TIME-TRIGGERED CONTROLLER AREA NETWORK DESIGN FOR FORMULA SAE RACECARS AND TECHNIQUE FOR MEASURING CPU USAGE ON SYSTEMS WITH NESTED AND NON-NESTED INTERRUPTS

by

RANDY KYLE LONG

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

August 2016

Acknowledgements

First and foremost my thanks goes to Dr. Taylor Johnson and his seemingly infinite patience with a student who changed his thesis topic every 15 minutes.

Also, my thanks goes to Dr. Woods, Michael Hibbard, David Campbell, Audrey Porter, Matthew Martin, and all other contributors to the UTA FSAE E-16 project. My thesis topic was chosen specifically to support that vehicle during the design defense event at the upcoming Formula Student Germany competition. Without their efforts, I would have had to write about something else much less interesting.

And lest we forget: The material presented in this thesis is based upon work supported by the National Science Foundation (NSF) under grant numbers CNS 1464311 and CCF 1527398, the Air Force Research Laboratory (AFRL) through contract number FA8750-15-1-0105, and the Air Force Office of Scientific Research (AFOSR) under contract numbers FA9550-15-1-0258 and FA9550-16-1-0246. The U.S. government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of AFRL, AFOSR, or NSF.

July 29, 2016

Abstract

TIME-TRIGGERED CONTROLLER AREA NETWORK DESIGN METHODOLOGY AND TECHNIQUE FOR MEASURING CPU USAGE ON SYSTEMS WITH NESTED AND NON-NESTED INTERRUPTS

RANDY K LONG, MS

The University of Texas at Arlington, 2016

Supervising Professor: Taylor Johnson

This document presents a design method and example for a time-triggered CAN (TTCAN) system which reduces latency, variability in transmission frequency, and increases data throughput relative to the traditional free-for-all CAN (FFACAN) transmission system.

In addition to performance considerations for a TTCAN transmission scheme, this document also presents a simple method for measuring CPU usage on embedded systems that use either nested or non-nested interrupts. Systems that use non-nested interrupts may have their performance measured with a single GPIO pin; systems that use nested interrupts require one pin per interrupt service routine.

Table of Contents

Acknowledgements iii
Abstract iv
Chapter 1 Project Overview 1
Chapter 2 CAN Bus Basics 4
Multi-Master Hardware Arbitration 4
Anatomy of a CAN Frame 6
The Arbitration Field 6
The Control Field 6
The Data Field 7
The CRC Field 7
Filters & Masks 8
Stuff Bits 10
CAN Bus Performance Considerations 11
Chapter 3 Time-Triggered CAN Network Traffic Management 14
TTCAN Design Considerations 15
TTCAN Design Method 17
Designing for Multi-Cycle Transmission 21
TTCAN System Design Example 23
Designing for Multi-Cycle Transmission (2x) 28
Designing for Multi-Cycle Transmission (5x) 31
Designing for Multi-Cycle Transmission (8x) 32
Designing for Multi-Cycle Transmission (9x) 34
Summary of Results 36
Chapter 4 Comparison of TTCAN to FFACAN 37
Baseline Values 39
Comparing the 2x Transmission Multiplier 41
Comparing the 5x Transmission Multiplier 42

Comparing the 8x Transmission Multiplier 43
Comparing the 9x Transmission Multiplier 44
Summary of Results 45
Chapter 5 CPU Usage Calculation 46
Non-Nested vs Nested Hardware Interrupts 47
CPU Usage Calculation – Method 1 49
CPU Usage Calculation – Method 2 51
CPU Usage Measurement Implementation Methodology 54
Method 1, for Non-Nested Interrupts 54
Method 2, for Non-Nested Interrupts 54
Method 1, for Nested Interrupts 55
Method 2, for Nested Interrupts 55
CPU Usage Measurement Implementation Example 56
Parameters and Expected Results 56
Method 1, for Non-Nested Interrupts 57
Method 2, for Non-Nested Interrupts 59
Method 1, for Nested Interrupts 63
Chapter 6 Conclusions 67

List of Illustrations

Figure 1-1 F14 driving with an active-aerodynamic system 3
Figure 2-1 CAN bus differential voltage signaling 4
Figure 2-2 A filtered messaged is received 9
Figure 2-3 A filtered message is rejected 9
Figure 2-4 Generalized form of NRZ bus resynchronization 10
Figure 2-5 CAN bus time quanta parameters 12
Figure 3-1 Desynchronization due to differences in the clock rates 16
Figure 3-2 An edge counter in action 18
Figure 3-3 Determining minimum Δtmsg 21
Figure 3-4 Determining maximum Δtmsg 21
Figure 3-5 Determining minimum Δtmsg (2x transmission) 28
Figure 3-6 Determining maximum Δtmsg (2x transmission) 28
Figure 3-7 Determining minimum Δtmsg (5x transmission) 31
Figure 3-8 Determining maximum Δtmsg (5x transmission) 31
Figure 3-9 Determining minimum Δtmsg (8x transmission) 32
Figure 3-10 Determining maximum Δtmsg (8x transmission) 32
Figure 3-11 Determining minimum Δtmsg (9x transmission) 34
Figure 3-12 Determining maximum Δtmsg (9x transmission) 34
Figure 4-1 Bus configuration for the baseline test 39
Figure 4-2 Bus configuration for the 1x transmission multiplier test 40
Figure 4-3 Bus configuration for the 2x transmission multiplier test 41
Figure 4-4 Bus configuration for the 5x transmission multiplier test 42
Figure 4-5 Bus configuration for the 8x transmission multiplier test 43
Figure 4-6 Bus configuration for the 9x transmission multiplier test 44
Figure 5-1 Interrupt vectors for a TI TM4C1294NCPDT microcontroller [3] 48
Figure 5-2 Non-nested interrupt instrumentation output 49
Figure 5-3 Nested interrupt instrumentation output 51
Figure 5-4 Nested interrupt instrumentation output with highlights 53
Figure 5-5 Oscilloscope Graphical Output 57
Figure 5-6 Output from Timer 2 ISR (1.0 ms per division) 59
Figure 5-7 Output from Timer 3 ISR (1.0 ms per division) 60
Figure 5-8 Output from Timer 4 ISR (1.0 ms per division) 60
Figure 5-9 Output from Timer 5 ISR (100 µs per division) 61
Figure 5-10 Output from main loop (100 µs per division) 61
Figure 5-11 Oscilloscope Graphical Output (Incorrect) 63

Figure 5-12 Oscilloscope Graphical Output (Correct) 65
Figure 5-13 Raw vs Masked Signal for Timer 2 65

List of Tables

Table 3-1 Oscillator Speed Differences 24
Table 3-2 Parameters for a 5x Transmission Multiplier Scheme 31
Table 3-3 Parameters for a 8x Transmission Multiplier Scheme 33
Table 3-4 Parameters for an 9x Transmission Multiplier Scheme 35
Table 3-5 Min/Max/Avg Frame Transmission Rates for all Transmission Multipliers 36
Table 3-6 Min/Max/Avg Data Transmission Rates for all Transmission Multipliers 36
Table 4-1 Node Descriptions for FFACAN Bus 38
Table 4-2 TTCAN Transmission & Data Rates 38
Table 4-3 Requested vs Actual Transmission Rates, Baseline 39
Table 4-4 1x Transmission Multiplier Test Results 40
Table 4-5 Transmission Multiplier Test Results 41
Table 4-6 5x Transmission Multiplier Test Results 42
Table 4-7 8x Transmission Multiplier Test Results 43
Table 4-8 9x Transmission Multiplier Test Results 44
Table 4-9 Summary of Requested vs Actual Transmission Rates 45
Table 5-1 Timer Interrupt Configuration 56
Table 5-2 Oscilloscope Measurement Output (Multi-Pin, Non-Nested) 58
Table 5-3 Oscilloscope Measurement Output (Single-Pin, Non-Nested) 62
Table 5-4 Oscilloscope Measurement Output (Incorrect, Nested) 64
Table 5-5 Oscilloscope Measurement Output (Correct, Nested) 66

Chapter 1

Project Overview

A time-triggered (TT) transmission scheme has two primary beneficial characteristics which can be applied to any packet-based, multi-master telecommunication technology:

- Determinism - since transmission events follow a defined schedule, variability in latency is greatly reduced in comparison to a free-for-all transmission scheme. Determinism is an important characteristic of safety critical control systems – modern automobiles contain dozens of safety systems (anti-lock brakes, traction control, supplemental restraint systems). [4]
- Increased throughput - since the media access control hardware does not need to spend time in channel access arbitration, transmission events can take place more often.

A TT transmission schedule is made up of transmission windows, periods of time during which one or more bus members may attempt to transmit. Channel members may only transmit during their designated transmission windows - members with their own dedicated transmission window are therefore guaranteed exclusive access to the channel during that time. Collections of windows are called cycles, and are usually accompanied by a schedule resynchronization event. All transmission takes place according to this schedule; transmission rates can be easily predicted as long as all channel members follow this schedule. Also, since a given channel member can transmit without interference from another channel member, the effective information throughput is increased, since the transmitters do not waste time arbitrating for control of the channel.

This document presents an analysis and design example of a time-triggered (TT) transmission scheme over a Controller Area Network (CAN) bus. Chapter 2 contains a

summary of major CAN bus features along with considerations that must be made when designing a TTCAN system including the number of stuff bits per frame and bus timing parameters.

This document also presents a method for measuring performance of interrupt-driven embedded applications. Devices with large amounts of computerized processing capability and random-access memory (RAM) (such as PCs, modern phones, video game systems, etc.) are capable of running their own central processing unit (CPU) usage profiling applications with minimal effect on system performance. This is compared this to applications that run on microcontrollers with MB of flash memory and only a few KB of RAM. Live system profiling software may effect performance in a way that prevents the application from running normally. In-circuit programming and profiling hardware is available from most microcontroller manufacturers, but this equipment can be expensive and cumbersome for an inexperienced developer.

Embedded CPU usage can be measured with a simple and minimally intrusive technique in which each hardware interrupt is associated with one general purpose input or output (GPIO) pin. The state of the pin reflects that state of the interrupt – if the interrupt is active, the pin state is “high” (logical 1), if the interrupt is inactive, the pin state is “low” (logical 0).

Systems with nested interrupts can be profiled using the same technique as long as one GPIO pin is available for each nesting interrupt. The instrumentation can be extended to include tasks as well (if an operating system is present and running). Performance is measured with an external device such as an oscilloscope, saving the embedded processor the trouble of calculating and transmitting its own CPU usage numbers.

Both of these topics – TTCAN and CPU usage calculation measurements – were applied to the 2013, 2014, 2015, and 2016 design year version of UTA Formula SAE's active-aero control systems (see Figure 1-1).

Figure 1-1 F14 driving with an active-aerodynamic system

Chapter 2

CAN Bus Basics

Controller Area Network (CAN) bus technology was developed in response to increasing demand for a low-cost, simple, and robust network communication protocol for use with automotive applications. The physical layer of a CAN bus requires two conductors, two termination resistors, and one differential transceiver per bus member (bus members are also called **nodes**). Data frames are transmitted with unique identifier codes (UIDs), a data payload, and cyclic-redundancy check to ensure that data is not corrupted during transmission. Bits transmitted over a bus are sent via differential voltage signaling, which provides some resilience to electrical noise. [1]

Multi-Master Hardware Arbitration

Any node attached to a CAN bus can attempt to transmit at any time. CAN technology supports a form of non-destructive hardware arbitration to manage collisions and ensure that all nodes will eventually have a chance to transmit (assuming the bus is not overloaded with traffic). [1]

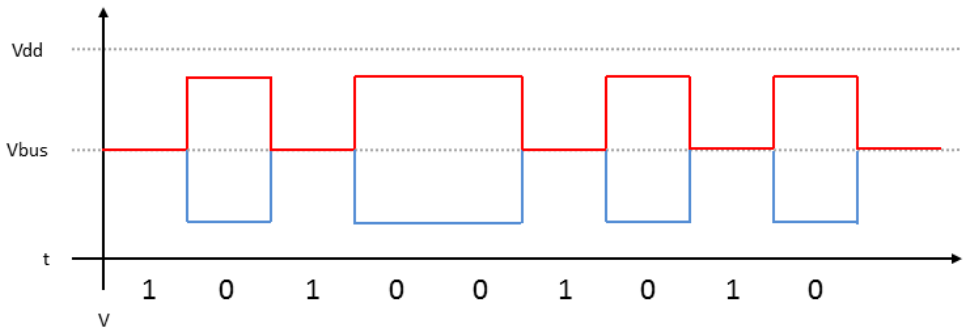

Figure 2-1 CAN bus differential voltage signaling

Figure 2-1 shows the generalized form of a CAN bus signal. Information is transmitted on a CAN bus in the form of differential voltage signals – instead of interpreting a signal by comparing it to some common reference, two signals are compared to each other. This ensures that information is received even if some noise causes severe corruption of the signal. [1]

The CAN standard distinguishes between 0s and 1s with the terms **dominant** and **recessive**, respectively. When no bits are being transmitted on the bus the termination resistors force the voltage on both conductors to an equal level. This is a recessive bus state, which represents a digital logic 1. When a node has control of the bus and is transmitting, the node's CAN transceiver is forcing the two conductors to different voltage levels. This is a dominant state, which represent a digital logic 0. [2]

Arbitration happens on a per-bit basis. A CAN bus transmitter silently listens to the bus state during transmission to ensure that the bus state is set correctly. If two nodes attempt to transmit at the same time, the node with the most dominant message will win arbitration and continue to transmit; the node that lost arbitration will wait a random amount of time before attempting to transmit again. Since a digital logic 0 represents a dominant bus state, CAN frames can be assigned some intrinsic priority through their unique identifier (UID) field – UID 0 has the highest priority. [2]

Anatomy of a CAN Frame

CAN frames are made up of four main parts:

- The arbitration (UID) field
- The control field
- The data field
- The cyclic-redundancy check (CRC) field

Each frame is also succeeded by at least 3 interframe spacing bits. [2]

The Arbitration Field

The arbitration field contains the CAN frame's UID and some control bits. The UID can be filtered and / or ignored by a CAN bus receiver hardware so that the microcontroller or microprocessor attached to the CAN receiver does not waste time processing a CAN frame that has no relevance to the receiver's tasks. The design example in Chapter 3 assumes that all CAN filters are disabled and thus have no effect on bus performance. However a short explanation of how CAN filters and masks work is given in a later section within this chapter.

Frames have two possible lengths: standard and extended. Standard frames have UIDs that are 11 bits in length (2047 possible unique values) while extended frames have UIDs that are 29 bits in length (536,870,912 possible unique values). Buses that follow the CAN2.0A specification use standard frames only and treat extended frames as bus errors. CAN2.0B buses support both standard and extended frames. [2]

The Control Field

CAN frames contain several control bits that identify the purpose and type of the frame. Standard and extended frames are distinguished by the value of the extended

identifier (IDE) bit, which is cleared for standard frames and set for extended frames. The CAN bus specification also supports Remote Transmission Requests, or RTR frames. This functionality allows a transmitter to trigger transmission of a message at a remote node for the purposes of polling or network flow control. RTR frames are rarely used in practice and thus are not included in the scope of this document. Extended frames have a Substitute Remote Request bit (SRR) that fills in the bit that would normally be used to trigger a standard RTR frame. [2]

The Data Field

CAN frames carry a 0 to 8 byte payload in their data field. [2] Officially, CAN2.0A and CAN 2.0B frames can only carry up to 8 bytes; however a clever application designer may notice that a standard frame (with 11 bits in the ID field) or an extended frame (with 29 bits in the ID field) allows for a few extra bytes to be stuffed into the frame's UID field. This technique is referenced as "byte stealing" throughout this document. Specifically, standard frames can accommodate an additional byte, leaving 3 bits left over for frame identification. Extended frames can accommodate an additional 3 bytes, leaving 5 bits left over for frame identification.

The CRC Field

To ensure that a CAN frame arrived at a receiver exactly as it was sent, each frame includes a 15 bit CRC before the end-of-frame field. Modern CAN receivers have hardware support for the CRC generation and check the CRC of each received message without any intervention from a microcontroller or microprocessor. If the CRC in the frame does not match the one that was calculated by the receiver, the frame is discarded without being copied to a receive buffer. [2]

Filters & Masks

Masks and filters work together to form simple “deny” or “allow” functionality for CAN bus receivers. This is particularly useful for nodes with low processing power or for nodes that cannot afford to spend time processing interrupts associated with CAN reception on a bus with a lot of traffic. Filters can be applied to both standard and extended frames. The number of bits in a filter is equal to the number of bits in the incoming frames’ UID. [2]

Upon reception of a frame the receiver first compares the frame’s type to all configured filters. Extended frames are not compared to standard filters, or vice versa. If a filter with a matching length is found, the CAN receiver will first mask the bits in the filter before comparing the filter against the incoming frame.

Masks, like filters, contain the same number of bits as the incoming frame’s UID. Masks allow an application designer to selectively enable and disable groups of bits within the filter so that only messages that lie within a certain UID range are received. Filters must be paired with masks; how this is done depends on the implementation of the CAN hardware. [2]

A filter can be disabled entirely by clearing all bits in the mask associated with that filter. The TTCAN design example in Chapter 3 quantifies CAN bus performance from the perspective of the bus, thus considerations for filter and masks are not included in the TTCAN design example.

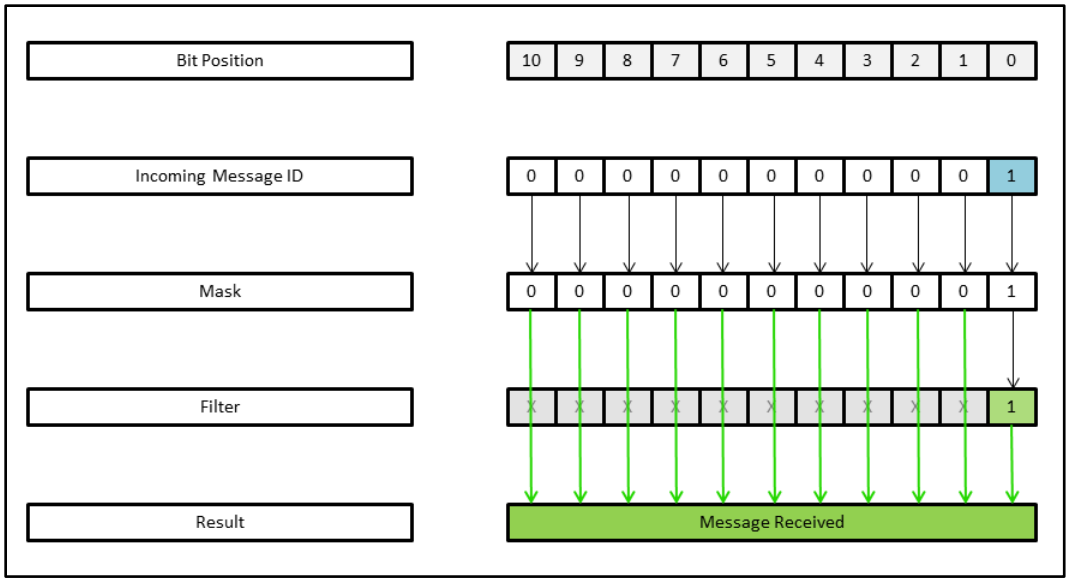

Figure 2-2 A filtered messaged is received

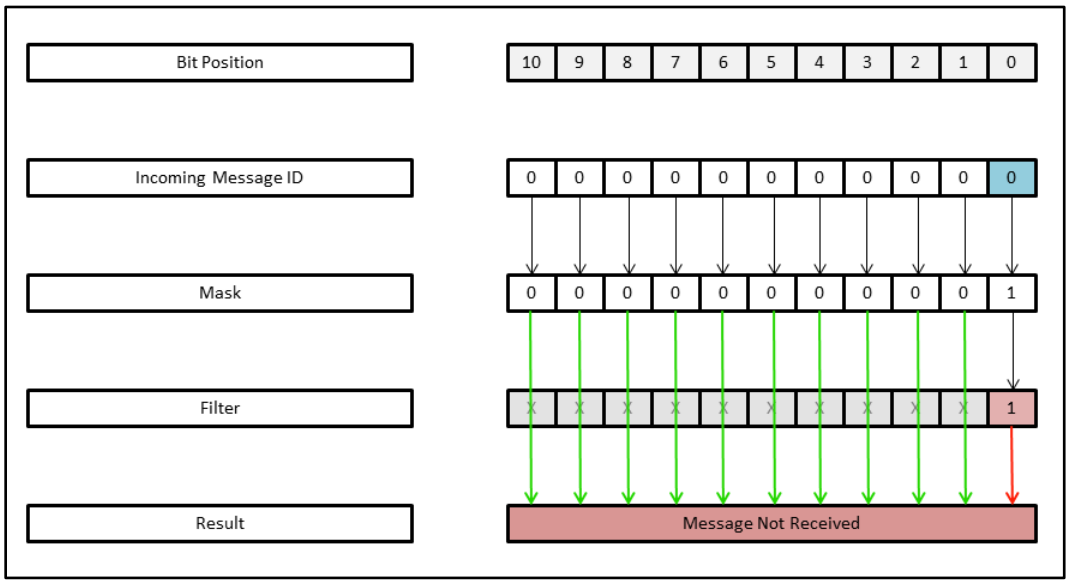

Figure 2-3 A filtered message is rejected

Stuff Bits

Information transmitted on a CAN bus is represented by a non-return to zero (NRZ) differential voltage signal. CAN receivers are responsible for maintaining a synchronized bus clock. Since there is no dedicated clock signal, the bus clock must be based on the rising edges of dominant bits. If a long string of recessive or dominant bits are received, a receiver's clock may drift out of time to the point where it becomes desynchronized with the bus. This problem is solved with **stuff bits**, special clock synchronization bits that are automatically inserted into CAN frames by the CAN media access control hardware during transmission. [2]

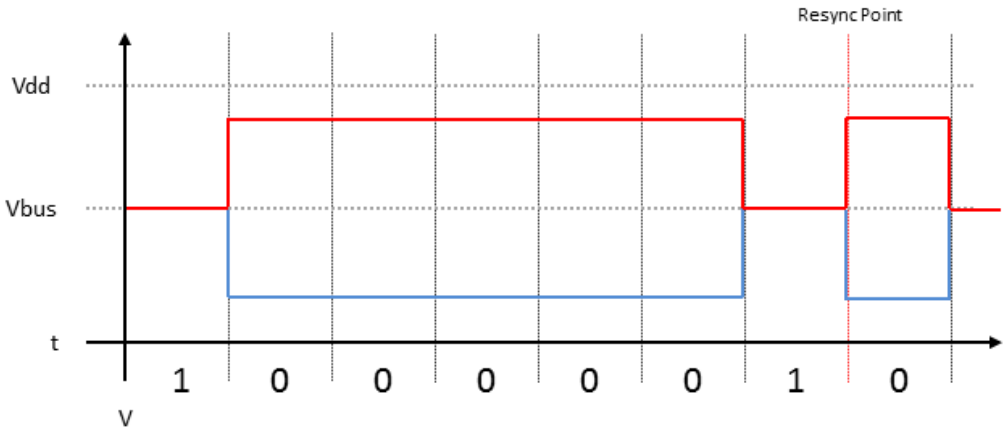

Figure 2-4 Generalized form of NRZ bus resynchronization

On a CAN bus, stuff bits are inserted following 5 recessive or 5 dominant bits. Stuff bits can be inserted at any time to ensure accurate clocking of the bus, except within the interframe space. CAN transmitters automatically insert stuff bits and CAN receivers automatically remove stuff bits. If more than 5 of the same type of bit are received in succession, CAN receivers will forcibly generate a bus error, interrupting transmission and forcing the transmitter to try again. [2]

Stuff bits require special consideration for CAN buses with tight timing requirements. Standard frames have a nominal maximum length of 108 bits, but this length can be as much as 130 bits if the pattern of bits within the frame causes a transmitter to generate the maximum number of stuff bits. Extended frames have a nominal maximum length of 128 bits, but with stuff bits, this length can be as high as 154 bits.

CAN Bus Performance Considerations

For periodic CAN frames performance can be parameterized by measuring the throughput, latency, and standard deviation of the frequency of transmission. One-shot frame transmission performance can be measured by examining the latency between transmission and reception.

CAN bus performance is dependent upon:

- Length of the physical bus
- Time quanta per bit
- Bus baud rate
- Contents of the frames (which may generate stuff bits)

When designing a new CAN bus, a system designer should first consider the length of the physical bus. Signal propagation through the CAN bus is dependent upon the speed of information in the conductor medium (usually near c = 3.0 x 10^8 m/s). The designer must also have some idea of the tolerances between different CAN node clock sources (and thus baud rate generators). Both of these bus parameters affect the value of the bus bit timing parameters.

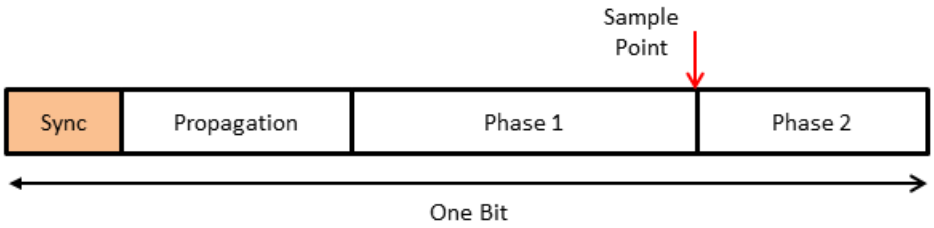

Figure 2-5 CAN bus time quanta parameters

Each bit on a CAN bus is made up of smaller components called **time quanta**. The CAN bus specification allows for anywhere from 9 to 25 time quanta per bit. Time quanta are associated with two general design rules: [1]

- Compensate for bus length by increasing the size of the propagation segment. Longer buses require more propagation time.
- Compensate for differences in local clock speeds by adjusting the sizes of the phase segments to place the **sample point**. A larger difference in clock rate requires larger phase segments.

The sample point is the position within each bit at which a CAN receiver will sample the bus to determine its state.

Once the bit timing parameters are determined, the bus baud rate can be set. The baud rate is also dependent on the physical length of the bus and the tolerances between different CAN node clock sources. A detailed explanation of how these parameters and characteristics interact is outside the scope of this document; however some sample calculations are presented in the TTCAN system design example in a later section.

Performance associated with individual CAN frames is dependent upon:

- The frame's intrinsic priority

- The size of the frame
- The periodicity of the frame
- Stuff bits within the frame

In general, frames with 0 data bytes, low periodicity, and high priority (low UID value) tend to be successfully received much more often than frames with a relatively large amount of data, high periodicity, or low priority (high UID value). [2]

Chapter 3

Time-Triggered CAN Network Traffic Management

A system designer must also decide whether to use any traffic management scheme on their CAN bus. According to the CAN2.0B specification the maximum allowable CAN bus speed is 1Mbit/sec. [2] Assuming the smallest possible frame size (44 bits + 3 interframe space bits), the highest achievable rate of transmission for a 1Mbit/sec bus is 21276.6 frames per second, disregarding stuff bits that may increase the size of each frame. On a bus with many nodes that transmit in a free-for-all (FFA) manner, the effective message rate may be much lower due to collisions.

Time-triggered CAN (TTCAN) is a form of time-based network traffic management in which each CAN transmitter is given a predetermined amount of time for transmission on the bus. This system guarantees that every transmitter will eventually be allowed to transmit, and that any active transmitter will not be interrupted by another transmitter. Each node keeps track of time with its own local transmission timer, and uses this timer to determine when to wait for an open transmission window or when to transmit.

TTCAN buses require at least one bus synchronization node. This node periodically sends a “sync” message which forces all other transmitters on the bus to resynchronize their local transmission clocks. Due to differences in clock source speeds, these local transmission clocks will eventually drift out of sync to a point where transmitters begin interrupting each other, generating bus collisions.

Designing a TTCAN system requires a system designer to understand basic CAN bus principles and also to investigate the timing tolerances and capabilities of their own CAN bus based system.

TTCAN Design Considerations

A system designer must first determine the maximum expected width of a CAN frame that will be used in their TTCAN system. For systems that use many differently formatted CAN frames, this is done easily by assuming the maximum theoretical width of a CAN frame (130 bits for standard frames, 154 bits for extended frames).

The maximum width of a CAN frame is then used to calculate the message transmission time, which is dependent upon baud rate. This is equal to the width of the message transmission window and forms the basis of the TTCAN schedule, which all nodes on the CAN bus must follow in order to transmit without interruption.

The message transmission window must be widened to compensate for differences in local clock sources. See the figure below for a graphical explanation. The amount of widening depends on the number of transmissions that take place between synchronization frames.

Finally, each transmission cycle must be padded to account for the node with the slowest transmission timer. This ensures that the slowest node has time to finish transmission before the synchronization message is transmitted again.

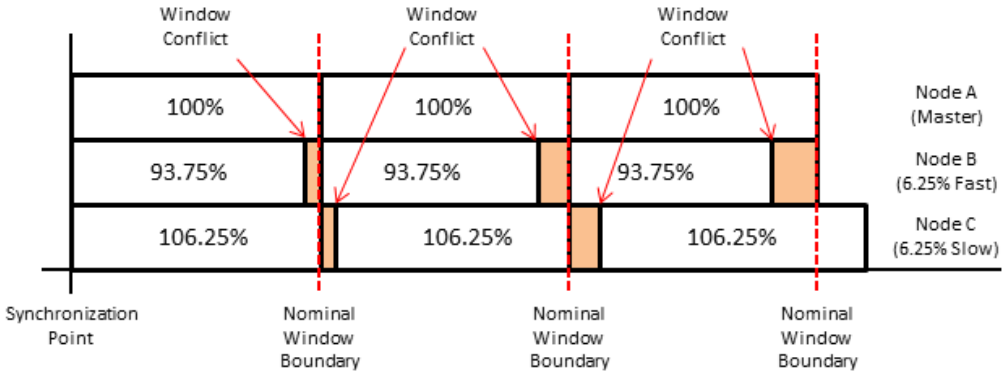

Figure 3-1 Desynchronization due to differences in the clock rates

Figure 3-1 shows the effects of desynchronization on three nodes. The master node (Node A) is responsible for synchronizing the transmission schedule, so its clock is used as a reference. Node B has a timer clock source that runs 6.25% faster than that of Node A, so its transmission windows start slightly early compared to Node A. Node C has a timer clock source that runs 6.25% slower than that of Node A, so its transmission windows start slightly later compared to Node A. As the transmission cycle progresses, the transmission windows become more and more desynchronized. Eventually transmission by one node is interrupted by that of another node and hardware arbitration must take over to ensure that transmission succeeds. In this case, transmission does not stop, but the efficiency gained by using a TTCAN transmission scheme is nullified.

TTCAN Design Method

1. Choose $\mathbf{L_{bit}}$, the maximum expected length of a CAN frame on the bus (in bits). This is dependent on whether the bus will use standard or extended frames. If the bus uses standard and extended frames together, use the maximum extended frame length. The absolute theoretical maximum possible frame lengths (accounting for the maximum possible number of stuff bits) for the two different frames types are given below.

Standard IDs: $L_{bit} = 130\ bits$

Extended IDs: $L_{bit} = 154\ bits$

2. Choose **TQ**, the number of time quanta per bit. This number can be found by adding the length of the bit propagation segment, phase 1 segment, phase 2 segment, and sync segment.
3. Choose $\mathbf{F_{baud}}$, the bus baud rate in bits per second.
4. Calculate the bus rate in terms of time quanta per second ($\mathbf{F_{TQ}}$) using $\mathbf{F_{baud}}$ and **TQ**.

$$F_{TQ} = F_{baud} \cdot TQ$$

5. Calculate the bus rate in terms of frames per second ($\mathbf{F_{frame}}$). The additional 3 bits in the denominator represent the 3 interfame space bits that must be transmitted on the bus before the next frame can begin.

$$F_{frame} = \frac{F_{TQ}}{TQ \cdot (L_{bit} + 3)}$$

6. Calculate the number of seconds per frame ($\mathbf{t_{frame}}$).

$$t_{frame} = {F_{frame}}^{-1}$$

7. Measure the oscillator speed difference between the bus master node and all bus slave nodes. If more than one master is being used, this process must be repeated for each master node. Oscillator speed may be directly measured with an edge counter attached to the clock source. Use this information to find $\mathbf{\Delta f_{max}}$ and $\mathbf{\Delta f_{min}}$, the maximum and minimum ratio between some slave node oscillator and the bus master oscillator.

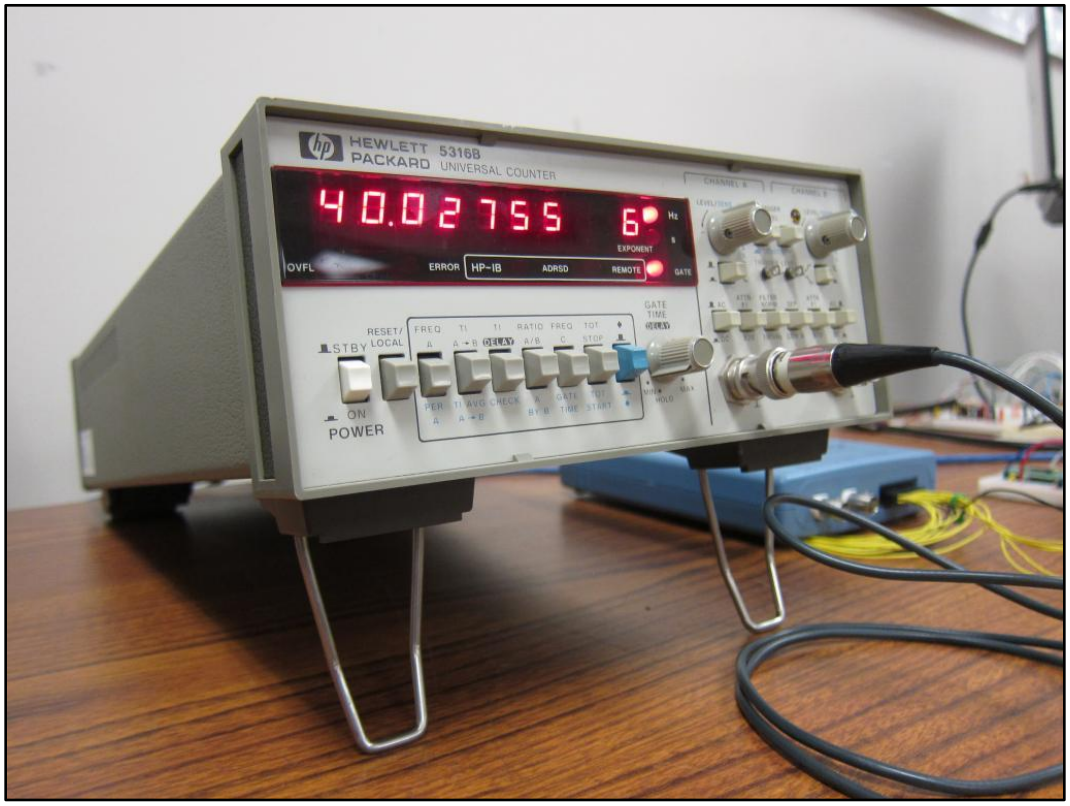

Figure 3-2 An edge counter in action

8. Calculate the frame overlap time, $\mathbf{t_{ol}}$.

$$t_{ol} = (\Delta f_{max} - \Delta f_{min}) \cdot t_{frame}$$

9. Choose the number of transmission windows per cycle, $\mathbf{N_{windows}}$.
10. Calculate the total cycle padding time, $\mathbf{t_{pad}}$.

$$t_{pad} = t_{ol} \cdot N_{windows}$$

11. Calculate the length of each transmission window, $\mathbf{t_{window}}$.

$$t_{window} = t_{frame} + t_{pad}$$

12. Calculate the length of each transmission cycle, $\mathbf{t_{cycle}}$.

$$t_{cycle} = t_{window} \cdot N_{windows}$$

13. Calculate the bus master resynchronization time, $\mathbf{t_{resync}}$.

$$t_{resync} = t_{cycle}\left(\frac{1}{1 + \Delta f_{min}} - 1\right)$$

14. Calculate the maximum deterministic frame transmission rate $\mathbf{F_{cycle}}$, which assumes transmission takes place once per cycle.

$$F_{cycle} = \left(N \cdot t_{frame} + t_{resync}\right)^{-1}$$

15. Calculate the data throughput rate for each message, $\mathbf{B_{throughput}}$, in bytes / sec. This value is dependent on the number of bytes that are transmitted per frame, $\mathbf{N_{bytes}}$. With byte stealing $\mathbf{N_{bytes}}$ can be any value between 0 and 9 bytes for standard length ID frames, or any value between 0 and 11 bytes for extended length ID frames..

$$B_{throughput} = F_{cycle} \cdot N_{bytes}$$

If $\mathbf{F_{cycle}}$ is equal to or higher than the minimum required sample rate, and $\mathbf{B_{throughput}}$ is equal to or higher than the minimum required throughput rate, the design

process is complete. The application designer may now create the TTCAN schedule such that each unique message is transmitted once per cycle at most.

If a higher sample rate or data throughput rate is required, proceed to the extra steps in the next section.

Designing for Multi-Cycle Transmission

Frames that are transmitted more than once per cycle must have their transmission windows distributed such that their effective periodicity has minimum variability. The figures below show how a message may be distributed within one transmission cycle to minimize variability and also to increase transmission rate by some integer factor $\mathbf{N_{message}}$. This integer factor is also called the **transmission multiplier.** These figures also show the minimum and maximum time between message transmission is affected by the amount of resynchronization time (represented in blue) at the end of a cycle.

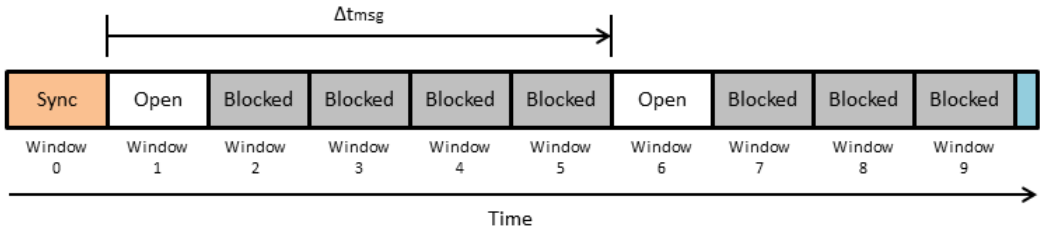

Figure 3-3 Determining minimum Δtmsg

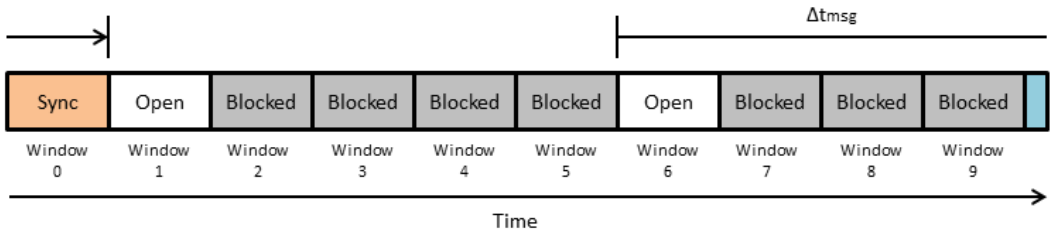

Figure 3-4 Determining maximum Δtmsg

Frequency of transmission and data throughput can be calculated using the procedure outlined below.

1. Calculate $\mathbf{F_{min}}$, $\mathbf{F_{max}}$, and $\mathbf{F_{avg}}$, which are the average, minimum, and maximum message transmission rate, respectively. $\mathbf{\Delta t_{msg}}$ represents the amount of time between message windows that transmit the same message.

$$F_{min} = max\left(\Delta t_{msg}\right)^{-1}$$

$$F_{max} = min\left(\Delta t_{msg}\right)^{-1}$$

$$F_{avg} = \frac{F_{max} + F_{min}}{2}$$

2. Calculate $\mathbf{B_{min}}$, $\mathbf{B_{max}}$, and $\mathbf{B_{avg}}$, which are the average, minimum, and maximum effective data transmission rates in bytes / sec, respectively. These values are dependent on the number of bytes that are transmitted per frame, $\mathbf{N_{bytes}}$. If using standard IDs, $\mathbf{N_{bytes}}$ can be any value between 0 and 9 bytes. If using extended IDs with byte stealing, $\mathbf{N_{bytes}}$ can be any value between 0 and 11 bytes.

$$B_{min} = N_{bytes} \cdot F_{min}$$

$$B_{max} = N_{bytes} \cdot F_{max}$$

$$B_{avg} = N_{bytes} \cdot F_{avg}$$

If a higher sample rate or data throughput rate is required, increase $\mathbf{N_{message}}$. If $\mathbf{N_{message}}$ cannot be increased, repeat the design process with a higher bus speed.

TTCAN System Design Example

This section contains a design example for a 500 kbps CAN2.0A bus. The resulting deterministic performance data is compared to the performance of a FFACAN bus running at the same speed in a later chapter.

1. Since all frames use standard length IDs, the maximum frame size (accounting for bit stuffing) is $\mathbf{L_{bit}}$ = 130 bits.
2. The bus is populated with nodes that use the dsPIC33EP256GP502 microcontroller (MCU). This MCU has a phase-locked-loop (PLL) clock source that is configured to run at 40 Mhz. To ensure that the bit sample point is nearest to the 87.5% position (as recommended by J1939 and CANopen) the number of time quanta per bit is set to 20 (**TQ** = 20). (Synchronization TQ = 1, propagation TQ = 8, phase 1 TQ = 8, phase 2 TQ = 3.)
3. The bus baud rate is 500,000 bits per second.
4. Calculate the bus rate in terms of time quanta per second:

$$F_{TQ} = F_{baud} \cdot TQ$$

$$F_{TQ} = (500000)(20)$$

$$F_{TQ} = 1x10^7\ TQ/s$$

5. Calculate the bus rate in terms of frames per second.

$$F_{frame} = \frac{F_{TQ}}{TQ \cdot (L_{bit} + 3)}$$

$$F_{frame} = \frac{1x10^7\ TQ/S}{\left(20\frac{TQ}{bit}\right)\left(130\frac{bits}{frame} + 3\frac{bits}{frame}\right)}$$

$$F_{frame} = 3759.4\ frames/s$$

6. Calculate the number of seconds per frame.

$$t_{frame} = {F_{frame}}^{-1}$$

$$t_{frame} = (3759.4\ frames/s)^{-1}$$

$$t_{frame} = 2.66\ x\ 10^{-4}\ s/frame$$

7. Measure the oscillator speed difference between the bus master node and all bus slave nodes. Results for this design example are shown in Table 3-1.

Table 3-1 Oscillator Speed Differences

Node Name	Frequency (Hz)	Ratio to Master	Diff from Master
GP0 (Master)	39979887	1.0000000	0.0000000
GP1	39959000	0.9994776	**-0.0005224**
GP2	40030806	1.0012736	**+0.0012736**
GP3	39975919	0.9999008	-0.0000992
GP4	39992614	1.0003183	+0.0003183
GP5	39972316	0.9998106	-0.0001894
GP6	40024292	1.0011107	+0.0011107
MC0	40018974	1.0009777	+0.0009777
MC1	39982050	1.0000541	+0.0000541
MC2	39960057	0.9995040	-0.0004960

The values of $\mathbf{\Delta f_{max}}$ (0.0012736) and $\mathbf{\Delta f_{min}}$ (-0.0005224) are highlighted in the table.

8. Calculate the overlap time, $\mathbf{t_{ol}}$.

$$t_{ol} = (\Delta f_{max} - \Delta f_{min}) \cdot t_{frame}$$

$$t_{ol} = (0.0012736 - (-0.0005224))(2.66\ x\ 10^{-4}\frac{s}{frame})$$

$$t_{ol} = 477.736x10^{-9}\ s$$

9. For this TTCAN design example, the number of windows ($\mathbf{N_{windows}}$) is 10.

10. Calculate the total cycle padding time, $\mathbf{t_{pad}}$.

$$t_{pad} = t_{ol} \cdot N_{windows}$$

$$t_{pad} = (477.736\ x\ 10^{-9}\ s) \cdot 10$$

$$t_{pad} = (4.77736\ x\ 10^{-6}\ s)$$

11. Calculate the length of each transmission window, $\mathbf{t_{window}}$.

$$t_{window} = t_{frame} + t_{pad}$$

$$t_{window} = (2.66\ x\ 10^{-4}\ s) + (4.77736\ x\ 10^{-6} s)$$

$$t_{window} = 270.777\ x\ 10^{-6}\ s$$

12. Calculate the length of each transmission cycle, $\mathbf{t_{cycle}}$.

$$t_{cycle} = t_{window} \cdot N_{windows}$$

$$t_{cycle} = (270.777\ x\ 10^{-6}\ s)(10)$$

$$t_{cycle} = 2.70777\ x\ 10^{-3} s$$

13. Calculate the bus master resynchronization time, $\mathbf{t_{resync}}$.

$$t_{resync} = t_{cycle}\left(\frac{1}{1 + \Delta f_{min}} - 1\right)$$

$$t_{resync} = (2.70777\ x\ 10^{-3}\ s)\left(\frac{1}{1 + (-0.0005224)} - 1\right)$$

$$t_{resync} = 1.41528\ x\ 10^{-6} s$$

14. Calculate the deterministic maximum frame transmission rate, $\mathbf{F_{cycle}}$.

$$F_{cycle} = \left(N_{windows} \cdot t_{frame} + t_{resync}\right)^{-1}$$

$$F_{cycle} = [(10)(2.66\ x\ 10^{-4}\ s) + (1.41528\ x\ 10^{-6}\ s)]^{-1}$$

$$F_{cycle} = 375.74\ Hz$$

15. Calculate the data throughput rate, $\mathbf{B_{throughput}}$. This example design assumes that each frame transmits 8 bytes, so $\mathbf{N_{bytes}}$ = 8.

$$B_{throughput} = F_{cycle} \cdot N_{bytes}$$

$$B_{throughput} = (375.74\ Hz)(8\ bytes)$$

$$B_{throughput} = 3005.92\ bytes/second$$

At this point, two design characteristics are known: the maximum sample rate for one message is 375.74 Hz, and the maximum possible throughput for one message is 3005.92 bytes per second.

Designing for Multi-Cycle Transmission (2x)

The figure below shows how frames that are transmitted twice per cycle may be arranged to minimize the variability in transmission and data rate.

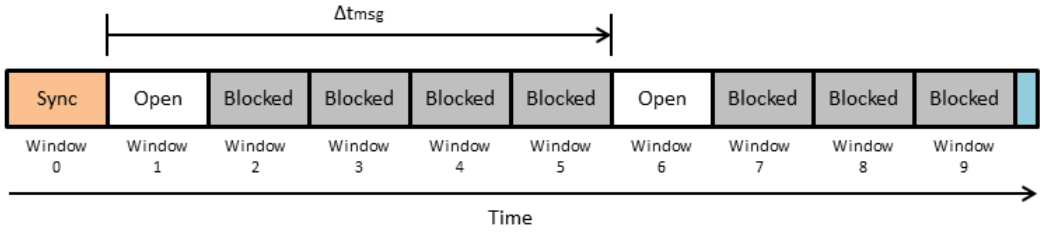

Figure 3-5 Determining minimum Δtmsg (2x transmission)

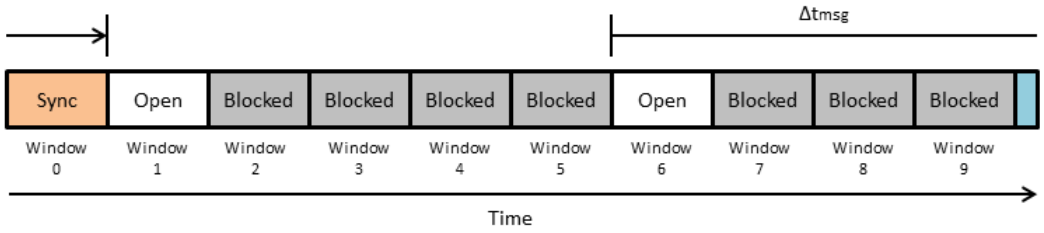

Figure 3-6 Determining maximum Δtmsg (2x transmission)

1. Calculate $\mathbf{F_{avg}}$, $\mathbf{F_{min}}$, and $\mathbf{F_{max}}$ for this transmission multiplier.

$\mathbf{\Delta t_{msg}}$ is found by measuring the amount of time between two message windows that transmit the same message. For a 2x transmission multiplier, the messages should be uniformly distributed such that they are 4 messages apart to minimize variability (see figure above). The total time elapsed between the first message and the second message in this figure is:

$$\Delta t_{msg} = (N_{between} + 1) \cdot t_{window}$$

$$\Delta t_{msg} = (4 + 1)(270.777\ x\ 10^{-6}\ s)$$

$$\Delta t_{msg} = 1.35389\ x\ 10^{-3}\ s$$

First, find F_{min} (accounting for resync time):

$$F_{min} = max\left(\Delta t_{msg}\right)^{-1}$$

$$F_{min} = \left((N_{between} + 1) \cdot t_{window} + t_{resync}\right)^{-1}$$

$$F_{min} = (1.35389\ x\ 10^{-3}s + 1.41528\ x\ 10^{-6}s)^{-1}$$

$$F_{min} = 737.841\ Hz$$

Then find F_{max}:

$$F_{max} = min\left(\Delta t_{msg}\right)^{-1}$$

$$F_{max} = \left((N_{between} + 1) \cdot t_{window}\right)^{-1}$$

$$F_{max} = (1.35389\ x\ 10^{-3}s)^{-1}$$

$$F_{max} = 738.612\ Hz$$

Then find $\mathbf{F_{avg}}$:

$$F_{avg} = \frac{F_{max} + F_{min}}{2}$$

$$F_{avg} = \frac{738.612\ Hz + 737.841\ Hz}{2}$$

$$F_{avg} = 738.227\ Hz$$

2. Calculate $\mathbf{B_{min}}$, $\mathbf{B_{max}}$, and $\mathbf{B_{avg}}$, which are the average, minimum, and maximum effective data transmission rates in bytes / sec, respectively. Recall from above that $\mathbf{N_{bytes}} = 8$.

First, calculate $\mathbf{B_{min}}$:

$$B_{min} = N_{bytes} \cdot F_{min}$$

$$B_{min} = (8\ bytes)(737.841\ Hz)$$

$$B_{min} = 5902.73\ bytes/sec$$

Then find $\mathbf{B_{max}}$:

$$B_{max} = N_{bytes} \cdot F_{max}$$

$$B_{max} = (8\ bytes)(738.612\ Hz)$$

$$B_{max} = 5908.90\ bytes/sec$$

Then find $\mathbf{B_{avg}}$:

$$B_{avg} = N_{bytes} \cdot F_{avg}$$

$$B_{avg} = (8\ bytes)(738.227\ Hz)$$

$$B_{avg} = 5905.82\ bytes/sec$$

The application designer may now decide if these message and data transmission rates are acceptable.

Designing for Multi-Cycle Transmission (5x)

Parameters for a 5x transmission multiplier scheme are summarized in Table 3-2 below.

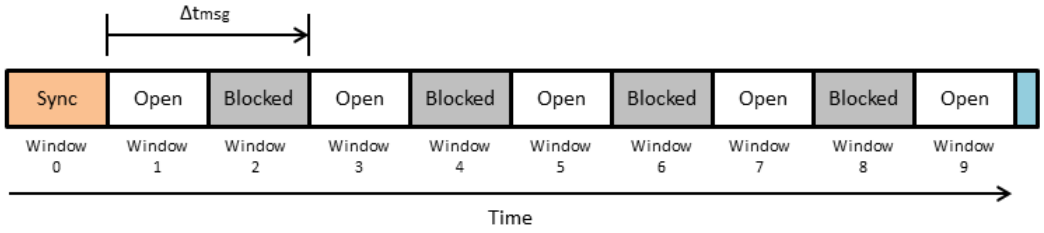

Figure 3-7 Determining minimum Δtmsg (5x transmission)

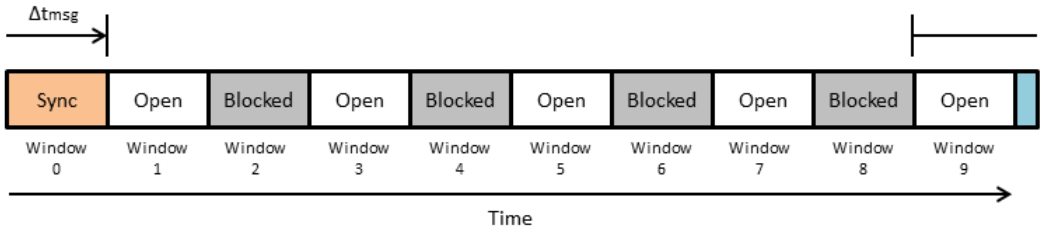

Figure 3-8 Determining maximum Δtmsg (5x transmission)

Table 3-2 Parameters for a 5x Transmission Multiplier Scheme

Parameter	Unit	Value
$\min(\Delta t_{msg})$	s	541.554×10^{-6}
$\max(\Delta t_{msg})$	s	542.969×10^{-6}
F_{min}	Hz	1841.73
F_{max}	Hz	1846.54
F_{avg}	Hz	1844.14
B_{min}	bytes /sec	14733.8
B_{max}	bytes /sec	14772.3
B_{avg}	bytes /sec	14753.1

Designing for Multi-Cycle Transmission (8x)

Parameters for a 8x transmission multiplier scheme are summarized in Table 3-3 below. Since $N_{windows}$ is not a least common multiple of the transmission multiplier, the transmission and data rate variability is greatly increased relative to transmission multipliers that divide evenly into $N_{windows}$.

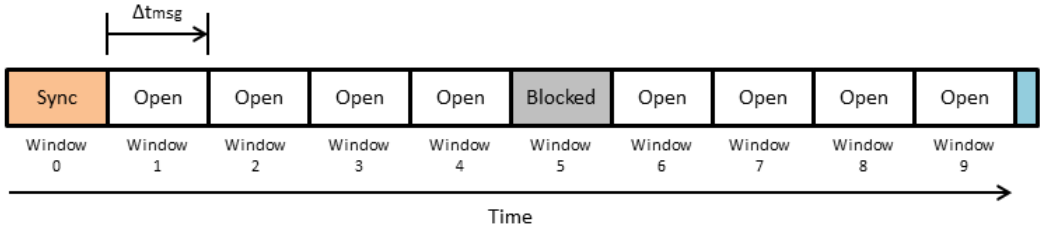

Figure 3-9 Determining minimum Δtmsg (8x transmission)

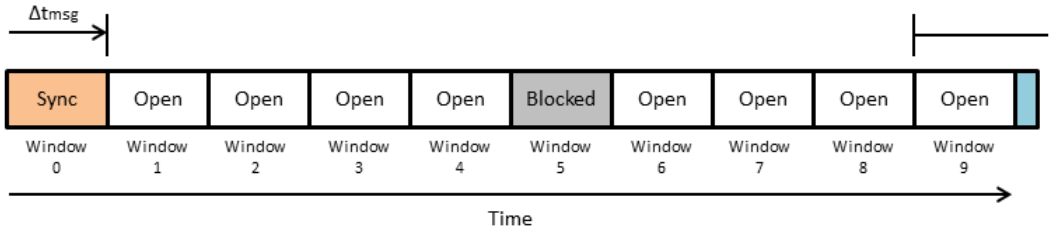

Figure 3-10 Determining maximum Δtmsg (8x transmission)

Table 3-3 Parameters for a 8x Transmission Multiplier Scheme

Parameter	Unit	Value
$\min(\Delta t_{msg})$	s	270.777×10^{-6}
$\max(\Delta t_{msg})$	s	542.969×10^{-6}
F_{min}	Hz	1841.72
F_{max}	Hz	3693.08
F_{avg}	Hz	2767.40
B_{min}	bytes /sec	14733.76
B_{max}	bytes /sec	29544.64
B_{avg}	bytes /sec	22139.20

Designing for Multi-Cycle Transmission (9x)

Since $\mathbf{N_{windows}}$ = 10, the maximum number of times a message may be transmitted per cycle is 9, accounting for the window taken up by the synchronization message.

Parameters for a 9x transmission multiplier scheme are summarized in Table 3-4 below. Note the high variability between the minimum and maximum transmission and data rates.

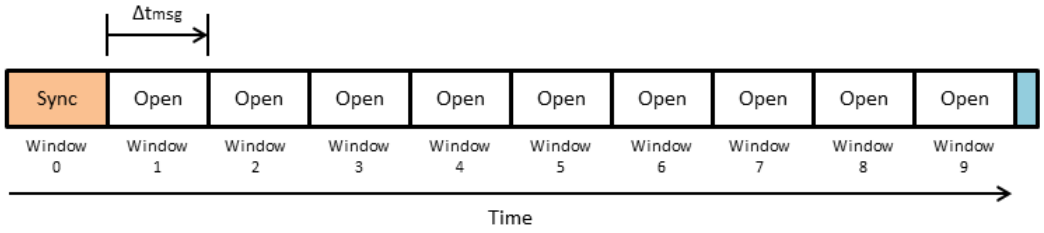

Figure 3-11 Determining minimum Δtmsg (9x transmission)

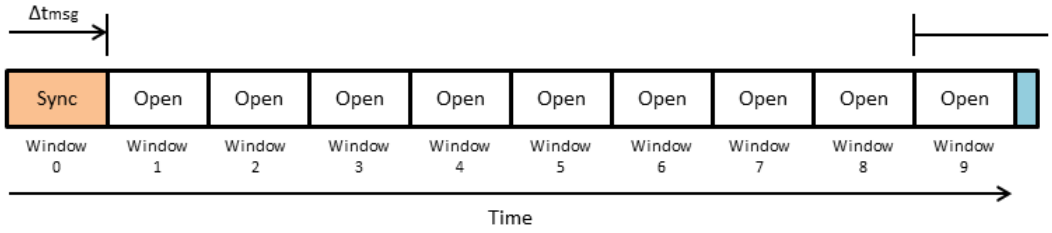

Figure 3-12 Determining maximum Δtmsg (9x transmission)

Table 3-4 Parameters for an 9x Transmission Multiplier Scheme

Parameter	Unit	Value
$\min(\Delta t_{msg})$	s	270.777×10^{-6}
$\max(\Delta t_{msg})$	s	542.969×10^{-6}
F_{min}	Hz	1841.72
F_{max}	Hz	3693.08
F_{avg}	Hz	2767.40
B_{min}	bytes/sec	14733.76
B_{max}	bytes /sec	29544.60
B_{avg}	bytes /sec	22139.20

Summary of Results

Frame and data transmission rates for all transmission multipliers in the design example are given below in Tables 3-5 and 3-6.

Table 3-5 Min/Max/Avg Frame Transmission Rates for all Transmission Multipliers

Transmission Multiplier	Minimum F (Hz)	Maximum F (Hz)	Average F (Hz)	Variability (Hz)
1	375.74	375.74	375.74	0
2	737.84	738.61	738.23	0.77
5	1841.73	1846.54	1844.14	4.81
8	1841.72	3693.08	2767.40	1851.36
9	1841.72	3693.08	2767.40	1851.36

Table 3-6 Min/Max/Avg Data Transmission Rates for all Transmission Multipliers

Transmission Multipler	Minimum F (bytes/sec)	Maximum F (bytes/sec)	Average F (bytes/sec)	Variability (bytes/sec)
1	3005.92	3005.92	3005.92	0
2	5902.72	5908.88	5905.84	6.16
5	14733.84	14772.32	14753.12	38.48
8	14733.76	29544.64	22139.2	14810.88
9	14733.76	29544.64	22139.2	14810.88

Chapter 4

Comparison of TTCAN to FFACAN

This section contains a comparison between a TTCAN design and an FFACAN design with similar bus characteristics. Both buses run at 500 kbits / sec and use 20 TQ per bit. (Sync TQ = 1, Propagation TQ = 8, Phase 1 TQ = 8, Phase 2 TQ = 3.) Every frame on the bus uses a standard length ID field and contains 8 data bytes. Transmission for each unique frame was attempted at their nominal TTCAN rates.

The data for this comparison was gathered from a CAN bus with 11 separate nodes with designations GP0 through GP6 and MC0 through MC3. Node GP0 was configured to transmit a frame with identifier 0x50F with some periodicity. Node GP1 was configured to receive frames transmitted from GP0. Upon successful reception, node GP1 momentarily set one of its GPIO pins to a digital logic “high” level so that an oscilloscope could measure the actual message periodicity and standard deviation of the actual periodicity. Actual transmission periodicity is dependent upon the traffic conditions on the bus. To ensure that hardware arbitration did not artificially skew the results, GP0 transmitted a message with a lower priority (higher value in the ID field) than any other node on the bus.

The TTCAN design example in Chapter 3 uses one window for the synchronization message out of 10 total windows ($N_{windows}$ = 10). The synchronization message does not transmit any application specific data and thus contributes to transmission overhead, which decreases the effective transmission and data rates. By comparison, a FFACAN bus does not require any such non-application specific synchronization, allowing more time on the bus for other nodes to transmit. To represent

this difference the FFACAN bus shown below uses 9 transmitters (as opposed to 10 for a TTCAN bus).

Table 4-1 summarizes the role of each node on this bus. GP1 is a dedicated receiver node and does not transmit any messages. This is done so that node GP1 does not spend any time attempting to transmit messages which could potentially affect interrupt service latency.

Table 4-1 Node Descriptions for FFACAN Bus

Node Name	Description	Transmit ID	Receive ID
GP0	Transmitter	0x50F	-
GP1	Receiver	-	0x50F
GP2	Traffic Generator	0x502	-
GP3	Traffic Generator	0x503	-
GP4	Traffic Generator	0x504	-
GP5	Traffic Generator	0x505	-
GP6	Traffic Generator	0x506	-
MC0	Traffic Generator	0x507	-
MC1	Traffic Generator	0x508	-
MC2	Traffic Generator	0x509	-

FFACAN transmission periodicity values are based on the values found in the TTCAN design example that can be found in Chapter 3. For convenience, they are summarized here:

Table 4-2 TTCAN Transmission & Data Rates

Transmission Multiplier	Transmission Rate (Hz)	Data Rate (bytes / sec)
1	375.740	3005.92
2	738.227	5905.82
5	1844.14	14753.12
8	2767.40	22139.20
9	2767.40	22139.20

Baseline Values

Before running any tests, nodes GP0 and GP1 were isolated on an empty bus. Figure 4-1 shows the bus configuration for this test. The transmitter is highlighted in green, the receiver is highlighted in orange, and the deactivated nodes are highlighted in grey.

Figure 4-1 Bus configuration for the baseline test

Messages were transmitted from GP0 at various rates. Instrumentation was attached to GP1 to measure the rate at which the messages were received. The results (summarized in Table 4-3) show that neither microcontroller CPU was overloaded by this task, even at the highest transmission rate.

The actual transmission rates are slightly higher than the requested transmission rates due to differences in the measured and actual oscillator speeds on the master node.

Table 4-3 Requested vs Actual Transmission Rates, Baseline

Transmission Multiplier	Requested Transmission Rate (Hz)	Actual Reception Rate (Hz)	Standard Deviation (Hz)
1	375.740	375.6	2.682
2	738.227	727.1	26.53
5	1844.14	1674.0	145.3
8	2767.40	2644.0	22.89

Comparing the 1x Transmission Multiplier

This test simulates one transmission event per TTCAN cycle per node (375.740Hz on a FFACAN bus).

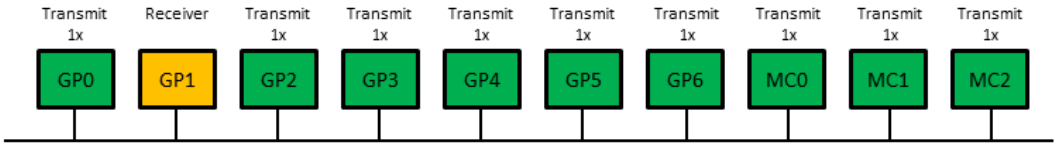

Figure 4-2 Bus configuration for the 1x transmission multiplier test

Table 4-4 1x Transmission Multiplier Test Results

Transmission Multiplier	1
Requested Transmission Rate (Hz)	375.740
Expected Data Rate (bytes / sec)	3005.92
Actual Reception Rate (Hz)	375.6
Actual Data Rate (bytes / sec)	3008.066
Transmission Standard Deviation (Hz)	2.682

Comparing the 2x Transmission Multiplier

This test simulates one transmission event per TTCAN cycle per node (375.740 Hz on a FFACAN bus) for all nodes except node GP0, which transmits twice per TTCAN cycle (738.227 Hz on a FFACAN bus).

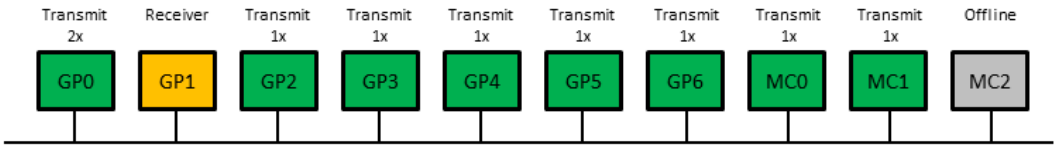

Figure 4-3 Bus configuration for the 2x transmission multiplier test

Table 4-5 Transmission Multiplier Test Results

Transmission Multiplier	2
Requested Transmission Rate (Hz)	738.227
Expected Data Rate (bytes / sec)	5905.82
Actual Reception Rate (Hz)	727.1
Actual Data Rate (bytes / sec)	5394.824
Transmission Standard Deviation (Hz)	26.53

Comparing the 5x Transmission Multiplier

This test simulates one transmission event per TTCAN cycle per node (375.740 Hz on a FFACAN bus) for all nodes except node GP0, which transmits 5 times per TTCAN cycle (1844.14 Hz on a FFACAN bus).

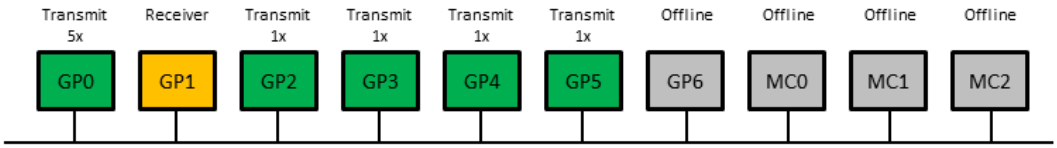

Figure 4-4 Bus configuration for the 5x transmission multiplier test

Table 4-6 5x Transmission Multiplier Test Results

Transmission Multiplier	5
Requested Transmission Rate (Hz)	1844.14
Expected Data Rate (bytes / sec)	14753.12
Actual Reception Rate (Hz)	1674.0
Actual Data Rate (bytes / sec)	13455.032
Transmission Standard Deviation (Hz)	145.3

Comparing the 8x Transmission Multiplier

This test simulates one transmission event per TTCAN cycle per node (375.740 Hz on a FFACAN bus) for all nodes except node GP0, which transmits 8 times per TTCAN cycle (2767.40 Hz on a FFACAN bus).

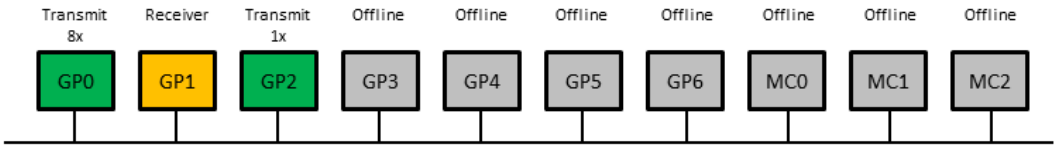

Figure 4-5 Bus configuration for the 8x transmission multiplier test

Table 4-7 8x Transmission Multiplier Test Results

Transmission Multiplier	8
Requested Transmission Rate (Hz)	2767.40
Expected Data Rate (bytes / sec)	22139.20
Actual Reception Rate (Hz)	2644.0
Actual Data Rate (bytes / sec)	21173.361
Transmission Standard Deviation (Hz)	22.89

Comparing the 9x Transmission Multiplier

The test simulates nine transmission events per TTCAN cycle (2830.54Hz on a FFACAN bus) for node GP0. All other transmitters are disabled.

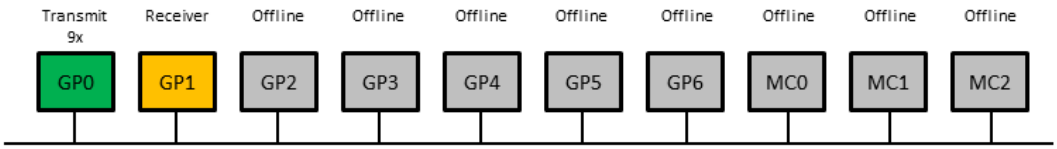

Figure 4-6 Bus configuration for the 9x transmission multiplier test

Table 4-8 9x Transmission Multiplier Test Results

Transmission Multiplier	9
Requested Transmission Rate (Hz)	2767.40
Expected Data Rate (bytes / sec)	22139.20
Actual Reception Rate (Hz)	2770.0
Actual Data Rate (bytes / sec)	22157.369
Transmission Standard Deviation (Hz)	0.1326

Summary of Results

Table 4-9 Summary of Requested vs Actual Transmission Rates

Transmission Multiplier	Requested Transmission Rate (Hz)	Actual Transmission Rate (Hz)	Standard Deviation (Hz)
1	374.74	374.60	2.682
2	738.23	727.10	26.53
5	1844.14	1674.0	145.3
8	2767.40	2644.0	22.89
9	2767.40	2770.0	0.1326

Chapter 5

CPU Usage Calculation

Microcontrollers and microprocessors spend all of their running time executing *something*, be it an interrupt service routine, an operating system task, or a simple main() loop with no context switching. However constantly executing some block of code does not mean a system is doing something useful. Systems may spend their time doing nothing (sometimes referred to as **spinning** or **idling**) while waiting for an interrupt to occur or for an OS task to wake up. Some systems take advantage of this time by disabling portions of their hardware to save power until the device or module is needed again.

Generally, systems that spend little or none of their time idling are overloaded. A system designer may find themselves alarmed to see that some or none of their tasks or interrupts are executing at the expected speed and then come to the same realization. The question then becomes “What is my system doing that is causing the overload?” followed by “How do I figure that out?”

CPU usage calculations are not completely accurate since they are typically done by the same system that is running the software being analyzed. (CPU usage calculations use CPU time, thus skewing the results.) This document presents two methods through which CPU usage may be determined with minimal interference to normal system execution.

The implementation and accuracy of CPU usage calculations depend on the system architecture. This document presents CPU measurement techniques for four possible architectures:

- Nested interrupts disabled, no operating system

- Nested interrupts enabled, no operating system
- Nested interrupts disabled, with an operating system
- Nested interrupts enabled, with an operating system

The analysis for each of these architectures does not make any special consideration for the operating system outside of assuming that the scheduler and context switching routines are given the highest interrupt priority. Thus, architecture classification may be reduced to:

- Systems without nested interrupts
- Systems with nested interrupts

Non-Nested vs Nested Hardware Interrupts

Many modern microcontrollers include some form of an interrupt controller for managing hardware interrupts. Interrupts are serviced by interrupt service routines (ISRs), special functions that are invoked by the CPU after an interrupt is asserted. The interrupt controller must also decide which interrupt to service when two or more interrupts are asserted. This is usually determined by examining an interrupt's priority, which is often user-configurable; if two interrupts have the same user-configured priority, they are executed in the order of their **intrinsic** priority, which is distinct from an interrupt's normal execution priority. How the intrinsic priority affects order of execution is dependent on the microcontroller's hardware architecture. For example, the Texas Instruments TM4C1294NCPDT microcontroller (see figure 5-1) uses the interrupt vector number as the intrinsic priority. Higher vector numbers are executed last. [3]

Vector Number	Interrupt Number (Bit in Interrupt Registers)	Vector Address or Offset	Description
0-15	-	0x0000.0000 - 0x0000.003C	Processor exceptions
16	0	0x0000.0040	GPIO Port A
17	1	0x0000.0044	GPIO Port B
18	2	0x0000.0048	GPIO Port C
19	3	0x0000.004C	GPIO Port D
20	4	0x0000.0050	GPIO Port E
21	5	0x0000.0054	UART0

Figure 5-1 Interrupt vectors for a TI TM4C1294NCPDT microcontroller [3]

Some microcontroller architectures have support for **interrupt nesting**, an execution scheme in which interrupts with lower priority may be interrupted by interrupts with a higher priority. This requires special consideration for application designers, especially when adding instrumentation for determining the amount of time the CPU spends servicing an interrupt.

Figures showing the flow of execution for systems with and without nested interrupt and shown in the sections below.

CPU Usage Calculation – Method 1

The first method assumes that the number of interrupts is equal to or less than the number of available GPIO pins, plus one pin for a main loop (no operating system) or idle task (with an operating system). This method is most accurate since it accounts for the near-exact amount of time that an interrupt may spend executing, but also for time used by aperiodic interrupts.

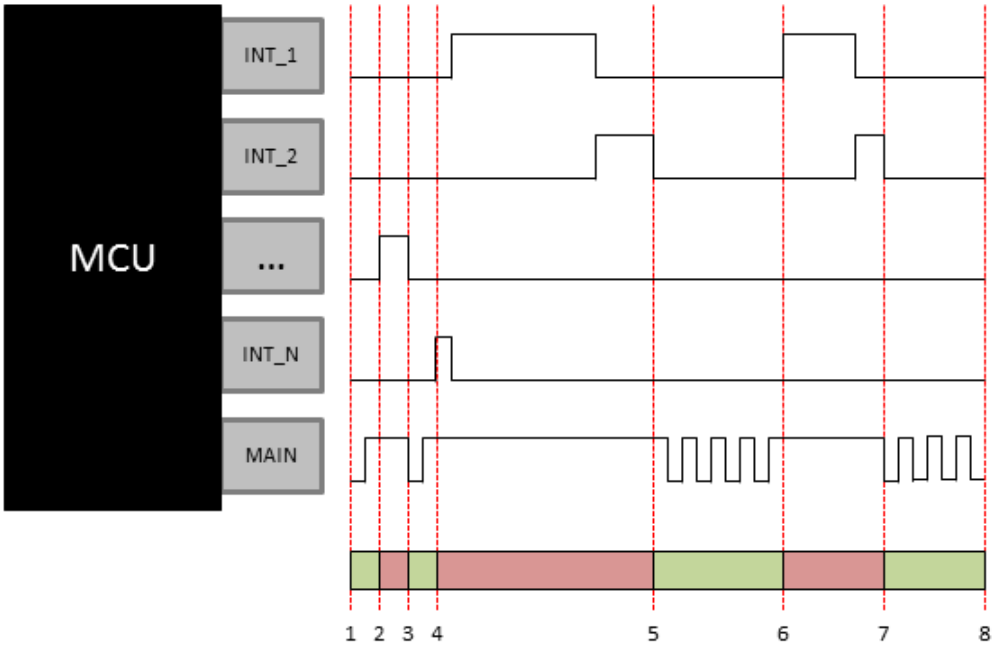

Figure 5-2 Non-nested interrupt instrumentation output

Figure 5-2 shows an example of some output from a microcontroller that has had instrumentation applied to its interrupt service routines. The sections highlighted in green (lighter) represent CPU time spent running the main loop, and the sections highlighted in red (darker) represent CPU time spent processing an interrupt service routine. The flow of this output is described below.

1. Initially, the microcontroller is running the main loop. No interrupts are active.
2. An interrupt is asserted and the CPU begins processing the interrupt service routine.
3. The interrupt service routine completes execution. The CPU resumes processing the main loop.
4. Another interrupt is asserted. While the associated interrupt service routine is executing, two more interrupts are asserted. Since nested interrupt handling is disabled, they must wait until the CPU core becomes available. They are then processed in order according to their priority. In the meantime the main loop stalls; the application designer must decide if a stall of this length is acceptable.
5. The last interrupt service routine finishes execution. The CPU resumes processing the main loop.
6. Two more interrupts are asserted. As before, the CPU executes their interrupt service routines in order according to their priorities.
7. The last interrupt service routine finishes execution. The CPU resumes processing the main loop.

By analyzing the duty cycle of the waveforms generated by each pin, a system designer can reliably determine the amount of CPU time spent executing an interrupt service routine or the main loop and then decide whether the system is processing its workload quickly enough.

CPU Usage Calculation – Method 2

For systems that do not support or do not use nested interrupts, the second method can be implemented with one GPIO pin, but requires that a system designer interpret the results after the fact and is also less accurate as it only measures the average time spent by each interrupt, not accounting for aperiodicity. The second method cannot be reliably implemented on systems that used nested interrupts.

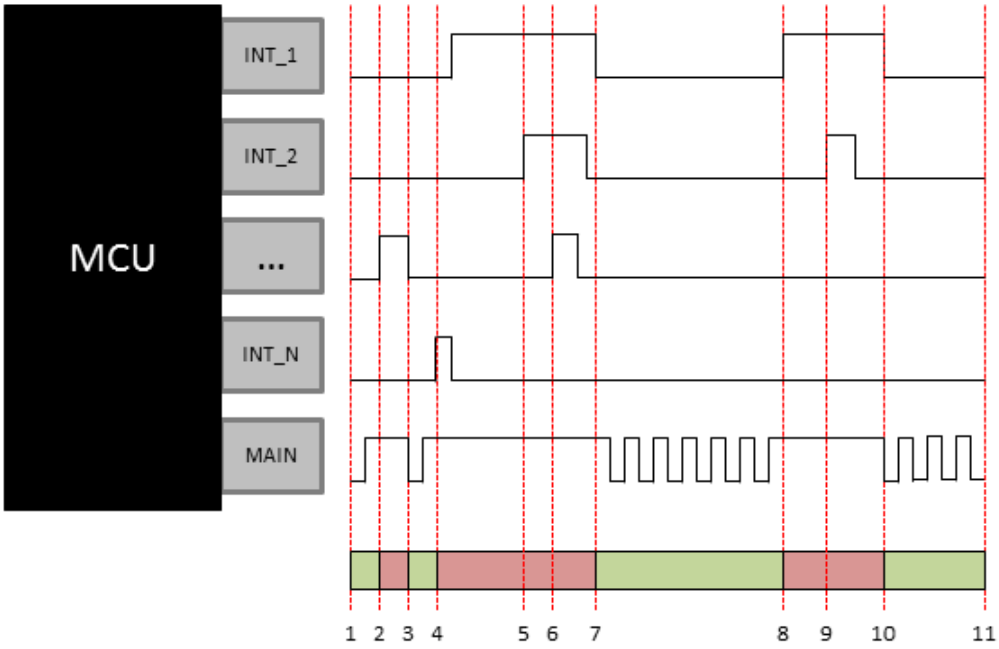

Figure 5-3 Nested interrupt instrumentation output

Figure 5-3 shows an example of some output from a microcontroller that has had instrumentation applied to its interrupt service routines, but with nesting enabled (as opposed to figure 5-2 in the previous section). These waveforms require interpretation to determine the actual amount of time the CPU uses to execute each interrupt service routine. The flow of this output is described below.

1. Initially, the microcontroller is running the main loop. No interrupts are active.
2. An interrupt is asserted and the CPU begins processing the interrupt service routine (ISR).
3. The ISR completes execution. The CPU resumes processing the main loop.
4. Another interrupt is asserted. In happenstance, the interrupt finishes processing just as another interrupt is asserted. At this point, the CPU only has one active interrupt.
5. Another higher priority interrupt is asserted, and the CPU suspends the lower priority interrupt. At this point the CPU has two active interrupts.
6. An even higher priority interrupt interrupts the second interrupt, forcing the CPU to nest even further. At this point the CPU has three active interrupts. Each nested ISR is processed in a last-in, first-out manner. Note how the width of the pulses associated with each interrupt could be misinterpreted as the total time that the CPU spent processing an ISR.
7. The lowest priority interrupt finishes execution. The CPU resumes processing the main loop.
8. An interrupt is asserted and the CPU begins processing the associated ISR.
9. A higher priority interrupt is asserted and nests inside the first. Again, the ISRs are processed in a last-in, first-out manner.
10. The CPU finishes executing all ISRs and resumes processing the main loop.

On a system with nested interrupts supported and enabled, the width of a pulse associated with each ISR may be misinterpreted as the total time that particular ISR spent executing. Figure 5-4 shows how to properly calculate the amount of time each ISR spends executing.

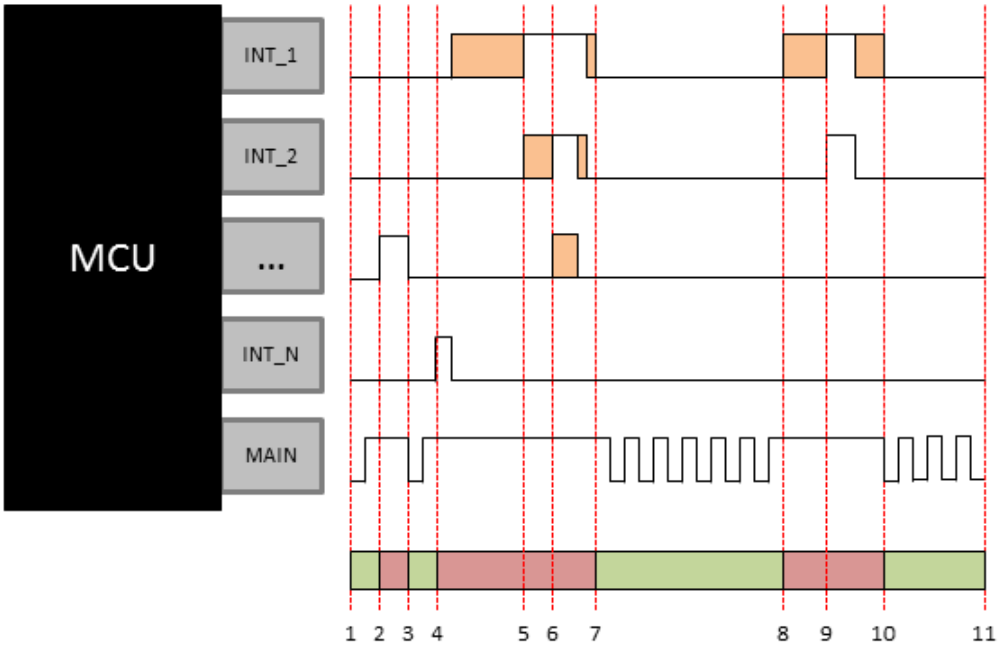

Figure 5-4 Nested interrupt instrumentation output with highlights

Figure 5-4 is a copy of Figure 5-3 shown above, but with actual CPU time highlighted within each pulse. It is easy to see how the highest priority interrupt has the shortest pulse width within a group of nested interrupts (see the interrupts between points 4 and 7).

CPU Usage Measurement Implementation Methodology

Method 1, for Non-Nested Interrupts

This method assumes that there are at least *n* + 1 GPIO pins available for a system with *n* interrupts.

1. At the entry point of each interrupt service routine, add a routine that toggles the interrupt's corresponding GPIO pin to a digital logic "high" (1). At the exit point, toggle the pin back to a digital logical "low" (0).
2. Add a routine that toggles a GPIO pin after an iteration of the main loop.
3. Monitor the state of each pin with an external device (such as an oscilloscope). Use snapshots of this data to determine the CPU usage of each interrupt and the speed of execution of the main loop.

Method 2, for Non-Nested Interrupts

This method assumes that there is only one GPIO pin available.

1. Instrument the first interrupt service routine as described in method 1 above.
2. Monitor the state of this pin with an external device over a period of time sufficient to determine the interrupt's average execution time.
3. Repeat steps 1 and 2 for all remaining interrupt service routines.
4. Instrument the main loop as described in method 1 above.
5. Monitor the state of this main loop pin and determine the speed of execution.

The system is overloaded if the main loop fails to execute at a sufficient speed, or if the main loop never executes due to blocking by interrupts that must constantly be serviced.

Method 1, for Nested Interrupts

This method assumes that there are at least $n + 1$ GPIO pins available for a system with n interrupts.

1. At the entry point of each interrupt service routine, add a routine that toggles the interrupt's corresponding GPIO pin to a digital logic "high" (1). At the exit point, toggle the pin back to a digital logical "low" (0).
2. Add a routine that toggles a GPIO pin after an iteration of the main loop.
3. Monitor the state of each pin with an external device (such as an oscilloscope).

The state of the pins must be considered when calculating CPU usage:

- The length of isolated pulses (indicating non-nested interrupt execution) may be used as-is for CPU usage calculations.
- Nested interrupts manifest as pulses that begin after another pulses has begun, and end before another pulse ends. In this case, the execution time of each interrupt is determined by subtracting the shortest execution time from the next shortest execution time.

Method 2, for Nested Interrupts

This method cannot be implemented on systems that use nested interrupts, as there is no way to differentiate between which interrupt toggled the instrumentation pin high or low. Systems with nested interrupts must either temporarily disable interrupt nesting, or use multiple GPIO pins.

CPU Usage Measurement Implementation Example

This section contains an example of each of the methods explained in the previous section. An example system was created using a dsPIC33EP256GP502 microcontroller with a 40 MHz system clock. Pin states were monitored with a multi-channel digital oscilloscope. Hardware timers were used to generate periodic interrupts. The main loop ran at a rate of 200 kHz.

Parameters and Expected Results

Four separate hardware timers were used to periodically generate interrupts. The rate of each timer is listed in the table below. Numbers in the “delay” column represent the amount of time that the ISR is active, in microseconds. Timers were also assigned a fixed interrupt service priority. The rates and delays for each timer interrupt are based on prime numbers to ensure that the interrupts would eventually overlap and conflict with each other, forcing the interrupt controller to execute the ISRs in order of priority. Expected CPU time was calculated by dividing the “delay” by “period”. This table suggests that the total CPU time spent servicing interrupts should be about 40.345%.

Table 5-1 Timer Interrupt Configuration

Timer	Period (µS)	Rate (Hz)	Delay (µS)	Priority	CPU Time (%)
2	2017	496.032	199	2	9.866
3	1013	987.167	101	3	9.970
4	233	4291.85	23	4	9.871
5	47	21276.6	5	5	10.638

Method 1, for Non-Nested Interrupts

Since the example system has four interrupts and a main loop, this method requires 5 free GPIO pins on the microcontroller. With instrumentation in place, the oscilloscope produced the waveform shown in Figure 5-5. The main loop toggle and timer signals are shown in order from top to bottom (main loop at the top, timer 2 second from top, and so on).

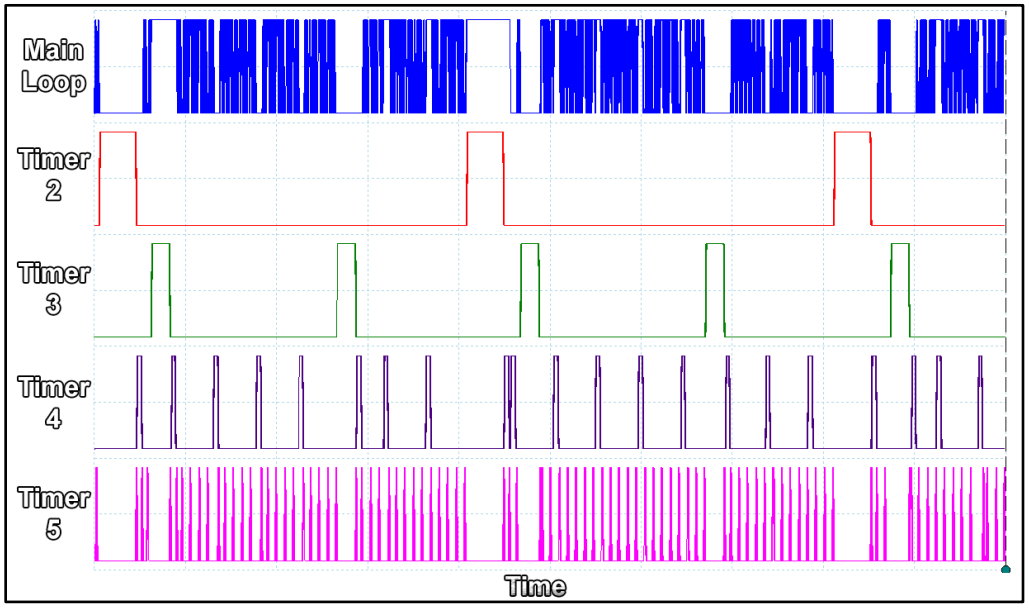

Figure 5-5 Oscilloscope Graphical Output

The oscilloscope's built-in measurement functions were used to find the duty cycle of each interrupt and the frequency of the main loop. These results are summarized in Table 5-2. Note that the measured values are within 0.5% of the expected values from Table 5-1, with the exception of main loop execution frequency.

Table 5-2 Oscilloscope Measurement Output (Multi-Pin, Non-Nested)

Name	Measurement	Value	Std Deviation	ΔExpected
Main	Frequency	53.22 kHz	1.597 kHz	-146.78 kHz
Timer 2	Duty Cycle	9.934%	0.111%	0.068%
Timer 3	Duty Cycle	10.01%	0.187%	0.04%
Timer 4	Duty Cycle	10.16%	0.124%	0.289%
Timer 5	Duty Cycle	10.32%	0.252%	-0.318%

Method 2, for Non-Nested Interrupts

Method 2 requires only one GPIO pin, but also requires that the software instrumentation be configured at least once per interrupt. The oscilloscope produced the output shown in the figure 5-6 through 5-9. Note that the effects of the non-nested interrupt configuration can be seen in Figures 5-8 and 5-9, where there is some discontinuation in the periodicity of the signals since a higher priority ISR is executing.

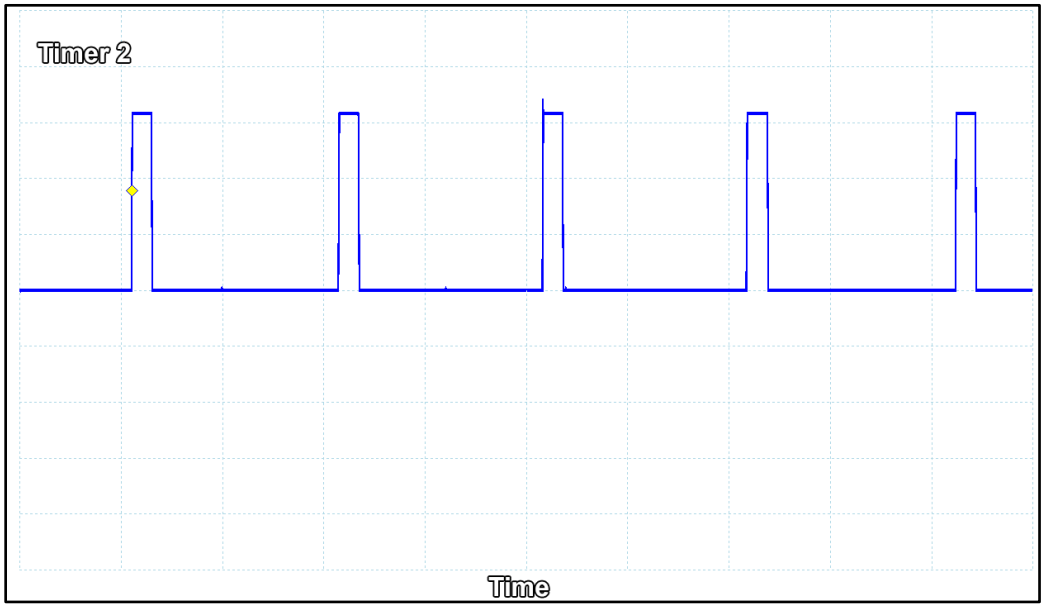

Figure 5-6 Output from Timer 2 ISR (1.0 ms per division)

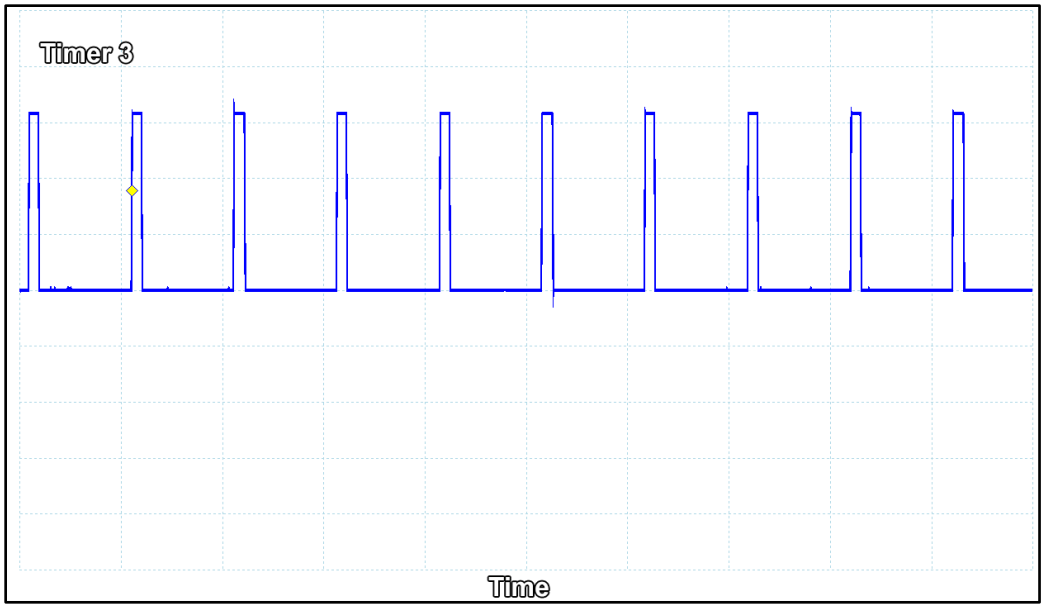

Figure 5-7 Output from Timer 3 ISR (1.0 ms per division)

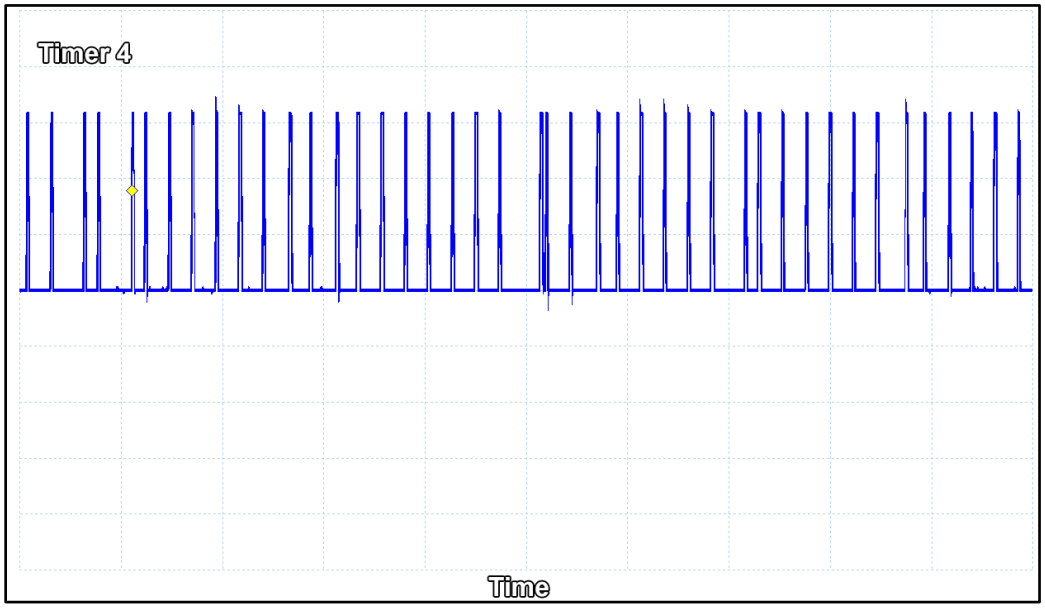

Figure 5-8 Output from Timer 4 ISR (1.0 ms per division)

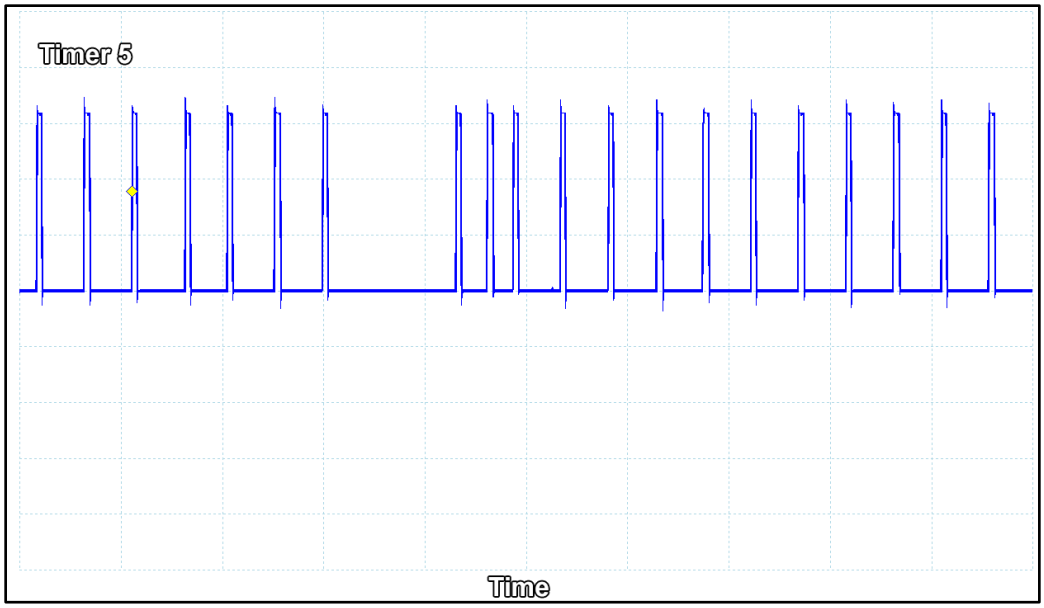

Figure 5-9 Output from Timer 5 ISR (100 µs per division)

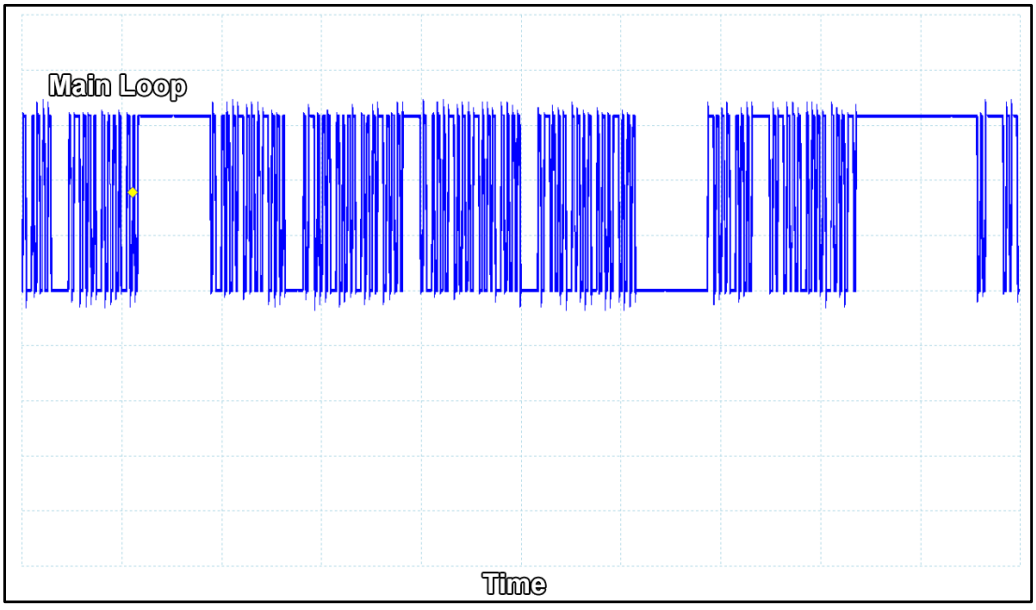

Figure 5-10 Output from main loop (100 µs per division)

As with the example for Method 1 (non-nested interrupts), the oscilloscope's built-in measurement functions were used to find the duty cycle of each interrupt and the frequency of the main loop. These results are summarized in Table 5-3. Note that the measured values are within 0.5% of the expected values from Table 5-3.

Table 5-3 Oscilloscope Measurement Output (Single-Pin, Non-Nested)

Name	Measurement	Value	Std Deviation	ΔExpected
Main	Frequency	55.69 kHz	4.449 kHz	-144.31 kHz
Timer 2	Duty Cycle	9.927%	0.043%	-0.061%
Timer 3	Duty Cycle	10.06%	0.068%	0.09%
Timer 4	Duty Cycle	10.17%	0.050%	0.299%
Timer 5	Duty Cycle	10.21%	1.007%	-0.428%

Method 1, for Nested Interrupts

For this example the timer configurations were left unchanged, but the option for interrupt nesting was enabled in the microcontroller's interrupt controller. Recall that this configuration requires interpretation of the oscilloscope's output before calculating the actual CPU usage percentages. Figure 5-11 shows how the output may appear reasonable but is misleading. The main loop toggle and timer signals are shown in order from top to bottom (main loop at the top, timer 2 second from top, and so on).

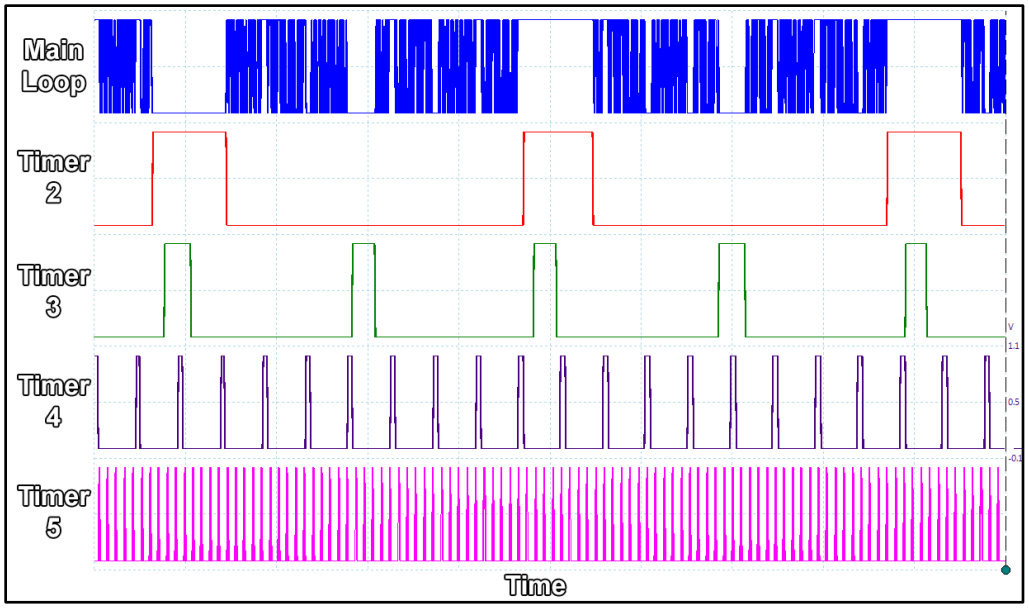

Figure 5-11 Oscilloscope Graphical Output (Incorrect)

Main loop stalls are easy to pick out by looking for the wide pulses in the waveform that normally toggles at a high frequency. The width of the pulses generated by timer 2's ISR is increased since the ISRs for timers 3, 4, and 5 nest for servicing. Table 5-4 contains the measurements taken by the oscilloscope.

Table 5-4 Oscilloscope Measurement Output (Incorrect, Nested)

Name	Measurement	Value	Std Deviation	ΔExpected
Main	Frequency	51.08 kHz	1.834 kHz	-146.78 kHz
Timer 2	Duty Cycle	15.28	3.157%	5.414
Timer 3	Duty Cycle	13.08	0.473%	3.110
Timer 4	Duty Cycle	11.53	0.160%	1.659
Timer 5	Duty Cycle	11.83	0.000%	1.192

Though these measurements are incorrect, the effect of enabling nested interrupts can be seen by examining the standard deviation of the duty cycle for timer 5's ISR. Since this interrupt has the highest priority, a standard deviation of zero is expected – timer 5's interrupt cannot be interrupted by anything else running on the CPU core.

The waveforms in Figure 5-11 could be used to find the correct CPU usage values by manually using the oscilloscope's built-in pulse measuring functions, but this process would be tedious and error-prone, especially for systems with many interrupt signals. An oscilloscope with some basic math function capabilities can be configured to automatically mask the pulses with each other so that the generated waveform shows the correct amount of CPU usage. An example is shown in Figure 5-12. The main loop and timer signals are shown in the same order as the previous figures.

The missing portions of the pulses in Figure 5-12 represent the time that the CPU spent servicing some other interrupt. Figure 5-13 shows a comparison between the raw signal created by the timer 2 interrupt, and the masked signal generated by the oscilloscope's math functions. The masked signal is shown at the top of Figure 5-13 (green). The raw signal is shown second from the top (red). It is easy to see how the other timer ISR signals can be subtracted from the raw signal generated by timer 2 to create the masked signal.

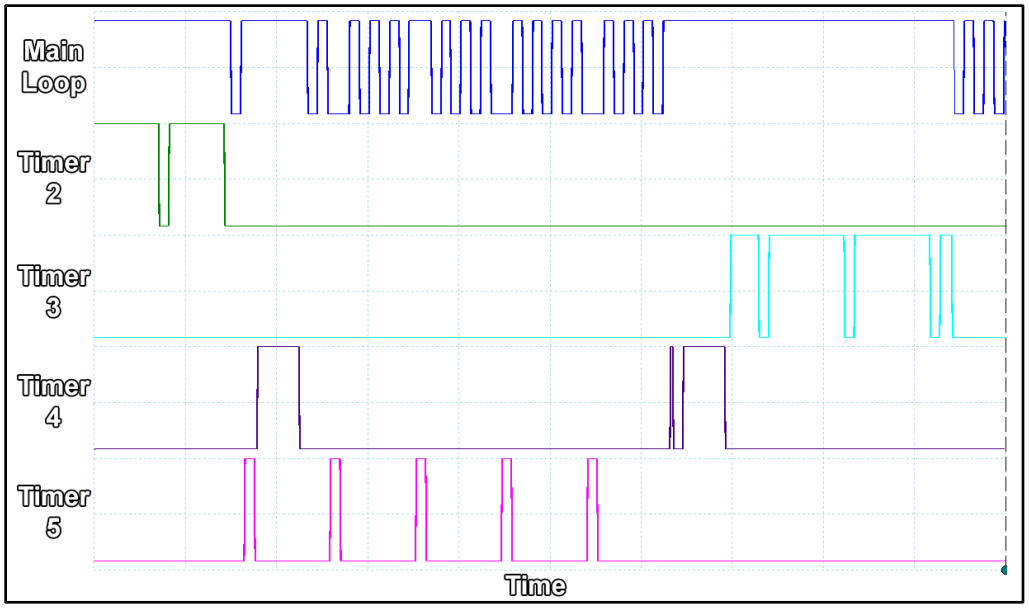

Figure 5-12 Oscilloscope Graphical Output (Correct)

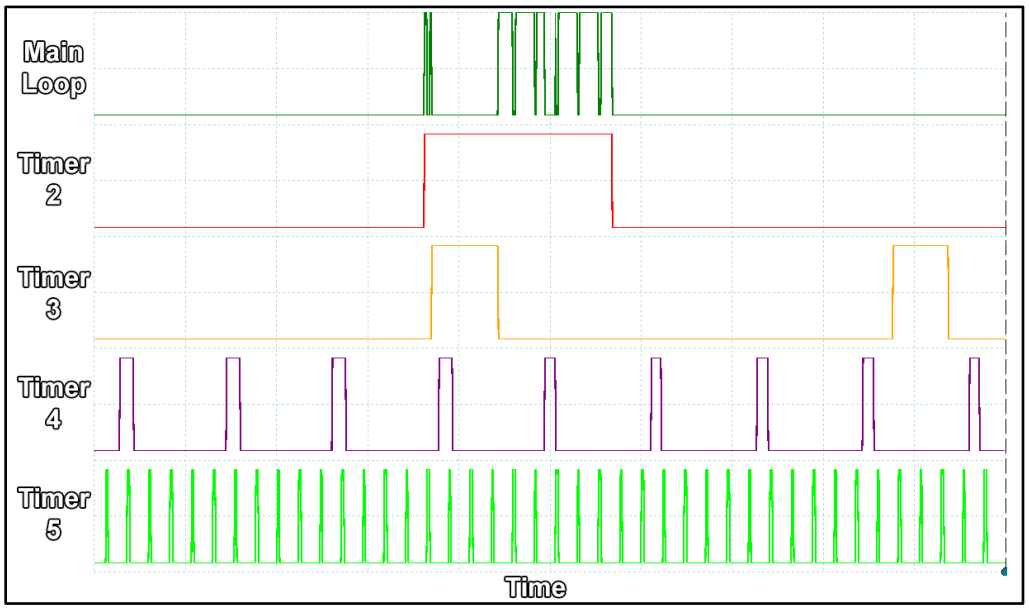

Figure 5-13 Raw vs Masked Signal for Timer 2

Mask creation was implemented using math functions that were included with the oscilloscope's software. Assuming that each signal is represented by some variable, *Dn,* where *n* is equal to the index of the signal, the equation for generating the masked signal for timer 2 is:

$$D_{masked} = (D_2 - D_3 - D_4 - D_1) + abs(D_2 - D_3 - D_4 - D_1)$$

When D_2 is low (logical level 0), subtracting the other signals result in a waveform with a negative amplitude, which confuses the oscilloscope software and produces a bogus value for the duty cycle. Adding the absolute value of the waveform to the original ensures that the negative portions of the waveform are cancelled.

A general form of the equation is:

$$D_{masked} = (D_{SUI} - D_1 - D_2 - \cdots - D_n) + abs(D_{SUI} - D_1 - D_2 - \cdots - D_n)$$

D_{SUI} represents the waveform of the signal under investigation.

Table 5-5 contains the measurements made by the oscilloscope on the masked signals shown in Figure 5-13.

Table 5-5 Oscilloscope Measurement Output (Correct, Nested)

Name	Measurement	Value	Std Deviation	ΔExpected
Main	Frequency	52.03 kHz	697 Hz	-146.78 kHz
Timer 2	Duty Cycle	12.25%	0.817%	2.384
Timer 3	Duty Cycle	10.18%	2.045%	0.210
Timer 4	Duty Cycle	8.091%	0.354%	-1.780
Timer 5	Duty Cycle	8.623%	0.165%	-2.015

Chapter 6

Conclusions

Telecommunication technologies are limited by two primary design characteristics: channel bandwidth and channel latency. Channel throughput is directly and positively correlated to channel bandwidth. Assuming that a given CAN bus carries standard length frames that contain the maximum number of stuff bits (L_{bit} = 130), and runs at 100% utilization at maximum baud rate, the highest possible achievable throughput is 58.74 KB/sec. If greater throughput is needed then the CAN bus could be replaced with (for example) an Ethernet bus which allows for a transmission rate of anywhere from 10 Mbps to 10 Gbps, or some other technology that is capable of higher throughput. New hardware designs may be able to accommodate this change assuming cost is not prohibitive and enough time for implementation is available, however in the case of an existing hardware design such modifications may not be possible. Any solutions must be implemented in software.

In addition to time-division multiple access (TDMA), total throughput of a channel can be increased further with some form of compression assuming the communication protocol has low overheard and a large data payload with low entropy. If any of these characteristics are not true (high overheard, small data payload, and/or data payload with high entropy) the compression ratio may not be sufficient to meet throughput requirements. Software-based compression also requires more CPU time, which may not be available on an existing design.

This document presents a design method and example of TDMA applied to a CAN bus wherein time-division is accomplished through time-based triggered transmission, hence the term time-triggered CAN (TTCAN). Chapter 4 contains a comparison between the theoretical performance of a TTCAN bus and an existing CAN bus that uses free-for-all transmission (FFACAN). The increase in effective throughput is marginal – with a 5x transmission multiplier, the throughput on the TTCAN bus increased by only 10.1% compared to the FFACAN bus. However the variability in frame transmission rate on the FFACAN bus varied widely from 1844.14 Hz to 1674.0 Hz (Table 4-9) whereas the variability in frame transmission rate on the TTCAN bus varied minimally from 1841.7 Hz to 1846.5 Hz. Whether these differences in transmission rates are acceptable is up to the system's designer.

This document also presents a simple method for measuring CPU usage of a microcontroller that uses both nested and non-nested interrupts. Though the applied method may seem trivial (monitor pin states with an oscilloscope), measurements become complicated for systems on which there few general purpose input and output (GPIO) pins are available, or for systems that use nested interrupts. The Boolean masking equation shown in Chapter 5 may be used with oscilloscope software that supports math channels. On systems that use nested interrupts these masking equations eliminate the need for interpreting pin state data after the fact; in other words, these masking equations allow for real-time CPU usage measurement on systems that use nested interrupts.

References

[1] Texas Instruments, “Controller Area Network Physical Layer Requirements,” SLLA270 datasheet, Steve Corrigan, Jan 2008.

[2] Microchip Corporation, “Stand-Alone CAN Controller with SPI Interface,” MCP2515 datasheet, Aug 2012 (Rev . G)

[3] Texas Instruments, “Tiva™ TM4C1294NCPDT Microcontroller”, SPMS433B datasheet, Jun 2014

[4] Taylor T. Johnson *et al,* “A Survey of Electrical and Electronic (E/E) Notifications for Motor Vehicles,” Dept. of Electrical and Computer Engineering, Univ. of Texas at Arlington, TX, USA.

Biographical Information

Randy Long first graduated in 2007 with a B.S. in Civil & Environmental Engineering. After several years of designing sanitary sewer pipelines by day and building robotic RC cars by night, he entered UTA's Electrical Engineering program in August of 2012 and joined UTA's Formula SAE team shortly thereafter.

Since then, Randy served as the lead embedded software developer for UTA's Formula SAE from December of 2012 to August of 2016. In April of 2013 he began development of UTA's first active-aerodynamic control system (AACS), which earned special recognition at the 2013 Formula SAE Lincoln competition and won the Cummins Applied Technology Award at the 2015 Formula SAE Michigan competition. UTA's AACS has been iterated several times since then for the 2014, 2015, and 2016 Formula SAE competitions in Lincoln, NE and Brooklyn, MI. The most recent version of the system uses separate controller boards for each wing along with a central controller, all of which run FreeRTOS and communicate using a custom CAN high-level protocol that was also developed by Randy.

The AACS was developed alongside the UTA FSAE E-16 electric racecar project. Randy contributed to this project by writing software for the electric powertrain control system, battery management systems, charge management system, the driver input body controller, and the driver information display. These efforts were rewarded in 2015 when he was given the opportunity to work as an intern in the gateway firmware group at Tesla Motors in Palo Alto, CA from June to December of the same year.

As for what comes next: “I’m going back to Cali. Uh, probably.”