow to retrieve data using an acceptance filter would be as follows:

- A message arrives on the CAN bus.
- The CAN controller interrogates the message identifier.
- The identifier passes an acceptance filter and the CAN controller stores the message in the FIFO.
- The CAN controller will either generate in interrupt, or raise a poll-able flag to indicate message arrival to the microcontroller.
- The microcontroller responds to the flag and reads the message from the FIFO, and interrogates the identifier in order to determine where to store the data.

In modern hardware, CAN controllers can be integrated into the microcontroller silicone. This has the advantage that it is possible for CAN messages to be stored directly in Direct Memory Access (DMA) registers, allowing for faster retrieval of data [8]. This advance has brought with it more sophisticated methods of handling CAN messages.

An example of a microcontroller with an integrated 'basic CAN' controller is the *STM32F407ZGT6* from STMicroelectronics, which provides 28 filter banks, each capable of holding four 16-bit Identifiers [8]:

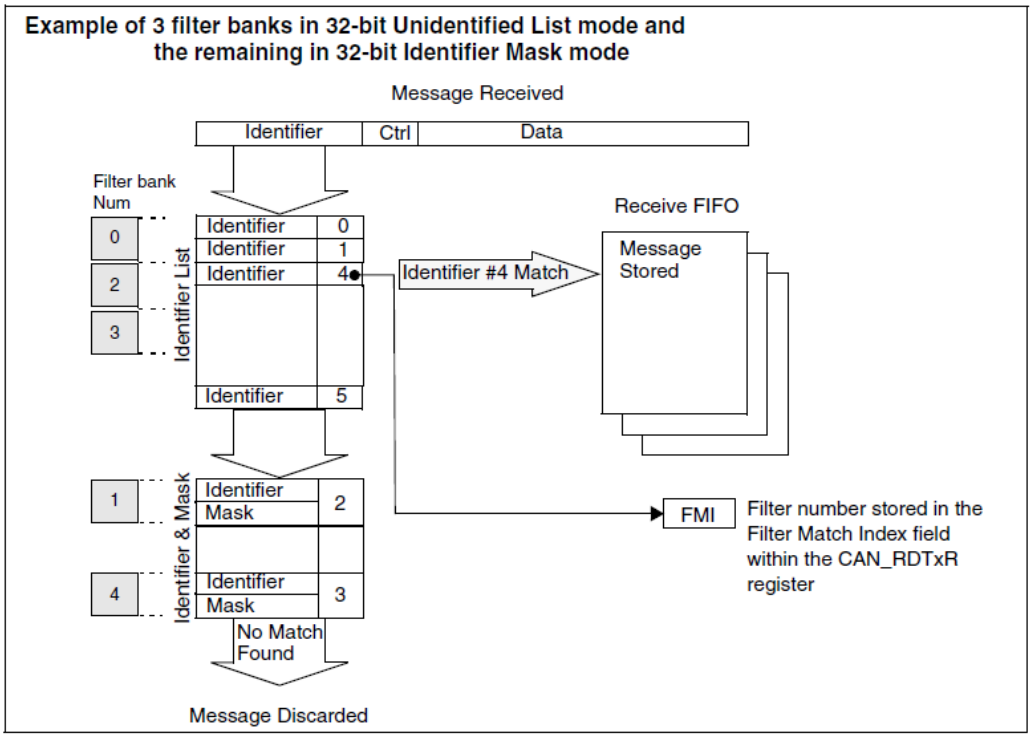


FIGURE 2-3: CAN ACCEPTANCE FILTERING ON AN STM STM32F407ZGT6 PROCESSOR [8]

As a message arrives from the network, the controller transparently compares the identifier with those in the filter lists, and stores it in a specific memory location. If the message does not match any of the filters, it is discarded. The controller stores the FIFO number for each successful acceptance filter match in a 'Filter Match Index' registry field, supplying the embedded application with message context on which to perform storage logic.

2.2.3 Full CAN

In 'full CAN', the hardware comprises dedicated areas of memory, known as 'message objects' or 'mailboxes', which are configured to receive or transmit messages with pre-set identifiers. An example of this is the eCAN arrangement present Texas Instruments C2000 family processors [9]. Instead of storing all CAN messages in one FIFO and leaving it up to the microcontroller to move the data to RAM, the eCAN controller contains 32 mailboxes in which can be configured to be either transmit (Tx) or receive (Rx) mailboxes, each with individual acceptance filters.

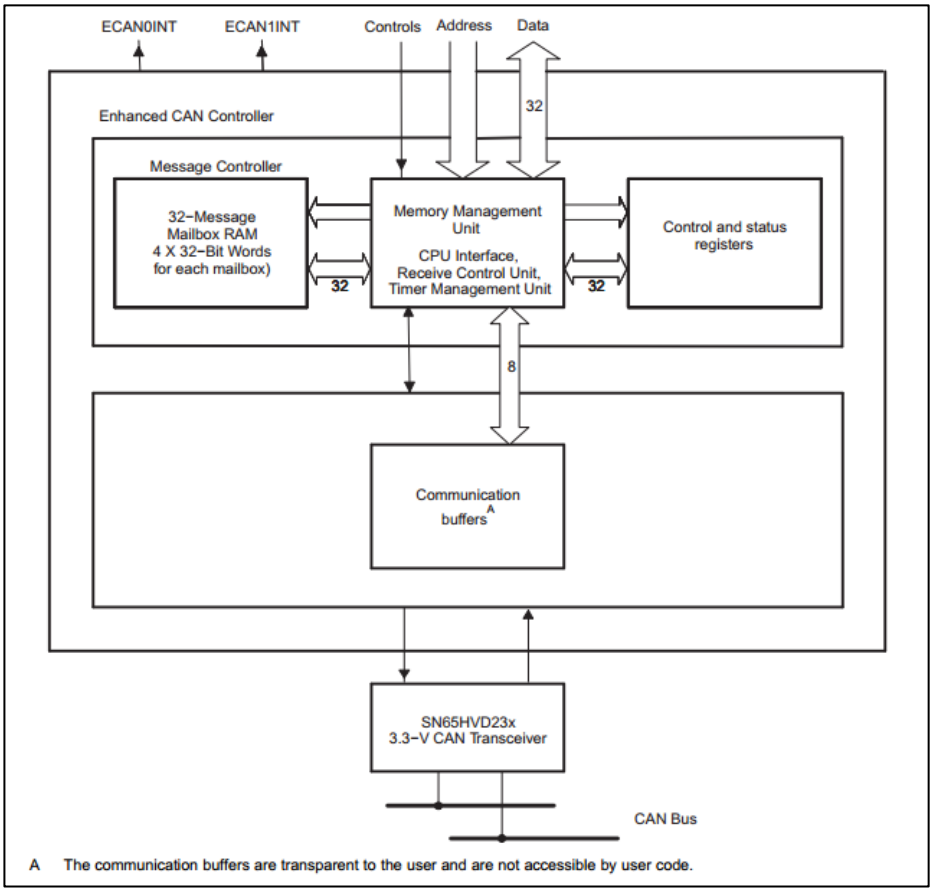


FIGURE 2-4: ECAN BLOCK DIAGRAM AND INTERFACE CIRCUIT [9]

With this arrangement when a message arrives, the microcontroller can read the data directly from the mailbox and has knowledge of the data context without using any look-up mechanisms in software:

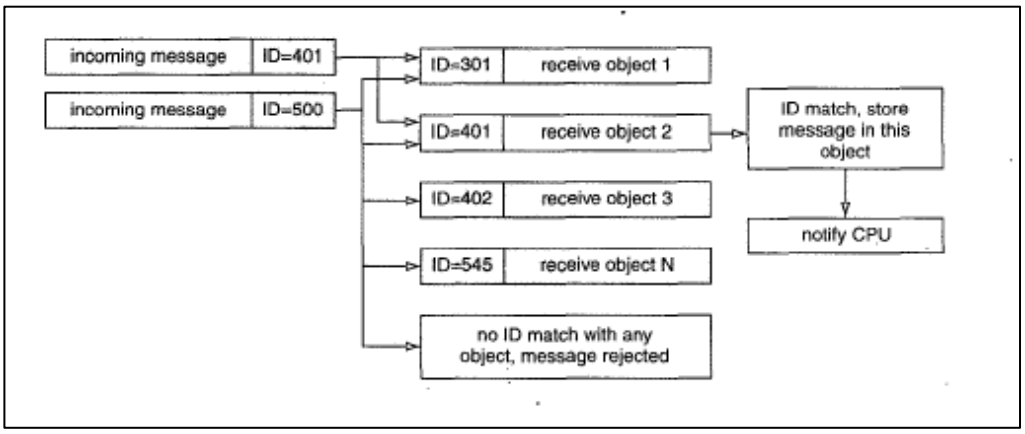


FIGURE 2-5: PRINCIPLES OF 'FULL CAN' OPERATION [4]

2.3 Real-Time and Time-Triggered Software Architecture

Real-time software can be defined as software that must complete tasks to a specified deadline. In embedded systems, software must respond to one or many 'events', which include interrupts from CPU peripherals and inputs from other systems or devices.

One way of controlling how the software responds to these events is to use interrupts for each event. These interrupts cause the software to suspend its current process, service the interrupt, and then resume from where it left off. This means that asynchronous external inputs have control over the execution order and timing of the software, which can lead to the system behaviour becoming highly unpredictable. A software layer known as a Real Time Operating System (RTOS) can be added to these systems to give some control over the priority given to individual software tasks; however, there are many complexities involved in these systems [10].

A preferred method for safety-critical or time-critical embedded software is to use Time-Triggered (TT) architecture [2]. By using this approach, the timing of software task execution can be guaranteed, with an accuracy measured in fractions of microseconds, to follow a fixed, pre-determined schedule.

The backbone of this architecture is a scheduler driven by a single, a timer-driven interrupt event. Software operations are divided into 'tasks' and the timer interrupt is used to generate periodic 'ticks' that allow the scheduler to keep track of elapsed time. Hard-coded properties control the timing of the tasks using an 'offset' (the time until first dispatch) and 'period' (the time between subsequent dispatches). Each tick invokes a scheduler that decrements a timer associated with each task. When the timer reaches zero, the task is dispatched, and the timer reloaded with the period value. Figure 2-6 below shows the behaviour of a 'Time Triggered Co-operative' (TTC) scheduler.

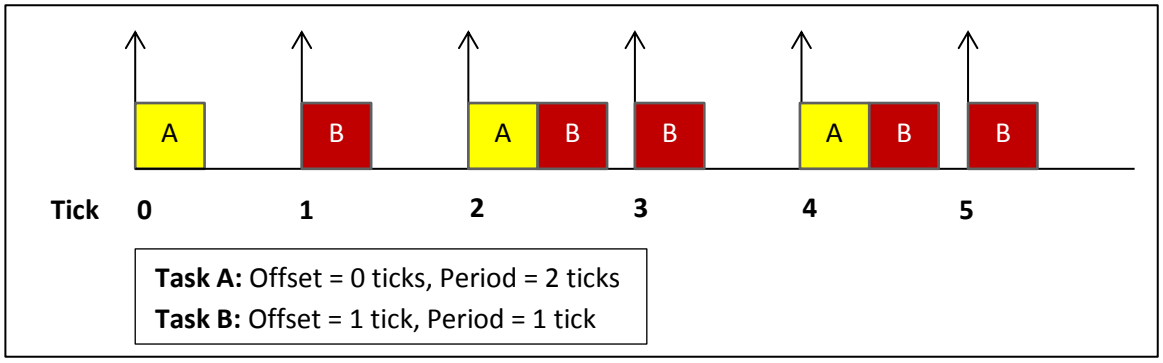


FIGURE 2-6: A SIMPLE TIME-TRIGGERED CO-OPERATIVE (TTC) SCHEDULER

Task A has a higher priority than Task B, so Task A is executed first when the two tasks share the same tick. This method works as long as a task does not overrun a tick occupied by a time-critical process. This would cause the proceeding task to be executed once the overrunning task has completed.

Figure 2-7 shows a 'Time Triggered Hybrid' (TTH) scheduler which uses 'planned pre-emption' to ensure that time-critical tasks are executed on time [11]:

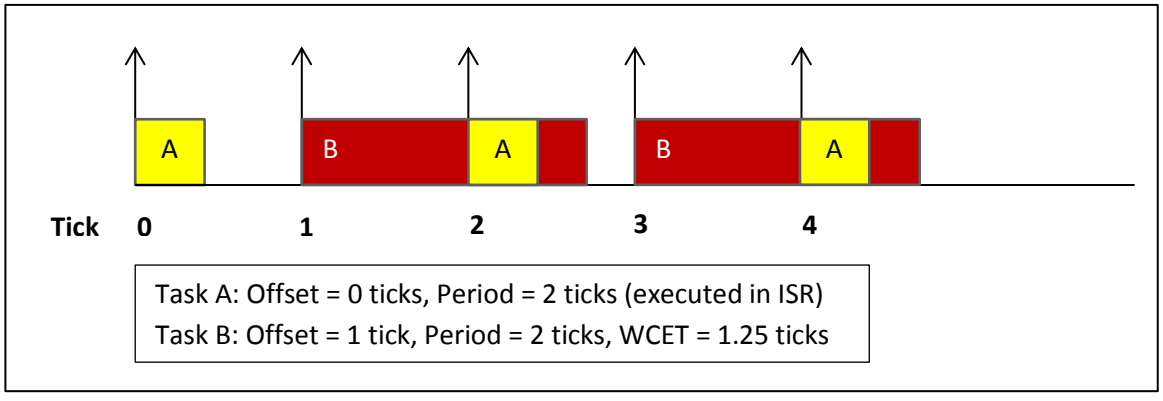


FIGURE 2-7: A SIMPLE TIME-TRIGGERED HYBRID (TTH) SCHEDULER

Here, Task B takes longer than one tick to complete, however Task A is configured to execute from the Interrupt Service Routine (ISR) and therefore pre-empts Task B. This means that Task A is guaranteed to run on time, and Task B will be suspended until Task A completes.

One rule that guarantees accurate timing in a TT system is that the timer driving the scheduler is the only interrupt allowed. This means the software must poll peripherals in order to detect any external events such as GPIO state changes and data reception, but ensures that asynchronous events will not prevent the CPU from executing a task on time. With knowledge of the CPU instruction timing, it is possible to model and predict software timing very accurately, as well as guaranteeing processor loading [3].

2.4 Using CAN with TT Architectures

As mentioned in 2.3, Time-Triggered architecture can only behave predictably if there are no interrupts active other than the timer that drives the scheduler. This means that, in order to handle CAN message arrival events, there is no choice but to poll the CAN controller.

When using a 'basic CAN' controller, messages arriving when the buffer is full will either be discarded or replace a message already in the buffer (depending on the configuration of the hardware). This means that messages will be lost, or missed, if the controller-polling period is longer than the worst-case timing between messages. This timing, also known as 'inter-frame space', varies depending on the configured baud rate of the CAN bus [4]. Timing examples for various baud rates are shown in Table 2-1:

TABLE 2-1: WORST CASE INTER-FRAME SPACING [4]

Baud Rate (kbit/s)	Inter-frame space, t_s (μs)
1000	47
500	94
250	188
125	376

Therefore, in order to receive every incoming message, a 'basic CAN' controller must be polled at least as often as:

$$t_{polling} = t_S \times N_{buffer} \qquad (2.1)$$

Where:

- $t_{polling}$ *is the polling period*
- t_s *is the worst-case Inter-frame space (from Table 2-1)*
- N_{buffer} *is the number of messages that the CAN controller's receive buffer can hold at any one time.*

For example, a CAN bus set to 500 kbit/s baud rate with a FIFO buffer capable of holding 16 messages needs to be polled at least every 1504 μs to guarantee reception of all messages in a worst-case scenario. This could pose problems in situations where there are more unique messages present on the CAN bus than buffer locations, particularly when the data is transmitted in 'bursts' by the other node(s) on the network.

'Full CAN' controllers, conversely, could be considered better suited to TT architectures as the message objects always hold the latest value for the configured CAN message. This means that even if it is not possible to poll the CAN controller as often as the messages arrive, the hardware will still handle the message arrival and no messages will be lost (although 'spikes' in message values could still be missed). Unfortunately, this method is only useful for devices interested in a relatively small subset of messages, where the number of messages to be accepted is less than or equal to the number of message objects. Further to this, some hardware, such as the eCAN controller described in 2.2.3 above, share the message objects between Tx and Rx functions. This means that if the target system is required to transmit messages as well as receive them, the acceptance scope of the controller is reduced further. Due to the maximum possible number of standard CAN identifiers being 2048 (000h to 7FFh), a device such as a data logger may need to accept a greater number of message identifiers than is allowed by a 'full CAN' controller.

2.5 Conclusions

There are several challenges involved in the development of a real-time embedded system that incorporates Controller Area Network in a Time Triggered software environment. These problems primarily surround the timing and predictability of the messages that are transmitted by other nodes on the network, particularly when the listening node is interested in a large subset of messages. The next chapter will review various related works that are themed around some of these challenges.

Chapter 3 Related Work

3.1 Introduction

The use of CAN as a communications medium in real-time embedded systems has been a topic of research for many years. This chapter will discuss several publications that address the subjects of CAN in embedded systems and message-related scheduling. Also discussed are links to cache behaviour. Contrasts with the proposed system will be posed, highlighting some of the novel aspects to this project.

3.2 CAN Message Scheduling in Embedded Systems

The operation of CAN bus networking and its integration into embedded systems is introduced by Farsi et al in [4]. The paper presents the distinction between 'full CAN' and 'basic CAN', and offers the merits of systems that allow the CAN controller to perform acceptance at the hardware level. Various hardware options are also discussed, as well as the timing constraints that application engineers are required to consider when integrating CAN hardware into an embedded system. This work provides a good grounding for CAN-based research, laying down the fundamental considerations required when building a distributed embedded system using the protocol.

In [12] and [13], Almeida et al discuss the various paradigms for message scheduling in real-time systems, highlighting a market gap for a communications system that supports both event and time-triggered traffic on a CAN bus. This problem is answered with a new application layer protocol, FTT-CAN, which provides flexible, dynamic scheduling of messages on a CAN bus. A centralised master node uses a "Planning Scheduler" [14] to broadcast a "Master Message" which indicates to the slave node(s) which message is required next. The planning scheduler uses knowledge of a "variable set" to determine the network activity for a set time window, or "plan". During execution of one plan, the scheduler builds the next plan, allowing reconfiguration of the system in time for the next time slot. [15] Expands on this protocol and discusses the technicalities of changing the variable set at run-time.

Tindell et al [1] discuss a single-processor message scheduling method and introduce the notion that message scheduling can be treated in the same manner as task scheduling in a real-time system. An off-line scheduling algorithm is proposed by Dobrin and Fohler in [16] and CAN message scheduling for time-critical control systems is discussed by Martí et al in [17].

Pop et al discuss, in [18], timing behaviour and schedulability when systems running in both Event Triggered and Time Triggered domains communicate over the same bus network.

All of these works surround the notion of controlling the transmission of messages over the bus network in order to achieve correct reception by the other nodes. In the automotive industry, however, where nodes are often designed independently from each other (e.g. by third parties), it is not always possible to control when the nodes transmit; it is only known roughly the order in which the nodes will transmit their messages. The research in this project is novel as it approaches the problems specifically surrounding the reception of CAN messages when there is no control over transmission and the transmitted message set unknown at compile-time.

3.3 Cache Behaviour

The incoming data stream on the CAN bus could also be viewed as a sequence of data storage requests, with the identifiers representing the memory locations. This could lead one to describe the proposed system as a cache.

Belady describes the factors that determine the optimal replacement policy for a cache or virtual memory system in [19]:

"To minimize the number of replacements, we attempt to first replace those blocks that have the lowest probability of being used again. Conversely, we try to retain those blocks that have a good likelihood of being used again in the near future."

In [20], Reineke et al discuss the use of caches in hard real-time systems. Various replacement policies are examined that are comparable to the behaviour of the proposed system. Weikle et al further describe the modelling of the sequence of requests as a data stream in [21]. The replacement policies described for caches, however, generally keep addresses alive if they achieve a hit. In contrast, when applying these techniques to CAN data stream, which is cyclic in nature, a hit for a specific identifier indicates that there will not be another like hit until the next message cycle.

3.4 Conclusions

Several paradigms for the scheduling of CAN messages are present in these disciplines, although the research focusses predominately on the synchronisation of message transmission, as opposed to message reception. The notion of treating the CAN bus acceptance filter as a cache has also been considered, with emphasis on the contrasts between memory address accesses in a traditional 'cache' and the cyclic message arrival on a CAN bus.

The next chapter will discuss how an existing embedded system, used to log data from an automotive CAN bus, puts some of these concepts into practice and shows areas for improvement that the proposed system will address.

Chapter 4 Existing Automotive Telemetry System

4.1 Introduction

This chapter will discuss an existing CAN data logging system used by an electric commercial vehicle company to gather and transmit vehicle diagnostics data via General Packet Radio Service (GPRS) to a data centre. Some fundamental flaws in the software architecture will be highlighted, that the proposed system will aim to improve upon. A method of determining the performance of the existing system will also be described, the results of which will be used as a benchmark to measure the success of the proposed system.

4.2 Software Architecture

The existing software architecture is a complex combination of interrupt-driven, event-triggered operations, and functions driven by multiple timer interrupts. This arrangement makes it very difficult to determine the state of the software at any given time. Although this project is primarily concerned with the CAN data logging aspect of the system, it is important to understand these complexities when forming comparisons with the new system.

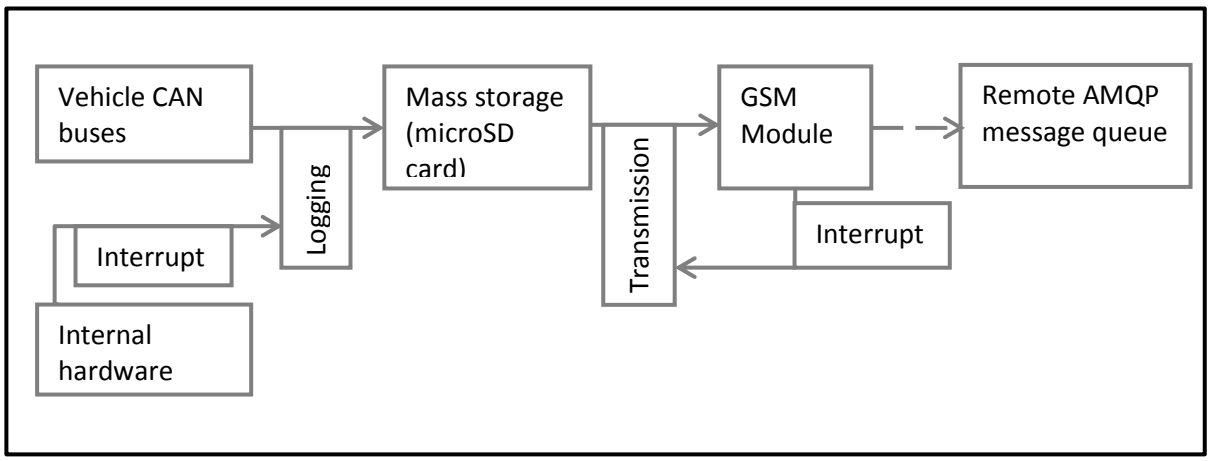


FIGURE 4-1: EXISTING REMOTE DEVICE SOFTWARE ARCHITECTURE

4.2.1 Data logging functions

These functions are performed using a combination of timer-driven interrupts and hardware interrupts:

- CAN, RS232, GPS and sensor data are collected from the vehicle and buffered internally.
- Buffered data is compressed and stored to RAM every 1 second.
- Every 30 seconds, the data is copied from RAM to the microSD card in data 'blocks' of around 4 – 10 kB.

4.2.2 Data transmission functions

These functions are performed using a UART interrupt from the GSM module, indicating that the module is connected to the network, and a looped polling function that waits for an unsent data block to become available:

- A connection is made to a remote AMQP message queue.
- Every 30 seconds, the most recent data block is read from the SD card. The large data blocks are split into several 1kB 'chunks' and sent to the message queue.
- If the device loses network signal, the software sits in one of several 'while' loops until a GSM interrupt event occurs. During this time, the data storing functions are still operating from timer interrupts, allowing the device to continue data collection during periods of low or no GPRS signal.

4.2.3 Event handling

There are several different types of event handling present in the software:

Event Interrupts

Several hardware events generate interrupts to the software. These include Analogue to Digital Conversion (ADC), four RS232 channels, a Global Positioning System (GPS) module and a GPRS module.

Time-released functions

These functions are driven by a timer interrupt operating at a 1 kHz frequency. However, since the architecture breaks the 'one interrupt per CPU' rule, these cannot be referred to as 'Time Triggered'. The time-released functions perform the CAN logging functionality, as well as sampling from the ADC, GPS and RS232 data, data compression and data storage to an on-board SD card.

'Super-loop' polling

Once a data connection is established with the server, a data transmission function stays in a 'super-loop', reading data from the SD card when it becomes available, and transmitting it to the server.

4.2.4 CAN Message Storage

The device stores the most recent data for each CAN message in a fixed memory location to allow compression logic to be performed. The device uses Microchip MCP2515 CAN controllers that only support 'basic CAN' [22], so the incoming data is filtered and stored by the software. This process is shown in Figure 4-2. Using this system, the software needs to process every CAN message that arrives on the CAN bus, whether logging of the message is required or not.

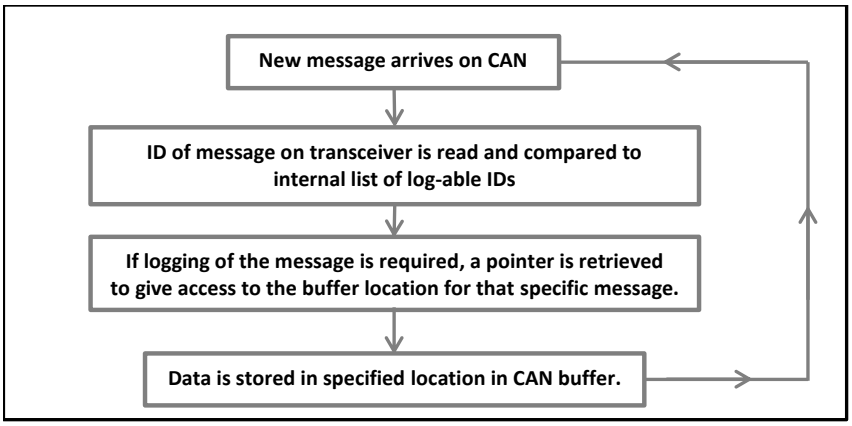


FIGURE 4-2: SOFTWARE FILTERING IN THE CAN DATA LOGGING PROCESS

Due to the large number of interrupts enabled in the system, it is not possible to guarantee the 1 kHz polling frequency demanded by the source code during normal operation. Moreover, due to the asynchronous nature of the connected CAN bus, it is not possible to predict or guarantee the hit rate of the CAN messages.

4.3 Instrumentation

In order to provide a benchmark to measure the performance of the proposed system, the existing embedded software was 'instrumented' by adding counters to the message storage arrays. The counter for a successful message 'hit' is incremented at the point in the software in which the CAN data is stored to RAM. The values of these counters were output using the device's existing RS485 debug port every 30 seconds. These values could be displayed using a terminal program, and copied to spreadsheet software for further analysis. This method was found to have the lowest impact on the behaviour of the system.

4.4 Conclusions

Several areas in this existing software are sub-optimal for a high-precision data logger. The design goes some way to utilise timer interrupts to control the sampling frequency of the CAN data; however, there are many other conflicting interrupt events in the design. The consequence of this is that the sampling rate cannot be guaranteed.

The next chapter will discuss a filtering algorithm proposed to allow such guarantees to be made. The results gathered by instrumenting this existing system will allow performance comparisons to be made in subsequent chapters.

Chapter 5 Dynamic CAN Filtering

5.1 Introduction

The proposed dynamic CAN filtering system aims to address a number of problems mentioned in the preceding chapters. An iterative development process was followed, first producing a Feasibility Simulation. Lessons learned from initial testing were fed into the parallel development of the Embedded Software, designed to run on a Texas Instruments C2000 family microcontroller, and a Remote Configuration and Analysis Tool (RCAT). Testing of these applications highlighted further areas for improvement that were fed back into the Feasibility Simulation.

This chapter describes the final product of this development process, with various development challenges discussed in further detail in Chapter 7 below.

5.2 Proposed Algorithm

The proposed system is a dynamic software acceptance filter designed around a Texas Instruments *TMS320F28335* processor, which incorporates a 'standard CAN' controller, referred to by the manufacturer as 'eCAN' [9]. The filter acts as a software extension to the mailbox system that is suited to periodic polling and takes advantage of the hardware rejection of unwanted identifiers. This provides a level of abstraction from the eCAN hardware, removing the limitation that the quantity of mailboxes would normally impose on the number of identifiers that can accepted by the device. This abstraction layer also allows the acceptance identifiers to be specified at run-time, allowing for a degree of remote configuration.

The system is provided a 'logging list' which comprises a description of the CAN IDs required to be logged by the device, along with their cycle times. CAN message acceptance filtering is handled in hardware by the CAN controller and a software layer built on the TI eCAN library allows tasks running from a Time-Triggered Hybrid (TTH) scheduler to modify the mailbox configurations. Using a pre-emptive task executed in the ISR, the software continually reads the incoming data from the CAN mailboxes and copies them into RAM.

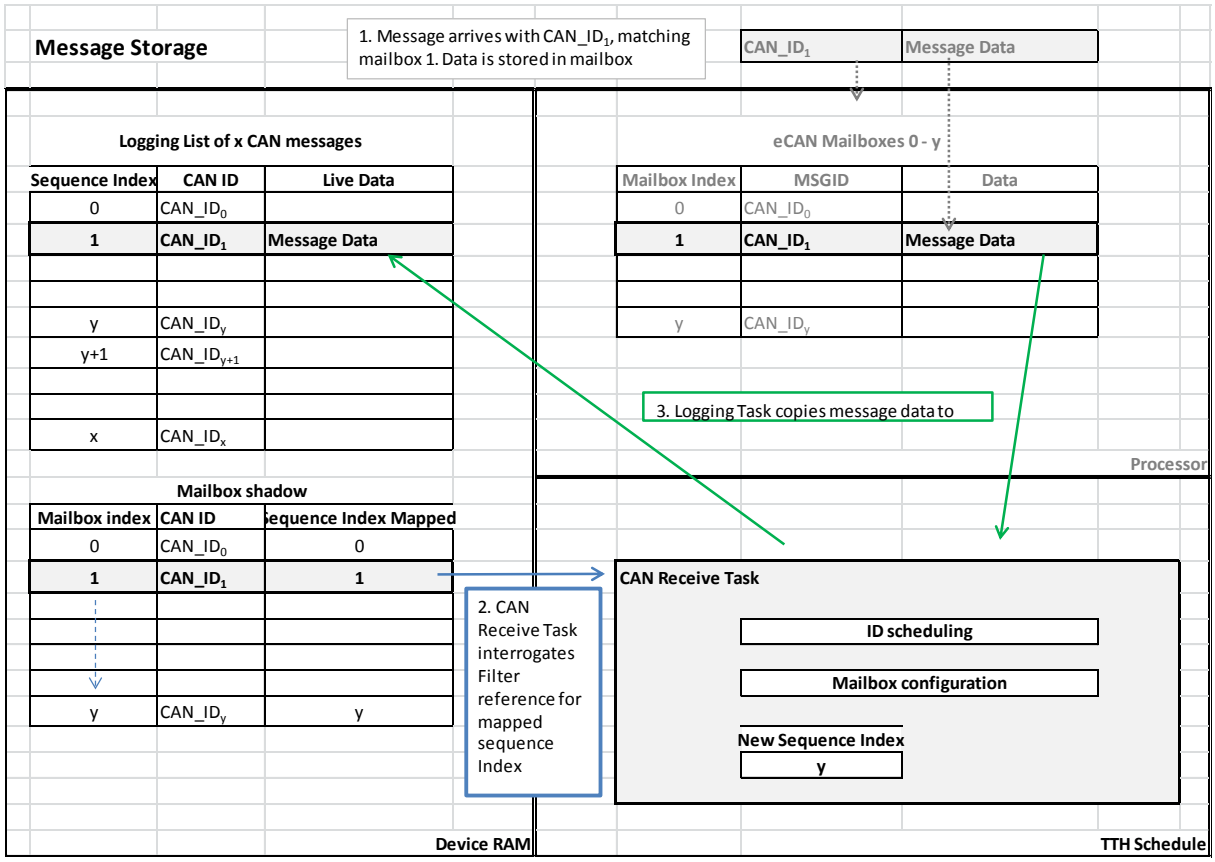


FIGURE 5-1: MESSAGE ARRIVAL IN DYNAMIC ACCEPTANCE FILTERING

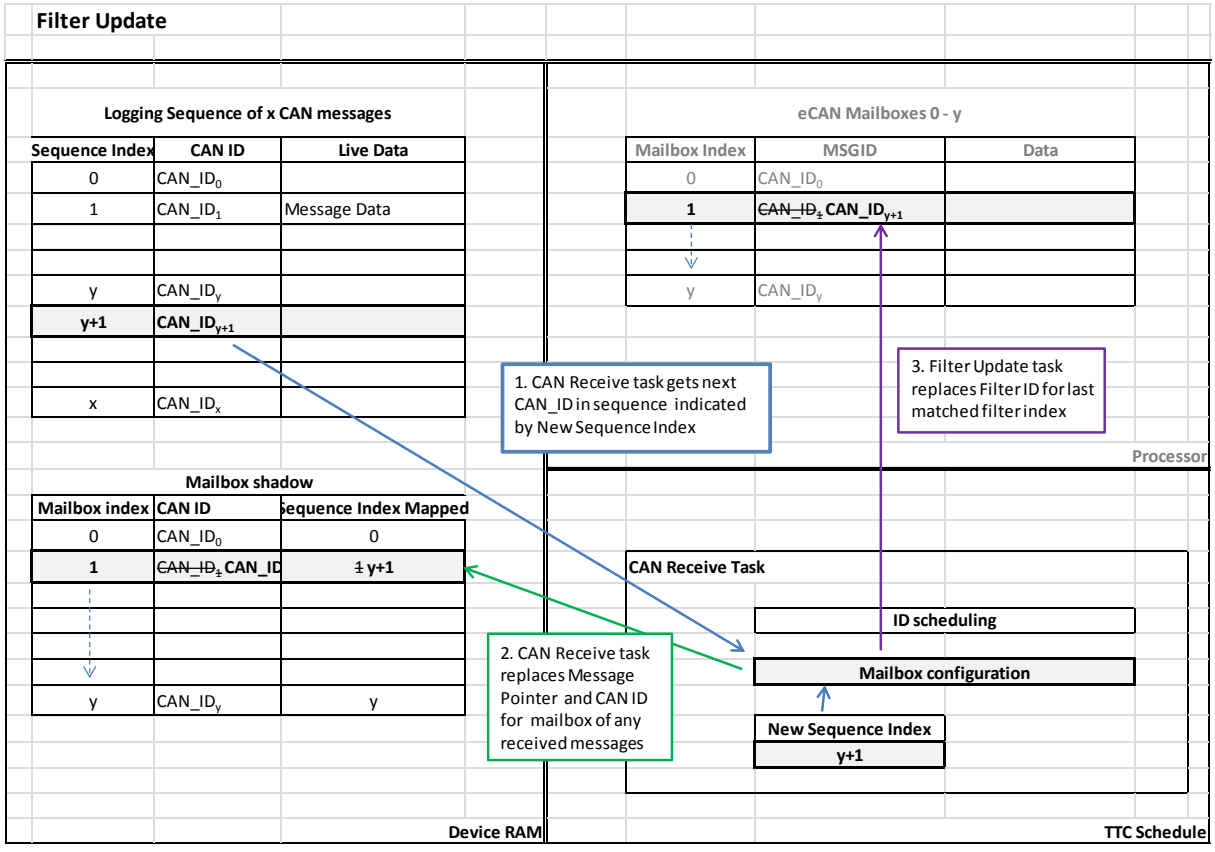


FIGURE 5-2: FILTER REPLACEMENT IN DYNAMIC ACCEPTANCE FILTERING

The source of the logging list is flexible and, if required, can be provided to the embedded system by a remote server, providing the ability to configure the filter remotely. The logging list structure is used to store the incoming message data, and a shadow table is used to keep track of the mapping between the logging list and the eCAN mailboxes.

As mentioned in Chapter 3, by viewing the filter as a cache, and the CAN data stream as a series of memory write requests, rules can be imposed to dictate how the acceptance filters are configured based on message arrival. It is ordinarily accepted that the optimum replacement policy is to 'evict' the block with the lowest probability of being accessed next [19]. When applied to CAN, where the data is predominately cyclic, we can predict that the ID with the lowest probability of arriving next is the one that has just arrived, and that the ID with the highest probability of arriving is the next one in the sequence. Therefore, as a message arrives in a mailbox, the acceptance filter for that mailbox is updated with the next identifier in the 'logging list'.

This method exploits the assumption that the order and timing in which individual messages are published over the CAN bus is relatively predictable. Further to this, there is no CPU processing time wasted on discarding unwanted messages, as all messages 'visible' to software have already been filtered by the hardware.

5.3 Feasibility Simulation

A simulation application was written in 'C' that reads through an ASCII text-based CAN message log, or 'trace', and uses the trace timestamps to determine whether a message would be accepted by the proposed filtering algorithm. The advantage of this approach is the ability to read the CAN trace faster than real-time, allowing for rapid development of the algorithm, and for automatic cycling of the simulation with varying control parameters for quicker analysis.

A time-triggered, periodic logging task is simulated that, in the target embedded system, will read all logged messages from the CAN mailboxes, and update the acceptance filters. The following parameters and variables are used to control the simulated behaviour:

- `LOGGING_TASK_PERIOD_us` - Controls the period of the simulated TT task.
- `loggingSequence[]` – Holds the sequence of CAN IDs to filter from the given trace.
- `sequencePointer` – used to keep track of the location in `loggingSequence[]`.
- `acceptanceFilter[filterSize]` – Represents the acceptance filter of size `filterSize`.

At initialisation, this array is filled, index-for-index, with the CAN identifier values from the top of `loggingSequence[]`.

5.3.1 Message acceptance

The application simulates a hardware acceptance filter by examining the CAN identifier in each line of the CAN trace. The identifier is compared to those in the logging sequence to determine whether logging is required. Identifiers not found in the logging sequence are ignored and the simulation moves on to the next line of the CAN trace.

If the identifier is contained in the logging sequence, the acceptance filter array is interrogated to see if the identifier is present. If the identifier is present in the array, the CAN message would be seen by a hardware acceptance filter. A 'hit' is recorded for the identifier by incrementing a counter relating to the captured identifier. A further counter, `IDLogCount`, is incremented to keep track of the overall hit rate. The identifier is marked as 'logged' in the acceptance filter, but the acceptance filter is not yet updated. This simulates the 'blocking' effect of identifiers between executions of the simulated Time Triggered task. If the identifier is in the logging sequence, but not present in the acceptance filter, the identifier has been 'missed' and `IDMissedCount` is incremented.

5.3.2 Simulated Time Triggered task

In order to achieve a realistic simulation of the system, it is important that the application behaviour is as close as possible to that of a periodic task running from a TT scheduler. To accomplish this, the simulation exploits the timestamps recorded alongside each message in the CAN trace. This provides an indication, correct to 1 μs, of the time that elapsed between the recorded CAN messages, allowing the simulation to determine when the logging task would run.

```
;   Message Number
;   |         Time Offset (ms)
;   |         |       Bus
;   |         |       |    Type
;   |         |       |    |       ID (hex)
;   |         |       |    |       |    Reserved
;   |         |       |    |       |    |   Data Length Code
;   |         |       |    |       |    |   |    Data Bytes (hex) ...
;---+-- ------+------ +- --+-- ----+--- +- -+-- -+ -- -- -- -- -- -- --
     1)        11.449 1  Rx        0050 -  0
     2)        12.058 1  Rx        0185 -  8    00 00 00 00 00 00 00 00
     3)        12.684 1  Rx        0385 -  8    00 00 00 00 00 00 09 02
     4)        12.958 1  Rx        018B -  8    00 00 00 00 00 00 00 00
     5)        13.298 1  Rx        0188 -  8    28 E2 02 00 00 00 00 00
```

FIGURE 5-3: A PCAN TRACE FILE EXAMPLE

When the elapsed time, indicated by the CAN trace 'time offset', exceeds `LOGGING_TASK_PERIOD_us`, the acceptance filter is scanned and each identifier that has been marked as 'logged' (see 5.3.1) is replaced by the next identifier in the logging sequence. This approach closely simulates the periodic execution of the time-triggered task, and provides an understanding of the limitations caused by this behaviour.

5.4 Embedded Software Implementation

Development of the embedded software implementation involved porting the code from the feasibility simulation to tasks running in a TTH scheduler. The software includes two tasks running from the ISR:

- `handleCAN_update()` - A CAN mailbox handler developed specifically for periodic polling of the mailboxes, which are accessed through the Texas Instruments eCAN library [9]. This handler runs every other tick and updates the mailbox state, indicating when mailboxes have data pending.
- `receiveCAN_update()` - The filter handler is executed in alternate ticks to the mailbox handler and reads from the mailboxes and updates the filter using the algorithms described above.
- A further task, `controlSCI_update()` runs from the scheduler, and communicates with the Remote Configuration and Analysis Tool (RCAT) over an RS232 connection (see 5.5).

The high-level operation of the embedded system can be modelled as a simple state machine, as described in Figure 5-4 and Table 5-1. There are various key differences and additions to the logic in the embedded software compared to the Feasibility Simulation. Firstly, the sequence is built from the logging list received from the external Remote Configuration application (see 5.5).

In order to prevent lower frequency messages from 'blocking' higher frequency messages, the filter is split into 'segments'. Each segment is dedicated to a specific set of message grouped by expected cycle time. The number of mailboxes per segment is a ratio controlled by a configurable constant in the source code. This segmentation is equivalent to limiting the simulation to filter messages for like cycle times (see 7.2 below).

In addition, the embedded system is unable to count misses, because it is only aware of the CAN messages that it logs successfully. In order to evaluate the performance, the results must be used in conjunction with those from the simulation tool. This does not present a problem as long as the simulation is run on the same trace file that is replayed through the embedded system.

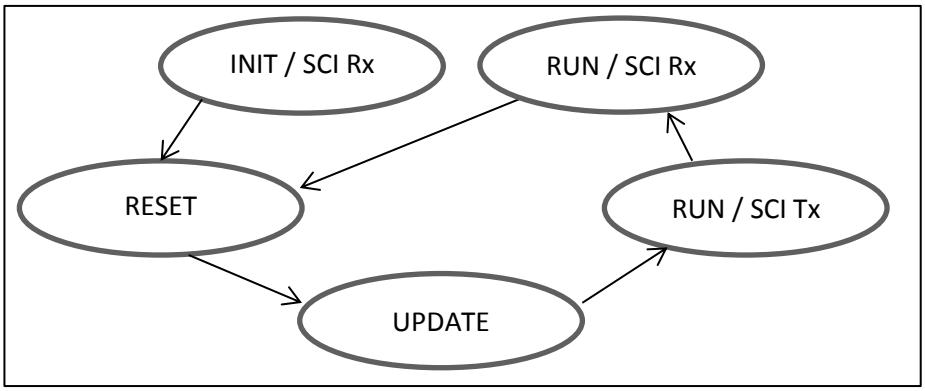


FIGURE 5-4: SIMPLIFIED EMBEDDED SOFTWARE STATE MACHINE

TABLE 5-1: STATE DESCRIPTIONS AND TRANSITION RULES

State	Description
INIT / SCI Rx	• Embedded system powered on. No CAN messages can be received until filter is configured. • SCI reads incoming configuration data on successful handshake. • When all configuration data is received, sequencing and segmentation logic is performed before transitioning to RESET.
RESET	• All mailboxes are disabled and counters reset before transitioning to UPDATE.
UPDATE	• Mailboxes are initialised (one per task execution) with the identifiers at the top of the sequence. • When all required mailboxes are configured, the system transitions to RUN / Tx
RUN / SCI Tx	• System polls mailboxes for incoming data and performs identifier replacement according to the rules set out in 5.3.1. • Hit counts and filter mapping information are transmitted to the RCAT. • System transitions to RUN / SCI Rx on reception of reset request character ('?') from the RCAT.
RUN / SCI Rx	• System continues to store CAN messages and perform filter replacement as in RUN / SCI Tx. • SCI reads incoming configuration data. • When all configuration data is received, sequencing and segmentation logic is performed before transitioning to RESET, initiating a re-configuration of the filter.

5.5 Remote Configuration and Analysis Tool

In order to satisfy the 'remote configuration' aspect of the project, an application was written in the 'Processing' programming language [23] to communicate with the embedded system from a desktop computer.

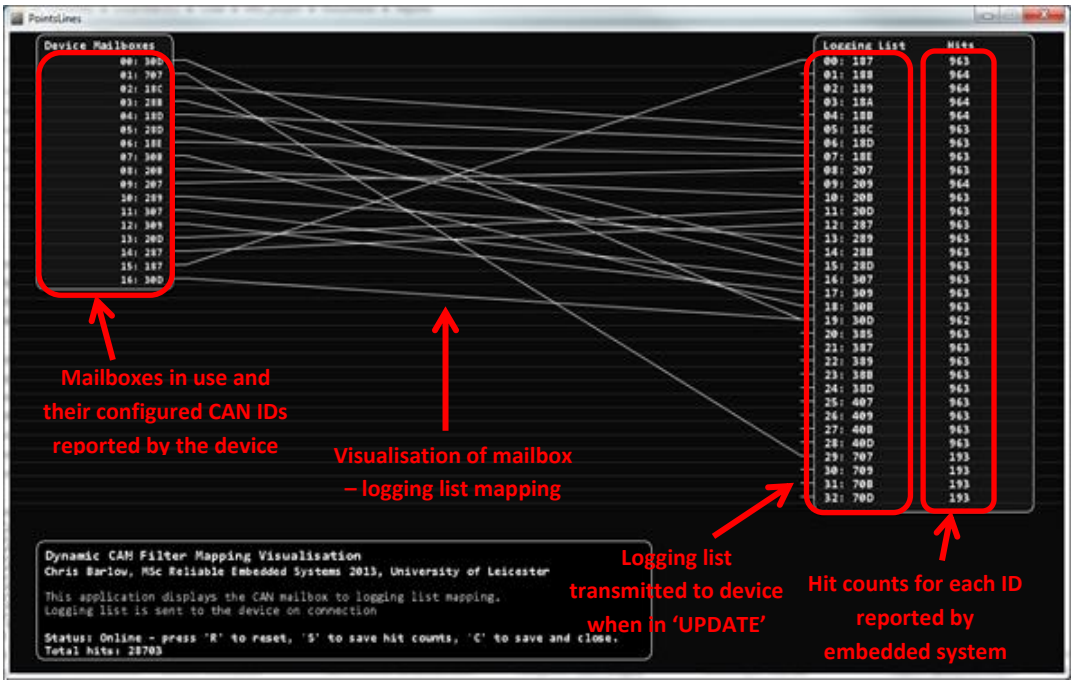


FIGURE 5-5: VISUAL FEEDBACK AND HIT RATE ANALYSIS FROM RCAT

The function of this application varies depending on the current state of the system:

- To handshake with and remotely switch the mode of the embedded system between 'SCI Tx' and 'SCI Rx'.
- To transmit the logging list configuration to the embedded system over a serial (RS232) connection when the embedded system is in 'INIT / SCI Rx' state.
- To display mapping and hit count feedback from the filter mechanism, which provides a visualisation of the behaviour of the algorithm.
- To save hit counts to a comma-delimited text file (*.csv) for further analysis.

5.6 Conclusions

The use of an iterative development process provided the opportunity to rapidly prototype the filtering algorithm, exploiting the timestamps logged in the CAN traces to perform tests faster than real-time. Once the feasibility of the algorithm was determined, the embedded system was developed. The result is a simulation that can quickly evaluate the effectiveness of the system on a given CAN trace, and an embedded system that can be remotely configured and monitored in real-time using a bespoke PC application. The next chapter will analyse the results found through testing these final systems, making comparisons to the existing system described in Chapter 4.

Chapter 6 Analysis

6.1 Introduction

In order to analyse the behaviour and performance of the proposed algorithm there were a number of tests to be conducted. This chapter describes tests carried out on the finished system, i.e. the Feasibility Simulation and Embedded Software Implementation discussed in Chapter 5, with further discussion on results found during the development process discussed in Chapter 7. This testing set out to answer the following questions:

- Is there a relationship between the acceptance filter size and the logging list size? If so, what is the optimum size for the acceptance filter for a given logging list?
- How closely does the performance of the feasibility simulation match that of the embedded system?
- How does the hit rate per identifier compare to that of the existing system?
- What is the timing behaviour of the embedded system?

6.2 Method

6.2.1 CAN trace samples

Two CAN trace samples were used during testing. These are outlined in Table 6-1 below:

TABLE 6-1: SUMMARY OF SAMPLE CAN TRACES

Trace	Total Unique IDs	Cycle Time (ms)	Number of Unique IDs
A	33	20	29
		100	4
B	82	100	22
		1000	60

These traces provided a good variation of different bus conditions. Trace A has a greater number of high frequency than low frequency messages. Trace B gives a good indication of how the algorithm behaves when this ratio is reversed. Trace B is also significant, because the number of unique IDs is more than twice the number of mailboxes available on the target microcontroller.

6.2.2 Simulation Testing

Trace A was used initially to determine the relationship between the filter size and logging list size. The logging list was set up to include all 33 of the unique IDs present in the trace and the simulation was executed in 'full sweep' mode, which repeats the analysis for different filter sizes, varying from `filterSize = 1` to `filterSize =(loggingListSize-1)`.

The simulation outputs, to a comma-delimited log file, the total hit and miss counts for each value of `filterSize`. The size of the logging list was varied by removing identifiers, and the simulation repeated.

The optimum filter size found from the above tests was used for an in-depth test to use as a performance comparison. For this test the simulation was run again, but this time the hit and miss rate per identifier for the given filter size were recorded. This in-depth test was executed for both Trace A and Trace B.

6.2.3 Hardware Testing

The hardware test involved connecting a PC, a development board running the target processor, a PCAN USB to CAN bus adapter and the existing telemetry device running the instrumented firmware (see 4.3). The connections between these components are shown in Figure 6-1:

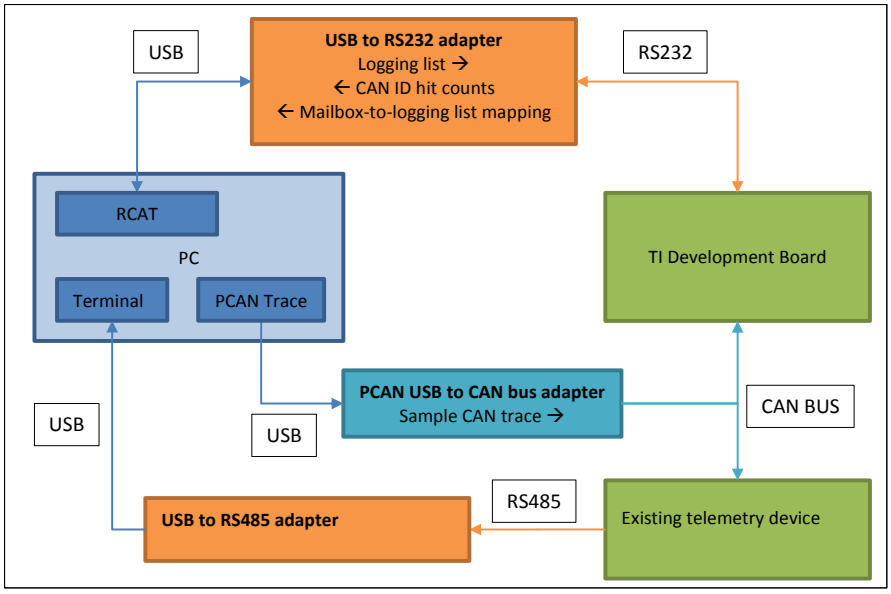


FIGURE 6-1: HARDWARE TEST SETUP

Traces A and B were transmitted over the CAN bus in real-time using a Peak USB to CAN adapter and the 'PCAN Trace' software. This allowed the target development board to see the same CAN messages in the same conditions as the existing telemetry device.

Identifier hit counts were recorded by the RCAT and those for the existing device were recorded using a serial terminal program, 'Tera Term'. The test was repeated in order to gauge the consistency and predictability of the hit counts.

The execution times of the filtering tasks were measured by instrumenting the scheduler around the tasks running in the ISR. A timer was started just before the tasks were executed, and read just after the task finished. The timer values were read by 'watching' the associated variables using the debug feature of the Texas Instruments Code Composer Studio IDE.

6.3 Results

6.3.1 Performance – filter size relationship

Figure 6-2 below shows how the total filter hit rate for different list sizes varies with the size of the filter:

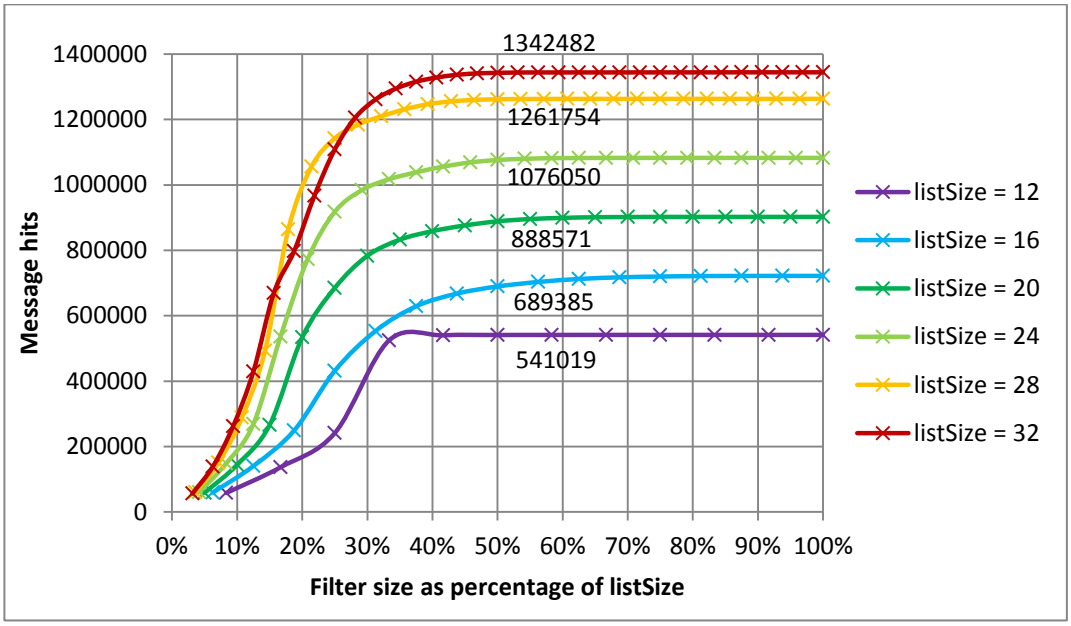


FIGURE 6-2: MESSAGE HITS VS FILTER SIZE FOR VARYING LIST SIZES

Displaying the filter size as a percentage of the list size shows that the number of message hits levels off when the filter size is around 50% of the total size of the logging list. This shows a predictable relationship between the size of the filter relative to the size of the logging list.

6.3.2 Optimum filter size

Figure 6-3 shows the typical hit rates for a 32-identifier logging list with a filter of varying sizes plotted against the maximum and minimum hit rates achieved for the existing system. It can be seen that the simulated system begins to better the existing system when the filter size = 10. When the filter reaches 16 (50% of the logging list size) the hit rate of the proposed system is significantly improved over the existing.

These results indicate that using a filter size equal to 50% of the logging list size will offer an acceptable trade-off between hit rate and system resources. A system based around this algorithm would theoretically be able to log twice as many identifiers as the number of mailboxes available to the CAN controller.

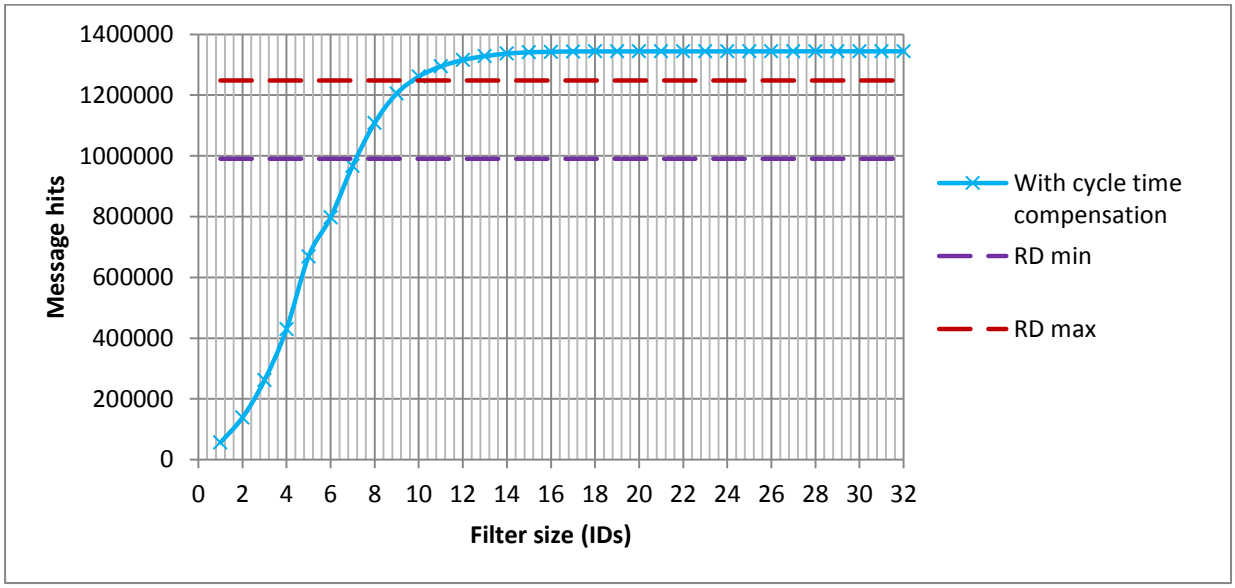


FIGURE 6-3: HIT RATE VS FILTER SIZE FOR 32-ID LOGGING LIST

6.3.3 Comparing the Simulation and Hardware Implementation

Table 6-3 show a comparison between the hit rates for the two sample CAN traces predicted by the Simulation tool and the Target Hardware. The simulation predicted the performance to within 3.6% of the embedded hardware, as indicated by the 'Hit Rate Delta':

TABLE 6-2: COMPARISSON BETWEEN SIMULATION AND HARDWARE – TRACE A

	Total In Trace	Simulation				Hardware Hits		Hit Rate Delta
		Hits		Misses				
Seg. Cycle Time		Count	Percent	Count	Percent	Count	Percent	
20 ms	1308177	1307812	99.972%	365	0.028%	1301518	99.491%	0.481%
100 ms	36072	35936	99.623%	136	0.377%	34665	96.099%	3.524%
Both segments	1344249	1343748	99.963%	220	0.016%	1336183	99.400%	0.563%

TABLE 6-3: COMPARISSON BETWEEN SIMULATION AND HARDWARE – TRACE B

	Total In Trace	Simulation				Hardware Hits		Hit Rate Delta
		Hits		Misses				
Seg. Cycle Time		Count	Percent	Count	Percent	Count	Percent	
100 ms	140345	137050	97.652%	3295	2.348%	137049	97.652%	0.001%
1000 ms	24900	24900	100.000%	0	0.000%	24899	99.996%	0.004%
Both segments	165245	161950	98.006%	3295	1.994%	161948	98.005%	0.001%

It can be seen that there is some variation in accuracy between the two traces. This is possibly caused by different timing conditions in the CAN trace, however the exact cause is unclear and investigation is out of the scope of this project. These results do indicate, however, that the simulation would be a useful pre-implementation tool to allow a target CAN bus to be analysed for suitability for the application of this algorithm. It would also allow predictions to be made, to a reasonable level of confidence, about the expected hit rate of a given system.

6.3.4 Comparison with Existing Device

The hit rates per CAN identifier for both the existing and new hardware systems can be seen in Figure 6-4 and Figure 6-5:

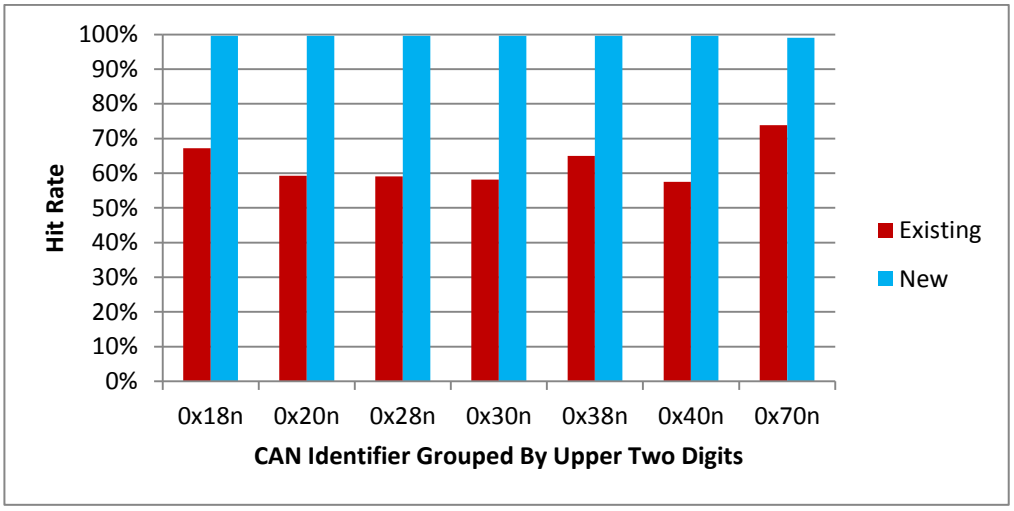


FIGURE 6-4: HIT RATES FOR IDENTIFIERS GROUPED BY UPPER TWO DIGITS – TRACE A

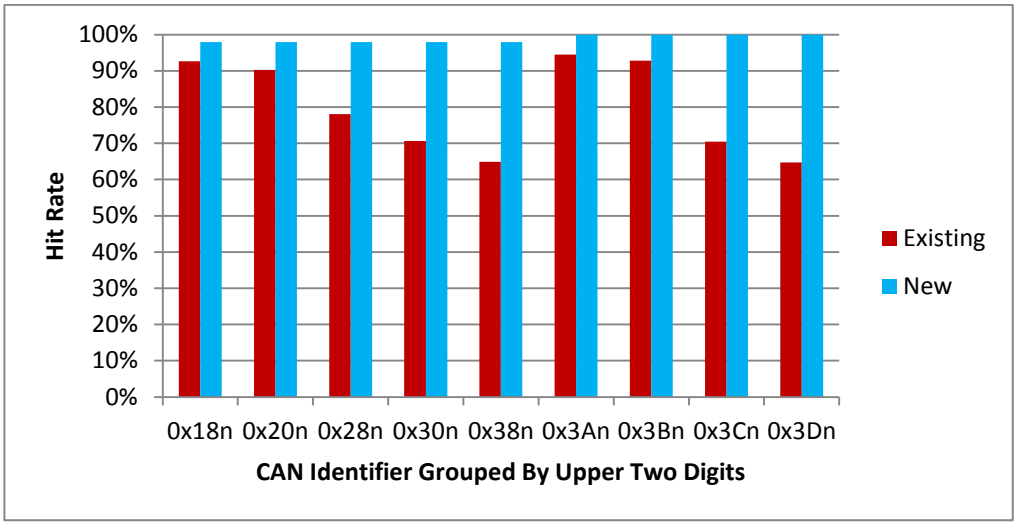


FIGURE 6-5: HIT RATES FOR IDENTIFIERS GROUPED BY UPPER TWO DIGITS – TRACE B

Here it can be seen that the hit rates for the new system are not only higher than the existing device, but also more consistent across the whole range of identifiers. This indicates a marked improvement in logging performance over the existing device.

6.3.5 Timing behaviour

The timing results for the time-triggered tasks are shown in Figure 6-6. `ISRbcet` contains the Best Case Execution Time (BCET) of the tasks; `ISRwcet` contains the Worst Case Execution Time (WCET). Index [0] refers to `handleCAN_update()` and index [1] refers to `receiveCAN_update()`. Timing is measured in CPU cycles.

It can be seen that there is a large amount of variation (jitter) in the timing to task [1]. This is due to the task being idle when there are no CAN messages pending, and requiring extra work during data burst periods. Task [0] is relatively stable, with only 5408 cycles of jitter. This results in a pattern of CPU activity as shown in Figure 6-7. Because of the alternating execution of these two tasks, there are periods of stability every two ticks.

'scheduler.c'::ISRbcet	unsigned long[2]	0x0000CAEA@Data
(x)= [0]	unsigned long	23474
(x)= [1]	unsigned long	1075
'scheduler.c'::ISRwcet	unsigned long[2]	0x0000CAE2@Data
(x)= [0]	unsigned long	28882
(x)= [1]	unsigned long	139362

FIGURE 6-6: DEBUG OUTPUT FROM CODE COMPOSER STUDIO SHOWING BCET AND WCET FOR ISR TASKS

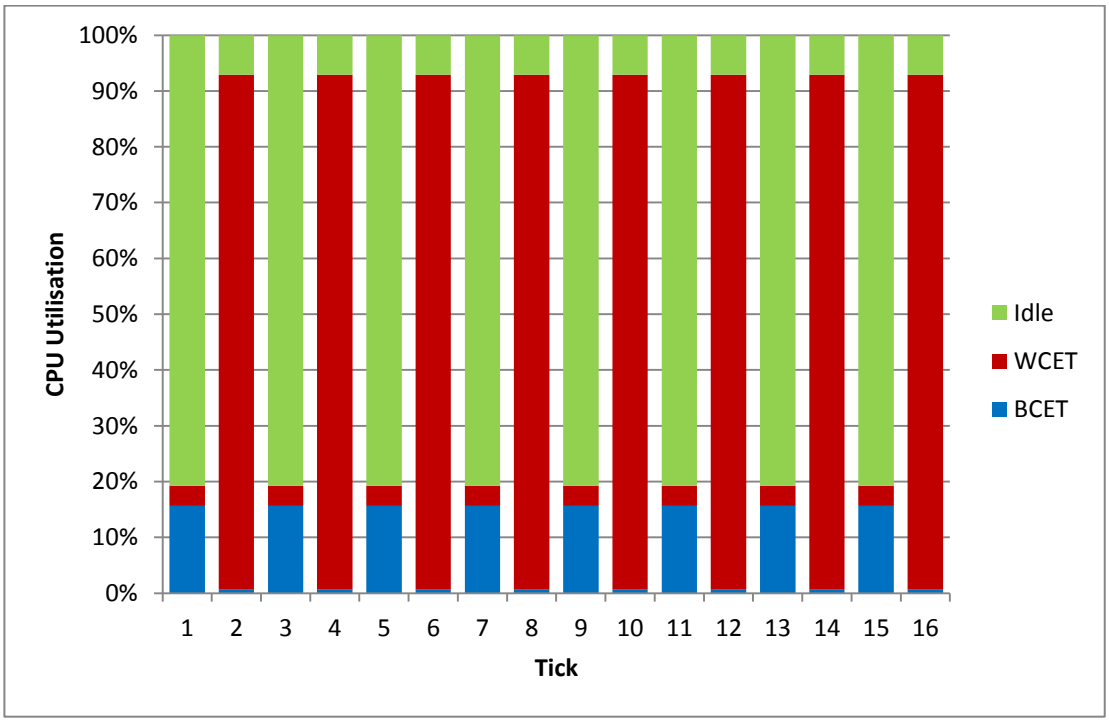


FIGURE 6-7: ISR CPU UTILISATION

This means that there is scope for the remainder of the processing time (minus scheduler overhead) to be used for a target application as long as the tasks were carefully scheduled so that any time-critical tasks execute in the 'stable' ticks and have a period no shorter than 2 ticks. The example in this project used this time to communicate with the remote configuration tool, however this shows there is scope to build the filter mechanism into any application where there is a requirement to accept more CAN identifiers than the 'standard CAN' hardware allows.

6.4 Conclusions

It can be seen from these results that there is a clear relationship between size of the filter relative to the number of unique messages in the logging list. It can be seen that, from these test traces, the filter will catch the majority of messages when the filter size is over 50% of the list size. This can be used to assign dynamically an 'optimum' number of mailboxes depending on the size of the logging list supplied to the system. Further testing with a greater variation of CAN traces with different timing conditions is required, however, to prove this holds true for all situations. Further testing is also required to investigate the causes of some inaccuracy found between the simulation and embedded system. The comparison with the existing system indicates that the dynamic filtering algorithm offers significant improvement over the predictability and consistency of the CAN data sampling.

In the chapter to follow, some behavioural factors will be discussed that were discovered during the development of the system.

Chapter 7 Discussion

7.1 Introduction

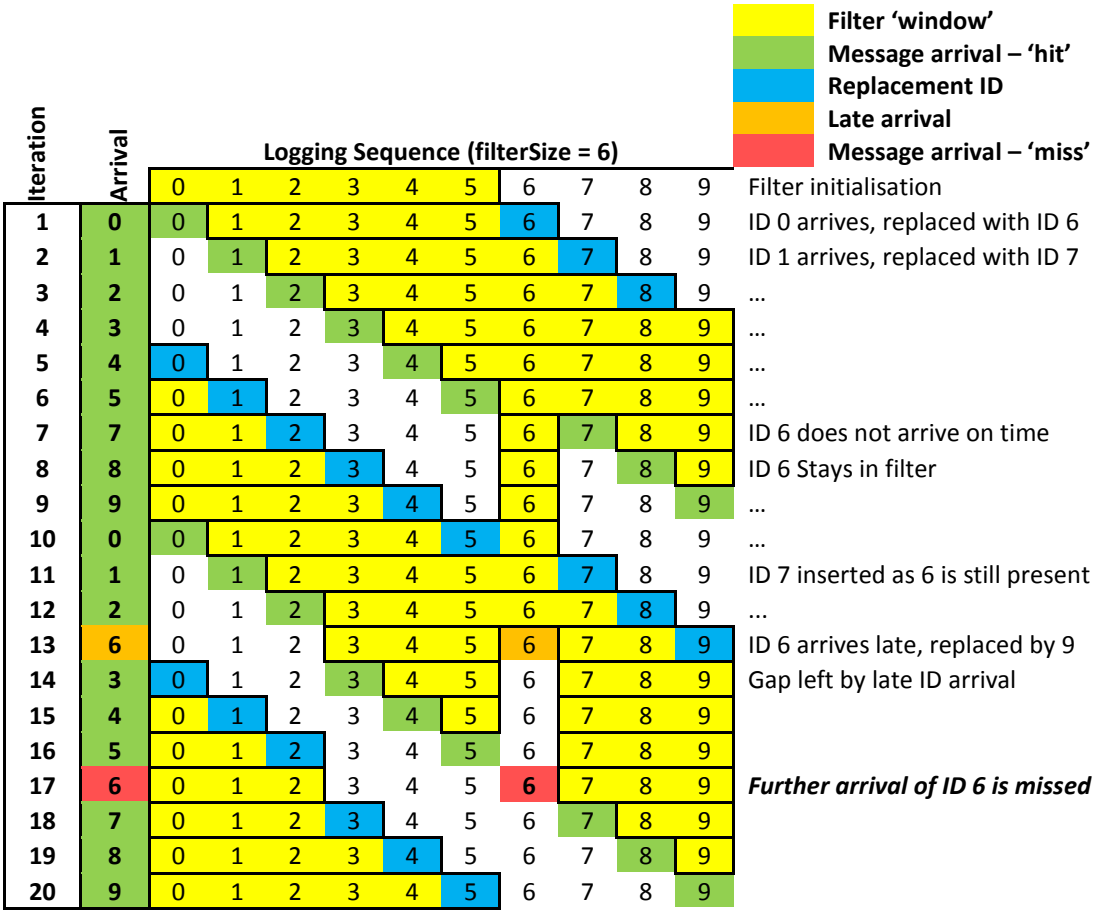
The previous chapters have addressed the testing of the final implementations of the system. This chapter will discuss some of the factors influencing the behaviour of the system that were discovered during the development process.

7.2 Duplication in Filter

One discovery during the development process was the concept of 'controlled duplication' in the filter. It was assumed at the beginning of the project that allowing more than one occurrence of an identifier into the filter would block other identifiers from being recorded by the system. This is true as the effective size of the filter becomes smaller as the number of duplicates increases; however, it was found that allowing a controlled number of duplicates into the filter was beneficial to the logging consistency of the system across identifiers. This is because the duplicates allow the filter to catch up when CAN messages arrive out of the expected sequence.

This behaviour is demonstrated in Figures Figure 7-1 and Figure 7-2. The filter is shown as a 'window' moving across the logging sequence. At iteration 13, with no duplication allowed in the filter, the late arrival of ID 6 causes a gap to form in the filter 'window'. This causes the next, on time, arrival of ID 6 to be missed in iteration 17. This is very damaging to the hit rates for identifiers that repeatedly arrive out of sequence.

With duplication allowed, the sequencing logic configures an additional mailbox for ID 6 between iterations 11 and 12. This means that when ID 6 arrives late in iteration 13, there is still a mailbox available to accept the next arrival. In this scenario, both instances of the identifier are logged.



Iteration	Arrival	Logging Sequence (filterSize = 6)										
		0	1	2	3	4	5	6	7	8	9	Filter initialisation
1	0	0	1	2	3	4	5	6	7	8	9	ID 0 arrives, replaced with ID 6
2	1	0	1	2	3	4	5	6	7	8	9	ID 1 arrives, replaced with ID 7
3	2	0	1	2	3	4	5	6	7	8	9	…
4	3	0	1	2	3	4	5	6	7	8	9	…
5	4	0	1	2	3	4	5	6	7	8	9	…
6	5	0	1	2	3	4	5	6	7	8	9	…
7	7	0	1	2	3	4	5	6	7	8	9	ID 6 does not arrive on time
8	8	0	1	2	3	4	5	6	7	8	9	ID 6 Stays in filter
9	9	0	1	2	3	4	5	6	7	8	9	…
10	0	0	1	2	3	4	5	6	7	8	9	…
11	1	0	1	2	3	4	5	6	7	8	9	ID 7 inserted as 6 is still present
12	2	0	1	2	3	4	5	6	7	8	9	…
13	6	0	1	2	3	4	5	6	7	8	9	ID 6 arrives late, replaced by 9
14	3	0	1	2	3	4	5	6	7	8	9	Gap left by late ID arrival
15	4	0	1	2	3	4	5	6	7	8	9	
16	5	0	1	2	3	4	5	6	7	8	9	
17	6	0	1	2	3	4	5	6	7	8	9	***Further arrival of ID 6 is missed***
18	7	0	1	2	3	4	5	6	7	8	9	
19	8	0	1	2	3	4	5	6	7	8	9	
20	9	0	1	2	3	4	5	6	7	8	9	

FIGURE 7-1: SIMPLIFIED VISUALISATION OF OUT-OF-SEQUENCE CAN MESSAGE - NO DUPLICATION

		0	1	2	3	4	5	6	7	8	9	Filter initialisation
1	0	0	1	2	3	4	5	6	7	8	9	ID 0 arrives, replaced with ID 6
2	1	0	1	2	3	4	5	6	7	8	9	ID 1 arrives, replaced with ID 7
3	2	0	1	2	3	4	5	6	7	8	9	…
4	3	0	1	2	3	4	5	6	7	8	9	…
5	4	0	1	2	3	4	5	6	7	8	9	…
6	5	0	1	2	3	4	5	6	7	8	9	…
7	7	0	1	2	3	4	5	6	7	8	9	ID 6 does not arrive on time
8	8	0	1	2	3	4	5	6	7	8	9	ID 6 Stays in filter
9	9	0	1	2	3	4	5	6	7	8	9	…
10	0	0	1	2	3	4	5	6	7	8	9	…
11	1	0	1	2	3	4	5	**6(x2)**	7	8	9	ID 1 arrives, duplicate added for ID 6
12	2	0	1	2	3	4	5	**6(x2)**	7	8	9	Window effectively reduced to 5 IDs
13	6	0	1	2	3	4	5	6	7	8	9	ID 6 arrives late, duplicate replaced by 8
14	3	0	1	2	3	4	5	6	7	8	9	Effective window size returns to 6 IDs
15	4	0	1	2	3	4	5	6	7	8	9	…
16	5	0	1	2	3	4	5	6	7	8	9	…
17	6	0	1	2	3	4	5	6	7	8	9	***Further arrival of ID 6 is caught***
18	7	0	1	2	3	4	5	6	7	8	9	…
19	8	0	1	2	3	4	5	6	7	8	9	…
20	9	0	1	2	3	4	5	6	7	8	9	…

FIGURE 7-2: SIMPLIFIED VISUALISATION OF OUT-OF-SEQUENCE CAN MESSAGE - CONTROLLED DUPLICATION

The effects of this algorithm with a real CAN trace are shown in Table 7-1. It should be noted that the hit rates for some identifiers have reduced. It is assumed that this is due to the effective filter window decreasing during periods of duplication. However, since the emphasis in this project is in making the hit rate more predictable, the slight reduction in hit rate is acceptable in exchange for more consistency across the identifier range.

TABLE 7-1: HIT RATES PER ID FOR DIFFERENT LEVELS OF DUPLICATION

CAN Message		Hit Rate		
Cycle time	**CAN ID**	**No duplicates**	**1 duplicate**	**2 duplicates**
~20 ms	0x187	99.856%	99.550%	99.585%
	0x188	96.469%	99.541%	99.577%
	0x189	99.898%	99.543%	99.579%
	0x18A	96.404%	99.541%	99.577%
	0x18B	99.909%	99.541%	99.577%
	0x18C	96.402%	99.541%	99.577%
	0x18D	99.936%	99.546%	99.581%
	0x18E	96.389%	99.541%	99.577%
	0x207	99.911%	99.550%	99.585%
	0x209	99.942%	99.543%	99.579%
	0x20B	99.936%	99.541%	99.577%
	0x20D	99.953%	99.546%	99.581%
	0x287	99.925%	99.550%	99.585%
	0x289	99.967%	99.543%	99.579%
	0x28B	99.973%	99.541%	99.577%
	0x28D	99.829%	99.543%	99.581%
	0x307	99.967%	99.550%	99.585%
	0x309	99.984%	99.543%	99.579%
	0x30B	99.971%	99.541%	99.577%
	0x30D	99.960%	99.546%	99.581%
	0x385	96.353%	99.539%	99.574%
	0x387	99.996%	99.550%	99.585%
	0x389	99.982%	99.543%	99.579%
	0x38B	99.996%	99.541%	99.577%
	0x38D	99.998%	99.546%	99.581%
	0x407	99.987%	99.550%	99.585%
	0x409	99.980%	99.543%	99.579%
	0x40B	99.996%	99.541%	99.577%
	0x40D	99.996%	99.546%	99.581%
Standard Deviation		**1.3648%**	**0.0035%**	**0.0035%**
~100 ms	0x707	99.257%	99.046%	99.157%
	0x709	99.346%	99.046%	99.124%
	0x70B	99.479%	99.057%	99.124%
	0x70D	99.501%	99.046%	99.135%
Standard Deviation		**0.1151%**	**0.0055%**	**0.0157%**

7.3 Cycle time compensation

It became apparent during early iterations of the simulation that including identifiers of different cycle times in the same logging list would be problematic. This is because the lower cycle identifiers have the effect of 'blocking' the higher cycle messages. Following the basic sequential replace strategy outlined below, the acceptance filter would quickly fill up with identifiers for messages of longer cycle times. In order to compensate for this, two different strategies were evaluated.

7.3.1 Weighted Sequencing

One compensation approach was to use a sequence scheduling strategy that weighted the filter insertion rate of an identifier based on its cycle time. Each identifier is given a counter, reloaded with a value determined by:

$$counter_{reload} = \left\lceil \frac{cycleTime_n}{cycleTime_{min}} \right\rceil \qquad (7.1)$$

Each time an identifier is inspected for insertion into the acceptance filter, the counter is decremented. If the counter reaches zero, the identifier is used in the filter. If the counter is still greater than zero, the next identifier in the sequence is inspected. This approach provided promising results for Trace A; where there are fewer long-cycle than short-cycle identifiers, as the hit rates became more consistent between identifiers:

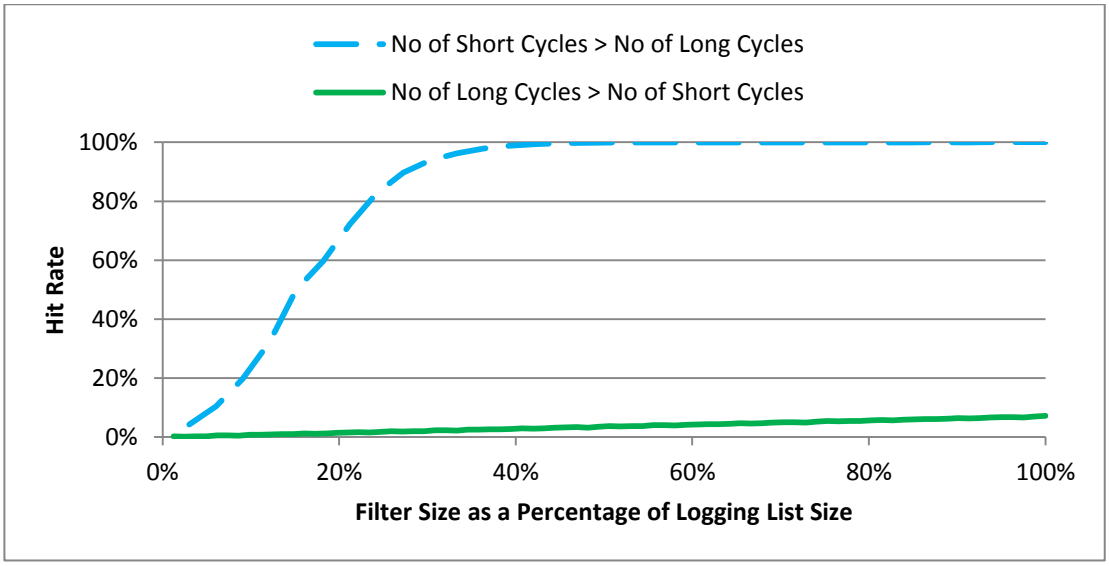


FIGURE 7-3: EFFECT OF CYCLE TIME BALANCE ON HIT RATE FOR SINGLE-SEGMENT FILTER

For sequences such as Trace B, however, where the number of long-cycle identifiers is greater than the number of short cycle identifiers, this scheduling strategy became less effective. Under these conditions, the filter would eventually 'stall' as the filter was would fill with long cycle identifiers, blocking the short-cycle ones (see Figure 7-3). For configurable applications with unknown data sets, this strategy was deemed impractical.

7.3.2 Filter Segmentation

The second strategy was to segment the filter into multiple time domains. Each filter segment is associated with identifiers of a specific cycle time and the number of mailboxes dedicated to each cycle time is determined by the number of messages that occupy that time domain.

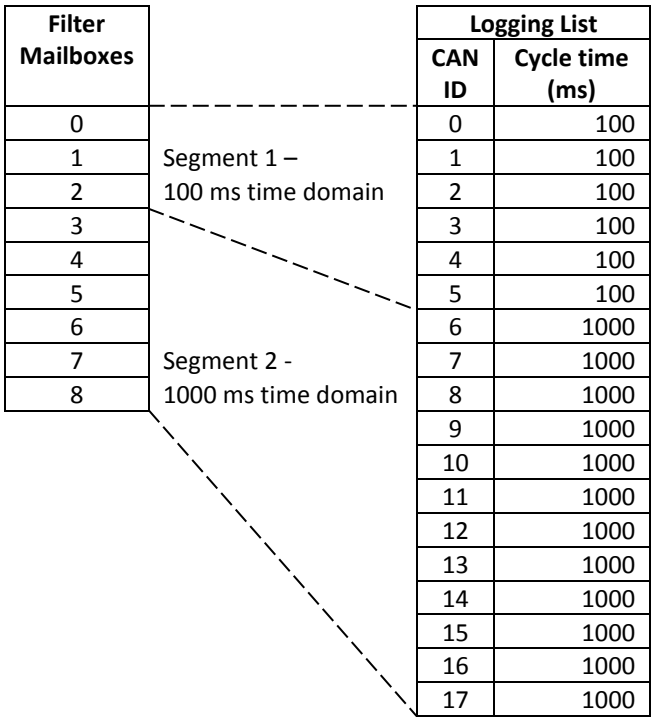


FIGURE 7-4: FILTER SEGMENTATION FOR DIFFERENT MESSAGE CYCLE TIMES

This method ensures that the less frequent messages are no longer blocking the filter. Each time domain is able to update on every successful message arrival, producing the most consistent results across all identifiers for various combinations of CAN message timing conditions.

Figure 7-5 shows the hit rates for Trace B for both single-segment and multi-segment filters. It can be seen that the optimum filter size falls well within the 50 % ratio found for Trace A.

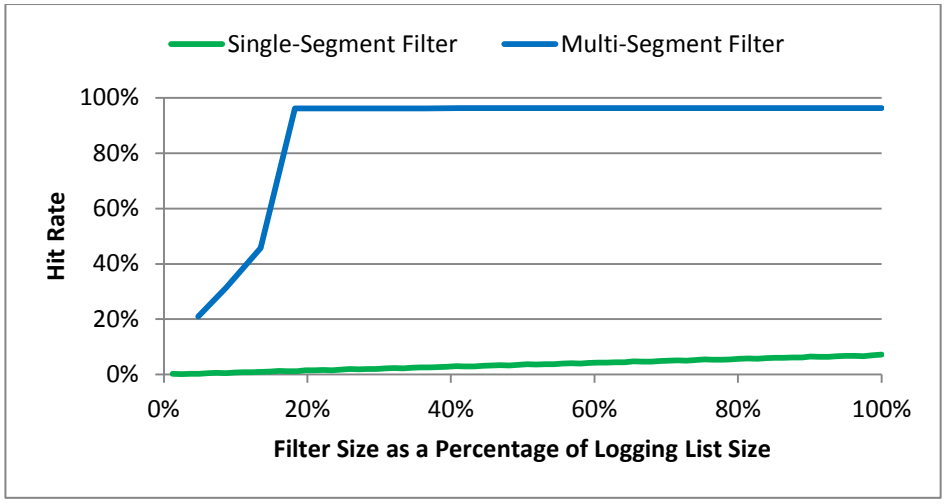


FIGURE 7-5: HIT RATE FOR SEGMENTED FILTER COMPARED TO SINGLE-SEGMENT FILTER

7.4 Data bursts

It is possible for poorly designed nodes to attempt to commit a large set of messages for transmission simultaneously. If this occurs, the arbitration techniques defined in the CAN Specification mean that messages are received by the other nodes on the bus in numerical order by identifier [5]. Moreover, these messages will arrive in bursts with inter-frame spacing dependent on the configured baud rate of the network.

It has been shown in Chapter 2 above that the shortest time between frames on a 500 kbit/s CAN bus is 94 μs. This means that in our TT system, polling the mailboxes at 1000 μs, there will be a maximum of 10 frames arriving between filter updates. Given equation 2.1, the system will still catch the contents of such data bursts as long as the filter contains a minimum of 10 configured mailboxes, or there are less than 10 unique identifiers present on the network.

7.5 Limitations

There are several limitations to the current system. One limitation is that the filter currently relies on a known filter size – list size ratio. Through testing, the best compromise for this value was found to be 50 %; however, it has been suggested in some tests that this could be lower for some traces (see Figure 7-5). In order to truly dynamically optimise the filter, more information is needed about the CAN sequence characteristics that affect this ratio.

Another limitation is that the segmenting algorithm assigns to each time domain a number of mailboxes equal to 50 % of the number of identifiers. This means that the last segment to be configured is only given the remainder of mailboxes available in the hardware. It is possible for a CAN sequence running many different time domains to fail this process. The segmentation process also assumes that messages are grouped in the logging list by time domain.

7.6 Future Development

Further development of the project could include the addition of 'hard' mailboxes for specific critical identifiers. This concept would allow a 100 % hit rate for these specified messages, with the remainder of the mailboxes being dynamically assigned to receive the remainder of the messages. In a similar approach, some of the mailboxes could be reserved for message transmission, allowing the system to transmit CAN messages as well as receive them. There is also scope to allow configurable sampling rates of the identifiers. For some applications, the transmission rate of a particular message could be greater than the desired sampling rate. For example if a message is being transmitted at 100 Hz, but the receiving node is only capable of processing the data at 10 Hz, the filter could be configured, on a per-ID basis, to slow down the mailbox configuration rate and save processing time.

The accuracy of the simulation tool requires further investigation, particularly into the causes for the varying accuracy between different CAN bus conditions. If the causes for this variation could be determined, a more accurate simulation tool could be developed, providing improved pre-implementation predictions.

Chapter 8 Conclusion

This project has presented a novel way to filter data reception from a CAN bus using a Time-Triggered Hybrid Scheduler. The filtering algorithm takes advantage of the hardware message-rejection properties of a 'standard CAN' mailbox system, whilst extending the number of messages that can be accepted to at least double the number of mailboxes. The project demonstrates that, with control of the number of duplicate identifiers in the filter, identifiers that are continually transmitted out of sequence by the host nodes can be captured as often as those transmitted more consistently can. Moreover, the data set to be filtered can be configured by a remote system, allowing for a flexible design, and allowing changes to be made to the filter at run-time.

The project also shows that, due to the highly predictable nature of Time-Triggered architecture, a desktop simulation application can be built. This application has the potential to predict, to great accuracy, the hit rate for a particular data set given a trace file from the target CAN bus. The result is an application executed faster than real-time in a desktop environment that would allow performance guarantees to be made to the client prior to integration of the embedded system.

The predictable execution time of the configurable filter mechanism means the embedded application could be used as a framework to build a sophisticated data logging device, whilst being able to sample over 99% of the messages on a busy CAN bus.

Works Cited

[1] K. W. Tindell, H. Hansson and A. J. Wellings, “Analysing real-time communications: controller area network (CAN).,” in *Real-Time Systems Symposium*, 1994.

[2] M. J. Pont and A. Wesley, Patterns for Time Triggered Embedded Systems, Oxford: ACM Press, 2001.

[3] M. J. Pont, “Applying time-triggered architectures in reliable embedded systems: challenges and solutions,” *Elektrotechnik & Informationstechnik,* vol. 125, no. 11, pp. 401-405, 2008.

[4] M. Farsi, K. Ratcliff and M. Barbosa, “An overview of Controller Area Network,” *Computing and Engineering Journal,* vol. 10, no. 3, pp. 113-120, 1999.

[5] Robert Bosch GmbH, “CAN Specification Version 2.0,” Bosch, Stuttgart, 1991.

[6] H. Zimmermann, “OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection,” *IEEE Transactions on Communcations,* vol. 28, no. 4, pp. 425-432, 1980.

[7] P. Richars, “A CAN Physical Layer Discussion,” Microchip Technology Inc., Chandler, AZ, 2002.

[8] STMicroElectronics, “Controller area network (bxCAN),” in *RM0090 Reference Manual: STM32F405xx, STM32F407xx, STM32F415xx and STM32F417xx advanced ARM-based 32-bit MCUs*, 2012, pp. 784 - 825.

[9] Texas Instruments, TMS320F2833x, 2823x Enhanced Controller Area Network (eCAN) Reference Guide, Dallas, Texas, 2009.

[10] Netburner Inc., “Comparing RTOS to Infinite Loop Designs,” 2005. [Online]. Available: http://www.netburner.com/support/documents/mod5213/development-kit-2/77-850-2.

[Accessed 20 December 2013].

[11] A. Maiita and M. J. Pont, “Using 'planned pre-emption' to reduce levels of task jitter in a time-triggered hybrid scheduler,” in *Processfings of the 2nd UK Embedded Forum*, 2005.

[12] L. Almeida, J. Fonseca and P. Fonseca, “Flexible time-triggered communication on a controller area network.,” in *Proc. of the Work in Progress Session of the 19th IEEE Real-Time Systems Symposium*, 1998.

[13] L. Almeida, P. Padreiras and J. A. G. Fonseca, “The FTT-CAN protocol: Why and how,” *IEEE Transactions on Industrial Electronics,* vol. 49, no. 6, pp. 1189-1201, 2002/12.

[14] L. Almeida, R. Pasadas and J. A. Fonseca, “Using a planning scheduler to improve the flexibility of real-time fieldbus networks,” *Control Engineering Practice,* vol. 7, no. 1, pp. 101-108, 1999.

[15] P. Pedreiras, L. Almeida and J. A. Fonseca, “A Proposal to Improve the Responsiveness of FTT-CAN,” *Electrónica e Telecomunicações,* vol. 3, no. 2, pp. 108-112, 2012.

[16] R. Dobrin and G. Fohler, “Implementing off-line message scheduling on Controller Area Network (CAN).,” in *8th IEEE International Conference on Emerging Technologies and Factory Automation*, 2001.

[17] P. Martí, J. Yépez, M. Velasco, R. Villà and J. M. Fuertes, “Managing quality-of-control in network-based control systems by controller and message scheduling co-design.,” *IEEE Transactions on Industrial Electronics,* vol. 51, no. 6, pp. 1159-1167, 2004.

[18] T. Pop, P. Eles and Z. Peng, “Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems.,” in *Proceedings of the tenth international symposium on Hardware/software codesign*, 2002.

[19] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer.," *IBM Systems journal,* vol. 5, no. 2, pp. 78-101, 1966.

[20] J. Reineke, D. Grund, C. Berg and R. Wilhelm, "Timing predictability of cache replacement policies," *Real-Time Systems,* vol. 37, no. 2, pp. 99-122, 2007.

[21] D. A. B. Weikle, S. A. McKee, K. Skadron and W. A. Wuld, "Caches as filters: A framework for the analysis of caching systems.," in *In Grace Murray Hopper Conference.*, 2000.

[22] Microchip Technology Inc., "MCP2515: Stand-Alone CAN Controller With SPI Interface," Microchip, 2007.

[23] B. Fry and C. Reas, "Processing 2," Media Template, [Online]. Available: http://processing.org/. [Accessed November 2013].

[24] K. M. Zuberi and K. G. Shin, "Design and implementation of efficient message scheduling for controller area network.," *IEEE Transactions on Computers,* vol. 49, no. 2, pp. 182-188, 2000.

[25] H. Kopetz, "The time-triggered model of computation," in *The 19th IEEE Real-Time Systems Symposium, 1998. Proceedings*, 1998.

Appendix A: Source Code – Embedded Software

(Header files and TTH scheduler source omitted for clarity)

```
/*****************************************************************************
 * CAN_Rx_global.c
 *
 * Global CAN message receive buffers and control variables
 *
 *  Created on: 7 Mar 2013
 *      Author: chris.barlow
 * **********************/

#include "CAN_Rx_Filter_global.h"

/* Filter shadow is necessary due to being unable to read a mailbox's ID from registry */
filterShadow_t mailBoxFilterShadow_G[NUM_MESSAGES_MAX];

/* The main CAN Rx buffer - holds all received data in logging sequence order */
canRxMessage_t CAN_RxMessages_G[NUM_MESSAGES_MAX];

/* The logging list - list of CAN IDs transmitted to device in logging sequence order */
logging_list_t loggingList_G[NUM_MESSAGES_MAX];

/* The global sequence update state */
updateFlags_t updateSequenceRequired_G = INIT;



/* Global control variables */
Uint16 numRxCANMsgs_G = 0;
Uint16 filterSize_G = 0;
Uint16 numSegments_G = 0;



/* Filter segment array */
filterSegment_t segments[NUM_FILTER_SEGMENTS_MAX];
static Uint16 segmentNumber = 0;
```

```
/*****************************************************************************
 * Sets the mailbox at filterIndex to initial state
 * **********************/

void initFilter(Uint16 filterIndex){
   Uint16 sequenceIndex_init, segmentIndex;

   segmentIndex =  findSegment(filterIndex);
   sequenceIndex_init = segments[segmentIndex].sequenceIndex++;

   /* Replicates the duplicates mechanism for first use of ID */
   CAN_RxMessages_G[sequenceIndex_init].duplicates = 0;

   /* ID replacement in shadow */
   mailBoxFilterShadow_G[filterIndex].canID_mapped = CAN_RxMessages_G[sequenceIndex_init].canID;
   mailBoxFilterShadow_G[filterIndex].sequenceIndex_mapped = sequenceIndex_init;

   /* Real ID replacement - also re-enables mailbox*/
   configureRxMailbox(CANPORT_A, filterIndex, ID_STD, CAN_RxMessages_G[sequenceIndex_init].canID,
CAN_RxMessages_G[sequenceIndex_init].canDLC);
}


/*****************************************************************************
 * Copies sequence details from temporary buffers to global message sequence array.
 * Since we don't know where in the sequence we will start, the schedule timer for all messages is set to 1.
 * **********************/
void buildSequence(Uint16 listSize){
   Uint16 i, cycleTime_min = 0;

   segmentNumber = 0;

   /* Finds the minimum cycle time in the logging list */
   cycleTime_min = 0;
   numRxCANMsgs_G = listSize;
   printf("msgs:%u\n",numRxCANMsgs_G);
   for(i=0;i<listSize;i++){

/* Segments are assigned dynamically -
 * Limitations: ID's must be sent to the device ordered by cycle time, lowest - highest. */
         if(loggingList_G[i].cycleTime_LLRx > cycleTime_min){
            cycleTime_min = loggingList_G[i].cycleTime_LLRx;
            newSegment(i);
         }
```

```
        CAN_RxMessages_G[i].canID = loggingList_G[i].canID_LLRx;
        CAN_RxMessages_G[i].canData.rawData[0] = 0;
        CAN_RxMessages_G[i].canData.rawData[1] = 0;
        CAN_RxMessages_G[i].canDLC = loggingList_G[i].canDLC_LLRx;

        /* Force all timers to 1 for first iteration - level playing field */
        CAN_RxMessages_G[i].duplicates = 1;
        CAN_RxMessages_G[i].counter = 0;
   }

   /* final call to newSegment initialises the end point of the last segment */
   newSegment(listSize);
   numSegments_G = segmentNumber;
 }


/*****************************************************************************
 * Replaces the ID in the filter at location filterPointer, with ID from sequence at location sequencePointer.
 * ***********************/
void updateFilter(Uint16 filterIndex){
   Uint16 sequenceIndex_new = 0;
   Uint16 segment;

   sequenceIndex_new = getNextSequenceIndex(filterIndex);

   /* Message scheduling */
   CAN_RxMessages_G[mailBoxFilterShadow_G[filterIndex].sequenceIndex_mapped].duplicates = 0;
/* We allow more duplicates if a message is accepted by the filter (regardless of the number of duplicates already present) */

   /* ID replacement in shadow */
   mailBoxFilterShadow_G[filterIndex].canID_mapped = CAN_RxMessages_G[sequenceIndex_new].canID;
   mailBoxFilterShadow_G[filterIndex].sequenceIndex_mapped = sequenceIndex_new;

   /* Real ID replacement - also re-enables mailbox*/
   configureRxMailbox(CANPORT_A, filterIndex, ID_STD, CAN_RxMessages_G[sequenceIndex_new].canID,
CAN_RxMessages_G[sequenceIndex_new].canDLC);
}
```

```
/*****************************************************************************
 * Controls the scheduling of the IDs in the filter.
 * **********************/
Uint16 getNextSequenceIndex(Uint16 mailbox_num){
   Uint16 sequenceIndex_next = 0;
   Uint16 segment;

   segment =  findSegment(mailbox_num);
   sequenceIndex_next = segments[segment].sequenceIndex;

   /* Find next required CAN ID in sequence */
   do{
        /* Wrap search */
        if(sequenceIndex_next < segments[segment].sequenceEnd){
           sequenceIndex_next++;
        }
        else{
           sequenceIndex_next = segments[segment].sequenceStart;
        }

        /* Duplication logic - decides if the ID is allowed into the mailbox */
        if(CAN_RxMessages_G[sequenceIndex_next].duplicates <= DUPLICATES_LIMIT){
           segments[segment].sequenceIndex = sequenceIndex_next;
           CAN_RxMessages_G[sequenceIndex_next].duplicates++;
        }
        else{
           CAN_RxMessages_G[sequenceIndex_next].duplicates = CAN_RxMessages_G[sequenceIndex_next].duplicates;
           segments[segment].sequenceIndex = segments[segment].sequenceIndex;
        }
   }    /* Search will abort if all messages have been checked */
   while(sequenceIndex_next != segments[segment].sequenceIndex);
   return sequenceIndex_next;
}
/*****************************************************************************
 * Returns the filter segment that matches the requested mailbox. *
 * **********************/
Uint16 findSegment(Uint16 mailbox){
   Uint16 segmentNumber = 0, i;

   for(i = 0; i < numSegments_G; i++){
        if((mailbox >= segments[i].filterStart) && (mailbox <= segments[i].filterEnd)){
           segmentNumber = i;
        }
   }
   return segmentNumber;
}
```

```
/*****************************************************************************
 * Ends previous segment, and begins a new one. *
 * **********************/
void newSegment(Uint16 SequenceIndex){
   Uint16 filterIndex = 0;
   printf("I:%u",SequenceIndex);

   /* Dynamically assigns a space in the filter depending on the current SequenceIndex location and the FILTERSIZE_RATIO */
   filterIndex = SequenceIndex/FILTERSIZE_RATIO;
   if((SequenceIndex%FILTERSIZE_RATIO)!=0){
        filterIndex += 1;
   }

   /* Makes sure the filter doesn't overflow */
   if(filterIndex > NUM_MAILBOXES_MAX){
        segments[segmentNumber].filterEnd = (NUM_MAILBOXES_MAX-1);
   }
   else{
        segments[segmentNumber].filterEnd = (filterIndex-1);
   }

   /* Global used for filter looping */
   filterSize_G = (segments[segmentNumber].filterEnd + 1);

   /* Set sequence end point */
   segments[segmentNumber].sequenceEnd = (SequenceIndex-1);

   printf("Seg:%uSE:%uFE:%u\n",segmentNumber,segments[segmentNumber].sequenceEnd,segments[segmentNumber].filterEnd);

   /* Don't increment on initial function call */
   if(SequenceIndex > 0){
        segmentNumber++;
        printf("Seg:%u\n",segmentNumber);
   }

   /* Start a new segment if there are logging list items left */
   if((SequenceIndex < numRxCANMsgs_G) && (segmentNumber < (NUM_FILTER_SEGMENTS_MAX-1))){
        segments[segmentNumber].filterStart = filterIndex;
        segments[segmentNumber].sequenceStart = SequenceIndex;
        segments[segmentNumber].sequenceIndex = SequenceIndex;
        printf("SS:%uFS:%u\n",segments[segmentNumber].sequenceStart,segments[segmentNumber].filterStart);
   }
}



/*****************************************************************************
```

```
 *  receiveCAN.c
 *      checks the status of mailboxes. When a message is pending, the data is read
 *      and the dynamic filter mechanism updates the mailbox to the next valid CAN ID
 *
 *  Created on: 19 Jun 2013
 *      Author: chris.barlow
 * **********************/

#include "../../global.h"
#include "receiveCAN.h"
#include <stdio.h>
#include "../../CAN_Exchange/CAN_Rx_Filter_global.h"

/*****************************************************************************************************
 * Initialisation - called once when the device boots, before the scheduler starts.
 * ***************************************************************************************************/
void receiveCAN_init(void){
   /* mailboxes are configured in _update when first logging list is received from desktop app */
   updateSequenceRequired_G = INIT;
}

/*****************************************************************************************************
 * Update function - called periodically from scheduler
 * ***************************************************************************************************/
void receiveCAN_update(void){
   static Uint16 mailBox = 0;
   Uint16 sequenceIndex_received, sequenceIndex_new;

   /* updateSequenceRequired_G controls the sequence update mechanism when a new logging list is transmitted to the device */
   switch(updateSequenceRequired_G){
   /* Do nothing until first logging list arrival (RESET)*/
   default:
   case INIT:
        break;

   /* controlSCI will initiate RESET when new logging list is received */
   case RESET:

        /* Ensure all mailboxes are disabled */
        for(mailBox = 0; mailBox < NUM_MAILBOXES_MAX; mailBox++){
           disableMailbox(CANPORT_A, mailBox);
        }

        mailBox = 0;
        updateSequenceRequired_G = UPDATE;
        break;
```

```
/* Set up mailboxes for initial filter conditions */
   case UPDATE:
         /* Direct copy of first filterSize_G IDs in the sequence */
         initFilter(mailBox);
         mailBoxFilterShadow_G[mailBox].mailboxTimeout = MAILBOX_DECAY_TIME;

         /* Initialising one mailBox per tick ensures all mailboxes are initialised before moving to RUN (mainly so that we can
printf some debug info) */
         mailBox++;
         if(mailBox == filterSize_G){
            updateSequenceRequired_G = RUN;
         }
         break;

/* Checking for CAN messages and updating filters - normal running conditions */
   case RUN:
         /* look through mailboxes for pending messages */
         for(mailBox=0; mailBox<filterSize_G; mailBox++){
            updateSingleMailbox(CANPORT_A, mailBox);

            if(checkMailboxState(CANPORT_A, mailBox) == RX_PENDING){
               disableMailbox(CANPORT_A, mailBox);

/* read the CAN data into buffer (Nothing is done with the data, but nice to do this for realistic timing) */
               readRxMailbox(CANPORT_A,
mailBox,CAN_RxMessages_G[mailBoxFilterShadow_G[mailBox].sequenceIndex_mapped].canData.rawData);

               /* Count message hits */
               CAN_RxMessages_G[mailBoxFilterShadow_G[mailBox].sequenceIndex_mapped].counter++;

               /* update the filter for next required ID  */
               updateFilter(mailBox);     /* Mailbox is re-enabled in configureRxMailbox() - this is done last to help prevent new
message arrivals causing erroneous hits mid-way through process*/
            }
         }

         break;
   }
}
```

```
/**************************************************************************
 * controlSCI.c
 *
 * Controls the serial data transfer between device and desktop application via the TI SCI port
 *
 * Created on: 25 June 2013
 *      Author: chris.barlow
 * **********************/

#include "../../Lib/SCI/SCI.h"
#include "controlSCI.h"
#include <stdio.h>
#include "../../CAN_Exchange/CAN_Rx_Filter_global.h"

/* SCI states */
typedef enum{WAITING,RECEIVE,SEND_M,SEND_S}SCIstate_t;

/* Raw character receive buffer */
static char rxbuffer[350];
Uint16 rxbufferSize = (sizeof(rxbuffer)/sizeof(rxbuffer[0]));

/* Position control for packet data */
enum {
   FSC_DATAPOSITION = 1,
   DUP_DATAPOSITION = 2,
   IDH_DATAPOSITION = 3,
   IDL_DATAPOSITION = 4,
   DLC_DATAPOSITION = 5,
   CYT_DATAPOSITION = 6
};

/* Temporary arrays for data unpacking */
typedef struct{
   Uint16 sequenceIndex_SCITx;
   Uint16 canID_SCITx;
} tempShadow_t;
tempShadow_t mailBoxFilterShadow_SCITx[NUM_MESSAGES_MAX];

/**************************************************************************
 * Initialisation - called once when the device boots, before the scheduler starts.
 * **********************/
void controlSCI_init(void){
   /* This TI function is found in the DSP2833x_Sci.c file. */
   InitSciaGpio();
   scia_fifo_init();       /* Initialize the SCI FIFO */
   scia_init();                  /* Initalize SCI for echoback */
}
```

```
/*****************************************************************************
 * Update function - called periodically from scheduler
 * **********************/
void controlSCI_update(void){
   static SCIstate_t SCIstate = WAITING;
    static Uint16 i = 0, j = 0;
    Uint32 IDH = 0, IDL = 0;
    char tempCharOut;
    static Uint16 indexShift = 0;
    Uint16 sequenceNum = 0, sequenceSize_Rx = 0;


    /* state machine controls whether the device is transmitting or receiving logging list information
     * will always receive until first logging list is received */
    switch(SCIstate){

    case WAITING:
        /* First character received induces RECEIVE state */
        if(SciaRegs.SCIFFRX.bit.RXFFST != 0){
           for(i=0;i<rxbufferSize;i++){
              rxbuffer[i] = 0;
           }
           i=0;
           SCIstate = RECEIVE;
        }
        break;

    case RECEIVE:

        /* Checks SCI receive flag for new character */
        if(SciaRegs.SCIFFRX.bit.RXFFST != 0){
           rxbuffer[i] = SciaRegs.SCIRXBUF.all;

           /* "???" sent by desktop app indicates that a reset is required (someone pressed the 'R' key) */
           if((rxbuffer[i] == '?')&&(rxbuffer[i-1] == '?')&&(rxbuffer[i-2] == '?')){
              SCIstate = WAITING;
           }
           /* Received data packet looks like this:
            *              index:  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
            *              chars:  { f d a a A X b b B  Y  c  c  C  Z  ~  }
            * Where:
            *              f is the filter size control constant
            *              d is the duplication control constant
            *       aa is two byte CAN ID
            *              A is the CAN data length
            *       X is the CAN message cycle time
            *       etc * */
```

```
if((i>0)&&(rxbuffer[i-1] == '~')&&(rxbuffer[i] == '}')){
     /* flag tells receiveCAN to update the logging sequence */
     updateSequenceRequired_G = INIT;

     /* In above eg, i = 16 at end of packet, numRxCANMsgs_G = 3 */
     sequenceSize_Rx = (i-4)/4;

     /* Safeguard against mailbox overload */
     if(rxbuffer[FSC_DATAPOSITION] <= NUM_MAILBOXES_MAX){
     filterSize_G = rxbuffer[FSC_DATAPOSITION];
     }
     else{
     filterSize_G = NUM_MAILBOXES_MAX;
     }

     /* Unpackaging logging list info from data packet */
     for(sequenceNum=0;sequenceNum<sequenceSize_Rx;sequenceNum++){
     IDH = rxbuffer[(4*sequenceNum)+IDH_DATAPOSITION];
     IDH <<= 8;
     IDL = rxbuffer[(4*sequenceNum)+IDL_DATAPOSITION];

     loggingList_G[sequenceNum].canID_LLRx = (IDH|IDL);
     loggingList_G[sequenceNum].canID_LLRx &= 0x7FF;
     loggingList_G[sequenceNum].canDLC_LLRx = rxbuffer[(4*sequenceNum)+DLC_DATAPOSITION];
     loggingList_G[sequenceNum].cycleTime_LLRx = rxbuffer[(4*sequenceNum)+CYT_DATAPOSITION];
     }

     /* Initialise sequence */
     buildSequence(sequenceSize_Rx);

     /* flag tells receiveCAN to update the logging sequence */
     updateSequenceRequired_G = RESET;

     SCIstate = SEND_M;
   }
   else if(rxbuffer[0] == '{'){
     /* data packet reception still in progress */
     i++;

     /* Reset state if buffer overflows - can happen if data loss occurs */
     if(i >= (sizeof(rxbuffer)/sizeof(rxbuffer[0]))){
     SCIstate = WAITING;
     }
   }
}
break;
```

```
case SEND_M:
/* check for reset request from desktop app */
if(SciaRegs.SCIFFRX.bit.RXFFST != 0){
   rxbuffer[0] = SciaRegs.SCIRXBUF.all;
}

if(rxbuffer[0] == '?'){
   SCIstate = WAITING;
}
else{
   /* Take snapshot of filters (should prevent updates halfway through transmission)*/
   for(i=0;i<filterSize_G;i++){
      j = mailBoxFilterShadow_G[i].sequenceIndex_mapped;
      mailBoxFilterShadow_SCITx[i].sequenceIndex_SCITx = j;
      mailBoxFilterShadow_SCITx[i].canID_SCITx = mailBoxFilterShadow_G[i].canID_mapped;
   }
   /* Transmit mailbox data
    *
    * Data packet looks like this:
    *    index:  0 1 2 3 4 5 6 7
    *    chars:  { M A a a X ~ }
    * Where:
    *    A is the sequence location mapped to mailbox
    *    aa is two byte CAN ID
    *    X mailbox location
    *    This is fixed length.
    * */
   scia_xmit('{');
   scia_xmit('M');

   for(i=0;i<filterSize_G;i++){
      j = mailBoxFilterShadow_SCITx[i].sequenceIndex_SCITx;

      tempCharOut = ((mailBoxFilterShadow_SCITx[i].canID_SCITx>>8) & 0xFF);
      scia_xmit(tempCharOut);

      tempCharOut = (mailBoxFilterShadow_SCITx[i].canID_SCITx & 0xFF);
      scia_xmit(tempCharOut);

      tempCharOut = (j & 0xFF);
      scia_xmit(tempCharOut);
    }

   scia_xmit('~');
   scia_xmit('}');
```

```
        /* Instruct desktop app to refresh screen */
        scia_xmit('{');
        scia_xmit('~');
        scia_xmit('}');

        SCIstate = SEND_S;

      }

      break;
  case SEND_S:

      /* check for reset request from desktop app */
      if(SciaRegs.SCIFFRX.bit.RXFFST != 0){
         rxbuffer[0] = SciaRegs.SCIRXBUF.all;
      }

      if(rxbuffer[0] == '?'){
         SCIstate = WAITING;
      }
      else{
         /* Transmit message counts
          * Due to the large amount of data for the message counts
          * Data is transmitted as max 10 values, 6 apart, offset by pointerShift
          *
          * Data packet looks like this:
          *     index:  0 1 2 3 4 5 6 7 8
          *     chars:  { S A a a a a ~ }
          * Where:
          *    A is the sequence location
          *    aaaa is four byte hit count for the sequence location
          *
          *    This is fixed length.
          * */
         for(i=0;i<=SEQ_TX_CHUNK_SIZE;i++){

           j = (i*SEQ_TX_CHUNK_SPACING)+indexShift;

           if(j<=numRxCANMsgs_G){
           scia_xmit('{');
           scia_xmit('S');
           /* Need to tell the desktop app the array index for this value */
           tempCharOut = (j & 0xff);
           scia_xmit(tempCharOut);
```

```
            /* Send the 32-bit counter value */
            tempCharOut = ((CAN_RxMessages_G[j].counter>>24)&0xFF);
            scia_xmit(tempCharOut);
            tempCharOut = ((CAN_RxMessages_G[j].counter>>16)&0xFF);
            scia_xmit(tempCharOut);
            tempCharOut = ((CAN_RxMessages_G[j].counter>>8)&0xFF);
            scia_xmit(tempCharOut);
            tempCharOut = ((CAN_RxMessages_G[j].counter)&0xFF);
            scia_xmit(tempCharOut);

            scia_xmit('~');
            scia_xmit('}');
            }

         }
         /* Increment pointerShift to inter-space next set of values next time */
         indexShift++;
         if(indexShift > 10){
            indexShift = 0;
         }
         /* Instruct desktop app to refresh screen */
         scia_xmit('{');
         scia_xmit('~');
         scia_xmit('}');

         SCIstate = SEND_M;
      }

      break;

   default:
       break;
   }

   /* ? symbol acts as a handshake request with the desktop app */
  scia_xmit('?');

}
```

Appendix B: Source Code – Remote Configuration and Analysis Tool

```
/**
 * Remote CAN filter configuration and
 * Filter Mapping Visualisation
 *
 * A close-to-realtime visualisation of the mapping between
 * the device's mailboxes and the logging list.
 *
 * Also transmits the logging list to the device over RS232
 * as a form of remote configuration
 *
 */

/* Serial port definitions */
import processing.serial.*;
Serial myPort;
int[] serialInArray = new int[1000];
int serialCount = 0;

File lastFile;

/* Global state flags */
boolean readyState = false;
int status = 0;
boolean allRefresh = false;
int hsCount = 0;

/* fonts */
PFont font, fontBold;

/* standard positioning values and indexing vars */
int d = 11;
int s = 200;
int w = 900;
int i,j;

/* This array holds the variable end positions for the mapping lines (logging list end) */
int[] mapLineEnd = new int[100];

/* This array holds the variable ID's in the device mailboxes */
int[] IDs = new int[100];

/* logging list transmission progress */
int txPointer = 0;
```

```
/* Config */
int filterSizeTx = 32;      /* Set to zero to enable auto-sizing */
int duplicatesAllowed = 1; /* This isn't implemented in the device code. Still deciding if it's beneficial */

/* The logging list. This is transmitted to the device for filter configuration */

  int[][] loggingList = {
    {0x185,8,10},{0x385,8,10},
    {0x187,8,10},{0x207,8,10},{0x287,8,10},{0x307,8,10},{0x387,8,10},
    {0x189,8,10},{0x209,8,10},{0x289,8,10},{0x309,8,10},{0x389,8,10},
    {0x18B,8,10},{0x20B,8,10},{0x28B,8,10},{0x30B,8,10},{0x38B,8,10},
    {0x18D,8,10},{0x20D,8,10},{0x28D,8,10},{0x30D,8,10},{0x38D,8,10},
    {0x3A0,8,100},{0x3B0,8,100},{0x3C0,8,100},{0x3D0,8,100},
    {0x3A1,8,100},{0x3B1,8,100},{0x3C1,8,100},{0x3D1,8,100},
    {0x3A2,8,100},{0x3B2,8,100},{0x3C2,8,100},{0x3D2,8,100},
    {0x3A3,8,100},{0x3B3,8,100},{0x3C3,8,100},{0x3D3,8,100},
    {0x3A4,8,100},{0x3B4,8,100},{0x3C4,8,100},{0x3D4,8,100},
    {0x3A5,8,100},{0x3B5,8,100},{0x3C5,8,100},{0x3D5,8,100},
    {0x3A6,8,100},{0x3B6,8,100},{0x3C6,8,100},{0x3D6,8,100},
    {0x3A7,8,100},{0x3B7,8,100},{0x3C7,8,100},{0x3D7,8,100},
    {0x3A8,8,100},{0x3B8,8,100},{0x3C8,8,100},{0x3D8,8,100},
    {0x3A9,8,100},{0x3B9,8,100},{0x3C9,8,100},{0x3D9,8,100},
    {0x3AA,8,100},{0x3BA,8,100},{0x3CA,8,100},{0x3DA,8,100},
    {0x3AB,8,100},{0x3BB,8,100},{0x3CB,8,100},{0x3DB,8,100},
    {0x3AC,8,100},{0x3BC,8,100},{0x3CC,8,100},{0x3DC,8,100},
    {0x3AD,8,100},{0x3BD,8,100},{0x3CD,8,100},{0x3DD,8,100},
    {0x3AE,8,100},{0x3BE,8,100},{0x3CE,8,100},{0x3DE,8,100}
  };


/* counters for counting */
long[] counters = new long[loggingList.length];
long[] countersTemp = new long[loggingList.length];
long countersTotal = 0;

/* the number of mailboxes used for the filter (this should be half the number of IDs in the logging list */
int filterSizeRx = 0;


/* setup is called by Processing once on startup */
void setup(){
   /* Serial port will be Serial.list()[0] when nothing else connected */
   try{
      println(Serial.list());
      String portName = Serial.list()[0];
      myPort = new Serial(this, portName, 9600);
   }
```

```
   catch(Exception e){
      /* App will close if no serial ports are found */
      println("No serial ports found. Use your dongle!");
      exit();
   }

   size(1200, 1000);
   background(0);
   font = loadFont("Consolas-16.vlw");
   fontBold = loadFont("Consolas-Bold-16.vlw");
   textFont(font, 10);
   stroke(153);
   /* RS232 reception triggers redraw */
   noLoop();
}

/* draw is called by processing for every redraw();
void draw(){
   int barLength = 0;
   String strg = "";

   try{
      /* Mailbox and logging list blocks */
      background(10);
      stroke(255);
      fill(20);
      rect((s-((4*d)+110)), 2, s-d-(s-((4*d)+110)), 25+(filterSizeRx*d),10);
      rect(w+d, 2, 250, 2*d+(loggingList.length*d),10);

      stroke(255);
      fill(255);
      textFont(fontBold, 10);
      text("Device Mailboxes", (s-((4*d)+100)), (d+4));
      text(" Logging List      Hits", (w+d+3), (d+4));

      if(allRefresh == true){
         countersTotal = 0;
      }

      /* Draws logging list details */
      for(i=0;i<loggingList.length;i++){

         if(allRefresh == true){
            counters[i] = countersTemp[i];
            countersTotal += counters[i];
         }
```

```
   stroke(45);
   line(0, standardSpacingY(i,d/2), 1200, standardSpacingY(i,d/2));
   stroke(255);
   strg = intToStr_02(i);
   text(strg+": "+hex(loggingList[i][0],3)+"           "+counters[i], (w+d+5), standardSpacingY(i,6));
   line(w, standardSpacingY(i,0), (w+d), standardSpacingY(i,0));
}

/* Draws device filter information and mapping lines */
textFont(fontBold, 10);
for(i=0;i<filterSizeRx;i++){
   /* Text and leader lines */
   strg = intToStr_02(i);
   text(strg+": "+hex(IDs[i],3), (s-((4*d)+20)), standardSpacingY(i,6));
   stroke(255);
   line(s, standardSpacingY(i,0), (s-d), standardSpacingY(i,0));

   /* Mapping lines */
   line(s, standardSpacingY(i,0), w, mapLineEnd[i]);
}

/* Title and status block */
fill(0);
rect((s-((4*d)+110)), standardSpacingY(70,15), 760, 150, 10);
fill(255);

switch(status){
case 0:
   strg = "Offline - Can't see device";
   break;
case 1:
   strg = "Offline - Device waiting\nPress 'R' to begin.";
   break;


case 2:
   strg = "Transmitting logging list: ";
   if(txPointer>6){
      strg += txPointer-6;
      rect((s-((4*d)+100)), standardSpacingY(79,15), (740/(loggingList.length/(txPointer-6))), 6);
   }
   break;
case 3:
   strg = "Online\nPress 'R' to reset, 'S' to save hit counts, 'C' to save and close, 'X' to exit without saving.\n";
   strg += "Total hits: ";
   strg += countersTotal;
   break;
```

```
      case 4:
         strg = "Connection lost - press 'R' to reset";
      default:
         break;
      }

      textFont(fontBold, 16);
      text("Dynamic CAN Filter Remote Configuration and Mapping Visualisation Tool", (s-((4*d)+100)), standardSpacingY(73,6));

      textFont(fontBold, 14);
      text("Chris Barlow, MSc Reliable Embedded Systems 2013, University of Leicester", (s-((4*d)+100)), standardSpacingY(74,8));

      textFont(font, 14);
      text("This application displays the CAN mailbox to logging list mapping.", (s-((4*d)+100)), standardSpacingY(76,6));

      text("Logging list is sent to the device on connection", (s-((4*d)+100)), standardSpacingY(77,6));

      textFont(fontBold, 14);
      text("Status: "+strg, (s-((4*d)+100)), standardSpacingY(79,6));
      allRefresh = false;
   }
   catch(Exception e){
      exit();
   }
}

/* Transmitted data packet looks like this:
 *
 *     index:  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
 *     chars:  { f d a a A X b b B  Y  c  c  C  Z  ~  }
 * Where:
 *     f is the filter size control constant
 *     d is the duplication control constant
 *    aa is two byte CAN ID
 *     A is the CAN data length
 *     X is the CAN message cycle time
 *    etc * */

void transmitLoggingList(){
   int txListPointer;

   print(txPointer+" ");

   if(txPointer == 0){ /* packet start */
      myPort.write('{');
      println("{");
   }
```

```
   else if(txPointer == 1){
      myPort.write(filterSizeTx);
      println(filterSizeTx);
   }
   else if(txPointer == 2){
      myPort.write(duplicatesAllowed);
      println(duplicatesAllowed);
   }
   else if(txPointer < loggingList.length+3){
      txListPointer =  txPointer-3;
      /* CAN ID high byte */
      myPort.write((loggingList[txListPointer][0]>>8)&0x87);
      /* CAN ID low byte */
      myPort.write (loggingList[txListPointer][0]&0xFF);
      /* Message length in bytes */
      myPort.write (loggingList[txListPointer][1]&0xFF);
      /* Message cycle time */
      myPort.write (loggingList[txListPointer][2]&0xFF);

      println(hex((loggingList[txListPointer][0]>>8)&0xFF,1)+" "+hex(loggingList[txListPointer][0]&0xFF,2)+"
                                 "+hex(loggingList[txListPointer][1]&0xFF,2)+" "+hex(loggingList[txListPointer][2]&0xFF,2));
   }

   else if(txPointer == (loggingList.length+3)){
   /* packet sign-off */
      myPort.write('~');
      println("~");
   }
   else if(txPointer == (loggingList.length+4)){
      myPort.write('}');
      println("}");

   }
   else{
      println("got here");
   }
}


void receiveLoggingDetails(){
   int loggingListPointer = 0;
   int IDhPointer,IDlPointer,lineEndPointer, txListPointer;
   long messageCounter = 0;

   /* First character of packet after { indicates packet type */
   switch(serialInArray[1]){
```

```
case 'M':
   /* Data packet contains mailbox information
    *
    * Data packet looks like this:
    *    index:  0 1 2 3 4 5 6 7
    *    chars:  { M A a a X ~ }
    * Where:
    *    A is the sequence location mapped to mailbox
    *    aa is two byte CAN ID
    *    X mailbox location
    *    This is fixed length.
    * */

   if(serialCount-3 == 1){
      filterSizeRx = 1;
   }
   else{
      filterSizeRx = (serialCount-3)/3;
   }

   for(loggingListPointer=0;loggingListPointer<filterSizeRx;loggingListPointer++){
      IDhPointer = (3*loggingListPointer)+2;
      IDlPointer = (3*loggingListPointer)+3;
      lineEndPointer = (3*loggingListPointer)+4;

      IDs[loggingListPointer] = ((serialInArray[IDhPointer]<<8) | serialInArray[IDlPointer]);

      mapLineEnd[loggingListPointer] = standardSpacingY(serialInArray[lineEndPointer],0);

   }
   break;

case 'S':
   /* Data packet contains loggingList information
    * Due to the large amount of data for the message counts
    * Data is transmitted as max 10 values, 6 apart, offset by pointerShift
    *
    * Data packet looks like this:
    *     index:  0 1 2 3 4 5 6 7 8
    *     chars:  { S A a a a a ~ }
    * Where:
    *    A is the sequence location
    *    aaaa is four byte hit count for the sequence location
    *
    *    This is fixed length.
    * */
   loggingListPointer = serialInArray[2];
```

```
      if(loggingListPointer < loggingList.length){
         /* Unpack 32 bit counter */
         messageCounter  = ((serialInArray[3]&0xFF)<<24);
         messageCounter |= ((serialInArray[4]&0xFF)<<16);
         messageCounter |= ((serialInArray[5]&0xFF)<<8);
         messageCounter |=  (serialInArray[6]&0xFF);

         /* counters stored in temp array until refresh required */
         countersTemp[loggingListPointer] = messageCounter;

         /* Only refresh loggingList counters on screen when all counters have been received (takes several packets) */
         if(loggingListPointer >= (loggingList.length-1)){
            allRefresh = true;
         }
      }
      break;

   case '~':
      /* Empty serial packet instructs screen refresh */
      redraw();
      break;

   default:
      break;
   }
}


void serialEvent(Serial myPort) {
   try{
      if(serialCount == 0){
         for(j=0;j<serialInArray.length;j++){
         serialInArray[j] = 0;
      }
   }

   /* read a byte from the serial port: */
   serialInArray[serialCount] = myPort.read();

   /* Device sends '?' character as a handshake / logging list request */
   if(serialInArray[serialCount] == '?'){

      /* Prevents '63' values in data stream from being misinterpreted as a handshake request */
      if(hsCount < 10){
         hsCount++;
      }
```

```
      else{
         hsCount = 0;
         if(status == 0){
            status = 1;
         }
         /* if we are in online state, we know that the device has been reset */
         else if(status == 3){
            status = 4;
         }
         print("HS ");
      }
   }
   else{
      hsCount = 0;
   }

   switch(status){
   case 0: /* Offline */
   case 4:
      serialCount = 0;
      txPointer = 0;
      delay_ms(5);
      /* Handshake signals device to wait for new filter information */
      myPort.write('?');
         if(readyState==false){
         status = 0;
      }
      redraw();
      break;

   case 1: /* Offline but Device found */
      myPort.write('?');
      if(readyState==true){
         txPointer = 0;
         status = 2;
      }
      redraw();
      break;

   case 2: /* Transmitting logging list */
      if(readyState==true){
         if((serialInArray[0] == '?')&&(txPointer <= (loggingList.length+4))){
            delay_ms(5);
            transmitLoggingList();
            txPointer++;
            redraw();
         }
```

```
        else if(serialInArray[0] == '{'){
           println("got here");
           status = 3;
        }
        else{
           println("staying here");
        }
     }
     else{
        serialCount = 0;
        status = 0;
        redraw();
     }
     break;

  case 3:    /* Online */
     if(readyState==true){
        /* End of data packet - update data arrays */
        if((serialCount>0)&&(serialInArray[serialCount-1] == '~')&&(serialInArray[serialCount] == '}')){
           receiveLoggingDetails();
           serialCount = 0;
        }
        /* Receiving data packet from TI chip */
        else if((serialInArray[0] == '{') && (serialCount < 999)){
           serialCount++;
        }
     }
     else{
        serialCount = 0;
        status = 0;
        redraw();
     }
     break;

  default:
     break;
  }     }
  catch(Exception e){
     exit();
  }
}
```

Appendix C: Source Code – Feasibility Simulation Tool

```
/*
 ============================================================================
 Name        : PCANalysis.c
 Author      : C Barlow
 Version     :
 Copyright   :
 Description : Performs timing analysis on a PCAN trace
 ============================================================================
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BUFFERSIZE               (81)
#define FILTERSIZE               (200)
#define MAX_TRACE_LINES          (0)  /* Set to zero to analyse entire trace */
#define LOGGING_TASK_PERIOD_us   (1000)


#define FREEZE_TRIES       (0)
#define MESSAGE_TIME_DELTA_MAX   (100500)
#define MESSAGE_TIME_DELTA_MIN   (0)

unsigned long  ID, timeDelta;
unsigned long  Task_WCET;
int noIDs;
unsigned int filterPointer;

typedef struct{
   int canID;
   unsigned int cycleTime;
} logging_list_t;

typedef struct{
   int canID;
   unsigned long counter;
   unsigned long loggedCounter;
   int timer;
   int timer_reload;
} logging_Sequence_t;

typedef enum {TRUE, FALSE}flag_t;
```

```
typedef struct{
   int canID;
   int sequencePointer;
   flag_t loggedFlag;
} filter_t;

logging_Sequence_t loggingSequence[BUFFERSIZE];
filter_t acceptanceFilter[FILTERSIZE];

/* The logging list is the list of CAN ID's to filter out */
logging_list_t loggingList[]={
    {0x185,10},{0x385,10},
    {0x187,10},{0x207,10},{0x287,10},{0x307,10},{0x387,10},
    {0x189,10},{0x209,10},{0x289,10},{0x309,10},{0x389,10},
    {0x18B,10},{0x20B,10},{0x28B,10},{0x30B,10},{0x38B,10},
    {0x18D,10},{0x20D,10},{0x28D,10},{0x30D,10},{0x38D,10},
    {0x3A0,100},{0x3B0,100},{0x3C0,100},{0x3D0,100},
    {0x3A1,100},{0x3B1,100},{0x3C1,100},{0x3D1,100},
    {0x3A2,100},{0x3B2,100},{0x3C2,100},{0x3D2,100},
    {0x3A3,100},{0x3B3,100},{0x3C3,100},{0x3D3,100},
    {0x3A4,100},{0x3B4,100},{0x3C4,100},{0x3D4,100},
    {0x3A5,100},{0x3B5,100},{0x3C5,100},{0x3D5,100},
    {0x3A6,100},{0x3B6,100},{0x3C6,100},{0x3D6,100},
    {0x3A7,100},{0x3B7,100},{0x3C7,100},{0x3D7,100},
    {0x3A8,100},{0x3B8,100},{0x3C8,100},{0x3D8,100},
    {0x3A9,100},{0x3B9,100},{0x3C9,100},{0x3D9,100},
    {0x3AA,100},{0x3BA,100},{0x3CA,100},{0x3DA,100},
    {0x3AB,100},{0x3BB,100},{0x3CB,100},{0x3DB,100},
    {0x3AC,100},{0x3BC,100},{0x3CC,100},{0x3DC,100},
    {0x3AD,100},{0x3BD,100},{0x3CD,100},{0x3DD,100},
    {0x3AE,100},{0x3BE,100},{0x3CE,100},{0x3DE,100}
};
int listSize = sizeof(loggingList)/sizeof(logging_list_t);
int sequenceSize;

char *logFormat           = "%4u.%06u 1  %3x             Tx%s";
char *detailedLogFormat   = "%4u.%06u 1  %3x             Rx   d %1u %02X %02X %02X %02X %02X %02X %02X %02X";

/* Function prototypes */
void buildSequence(void);
flag_t updateFilter(unsigned int filterPointer);
void CanSequenceMessageCounter(char *filename);
void checkLogability(char *filename, FILE *log, int filterSize, int sequenceSize);
void orderSequence(void);
int countSequence(void);
flag_t GetCAN1BufferPointer(unsigned int ID);
```

```
int main(void){
   char *CANlogFile = "MHI_Test_Drive_log-2013-10-21.asc";
   int i;

   FILE *outputFile = fopen("MHI_Test_Drive_log-2013-10-21.trc", "w");
   canTraceConverter(CANlogFile, outputFile);

   FILE *logFile = fopen("CAN_Logging_new.txt", "w");

   noIDs = 0;
   buildSequence();
   CanSequenceMessageCounter(CANlogFile);


/* Use to force sequence to numerical order by ID */
// orderSequence();

   sequenceSize = countSequence();
   printf("\n\n\n\n\n %u ID's\n\n",sequenceSize);

   printf("\n\n\nChecking logability...\r\n\n");
   fprintf(logFile,"\n\n,,Filter Size,Logged,Missed\n");
/* Use for 'full-sweep' mode */
   for(i = 1; i <= sequenceSize; i++)    {
      checkLogability(CANlogFile, logFile, i, sequenceSize);
   }

/* Use to check a specific filter size */
// checkLogability(CANlogFile, logFile, 16, sequenceSize);

   fclose(outputFile);
   fclose(logFile);

   return EXIT_SUCCESS;
}

/** Builds the logging sequence from the logging list. */
void buildSequence(void){
   int i, cycleTime_min;

   cycleTime_min = 0xFFFF;
   for(i=0;i<listSize;i++){
      if(loggingList[i].cycleTime<cycleTime_min){
         cycleTime_min = loggingList[i].cycleTime;
      }
   }
```

```
   for(i=0;i<BUFFERSIZE;i++){
      if(i<listSize){
         loggingSequence[i].canID = loggingList[i].canID;
         loggingSequence[i].timer_reload = loggingList[i].cycleTime/cycleTime_min;
         loggingSequence[i].timer = 1;
         printf("ID: %03X, Period: %u\n", loggingSequence[i].canID, loggingSequence[i].timer_reload);
      }
   }
}

/* Used to update mapping between filter and sequence.
 * Returns a TRUE if the mapping was changed successfully. */
flag_t updateFilter(unsigned int filterPointer){
   static int last_i = -1;
   int i, j;
   flag_t result = FALSE, IDfound = FALSE;

   i = last_i;
   do{
      if(i<(listSize-1)){
         i++;
      }
      else{
         i=0;
      }

      for(j = 0; j < FILTERSIZE; j++){
         if(acceptanceFilter[j].canID == loggingSequence[i].canID){
            IDfound = TRUE;
         }
      }

      if(IDfound == FALSE){
         loggingSequence[i].timer--;
      }

      if(loggingSequence[i].timer<=0){
         result = TRUE;
      }

   }while((result == FALSE)&&(i != last_i));

   if(result == TRUE){
      last_i = i;

      loggingSequence[i].timer = loggingSequence[i].timer_reload;
```

```
      acceptanceFilter[filterPointer].canID = loggingSequence[i].canID;
      acceptanceFilter[filterPointer].sequencePointer = i;
      acceptanceFilter[filterPointer].loggedFlag = FALSE;
   }
   return result;
}

/* Runs simulation on CAN trace compared to recorded sequence.
 * Outputs the number of hits, misses and total number of messages per CAN ID
 * Used to find optimum filter size for a given sequence. */
void checkLogability(char *filename, FILE *log, int filterSize, int sequenceSize){
   char inputStr[200];
   char canData[200];
   int i = 0, IDLogCount = 0, IDMissedCount = 0;
   flag_t IDlogged = FALSE;
   unsigned long timeNow_s ,timeNow_us, timeOrigin, lineCounter=0;

   int ID;

   /* open trace file */
   FILE *bufferFile = fopen(filename, "r");

   for(i = 0; i < filterSize; i++){
      if(updateFilter(i)==TRUE){
         printf("filter ID: %03X\n", acceptanceFilter[i].canID);
      }
      else{
         printf("updateFilter FAIL\n");
      }
   }
   timeOrigin = 0;

   while((fgets(inputStr, 190, bufferFile) != NULL) && ((lineCounter < MAX_TRACE_LINES) || (MAX_TRACE_LINES == 0))){
      /* Extract values from input string */
      unsigned int scanReturn = sscanf(inputStr, logFormat, &timeNow_s, &timeNow_us, &ID, &canData);
      if(scanReturn == 4)  { /* valid line in trace */
         lineCounter++;

         timeNow_us += (timeNow_s * 1000000);

         /* Find timeNow origin */
         if(timeOrigin == 0)       {
            timeOrigin = timeNow_us;
         }

         /* Find current time delta from origin */
         timeDelta = (timeNow_us - timeOrigin);
```

```
      printf("%u %lu\tChecking log line: %s", filterSize, timeDelta, inputStr);

      /* Logging task has run - replace logged IDs in filter */
      if(timeDelta >= LOGGING_TASK_PERIOD_us){

         for(i = 0; i < filterSize; i++){
         /* loggedFlag is set when ID is logged for first time */
            if(acceptanceFilter[i].loggedFlag == TRUE){
               if(updateFilter(i)==TRUE){
                  printf("filter ID: %03X\n", acceptanceFilter[i].canID);
               }
               else{
                  printf("updateFilter FAIL\n");
               }
            }
         }

         timeOrigin = timeNow_us;
      }

      /* ID is in logging list */
      if(GetCAN1BufferPointer(ID) == TRUE){
         IDlogged = FALSE;
         i = 0;

         /* look for ID in acceptance filter */
         do{
            if(acceptanceFilter[i].canID == ID){
               /* ID found, increment counters */
               IDLogCount++;
               loggingSequence[acceptanceFilter[i].sequencePointer].loggedCounter++;
               acceptanceFilter[i].loggedFlag = TRUE;

               IDlogged = TRUE;
            }
            i++;

         }while((IDlogged == FALSE) && (i < filterSize));

         /* ID not found in filter, so would be missed */
         if(IDlogged == FALSE){
            IDMissedCount++;
         }
      }
   }
}
```

```
   printf("filterSize: %u   Logged: %u    Missed %u\n", filterSize, IDLogCount, IDMissedCount);

   /* Use for full sweep output */
   for(i = 0; i < BUFFERSIZE; i++){
      if(loggingSequence[i].canID != 0){
         fprintf(log,",,0x%03X,%lu,%lu,%lu\n",loggingSequence[i].canID, loggingSequence[i].loggedCounter, \
                (loggingSequence[i].counter - loggingSequence[i].loggedCounter), loggingSequence[i].counter);
      }
   }
   fprintf(log,"\n");

   for(i = 0; i < BUFFERSIZE; i++){
      if(loggingSequence[i].canID != 0){
         printf(",,0x%03X,%lu,%lu,%lu\n",loggingSequence[i].canID, loggingSequence[i].loggedCounter, \
               (loggingSequence[i].counter - loggingSequence[i].loggedCounter), loggingSequence[i].counter);
      }
   }
   printf("\n");

/* Use for single filtersize output */
// fprintf(log,",,%u,%u,%u\n", filterSize, IDLogCount, IDMissedCount);

}

/* Counts the total number of messages per ID in the trace. */
void CanSequenceMessageCounter(char *filename){
   char inputStr[200];
   char canData[200];
   int i = 0;
   flag_t IDfound = FALSE;
   unsigned long timeNow_s ,timeNow_us, lineCounter;

   int ID;
   printf("Reading CAN log...\r\n");

   /* open trace file */
   FILE *bufferFile = fopen(filename, "r");

   while((fgets(inputStr, 190, bufferFile) != NULL) && ((lineCounter++ < MAX_TRACE_LINES) || (MAX_TRACE_LINES == 0))){
      /* Extract values from input string */
      unsigned int scanReturn = sscanf(inputStr, logFormat, &timeNow_s, &timeNow_us, &ID, &canData);

      if(scanReturn == 4){

         printf("%u Counting ID's... Log line: %s", scanReturn, inputStr);
```

```
         if(GetCAN1BufferPointer(ID) == TRUE){
            i = 0;
            IDfound = FALSE;

            do{
               if(loggingSequence[i].canID == ID){
                  loggingSequence[i].counter++;
               }

               if(loggingSequence[i].canID != 0){
                  i++;
               }

            }while((loggingSequence[i].canID != 0) && (i < listSize) && (IDfound == FALSE));
         }
      }
   }

   printf("Finished sequence:\n");

   i = 0;

   while((loggingSequence[i].canID != 0) && (i < BUFFERSIZE)){
      printf("0x%03X: %u\n",loggingSequence[i].canID,loggingSequence[i].counter);
      i++;
   }
}


/* Orders sequence by CAN ID */
void orderSequence(void){
   unsigned int i, j, minIDPrev = 0, minID = 0xFFFF, minIDPointer;
   logging_Sequence_t orderedSequence[BUFFERSIZE];

   for (i = 0; i < BUFFERSIZE; i++){
      for(j = 0; j < BUFFERSIZE; j++){
         if((loggingSequence[j].canID < minID) && (loggingSequence[j].canID > minIDPrev)){
            minID = loggingSequence[j].canID;
            minIDPointer = j;
         }
      }

      orderedSequence[i].canID = minID;
      orderedSequence[i].counter = loggingSequence[minIDPointer].counter;
      minIDPrev = minID;
      minID = 0xFFFF;
   }
```

```
   printf("\n\n");

   for(i = 0; i < BUFFERSIZE; i++){
      if(orderedSequence[i].canID == 0xFFFF){
         loggingSequence[i].canID = 0x000;
         loggingSequence[i].counter = 0;
      }
      else{
         loggingSequence[i].canID = orderedSequence[i].canID;
         loggingSequence[i].counter = orderedSequence[i].counter;

      }
      printf("0x%X\n",loggingSequence[i].canID);
   }
}


/* Counts number of ID's in sequence */
int countSequence(void){
   int i, counter = 0;

   printf("\n\n");
   for(i = 0; i < BUFFERSIZE; i++){
      if(loggingSequence[i].canID != 0x000){
         counter++;
      }
   }
   return counter;
}


/* Checks if ID is in logging list */
flag_t GetCAN1BufferPointer(unsigned int ID){
   int i;
   flag_t IDfound = FALSE;

   for(i=0;i<listSize;i++){
      if(ID == loggingList[i].canID){
         IDfound = TRUE;
      }
   }
   return IDfound;
}
```