|  |
| --- |
| [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](#_Toc373315847)

[Table of Contents 2](#_Toc373315848)

[Table of Figures 4](#_Toc373315849)

[1 Introduction 5](#_Toc373315850)

[2 Background 7](#_Toc373315851)

[2.1 Problem area 7](#_Toc373315852)

[2.2 Real-time and Time-Triggered Software Architecture 7](#_Toc373315853)

[2.3 Controller Area Network 9](#_Toc373315854)

[2.4 Using CAN with TT Architectures 12](#_Toc373315855)

[3 Related Work 14](#_Toc373315856)

[4 Automotive Telemetry System 16](#_Toc373315857)

[4.1 Overview 16](#_Toc373315858)

[4.2 Software Architecture 16](#_Toc373315859)

[4.3 CAN Message Storage 18](#_Toc373315860)

[4.4 Instrumentation 19](#_Toc373315861)

[5 Dynamic CAN Filtering Development 20](#_Toc373315862)

[5.1 Overview 20](#_Toc373315863)

[5.2 Feasibility Simulation 22](#_Toc373315864)

[5.3 Embedded Software Implementation 24](#_Toc373315865)

[5.4 Remote Configuration 24](#_Toc373315866)

[5.5 Existing Device Comparison 24](#_Toc373315867)

[6 Analysis 25](#_Toc373315868)

[6.1 Questions 25](#_Toc373315869)

[6.2 Method 25](#_Toc373315870)

[6.3 Results 26](#_Toc373315871)

[7 Discussion 28](#_Toc373315872)

[7.1 Algorithms 28](#_Toc373315873)

[7.2 Simulation and Hardware Implementation Comparison 30](#_Toc373315874)

[7.3 Duplication in filter 30](#_Toc373315875)

[7.4 Performance Comparison with Existing System 31](#_Toc373315876)

[7.5 Future Development 31](#_Toc373315877)

[8 Conclusions 32](#_Toc373315878)

[9 Works Cited 33](#_Toc373315879)

[Appendix 35](#_Toc373315880)

# Table of Figures

[Figure 2‑1: A simple Time-Triggered Co-operative (TTC) scheduler 8](#_Toc373315881)

[Figure 2‑2: A SIMPLE TIME-TRIGGERED Hybrid (TTH) SCHEDULER 9](#_Toc373315882)

[Figure 2‑3: Layered Architecture of CAN according to the OSI Reference Mode [4] 9](#_Toc373315883)

[Figure 2‑4: Variants of CAN Hardware [6] 10](#_Toc373315884)

[Figure 2‑5: CAN ACCEPTANCE FILTERING ON AN STM STM32F407ZGT6 PROCESSOR [8] 11](#_Toc373315885)

[Figure 4‑1: EXISTING REMOTE DEVICE SOFTWARE Architecture 16](#_Toc373315886)

[Figure 4‑2: SOFTWARE FILTERING IN THE CAN DATA LOGGING PROCESS 19](#_Toc373315887)

[Figure 6‑1: Message hits vs filter size for varying list sizes 26](#_Toc373315888)

[Figure 6‑2: Hit rate vs filter size for 32-ID logging list 27](#_Toc373315889)

# Introduction

Controller Area Network (CAN) has become a standard method of communication between embedded devices in automotive applications [1]. CAN Messages contain data that is transferred between sub-systems. 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.

# 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 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 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.

## 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 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 2‑2 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.

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 [3]. 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

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.

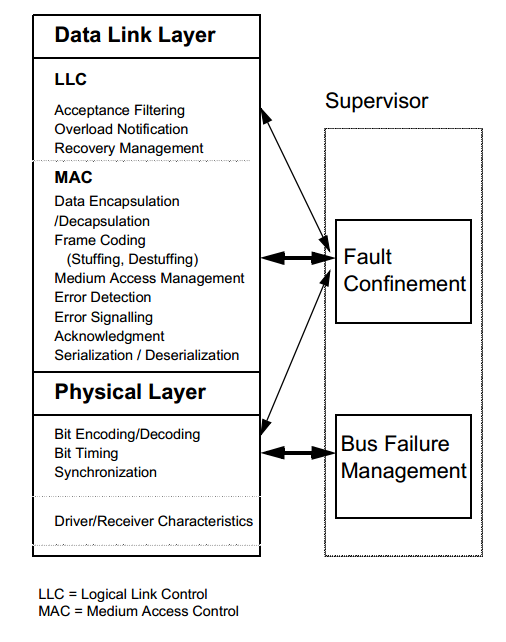


Figure 2‑3: Layered Architecture of CAN according to the OSI Reference Mode [4]

### 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.

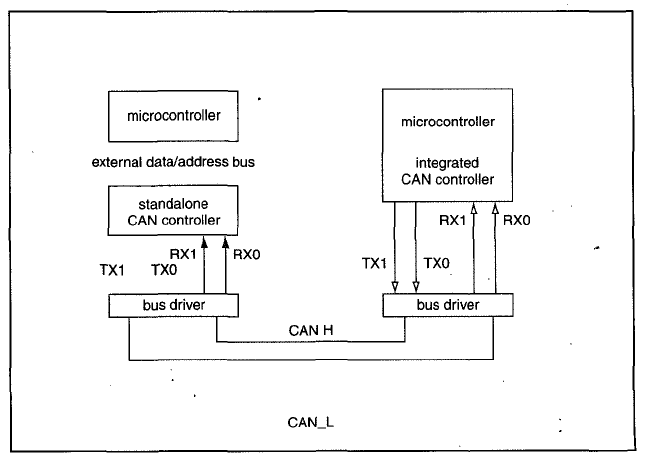


Figure 2‑4: Variants of CAN Hardware [6]

### 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 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 **Invalid source specified.**. 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 [8]:

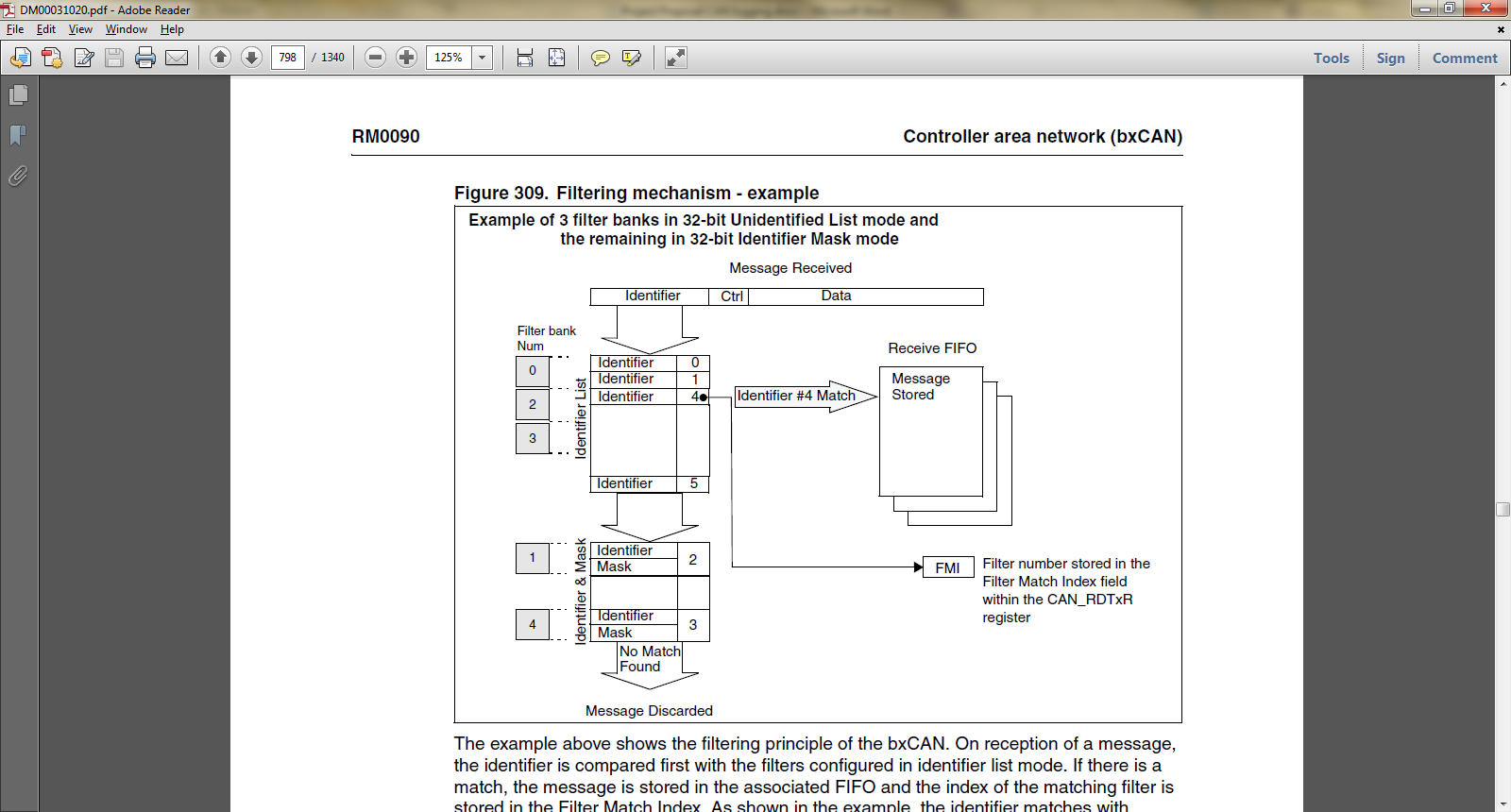


Figure 2‑5: CAN ACCEPTANCE FILTERING ON AN STM STM32F407ZGT6 PROCESSOR [8]

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.

Another method is the use of ‘mailboxes’, as used by Texas Instruments [9]. 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.

## 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

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.

In [10] and [11], 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 protocol, FTT-CAN, which provides flexible, dynamic scheduling of messages on a CAN bus.. A centralised master node uses a “Planning Scheduler” [12] 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. [13] 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 [14].

CAN message scheduling for time-critical control systems is discussed by Martí et al in [15].

[16] 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…

The research in this project is novel in that it approaches the problems surrounding CAN message reception on COTS hardware when the message set is unknown at compile time.

# Automotive Telemetry 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 a benchmark with which 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 an 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 addresses two of the problems mentioned above and provides 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 incorporates a mailbox-type CAN controller, referred to by the manufacturer as ‘eCAN’. The system is provided a ‘logging list’ which comprises of the CAN IDs required to be logged by the device, along with their cycle times. CAN message acceptance filtering is handles 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**
* **Configuration and Analysis application**

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.

Configuration Task

**TTH Scheduler**

Periodic filtering task

**CAN Bus**

**Logging Sequence**

**Data log**

Correct location for received data

**CAN Mailbox Transceiver**

RX Buffer

RX Mask / Filter

Unfiltered data in approximately known sequence

Next required CAN message

Filter set to next CAN ID in logging sequence

Data retrieval

Next CAN ID to be used in filter

CAN IDs to log (Logging list)

**Remote Server**

## 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, 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[].

### 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.

### 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.

### 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.

## 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 [9].

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).
* 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 Figure 5‑2).

|  |  |  |  |
| --- | --- | --- | --- |
| **Filter Mailboxes** |  | **Logging List** | |
|  | **CAN ID** | **Cycle time (ms)** |
| 0 |  | 0 | 100 |
| 1 | Segment 1 – | 1 | 100 |
| 2 | 100 ms time domain | 2 | 100 |
| 3 |  | 3 | 100 |
| 4 |  | 4 | 100 |
| 5 |  | 5 | 100 |
| 6 |  | 6 | 1000 |
| 7 | Segment 2 - | 7 | 1000 |
| 8 | 1000 ms time domain | 8 | 1000 |
|  |  | 9 | 1000 |
|  |  | 10 | 1000 |
|  |  | 11 | 1000 |
|  |  | 12 | 1000 |
|  |  | 13 | 1000 |
|  |  | 14 | 1000 |
|  |  | 15 | 1000 |
|  |  | 16 | 1000 |
|  |  | 17 | 1000 |

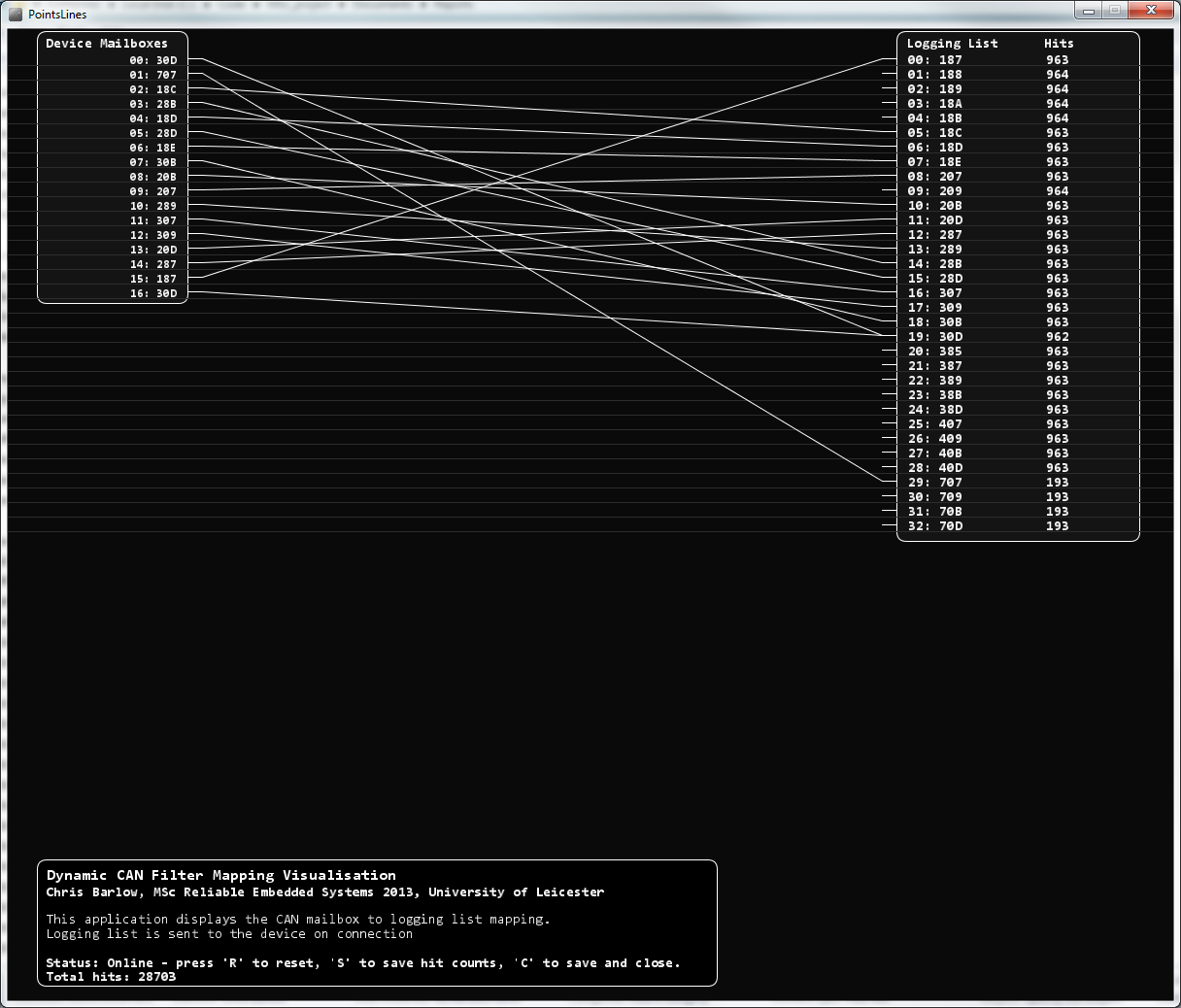
Figure 5‑2: Filter Segmentation for Different Message Cycle Times

* 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.

## Remote Configuration and Analysis Tool

In order to satisfy the ‘remote configuration’ aspect of the project, an application was written in ‘Processing’ programming language [17]. 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.



**Mailboxes in use and their configured CAN IDs reported by the device**

**Logging list transmitted to device when in ‘UPDATE’ mode**

**Hit counts for each ID reported by embedded system**

**Visualisation of mailbox – logging list mapping**

Figure 5‑3: Visual Feedback and Hit Rate Analysis from RCAT

# Analysis

## 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?

## Method

### 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.

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.

In order to gauge the impact of changing the order of identifiers in the sequence, the …..

### Hardware

PC

TI Development Board

**PCAN USB to CAN bus adapter**  
Sample CAN trace 🡪

**USB to RS232 adapter**

Logging list 🡪

🡨 CAN ID hit counts  
🡨 Mailbox-to-logging list mapping

Processing App

PCAN Trace software

Existing telemetry device

RS232

USB

USB

CAN BUS

Figure 6‑1: Hardware Test Setup

## Results

Figure 6‑1 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 6‑2: 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 6‑2 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 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.

# Discussion

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

* 1. 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?

## 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.

The first was a scheduling strategy that weighted an identifier’s insertion rate into the acceptance filter based on its cycle time. Each identifier is given a counter, reloaded with a value determined by:

Equation 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. The hit rates became more consistent between identifiers.

[need some example results here]

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 identifier scheduling was driven by the arrival of the more frequent messages. For configurable applications with unknown CAN domains, this strategy was deemed impractical.

[need some example results here]

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.

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.

[need some example results here]

## Simulation and Hardware Implementation Comparison

## Duplication in filter

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

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | Filter initialisation |
| 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | ID 0 arrives, replaced with ID 6 |
| 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | ID 1 arrives, replaced with ID 7 |
| 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 | ID 6 missing from sequence 1 |
| 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 | ID 6 arrives late leaving gap |
| 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 |

## 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:

Table 7‑1: Worst Case Inter-frame Spacing [6]

|  |  |
| --- | --- |
| Baud Rate (kbit/s) | Inter-frame space  (s) |
| 1000 | 47 |
| 500 | 94 |
| 250 | 188 |
| 125 | 376 |

From Table 7‑1 above, it can be seen that the shortest time between frames on a 500 kbit/s CAN bus is 94 s. This means that in our TT system, polling the mailboxes at 1000 s, there will be a maximum of 10 frames arriving between filter updates. The system will still catch the contents of such data bursts as long as the filter contains a minimum of 10 configured mailboxes.

## 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

The result is an application that can be executed in a desktop environment prior to integration of the embedded system that provides information about the expected performance of the system. The configurable embedded application could be used as a framework to build a sophisticated

# 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. |
| [3] |  |
| [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. |
| [8] | STMicroElectronics, “Controller area network (bxCAN),” in *RM0090 Reference Manual: STM32F405xx, STM32F407xx, STM32F415xx and STM32F417xx advanced ARM-based 32-bit MCUs*, 2012, pp. 784 - 825. |
| [9] | Texas Instruments, TMS320F2833x, 2823x Enhanced Controller Area Network (eCAN) Reference Guide, Dallas, Texas, 2009. |
| [10] | 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. |
| [11] | 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. |
| [12] | 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. |
| [13] | 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. |
| [14] | 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. |
| [15] | 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. |
| [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] | 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. |
| [18] | 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. |

# Appendix