|  |
| --- |
| [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.] |

# Table of Contents

[1 Table of Contents 1](#_Toc370070301)

[2 Acknowledgements 3](#_Toc370070302)

[3 Introduction 3](#_Toc370070303)

[4 Background 4](#_Toc370070304)

[4.1 Controller Area Network (CAN) 5](#_Toc370070305)

[4.1.1 CAN Hardware 6](#_Toc370070306)

[4.1.2 Acceptance Filtering 6](#_Toc370070307)

[4.2 Time-Triggered Architecture 8](#_Toc370070308)

[4.2.1 Limitations 9](#_Toc370070309)

[5 Related Work / Literature Review 10](#_Toc370070310)

[6 Review of Existing Software and Technology 11](#_Toc370070311)

[6.1 Electric Commercial Vehicle Telemetry System 11](#_Toc370070312)

[6.2 Message Storage in the Existing Device 12](#_Toc370070313)

[6.3 CAN Acceptance Filtering 12](#_Toc370070314)

[7 Dynamic CAN Filtering 13](#_Toc370070315)

[7.1 Overview 13](#_Toc370070316)

[7.2 Algorithm 14](#_Toc370070317)

[8 Testing 14](#_Toc370070318)

[8.1 Feasibility Simulation 14](#_Toc370070319)

[8.1.1 Overview 14](#_Toc370070320)

[8.1.2 Metrics 14](#_Toc370070321)

[8.1.3 Method 14](#_Toc370070322)

[8.1.4 Results 14](#_Toc370070323)

[8.2 Hardware Implementation incl. Remote Configuration 14](#_Toc370070324)

[8.2.1 Overview 14](#_Toc370070325)

[8.2.2 Metrics 15](#_Toc370070326)

[8.2.3 Method 15](#_Toc370070327)

[8.2.4 Results 15](#_Toc370070328)

[9 Discussion 15](#_Toc370070329)

[9.1 Simulation and Hardware Implementation Comparison 15](#_Toc370070330)

[9.2 Behaviour 15](#_Toc370070331)

[9.3 Performance Comparison with Existing System 17](#_Toc370070332)

[9.4 Future Development 18](#_Toc370070333)

[10 Conclusions 18](#_Toc370070334)

[11 Works Cited 18](#_Toc370070335)

[12 Appendix 19](#_Toc370070336)

# Acknowledgements

# 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.

# Background

* Intro will summarise all of this, but go into more detail here

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 approaches are now being explored for an updated device, including the ability to remotely modify which CAN messages 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.

## 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

* eCAN vs standard CAN

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.

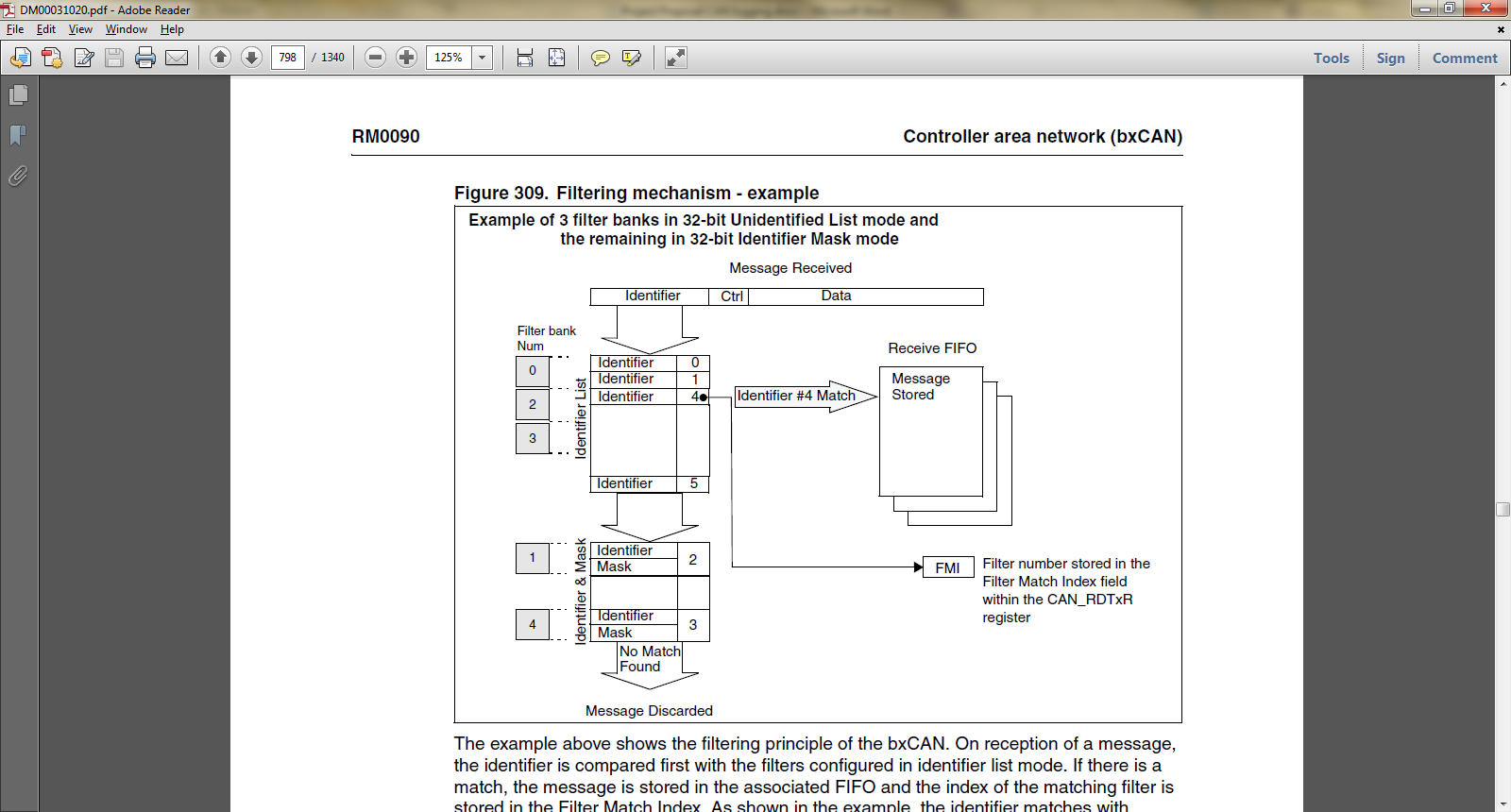


Figure 4.1: CAN ACCEPTANCE FILTERING ON AN STM STM32F407ZGT6 PROCESSOR [2]

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, either using an interrupt, or a flag, the microcontroller can now read the data directly from the mailbox and knows the context without using any look-up mechanisms.

This method is very useful for devices interested in a relatively small subset of messages, however with the maximum possible number of identifiers of a standard frame 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.

## Time-Triggered Architecture

Time-triggered 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 microseconds **Invalid source specified.**. This software runs on very strict rules that ensure the software is driven by elapsed time, rather than external ‘events’ ***(something about event triggered here).***

The backbone of this architecture is a scheduler driven by a single 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).

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 no unexpected events will 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].

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 4.3: 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

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

B

A

Figure 4.4: A SIMPLE TIME-TRIGGERED Hybrid (TTH) SCHEDULER

### Limitations

As mentioned previously, Time-Triggered architecture can only behave predictably if there are no interrupts active other than the timer that drives the scheduler. When handling events such as CAN message arrival, this means that 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, the controller must be polled at least as often as

F\_min \* depth\_buffer

Or

F\_min \* N\_mailboxes

* TT scheduling
  + Can only use polling
* Existing application
  + Some time-released functions
    - Don’t want to call these TT since the timing is not so accurate
  + event-triggered (interrupts, interrupts, interrupts!) functionality going on at the same time – typical of a non-time critical application built by two independent teams
  + No way of guaranteeing hit rate of CAN messages.
* Proposed Application

Data logging application running on Time Triggered scheduler occupying busy CAN bus (> 32 Different messages) alongside event-triggered components

* + - Cannot poll fast enough to collect data while performing other processor-intensive tasks
      * Not enough mailboxes in eCAN
      * At worst case, messages arrive in ‘burst’ (immediately after each other with period of cycle time) so FIFO not deep enough.
    - Cannot use interrupts because this breaks the “1 interrupt per CPU” rule.

Solution – Use mailboxes with dynamic assignment

# 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 ID’s on the network.

# Review of Existing Software and Technology

* Not sure where to put this…

## Electric Commercial Vehicle Telemetry System

Vehicle Comms buses

Mass storage (microSD card)

GSM Module

Remote AMQP message queue

Logging

Transmission

Interrupt

Interrupt

Internal hardware timer

Figure 6.1: EXISTING REMOTE DEVICE SOFTWARE OVERVIEW

* **Data logging functions** - performed using timer-driven interrupts.
  + CAN and sensor data are collected from the vehicle and buffered internally.
  + Buffered data is compressed and stored on a micro SD card in data ‘blocks’ of around   
    4 – 10 kB.
* **Data transmission functions** – performed using an interrupt on the GSM module, indicating that the module is connected to the network.
  + A connection is made to a remote AMQP message queue.
  + Data 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.

## Message Storage in the Existing Device

The compression logic used by the telemetry system requires each CAN message, identified by a unique ID, to be stored in the same location every time 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 6.2: SOFTWARE FILTERING IN THE CAN DATA LOGGING PROCESS – EXISTING REMOTE DEVICE

This means that whenever a CAN message is seen on the CAN bus, the device has to perform some processing on it, whether the device is interested in the message or not.

## 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.

# Dynamic CAN Filtering

## Overview

* More detail about the proposed system
* Questions
  + Is this feasible given the unpredictable nature of CAN
  + 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!

## Algorithm

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

# Testing

## Feasibility Simulation

### Overview

### Metrics

### Method

### Results

## Hardware Implementation incl. Remote Configuration

### Overview

* Integration into TTH scheduler
* Supporting software
* Remote configurator / analyser (Processing Sketch)
  + RS232 Comms
    - Simple {~} Protocol
  + Mapping visualisation
  + Log output
  + Instrumented scheduler – collect information about timing behaviour of the tasks (Impact ISR has on other tasks)

### Metrics

### Method

### Results

# Discussion

## 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