|  |
| --- |
| [Type the company name] |
| A Dynamic CAN Mailbox Extension Algorithm for a Time Triggered Hybrid Scheduler |
| [Type the document subtitle] |
|  |
| Chris Barlow |
| [Pick the date] |

|  |
| --- |
| [Type the abstract of the document here. The abstract is typically a short summary of the contents of the document. Type the abstract of the document here. The abstract is typically a short summary of the contents of the document.] |

# Acknowledgements

# Table of Contents

[Acknowledgements 1](#_Toc370237999)

[Table of Contents 2](#_Toc370238000)

[Table of Figures 4](#_Toc370238001)

[1 Introduction 5](#_Toc370238002)

[2 Background 7](#_Toc370238003)

[2.1 Controller Area Network (CAN) 7](#_Toc370238004)

[2.2 Time-Triggered Architecture 10](#_Toc370238005)

[3 Related Work / Literature Review 14](#_Toc370238006)

[4 Review of Existing Software and Technology 15](#_Toc370238007)

[4.1 Electric Commercial Vehicle Telemetry System 15](#_Toc370238008)

[4.2 Message Storage in the Existing Device 16](#_Toc370238009)

[4.3 CAN Acceptance Filtering 16](#_Toc370238010)

[5 Proposal: Dynamic CAN Filtering 18](#_Toc370238011)

[5.1 Overview 18](#_Toc370238012)

[6 Development and Testing 19](#_Toc370238013)

[6.1 Development Process Overview 19](#_Toc370238014)

[6.2 Feasibility Simulation 19](#_Toc370238015)

[6.3 Hardware Implementation incl. Remote Configuration 21](#_Toc370238016)

[7 Discussion 22](#_Toc370238017)

[7.1 Algorithms 22](#_Toc370238018)

[7.2 Simulation and Hardware Implementation Comparison 22](#_Toc370238019)

[7.3 Behaviour 22](#_Toc370238020)

[7.4 Performance Comparison with Existing System 23](#_Toc370238021)

[7.5 Future Development 23](#_Toc370238022)

[8 Conclusions 24](#_Toc370238023)

[9 Works Cited 25](#_Toc370238024)

[Appendix 26](#_Toc370238025)

# Table of Figures

[Figure 4.1: CAN ACCEPTANCE FILTERING ON AN STM STM32F407ZGT6 PROCESSOR [2] 10](#_Toc370237948)

[Figure 4.3: A simple Time-Triggered Co-operative (TTC) scheduler 12](#_Toc370237949)

[Figure 4.4: A SIMPLE TIME-TRIGGERED Hybrid (TTH) SCHEDULER 12](#_Toc370237950)

[Figure 6.1: EXISTING REMOTE DEVICE SOFTWARE OVERVIEW 15](file:///C:\Code\MScRES\MSc_project\MSc_Documents\Documents\Project_Report.docx#_Toc370237951)

[Figure 6.2: SOFTWARE FILTERING IN THE CAN DATA LOGGING PROCESS – EXISTING REMOTE DEVICE 16](file:///C:\Code\MScRES\MSc_project\MSc_Documents\Documents\Project_Report.docx#_Toc370237952)

# Introduction

* Academic bumf

Controller Area Network (CAN) has become a standard method of communication between embedded devices in automotive applications. CAN Messages contain data to be transferred between sub-systems, which are given unique identifiers to provide context about 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 all nodes can ‘see’ 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 list of ones that should 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 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.

This project focuses on the development of a novel approach to this problem whereby

Identifiers for CAN to be obtained are specified in a ‘logging list’ (a configurable list of CAN IDs that need to be logged) and storing this data in a precise location for retrieval by other software functions. This method will exploit the fact that the order and timing in which individual messages are published over the CAN bus is relatively predictable. The logging list will be collected from the remote server and used along with the known order / timing of CAN bus messages (‘CAN sequence’) to produce a timed ‘logging sequence’.

In a time-triggered environment, we can use this logging sequence to predict the IDs of the next messages on the CAN bus at any given ‘tick’. A periodic task can then modify the bxCAN acceptance filters to only accept the IDs of the expected messages on the bus for any given time.

As the sequence of CAN messages won’t be exactly uniform, we can specify multiple filters in the bxCAN’s identifier list to accept several IDs at the same time. This can then be modified dynamically as messages arrive.

The project draws from the subject of CAN, covered in module B3, Time triggered scheduling, which is a predominant theme in the MSc Reliable embedded systems, in particular the Time-Triggered Hybrid (TTH) scheduler introduced in A2. It also involves the subject of shared resources, which are covered in B2 and B3.

A novel software filtering mechanism is proposed, whereby…. Feasibility of the simulation is tested first through simulation before implementing on the target microcontroller. Comparisons are made between the embedded implementation and the simulation, and with an existing polled-buffer data logging device.

# Background

## Problem area

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 (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 remotely modify 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 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, while still being compatible with 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.

## 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 **Invalid source specified.**.

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 4.3 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 4.4 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.

The one rule that guarantees accurate timing is that only one interrupt is allowed on the CPU, which is the timer interrupt. 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 [5]. Software driven by more than one interrupt or event is known as ‘Event Trigered’.

A

0

1

2

3

4

5

Tick

B

A

B

B

A

B

B

Task A: Offset = 0 ticks, Period = 2 ticks

Task B: Offset = 1 tick, Period = 1 tick

Figure 2.1: A simple Time-Triggered Co-operative (TTC) scheduler

A

0

1

2

3

4

5

Tick

B

A

Task A: Offset = 0 ticks, Period = 2 ticks (executed in ISR)

Task B: Offset = 1 tick, Period = 2 ticks, WCET = 1.25 ticks

B

A

Figure 2.2: A SIMPLE TIME-TRIGGERED Hybrid (TTH) SCHEDULER

## Controller Area Network (CAN)

CAN is a standard for serial data communications over a 2 wire bus. The CAN specification (2.0) describes several aspects of this communications method.

* CAN overview
  + “Enhanced CAN” Mailboxes
  + Standard CAN
    - FIFO Layers
* current options for collecting data
  + Interrupt
  + Polling
  + Link to MSc

### CAN Hardware

The CAN physical layer comprises of two wires, one held low (CAN L) and the other high (CAN H). The transmission of data is dominant 0, therefore to transmit a 0 bit, CAN L is pulled high, and CAN H is pulled low.

* DIAGRAM HERE
* CAN physical layer
* CAN transceiver
* CAN controller

### Acceptance Filtering

Traditionally, a buffer built into the CAN controller hardware has handled arriving messages. This buffer is usually in First In, First Out (FIFO) arrangement, the depth of which varies between hardware manufacturers **Invalid source specified.**. 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 program 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 pollable flag to indicate message arrival to the microcontroller.
* The microcontroller 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 **Invalid source specified.**. This advance has brought with it more sophisticated methods of handling CAN messages.

The STM32F407ZGT6 uses 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.

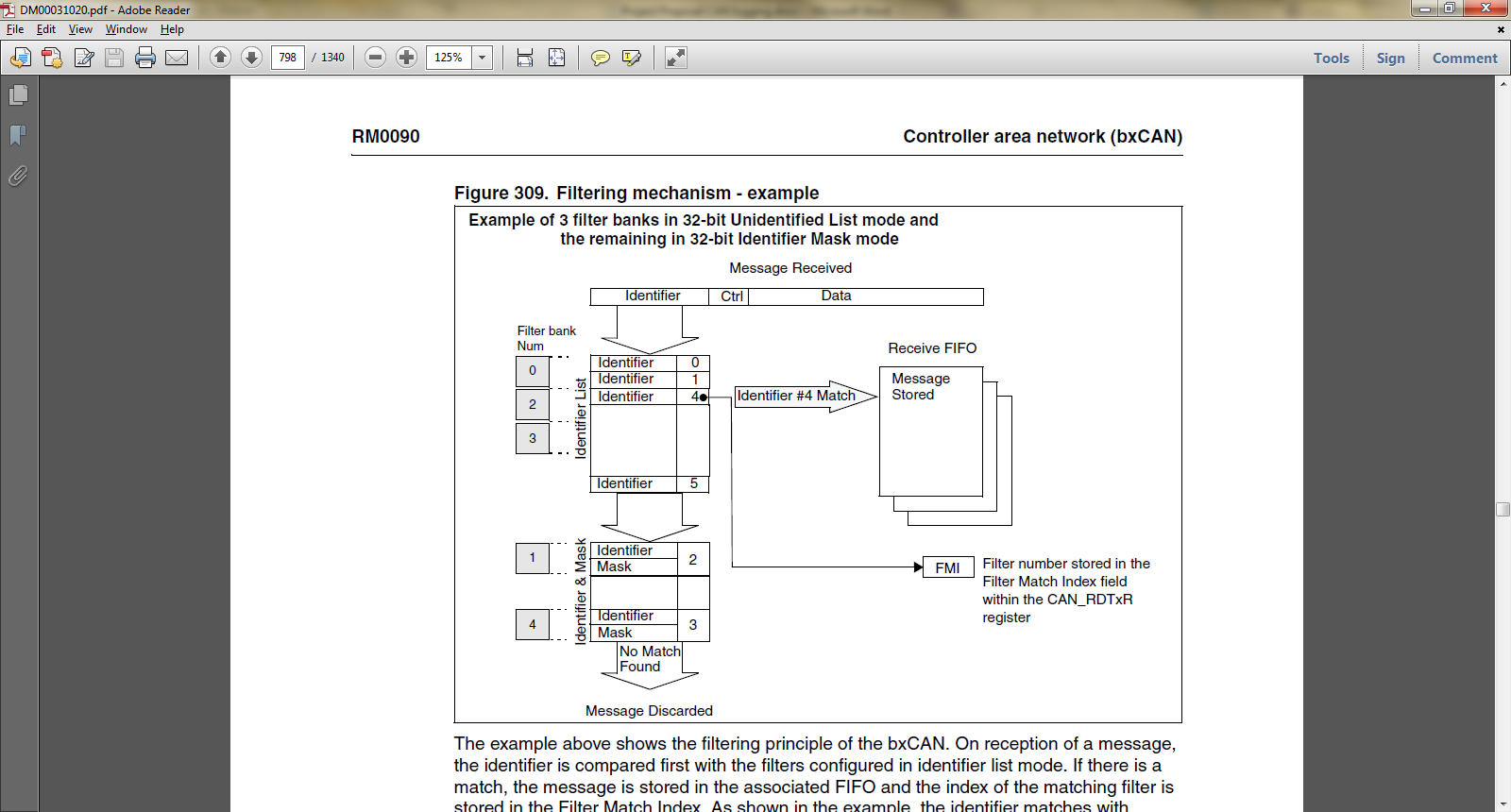


Figure 2.3: CAN ACCEPTANCE FILTERING ON AN STM STM32F407ZGT6 PROCESSOR [2]

Another method is the use of ‘mailboxes’, as used by Texas Instruments [3]. 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.

* CAN Acceptance Filtering
* Plans are to build the new Remote Device around an ARM Cortex M4 processor such as the STM32F407ZGT6. This processor provides 28 filter banks, each capable of holding four 16-bit Identifiers [2]. 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.
* The use of this acceptance filtering feature at hardware level could potentially free up processing time in the device software as there is no need to perform the comparison to check that the message is needed.

## Using CAN with TT Architectures

As mentioned previously, Time-Triggered architecture can only behave predictably if there are no interrupts active other than the timer that drives the scheduler. Depending on the configuration of the buffer, messages arriving when the buffer is full will either be discarded, or replace a message already in the buffer. 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. In order to see every message on the CAN bus, a buffer-based controller must be polled at least as often as:

Where *fpolling*is the polling frequency, *fmax* is the maximum frequency of incoming messages, and *Dbuffer* is the depth of the CAN controller’s receive buffer.

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.

Mailbox-based CAN controllers, on the other hand, are much better suited to TT architecture in that they 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. Unfortunately, this method is only useful for devices interested in a relatively small subset of messages however, 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 the hardware mailbox mechanism.

# Related Work / Literature Review

* Pont’s work on TT systems (obviously)
* Caching behaviour – link to what we’re trying to achieve but explain the differences.
* Shared resources? We’re treating the mailbox as a resource ‘shared’ by all of the Identifiers on the network.

# Review of Existing System

## 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.

## 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.

Vehicle CAN buses

Mass storage (microSD card)

GSM Module

Remote AMQP message queue

Logging

Transmission

Interrupt

Interrupt

Internal hardware timer

Figure 4.1: EXISTING REMOTE DEVICE SOFTWARE Architecture

### 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.

### 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.

### Interrupts

Interrupts are used for the following software operations:

* **Interrupt events**

There are several hardware events that cause interrupts to be generated. 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.

## CAN Message Storage

The compression logic used by the telemetry system requires that each CAN message, identified by a unique ID, is stored in a fixed memory location so that comparisons can be made between old and new data. For this reason the CAN data is filtered and stored by the software as follows:



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 the device is interested in the message 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.

## Instrumentation

In order to provide the required benchmark, 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 debug terminal every 30 seconds.

This method was found to be the least expensive way of reporting the hit rates of each CAN ID.

# Dynamic CAN Filtering

## Overview

The proposed system attempts to address two of the problems mentioned above and provide an extension to the mailbox system that is suited to periodic polling, and accepts a greater number of identifiers than the current hardware platforms. The proposed system is designed to run on a TI C2000 processor, which possesses a mailbox-type CAN controller. The system is provided a ‘logging list’ which comprises of the CAN IDs that required to be logged by the device, along with their cycle times. A software layer sits between the hardware mailbox and a time-triggered hybrid scheduler (TTH). CAN message acceptance filtering is handles in hardware by the CAN controller. The [software] layer continually reads the incoming data from the CAN mailboxes and copies them into RAM. As messages arrive in a mailbox, 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.

One challenge involved in the development of this algorithm is t

* More detail about the proposed system
* Questions
  + How much do we need to know about a given CAN bus in terms of number of messages, cycle time, time between messages
    - In other words how ‘dynamic’ is dynamic?
  + How does this compare to the existing application?
    - Hit rate
    - Processor utilisation
    - SANITY!

An iterative development process was used to ensure that the resultant algorithm was as refined as possible. In order to provide a benchmark

* **Feasibility simulation**
* **Embedded software and** **Configuration and Analysis application**

Code in the existing telemetry device was ‘instrumented’ to allow further comparison with the existing system.

## 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 caught 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, periodical 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[].

Each line of the CAN trace is read by the simulation and the CAN identifier is examined to see if it is present in the Logging List. If the identifier is in the Logging List, acceptanceFilter[] is interrogated to see if the identifier is present. If the identifier is present in the acceptance filter, a ‘hit’ is recorded for the identifier. A counter relating to the captured identifier is incremented, as is a general “IDLogCount”. The identifier is marked as ‘logged’ in the acceptance filter, but the acceptance filter is not yet updated. If the identifier is not present in the acceptance filter, the identifier has been ‘missed’ and IDMissedCount is incremented.

The timestamp of each message is interrogated to identify when the logging task would run. When LOGGING\_TASK\_PERIOD\_us has expired, each identifier in the acceptance filter that 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 simulates the periodic polling behaviour of the time-triggered architecture, which causes logged identifiers to ‘block’ the filter until they are replaced.

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. In order to compensate for this, a compensation measure was included in the algorithm that uses a counter to ‘schedule’ the insertion of identifiers into the filter. This has the effect of weighting the filter in favour of the higher frequency messages. The reload value of the counter could be calculated dynamically from the known cycle times of the identifiers.

### 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 simulated task period affect the hit rate?
* How does the hit rate per identifier compare to that of the existing system?

### Method

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.

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.

These hit rates were compared to the output from the instrumented version of the existing embedded software.

### Results

Figure 5.1 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 5.1: Message hits vs filter size for varying list sizes

|  |  |  |  |
| --- | --- | --- | --- |
| List size | Total messages expected | Total hits when  filter size = list size/2 | Percent hits |
| 12 | 541318 | 541019 | 99.94% |
| 16 | 721755 | 689385 | 95.52% |
| 20 | 902192 | 888571 | 98.49% |
| 24 | 1082631 | 1076050 | 99.39% |
| 28 | 1263068 | 1261754 | 99.90% |
| 32 | 1344249 | 1342482 | 99.87% |

Figure 5.2 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 5.2: 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.

## Hardware Implementation incl. Remote Configuration

The hardware implementation involved porting the code from the feasibility simulation to tasks running in a TTH scheduler. The scheduler included a CAN mailbox handler layer developed specifically for periodic polling of the mailboxes.

In order to satisfy the ‘remote configuration’ aspect of the project, an application was written to transmit configuration data over a serial (RS232) connection. The application also displayed feedback on the behaviour of the filter mechanism, and allowed for fine-tuning to be made to the algorithm.

### Metrics

### Method

### Results

## Existing Device Comparison

# Discussion

## Algorithms

* Two parts
* Multiple mailbox ‘schedules’?
  + Group by similar cycle times – ie 20 ms / 100 ms divide in example trace.

## Simulation and Hardware Implementation Comparison

## Behaviour

With no duplication, sequencePointer skips 6 since the message is processed after the update

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 2 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 4 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 5 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 8 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 9 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 6 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 2 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 4 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 5 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 6 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 8 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 9 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

With duplication, 6 is allowed to be kept in the filter, despite being answered after the update

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 2 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 4 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 5 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 8 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 9 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 6 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 2 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 4 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 5 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 6 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 8 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 9 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

## Performance Comparison with Existing System

## Future Development

* Combined ‘hard’ mailboxes with dynamic mailboxes to guarantee collection of critical messages
* Hardware / VHDL implementation
  + Layer between COTS CAN controller
  + Built into custom CAN controller

# Conclusions

# Works Cited

|  |  |
| --- | --- |
| [2] | STMicroElectronics, “Controller area network (bxCAN),” in *RM0090 Reference Manual: STM32F405xx, STM32F407xx, STM32F415xx and STM32F417xx advanced ARM-based 32-bit MCUs*, 2012, pp. 784 - 825. |
| [3] | Texas Instruments, TMS320F2833x, 2823x Enhanced Controller Area Network (eCAN) Reference Guide, Dallas, Texas, 2009. |
| [5] |  |
| [6] | Microchip Corporation, “PIC32MX5XX/6XX/7XX Family Data Sheet: High-Performance, USB, CAN and Ethernet 32-bit Flash Microcontrollers,” 2011. |
| [7] | Wikipedia, “RS-485,” 25 August 2012. [Online]. Available: http://en.wikipedia.org/wiki/RS-485. [Accessed 11 September 2012]. |

# Appendix