Using a Configurable, Dynamic Acceptance Filter to Log Sequential CAN Data from a Time-Triggered Hybrid Scheduler

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

TODO…

Abstract

TODO…

Table of Contents

List of Tables and Figures

1 Introduction

Controller Area Network (CAN) has become a standard method of communication between embedded devices in automotive applications [1]. CAN Messages contain data transferred between nodes on a bus network. Each message is given a unique identifier (ID) to provide context to the content of the message. Nodes on the CAN bus use CAN controller hardware to buffer messages that have been transmitted by other nodes. The inherent nature of a 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 message in order to decide whether it needs to read the content. This interrogation can be carried out in software, by comparing the identifier to a table containing those to be accepted, or by using 'acceptance filters' in hardware to restrict the identifiers allowed into the CAN controller's buffer.

Both methods have drawbacks. If acceptance filters are used, the number of filters (or 'mailboxes') available within the CAN controller hardware limits the number of messages that the node can accept. Therefore, if a node has interest in a large subset of messages on a CAN bus, it the solution has to involve software filtering. Interrogating the identifier in software uses up space in the buffer of the CAN controller 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.

Limitations also become apparent 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, and so mechanisms have to be put in place to allow configuration in the field.

This project focuses on the development of a novel approach to these problems whereby CAN message identifiers to be accepted by real-time embedded system are specified in a 'logging list' along with known order / timing properties. The logging list is transmitted to the embedded system from a remote configuration application and used to produce filter configuration sequences.

Using a time-triggered architecture, these configuration sequences are used to predict the IDs of the next messages on the CAN bus at any given 'tick'. A periodic task uses this prediction to modify the CAN controller acceptance filters to accept only the IDs of the expected messages on the bus for any given time.

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 resources, which are covered in modules B2 and B3. Some monitoring and instrumentation techniques learned from module B2 are also applied.

2 Background

2.1 Project Theme

An electric commercial vehicle company uses a telemetry device to log data from a multi-bus CAN network on their vehicles. Data are transmitted over the mobile phone network using GPRS to an AMQP message queue, where a dedicated server performs the necessary post-processing to store the information in a database.

New hardware and software requirements are now being explored for an updated device, which include the ability to modify remotely the CAN messages that are 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 the use of a TT scheduler should allow performance guarantees to be made to the company , the logging of data events using a TT scheduler is not without its challenges, which will be addressed in this and later chapters.

2.2 Real-Time and Time-Triggered Software Architecture

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

Time-triggered (TT) architecture is method of guaranteeing when a software operation should run. It is predominantly used for safety-critical embedded systems where it is imperative that operations are performed on time with an accuracy measured in fractions of microseconds [2].

The backbone of this architecture is a scheduler driven by a single event; a timer-driven interrupt. This interrupt is used to generate periodic 'ticks' that allow the scheduler to keep track of time. Software operations are divided into 'tasks'. Hard-coded properties control the timing of the tasks using an 'offset' (time until first dispatch) and 'period' (time between subsequent dispatches). Figure 2-1 below shows the behaviour of a 'Time Triggered Co-operative' scheduler. Each tick executes the necessary tasks. Task A is given a higher priority than Task B, so Task A is executed first when the two tasks share the same 'tick'.

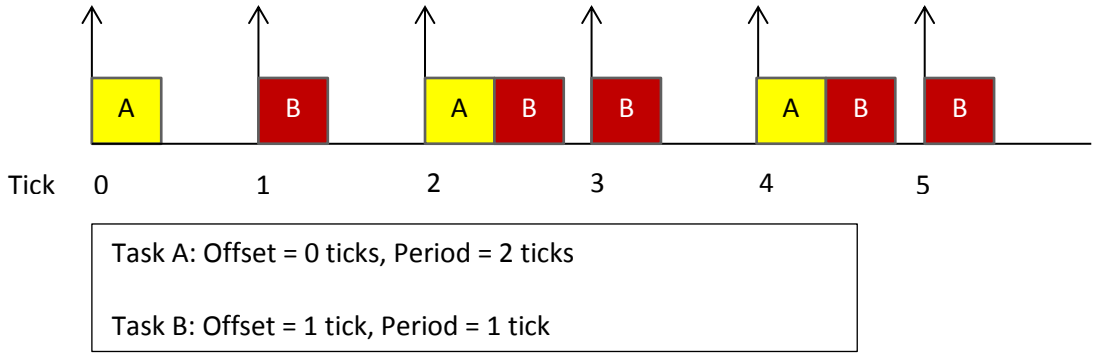

FIGURE 2-1: A SIMPLE TIME-TRIGGERED CO-OPERATIVE (TTC) SCHEDULER

Figure 2-2 shows a 'Time Triggered Hybrid' scheduler. Here, Task B takes longer than one tick to complete, however Task A is configured to execute from the Interrupt Service Routine (ISR). This means that Task A is guaranteed to run on time, and Task B will be suspended until Task A completes [3].

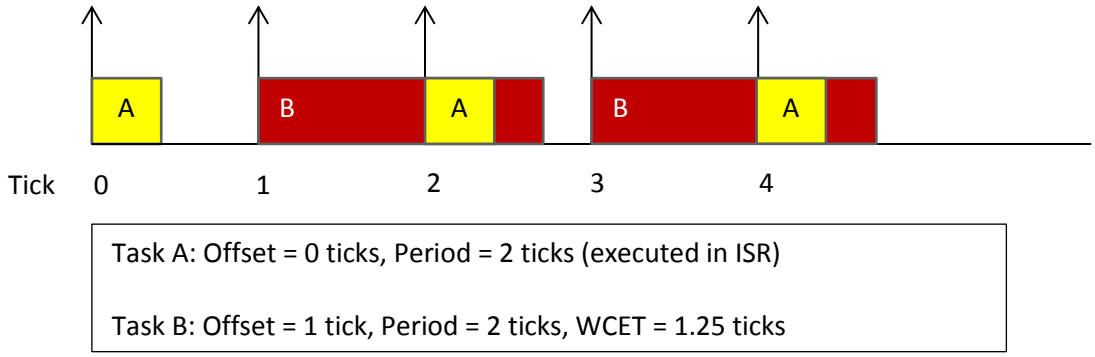

FIGURE 2-2: A SIMPLE TIME-TRIGGERED HYBRID (TTH) SCHEDULER

One rule that guarantees accurate timing in a TT system is that only one interrupt is allowed per CPU, which is the timer interrupt. This means that the software must poll peripherals in order to detect any external events such as GPIO state changes and data reception. This ensures that unexpected 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. Software driven by more than one interrupt or event is known as 'Event Trigered'.

2.3 Controller Area Network

Controller Area Network (CAN) is a standard for serial data communications over a 2-wire bus. Bosch's CAN Specification 2.0 [4] describes the Physical and MAC layers and part of the LLC layers of the OSI reference model [5]. 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. This project will be working within the realms of the Acceptance Filtering aspect of the LLC layer, and the Application Layer.

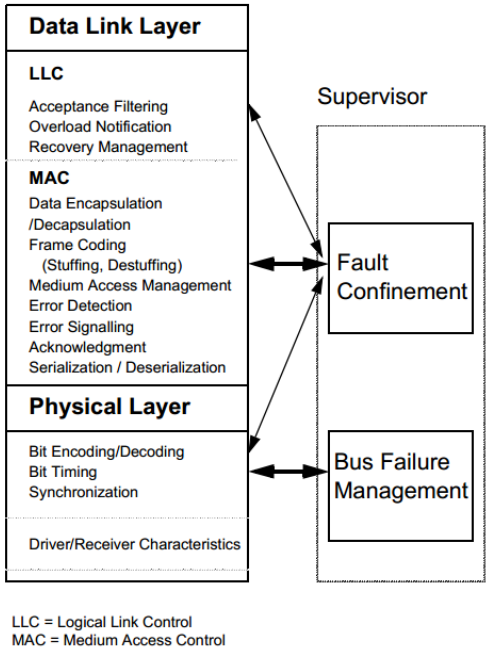

FIGURE 2-3: LAYERED ARCHITECTURE OF CAN ACCORDING TO THE OSI REFERENCE MODE [4]

2.3.1 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 the microcontroller [6]. The function of the CAN controller is to transmit data to the other nodes on the CAN bus, implementing the CAN specification. The CAN controller is also responsible for receiving data transmitted by the other nodes and storing it for the retrieval of the host microcontroller.

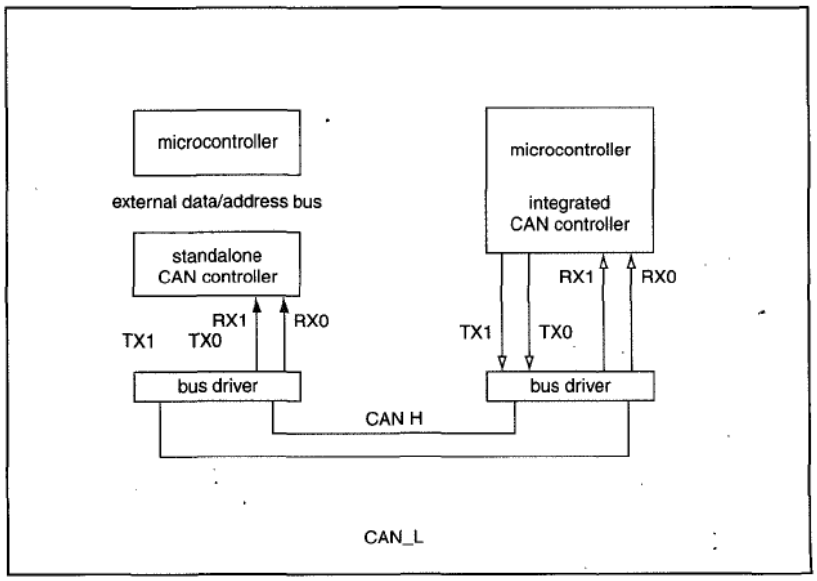

FIGURE 2-4: VARIANTS OF CAN HARDWARE [6]

2.3.2 Acceptance Filtering

The control of CAN message reception is classified as either 'basic CAN' or 'full CAN' [6]. 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.

- Include table of manufacturers, FIFO depths and acceptance filter sizes.

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 are integrated into the microcontroller silicone. This has the advantage that CAN messages can be stored directly in Direct Memory Access (DMA) registers, allowing for much faster retrieval of data [7]. This advance has brought with it more sophisticated methods of handling CAN messages.

The STM32F407ZGT6 from STMicroelectronics provides 28 filter banks, each capable of holding four 16-bit Identifiers [7]:

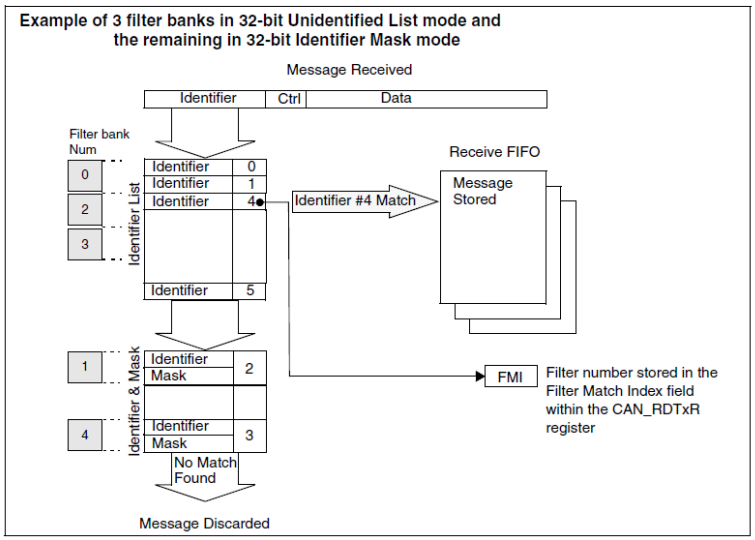

FIGURE 2-5: CAN ACCEPTANCE FILTERING ON AN STM STM32F407ZGT6 PROCESSOR [7]

As a message arrives on the CAN bus, the processor transparently compares the identifier with those in the filter lists, and stores it in a specific memory location. If the message doesn't match any of the filters, it is discarded. A 'Filter Match Index' registry field to store the acceptance filter that each FIFO entry matched. This gives the software visibility of the message context without needing to interrogate the identifier.

In 'full CAN', the hardware presents dedicated areas of memory, which are configured to receive or transmit messages with preset identifiers. An example of this is the eCAN arrangement present Texas Instruments C2000 family processors [8].

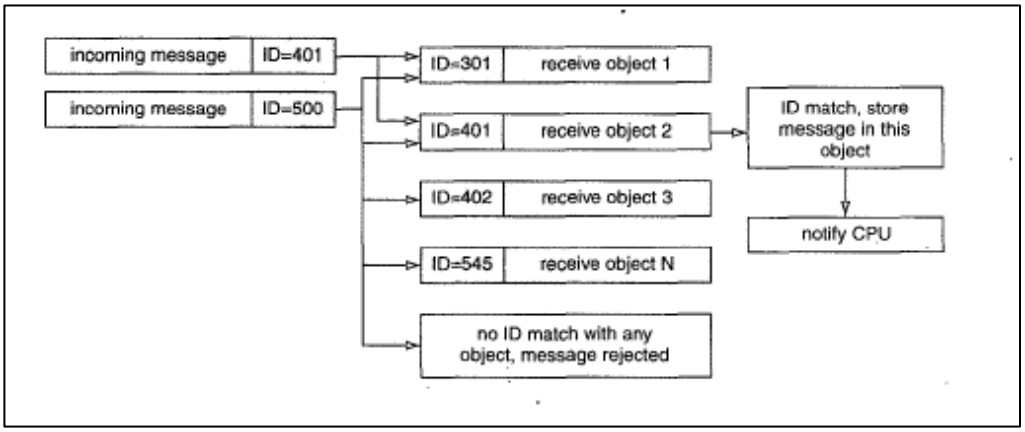

FIGURE 2-6: PRINCIPLES OF 'FULL CAN' OPERATION [6]

Instead of storing all CAN messages in one FIFO and leaving it up to the microcontroller to determine the context of the message, the CAN controller stores the message in a specific area of memory depending on its identifier. This now means that when the CAN controller indicates a message arrival, using either an interrupt or a flag, the microcontroller can read the data directly from the mailbox and knows the context without using any look-up mechanisms in software.

2.4 Using CAN with TT Architectures

As mentioned in 2.2, 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 other than to poll the CAN controller periodically to detect CAN messages. Using a 'basic CAN' controller, depending on the configuration of the hardware, messages arriving when the buffer is full will either be discarded, or replace a message already in the buffer. The worst case timing between messages for different baud rates can be seen in Table 2-1:

TABLE 2-1: WORST CASE INTER-FRAME SPACING [6]

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:

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

Where $t_{polling}$ is the polling period, t_S is the worst-case Inter-frame space (from Table 2-1) and D_{buffer} is the depth of the CAN controller's receive buffer.

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 architecture in that 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 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. 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.

3 Related Work

The use of CAN as a communications medium in real-time embedded systems has been a topic of research for many years. Several publications address the subject of CAN-related scheduling, although this research is predominately focussed on message transmission, as opposed to message reception.

The operation of CAN bus networking and its integration into embedded systems is introduced by Farsi et al in [6]. The distinction between 'full CAN' and 'basic CAN' is presented, and merits of systems that allow the CAN controller hardware to perform acceptance processing are offered. 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 system using the protocol.

In [9] and [10], 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" [11] 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 the system to be reconfigured in time for the next time slot. [12] Expands on this protocol and discusses the technicalities of changing the variable set in real-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 [13] and CAN message scheduling for time-critical control systems is discussed by Martí et al in [14].

Pop et al discuss, in [15], 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. The research in this project is novel in that it approaches the problems specifically surrounding the reception of CAN messages using COTS hardware when there is no control over the transmitted message set, which is unknown at compile-time.

The incoming data stream on the CAN bus could 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.

In [16], 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 [17]. The replacement policies described for caches, however, generally keep addresses alive if they achieve a hit. The proposed system needs to work in reverse as it assumes that a hit for an identifier indicates that there won't be another like hit until the next message cycle.

4 Automotive Telemetry System

4.1 Overview

This chapter will discuss an existing CAN data logging system used by an electric commercial vehicle company to gather and transmit vehicle diagnostics data to a data centre. The behaviour of this system will be used as a benchmark for 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 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.

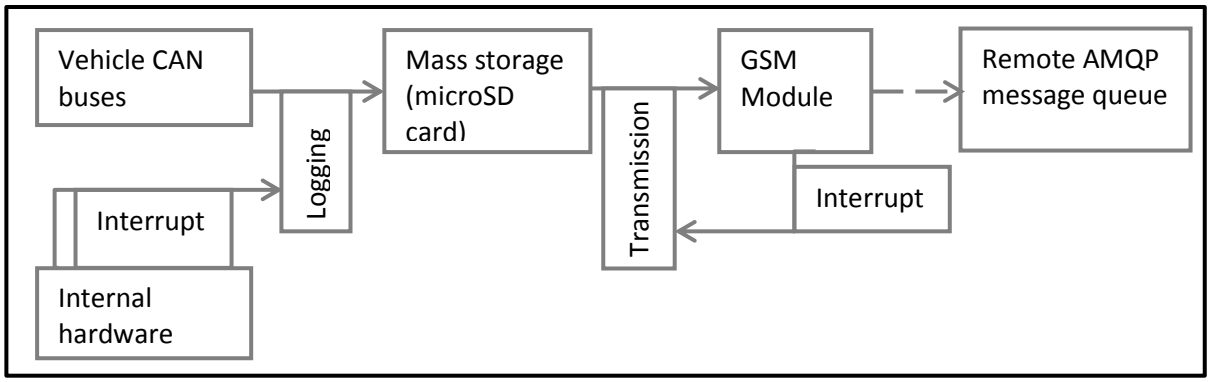

FIGURE 4-1: EXISTING REMOTE DEVICE SOFTWARE ARCHITECTURE

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 an interrupt on the GSM module, indicating that the module is connected to the network, and a looped polling function which 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 on timer interrupts, allowing the device to still collect data during periods of low or no GPRS signal.

4.2.3 Interrupts

Interrupts are used for the following software operations:

- **Interrupt events**

  There are several hardware events that generate interrupts to the software. These include Analogue to Digital Conversion (ADC), four RS232 channels, a Global Positioning System (GPS) module and a General Packet Radio Service module.

- **Time-released functions**

  These functions are driven by a timer interrupt operating at a 1 kHz frequency. The 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. Because the device uses a Microchip MCP2515 CAN controller that only supports 'basic CAN' [18], the incoming data is filtered and stored by the software. This process is shown in Figure 4-2.

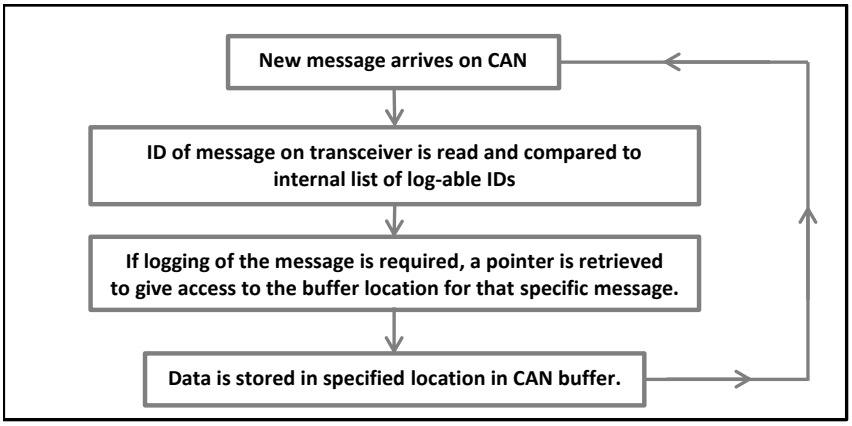

FIGURE 4-2: SOFTWARE FILTERING IN THE CAN DATA LOGGING PROCESS

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.

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. The values, separated by commas, could then be copied from a terminal program for further analysis. This method was found to be the least expensive way of reporting the hit rates of each CAN ID.

5 Dynamic CAN Filtering

5.1 Overview

The proposed system addresses a number of problems mentioned in Chapter 2. It is designed to run on a TI C2000 processor, which incorporates a 'standard CAN' controller, referred to by the manufacturer as 'eCAN' [8]. The filter acts as an extension to the mailbox system that is suited to periodic polling. The system takes advantage of the hardware rejection of unwanted identifiers, but provides a level of abstraction from the eCAN, removing the limitation that the 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 post-compilation, allowing for a level of remote configuration.

The system is provided a 'logging list' which comprises a description 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. The software continually reads the incoming data from the CAN mailboxes and copies them into RAM. As messages arrive in the mailboxes, the acceptance filter for that mailbox is updated with a new identifier, read from 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. The source of the logging list is flexible, and can be provided to the embedded system by a remote server.

An iterative development process was used to ensure that the resultant algorithm was as refined as possible producing the following applications:

- **Feasibility simulation**
- **Embedded software**
- **Remote Configuration and Analysis Tool (RCAT)**

The development process involved first producing the Feasibility Simulation. Lessons learned from initial testing were fed into the parallel development of the Embedded Software and Configuration / Analysis application. Testing on 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 more detail in Chapter 7 below.

5.2 Feasibility Simulation

A simulation application was written in 'C' that reads through a 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. This allowed 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, would read all logged messages from the CAN mailboxes, and update the acceptance filters.

- The period of this simulated task is controlled by "LOGGING_TASK_PERIOD_us".
- The number of identifiers in the acceptance filter (represented by the array, "acceptanceFilter[]") is configurable with the argument, "filterSize".
- At initialisation, the acceptanceFilter[] array is loaded with the CAN identifier values from the top of the loggingSequence[]
- A variable, "sequencePointer", is used to keep track of the location in loggingSequence[].

5.2.1 Identifier sequencing

The simulation program is capable of producing two types of sequence from the CAN trace; the sequence can be ordered numerically by identifier, or it can be built in the order in which the identifiers first appear in the trace file. These options are intended to provide an understanding of the affect the order of the sequence has on the performance of the system.

5.2.2 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 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 (see below). If the identifier is in the logging sequence, but not present in the acceptance filter, the identifier has been 'missed' and "IDMissedCount" is incremented.

5.2.3 Simulated Time Triggered task

In order to achieve a realistic simulation of the system, it is important that application behave as similar as possible to that of a periodic task running from a time-triggered scheduler. To accomplish this, the simulation exploits the timestamps recorded next to each message in the CAN trace. This provides a microsecond-accurate indication of the time that elapsed between the recorded CAN messages, allowing the simulation to determine when the logging task would run. When the elapsed time, indicated by the CAN trace timestamps, exceeds LOGGING_TASK_PERIOD_us, the acceptance filter is scanned and each identifier has been marked as 'logged' is replaced by the next identifier in the logging sequence that isn't already present in the acceptance filter. This approach more closely simulates the periodic execution of the time-triggered task, and provides an understanding of the limitations caused by this behaviour.

5.3 Embedded Software Implementation

The embedded software implementation involved porting the code from the feasibility simulation to tasks running in a TTH scheduler. The software includes a CAN mailbox handler developed specifically for periodic polling of the mailboxes, which are accessed through the Texas Instruments eCAN library [8]. The operation of the embedded system can be modelled as a simple state machine:

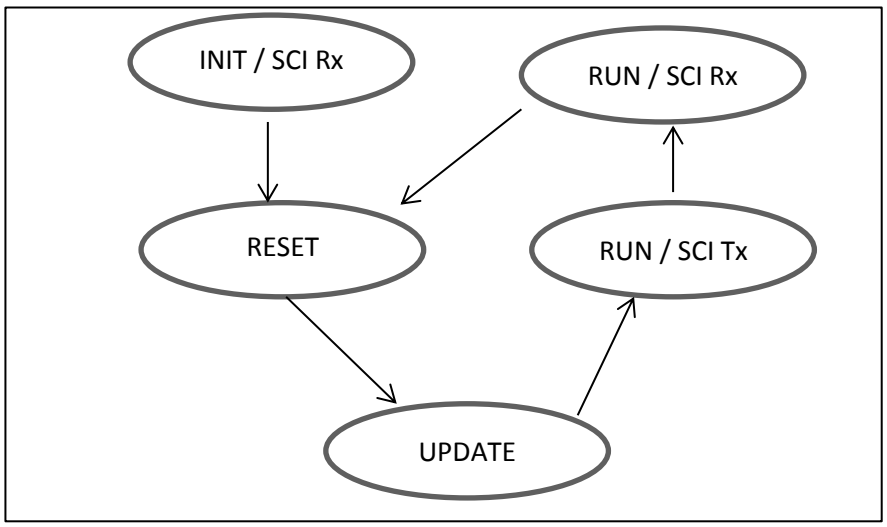

FIGURE 5-1: SIMPLIFIED EMBEDDED SOFTWARE STATE MACHINE

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 5.2.2. 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.

There are various differences and additions to the logic in the embedded software compared to the Feasibility Simulation:

- The sequence is built from the logging list received from the external Remote Configuration application (see 5.4).
- 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 7.1 below).
- A configuration / analysis task runs alongside the filter update task. This task is responsible for communicating with the Remote Configuration and Analysis Tool (RCAT) over an RS232 connection (see below).
- 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.

5.4 Remote Configuration and Analysis Tool

In order to satisfy the 'remote configuration' aspect of the project, an application was written in 'Processing' programming language [19]. 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.

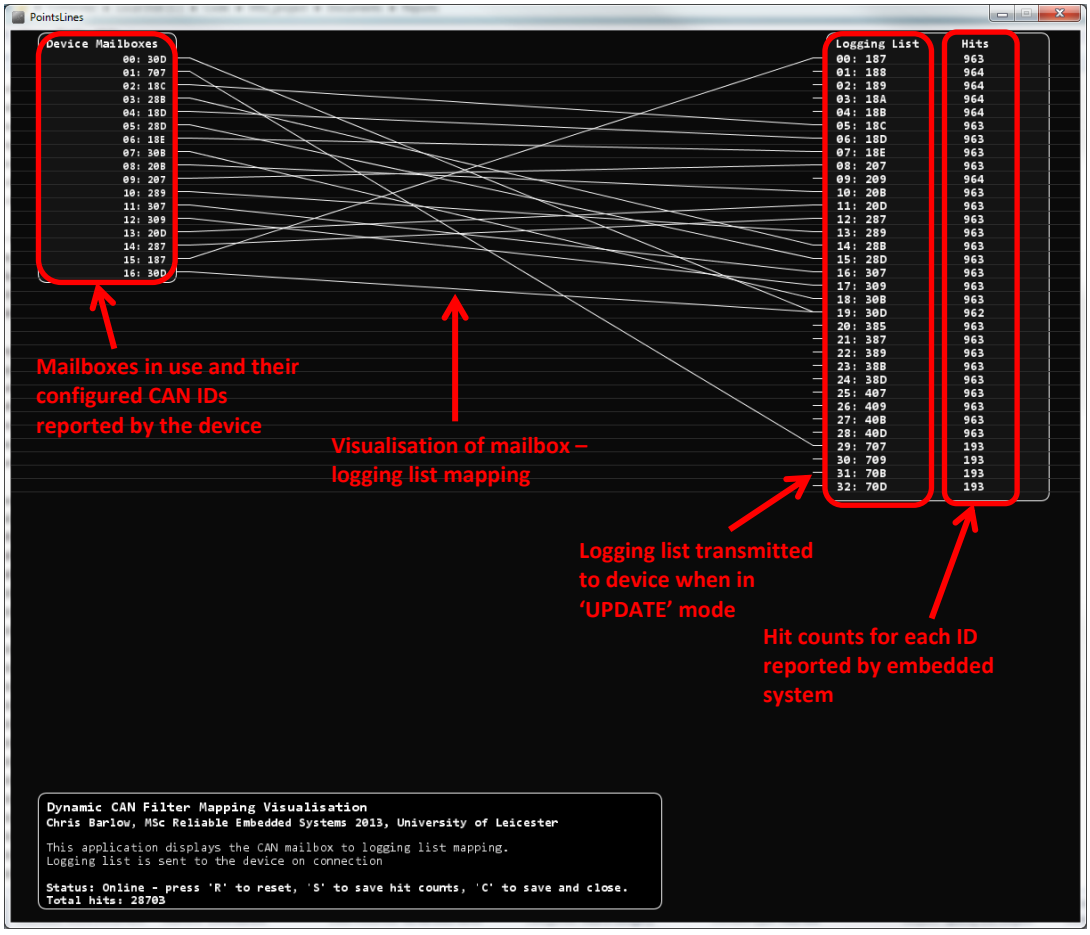

FIGURE 5-2: VISUAL FEEDBACK AND HIT RATE ANALYSIS FROM RCAT

6 Analysis

6.1 Questions

Testing of the feasibility simulation set out to answer the following questions:

- What is the optimum size for the acceptance filter for a given logging list?
- What is the relationship between the optimum acceptance filter size and the logging list size?
- How does the order of the identifiers in the logging sequence affect the hit rate of the algorithm?
- How does the logging task period affect the hit rate?
- 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?

6.2 Method

6.2.1 Simulation

The simulation was executed on a sample CAN trace recorded from the battery management system of an electric commercial vehicle.

The simulation was executed in 'full sweep' mode, which repeats the analysis for different filter sizes, varying from filterSize = 1 to filterSize = loggingListSize.

Tool was configured to build the sequence in the order in which they identifiers first appear in the trace.

The simulation outputs to a comma-delimited log file the hit and miss rates 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. This test runs the simulation again, but this time records the hit and miss rate per identifier for the given filter size.

In order to gauge the impact of changing the order of identifiers in the sequence, the test was repeated with the sequence built in numerical order by identifier.

6.2.2 Hardware

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:

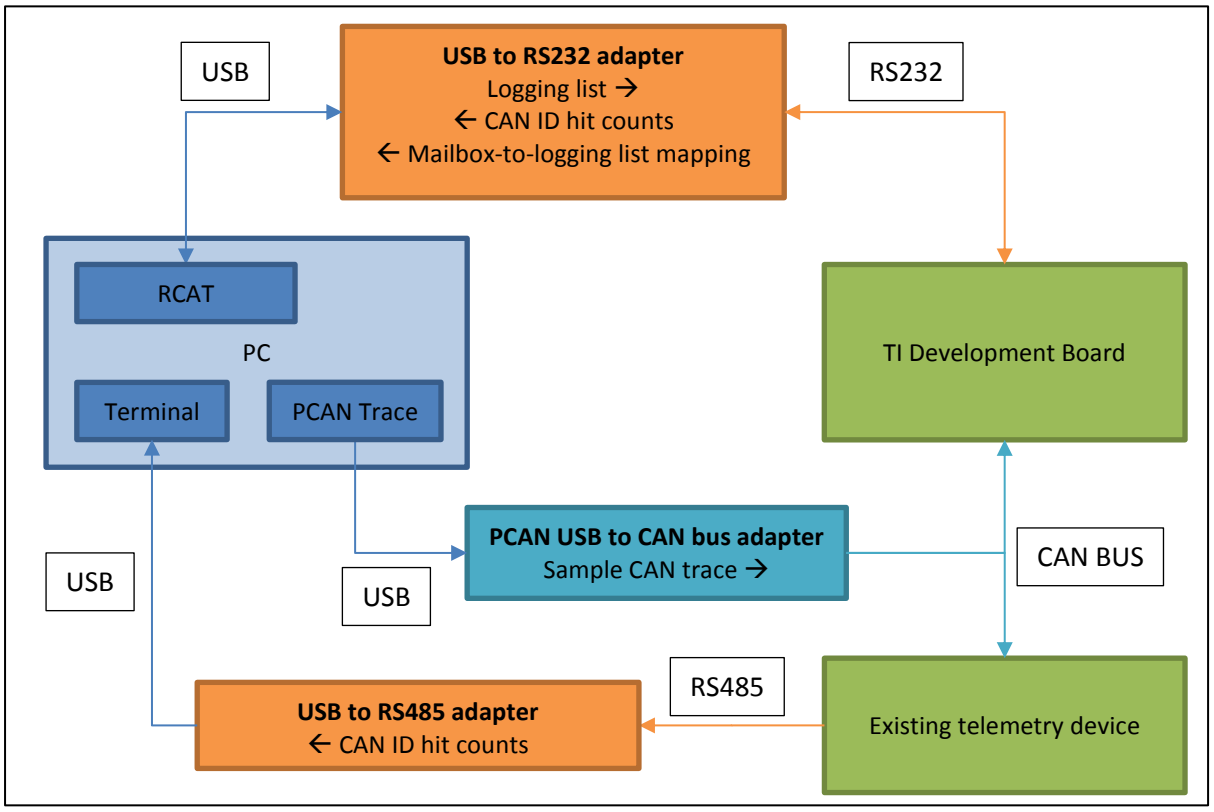

FIGURE 6-1: HARDWARE TEST SETUP

The sample CAN trace used in 6.2.1 above was transmitted to the CAN bus in real-time using 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 for the target were recorded by the RCAT and those for the existing device were recorded using a standard terminal program.

The test was repeated in order to gauge the consistency and predictability of the hit counts.

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. 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.

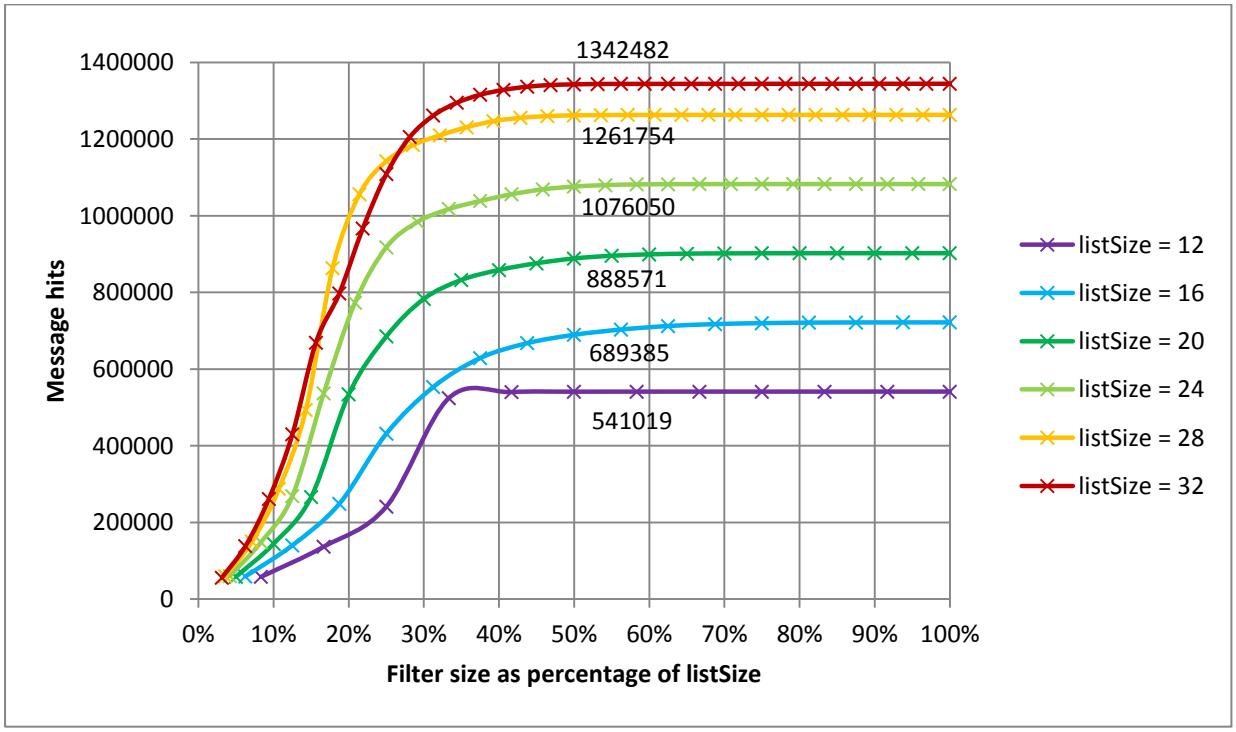

FIGURE 6-2: MESSAGE HITS VS FILTER SIZE FOR VARYING LIST SIZES

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.

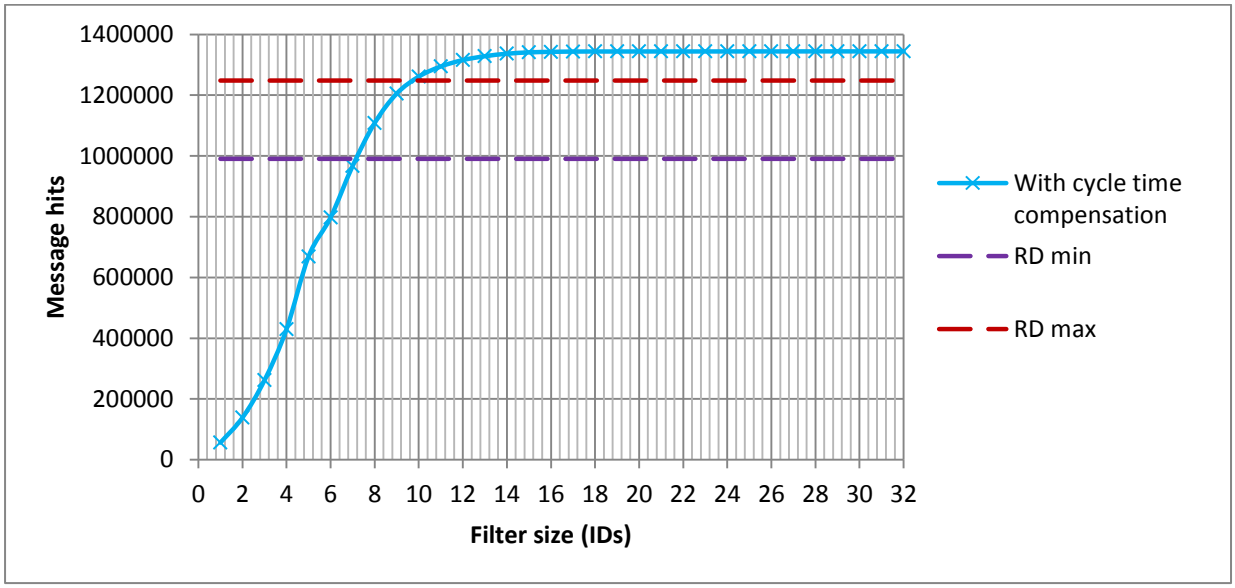

FIGURE 6-3: HIT RATE VS FILTER SIZE FOR 32-ID LOGGING LIST

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.

6.3.3 Comparing the Simulation and Hardware Implementation

Table 6-1 shows a comparison between the hit rate predicted by the Simulation tool and the Target Hardware. It can be seen that the Simulation predicted the performance to within 0.004% of the embedded hardware.

TABLE 6-1: COMPARISSON BETWEEN SIMULATION AND HARDWARE FOR 2-SEGMENT FILTER

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

This indicates that the Simulation would be a useful pre-implementation tool to allow guarantees to be made about the 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 system can be seen in Figure 6-4. Here it can be seen that the hit rates for the new system are more consistent across the whole range of identifiers. This indicates a marked improvement in logging performance over the existing device.

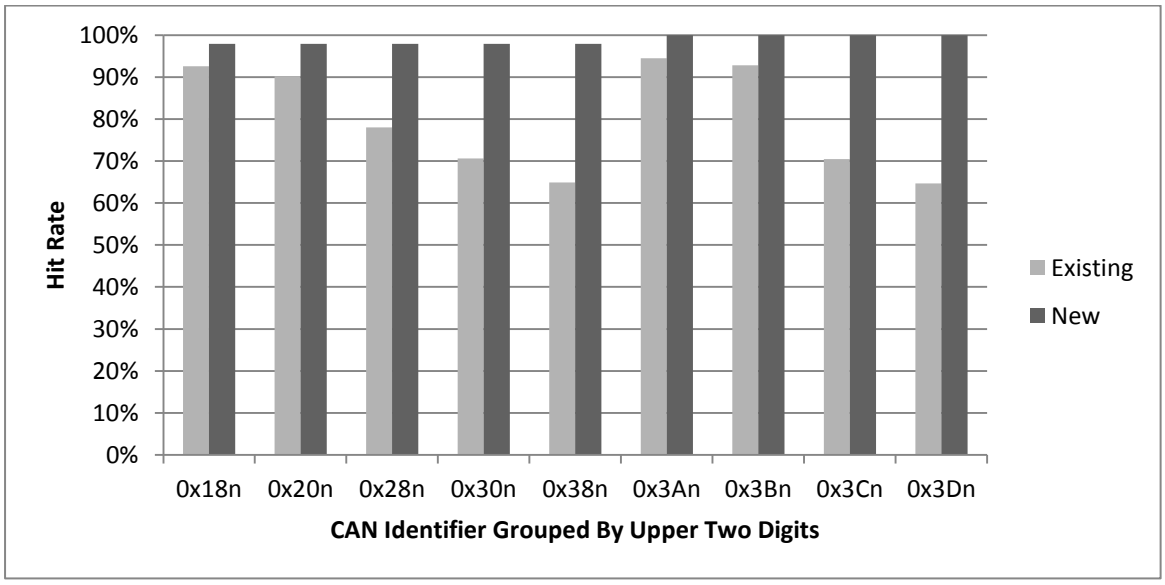

FIGURE 6-4: HIT RATES FOR IDENTIFIERS GROUPED BY UPPER TWO DIGITS

These questions can't be answered with my current data, need more testing to include them

- How does the logging task period affect the hit rate?

- How does the order of the identifiers in the logging sequence affect the hit rate of the algorithm?

7 Discussion

The previous chapters have addressed the testing of the final system. This chapter will discuss some of the factors influencing the development and behaviour of the system.

7.1 Duplication in filter

One enhancement that was a product of the development process was the concept of controlled duplication in the filter. It was assumed at the start 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 outside 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. With no duplication allowed in the filter, at iteration 13, 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.

Iteration	Arrival	Logging Sequence										
		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	
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+6	7	8	9	ID 1 arrives, duplicate added for ID 6
12	2	0	1	2	3	4	5	6+6	7	8	9	
13	6	0	1	2	3	4	5	6	7	8	9	ID 6 arrives late, duplicate replaced by 9
14	3	0	1	2	3	4	5	6	7	8	9	
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

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.

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%**

The effects of this algorithm with a real CAN trace are shown in Table 7-1 above. 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.

7.2 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.

7.2.1 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 when there were fewer long-cycle identifiers than short-cycle ones as the hit rates became more consistent between identifiers.

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

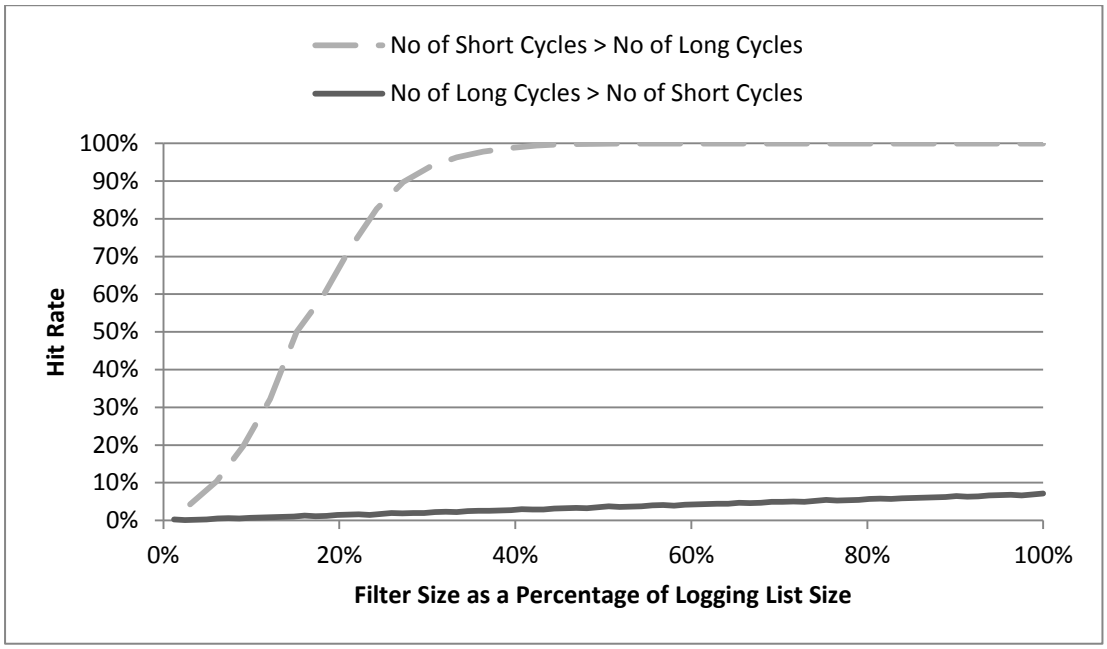

FIGURE 7-3: EFFECT OF CYCLE TIME BALANCE ON HIT RATE FOR SINGLE-SEGMENT FILTER

7.2.2 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.

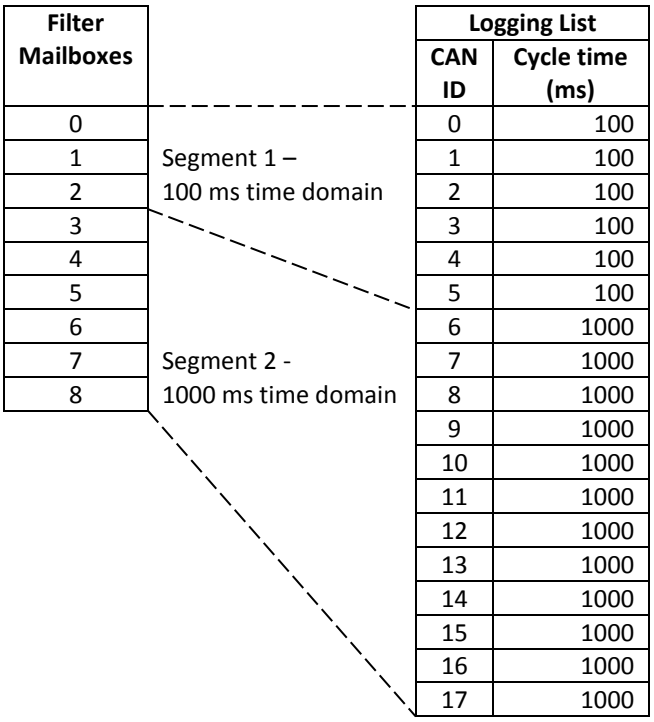

FIGURE 7-4: FILTER SEGMENTATION FOR DIFFERENT MESSAGE CYCLE TIMES

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 rate using the same CAN trace for both single-segment and multi-segment filters.

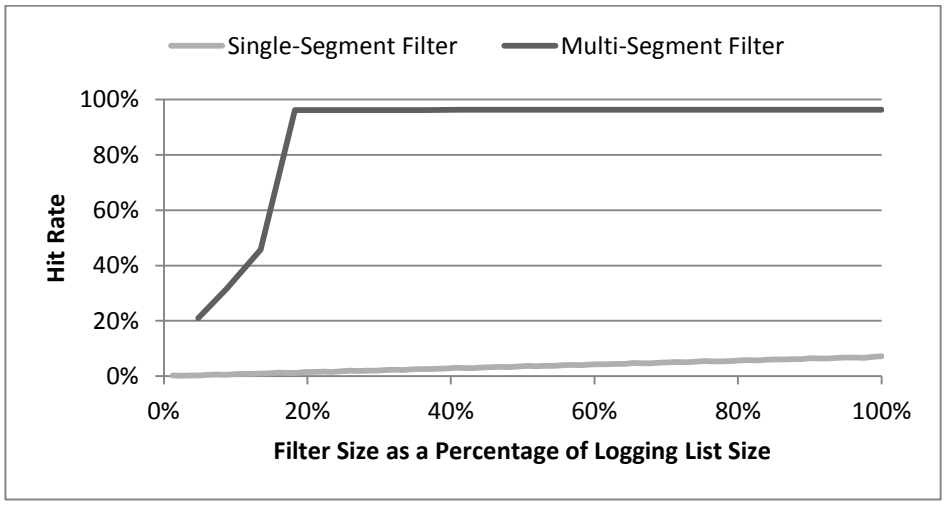

FIGURE 7-5: HIT RATE FOR SEGMENTED FILTER COMPARED TO SINGLE-SEGMENT FILTER

7.3 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 [4]. Moreover, these messages will arrive in bursts with inter-frame spacing dependent on the configured baud rate of the network.

From Table 2-1 above, it can be seen 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 $t_{polling} = t_{\mathrm{s}} \times D_{buffer}$ (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.

7.4 Task Execution Time

The execution time of the filtering task was measured by instrumenting the scheduler around that particular task. A timer was started just before the task was executed, and read just after the task finished. The timer indicated a Worst Case Exectution Time (WCET) of around 10000 CPU cycles. With the CPU running at 150 MHz, this equates to 66.7 μs. With the system running with 1500 μs ticks, this means the maximum processor utilisation of the filtering mechanism is 4.45 %.

This means that there is scope for the remainder of the processing time (minus scheduler overhead) to be used for a target application. The example in this project used this time to communicate with the remote configuration tool, however 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.

7.5 Limitations and Future Development

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 seen suggested in some tests that this could be less 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 sequencing 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 given the remainder. 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, ordered by identifier.

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.

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.

8 Conclusions

This project has presented a novel way to filter data reception 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 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 with a more consistent succession can. Moreover, the data set to be filtered can be configured by a remote system, allowing flexibility to the design, and allowing post-compilation changes to be made to the filter.

The project also shows that, due to the highly predictable nature of Time-Triggered architecture, a desktop simulation application can be built. This application can 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 that can be executed in a desktop environment that allows guarantees to be made to the customer prior to integration of the embedded system.

The short 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] 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.

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

[5] 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.

[6] 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.

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

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

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

[20] 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.

[21] 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 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].dup
licates = 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;
		}

/* ID not already in mailbox, decrement 'schedule' timer (timer sits between -
DUPLICATES ALLOWED and 0 while ID is in one or more mailboxes) */

		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((filterIndex%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[segmentN
umber].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.rawDa
ta);

				/* Count message hits */

	CAN_RxMessages_G[mailBoxFilterShadow_G[mailBox].sequenceIndex_mapped].counter
++;

/* Unsure whether mailbox decay is helpful. It appears not to make much difference
with the segmentation */
#ifdef DECAY_LOGIC
			}
			else if(mailBoxFilterShadow_G[mailBox].mailboxTimeout > 0){
				mailBoxFilterShadow_G[mailBox].mailboxTimeout--;
			}

			if(mailBoxFilterShadow_G[mailBox].mailboxTimeout == 0){
				mailBoxFilterShadow_G[mailBox].mailboxTimeout =
MAILBOX_DECAY_TIME;
#endif

				/* 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
*/
                            updateSequenceRrequired_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;

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();

}
```

```
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.\nTotal 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();
  }
}
```

```
void transmitLoggingList(){
  int txListPointer;


  /* 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
   * */
  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){
    /* Offline */
    case 0:
    case 4:
      /* App offline */
      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;

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

    /* Transmitting logging list */
    case 2:
      if(readyState==true){
        /* this delay is necesssary for the TI chip to keep up when it receives
erroneous null characters */
        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;
```

```
    /* Online */
    case 3:

      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();
  }
}

/* keyboard controls */
void keyPressed() {
  int k;

  /* Reset */
  if((key == 'r')||(key == 'R')){
     readyState = !readyState;
     println(readyState);
     for(k=0;k<filterSizeRx;k++){
        IDs[k] = 0x000;
        mapLineEnd[k] = standardSpacingY(k,0);
     }

     for (k = 0; k < loggingList.length; k++) {
      counters[k] = 0;
    }
    filterSizeRx = 0;
    redraw();
  }

  /* Save */
  if((key == 's')||(key == 'S')){
    selectOutput("Select file","saveCounters",lastFile);
  }
```

```
  /* Close and save */
  if((key == 'c')||(key == 'C')){
    selectOutput("Select file","saveCounters",lastFile);
    exit(); // Stop the program
  }

  /* Exit without saving */
  if((key == 'x')||(key == 'X')){
    exit(); // Stop the program
  }

  /* save frame image */
  if((key == 'f')||(key == 'F')){
    saveFrame("output/frames####.png");
  }
}


/* Save counters to text file */
void saveCounters(File selection){
  int k;
  String[] lines = new String[loggingList.length+6];

  lines[0] = ("CAN Filter Hit Rates");
  lines[1] = ("Tested on:,"+
day()+"/"+month()+"/"+year()+",at:,"+hour()+":"+minute());
  lines[2] = (",");
  lines[3] = ("CAN ID,Hits");

  for (k = 0; k < loggingList.length; k++) {
      lines[k+4] = (hex(loggingList[k][0],3)+","+counters[k]);
  }

  lines[k+4] = (" ");
  lines[k+5] = ("Total,"+countersTotal);

  if(selection != null){
    saveStrings(selection, lines);
    lastFile = selection;
  }
}

void delay_ms(int ms){
  long lastTime = millis();
  while (millis()-lastTime < ms);
}

int standardSpacingY(int mult, int offset){
  return ((d*mult)+(2*d)+offset);
}

String intToStr_02(int num){
  String returnStrg;

  if(i<10){
    returnStrg = (" 0"+num);
  }
  else{
    returnStrg = (" "+str(num));
  }

  return returnStrg;
}
```

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 if 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++;
//			printf("case(0x%03X)\n",loggingSequence[i].canID);
		}
	}

	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;
}
```