

## CAN in 30 minutes or less

By Hassane El-Khoury, Sr. Business Development Manager, Cypress Semiconductor Corp.

### Executive Summary

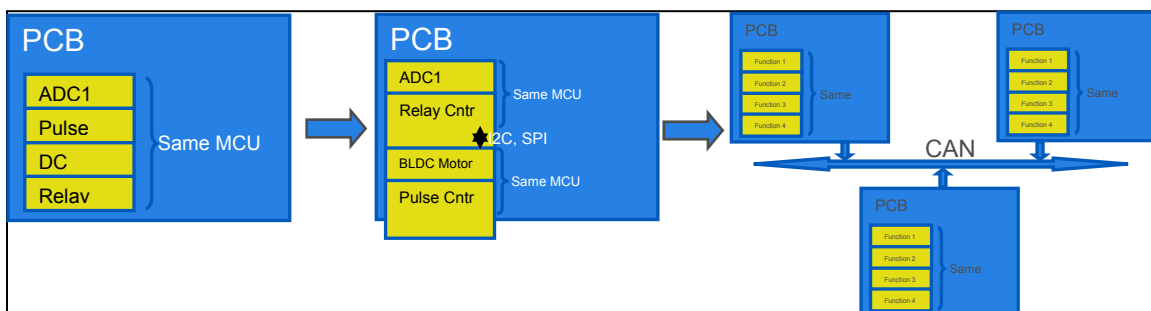
Since CAN was introduced in the 80's, it has seen a tremendous evolution in terms of specifications and requirements. Its extended capabilities have led to its wide adoption across applications, from automotive, to industrial machine and factory automation. With this growth, complexity of implementation has also increased on two levels:

- CAN controller design has gone from a basic controller to a full CAN controller and, in some cases, an extended full CAN controller.
- CAN software stacks vary, from an automotive communication stack, to CANOpen, and DeviceNet.

Given that CAN is only a single component within the automotive system, developers need to be able to implement it with as few challenges as possible so they can focus on system-level functionality rather than struggle with peripheral configuration. This article will explore the CAN interface and discuss different ways of implementing, configuring, and tuning interfaces to facilitate simplified design.

The Controller Area Network (CAN) was first introduced by Robert Bosch to address the growing complexity of vehicle functions and networks. In the early days of embedded systems development, modules contained a single MCU, performing a single or multiple simple functions such as reading a sensor level via an ADC and controlling a DC motor. As these functions became more complex, designers adopted distributed module architecture, implementing functions in 2 or more MCUs on the same PCB, and using I2C or SPI protocols to communicate between these functions. Using the same example above, a complex module would have the main MCU performing all system functions, diagnostics, and failsafe, while another MCU handles a BLDC motor control function. This was made possible with the wide availability of general purpose MCUs at a low cost.

In today's vehicles, as functions become distributed within a vehicle rather than a module, the need for a high fault tolerance, inter module communication protocol led to the design and introduction of CAN in the automotive market.



**Figure 1 – Introduction of CAN**

By the mid-nineties, CAN had seen wider adoption beyond automotive, with industrial controls with the introduction of DeviceNet and CANOpen protocols.

With this traction in the market, many MCU suppliers have integrated a CAN Controller as an integrated peripheral to address these markets. Although at a high level, CAN might seem similar in function to I2C or SPI, allowing communication between 2 nodes, CAN's communication is fundamentally different at the controller level, with the services listed below:

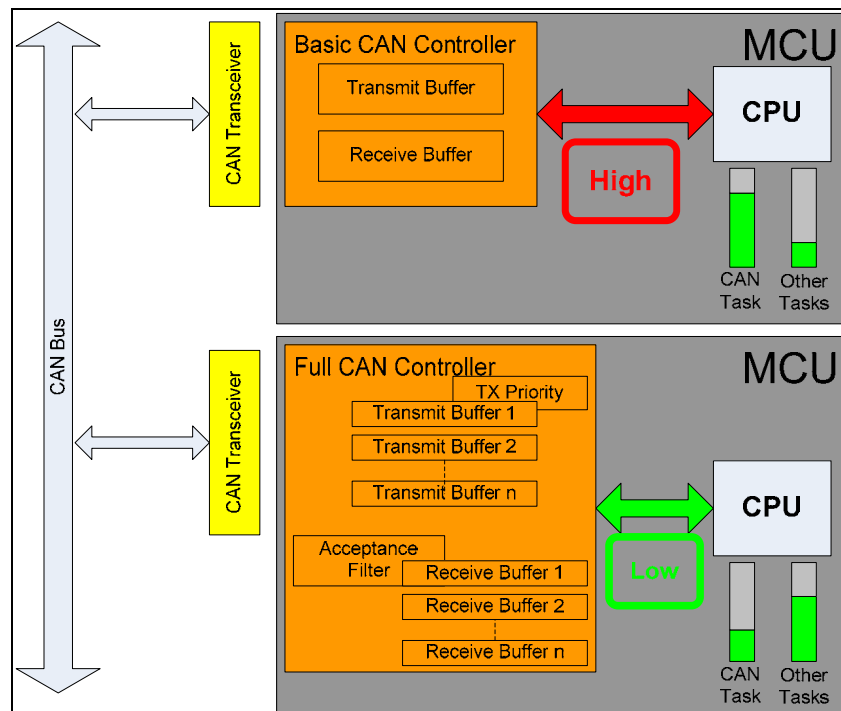
- CAN communication is message based, not address based.
- A CAN node can send or request a message on the bus.
- Complex error handling.
- Up to 5 corrupted bits in a row can be detected with CRC-15 protection.

Being message based versus address based, a node on the CAN bus could own multiple message for transmission; i.e. A Brake module might have a message containing vehicle speed information, a message containing sensor information, such as wheel speed sensors, and a message containing the diagnostics information, the latter having the highest transmit priority.

At first look at message priority and message ID decoding in a node, one might think that CAN would create a high load on the CPU, preventing additional more complex functions from being integrated. This issue is addressed by different types of CAN Controller also shows in figure 2.

- **Basic CAN Controller:** Very basic filtering implemented in the CAN Controller hardware, with reduced message handling, leading to a high CPU load. In a basic CAN controller, the CPU would receive multiple interrupts from the CAN controller in order to receive, acknowledge, and parse a message, and from the application side, decide whether or not a response should be transmitted based on the ID of the received message. Basic CAN controllers should only be used in low baud rate, and low message count communications, allowing the CPU to handle additional non-communication tasks.
- **Full CAN Controller:** provides an extensive implementation of message filtering, as well as message parsing in the hardware, thus releasing the CPU from the task of having to respond to every received message. Full CAN controllers can be configured to interrupt the CPU only when messages whose Identifiers have been setup as acceptance filters in the controller. Full CAN controllers are also setup with multiple message objects referred to as mailboxes, which can store specific message information such as ID and data bytes received for the CPU to retrieve. The CPU in this case would retrieve the message any time, however, must complete the task prior to an update of that same message is received and overwrites the current content of the mailbox. This scenario is resolved in the final type of CAN controllers.
- **Extended Full CAN controller:** provides an additional level of hardware implemented functionality, by providing a hardware FIFO for received messages. Such an implementation allows more than one instance of the same message to be stored before the CPU is interrupted therefore preventing any information loss for high frequency messages, or even allowing the CPU to focus on the main module function for a longer period of time.

Also important to note that DeviceNet extends the filtering criteria beyond the ID field, to the first two bytes of data, making a Full or Extended Full CAN controllers a must for implementing such a protocol.



**Figure 2 - Basic and Full CAN controllers**

Depending on the message architecture, the above 2 configurations can coexist in a single module, in order to implement high level message priority and improve received message handling for the CPU. For example, a module receiving failsafe information in one message (i.e. ID = 0x250) and temperature sensor information in another message (i.e. ID = 0x3FF) may configure the CAN controller as full for the first, and extended full with 4 buffer FIFO for the second: CPU is interrupted when each failsafe message is received, and once every 4 temperature sensor message received. Figure 3 illustrates such a CAN controller configuration with a visual CAN controller customizer for fast implementation of complex message handling schemes where all 3 types of CAN controllers co-exist:

- Message 5 → Basic CAN Mailbox.
- Message 0x250 → Full CAN Mailbox.
- Message 0x3FF → Extended Full CAN Mailbox.

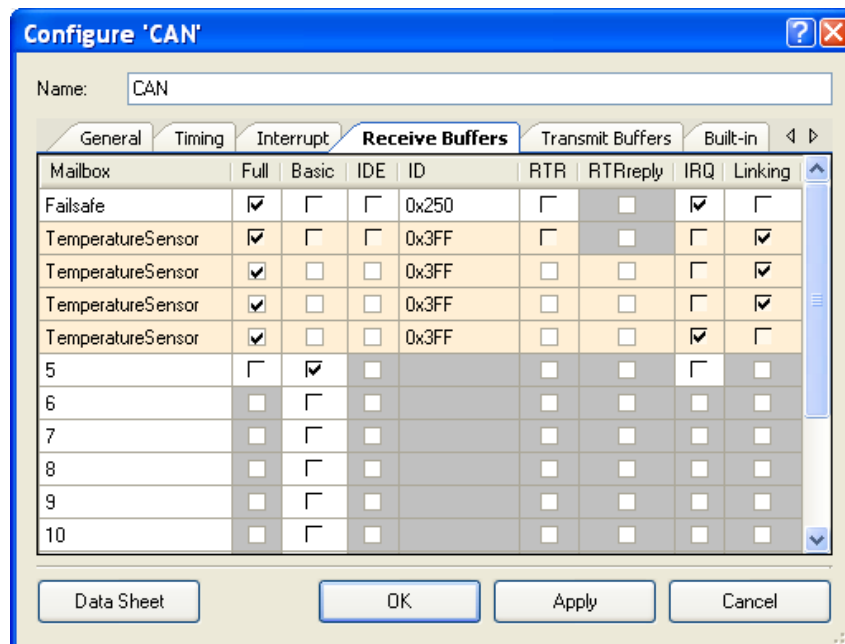


Figure 3 – PSoC Creator - CAN Controller Configuration

In addition to its functional capabilities, CAN has been widely adopted for its high fault tolerance. With bit rates up to 1Mbps, or bus length up to 1000m (at 50Kbps), CAN bit timing must be implemented to allow functionality in electrically noisy environment while maintaining high level of failure detection and correction.

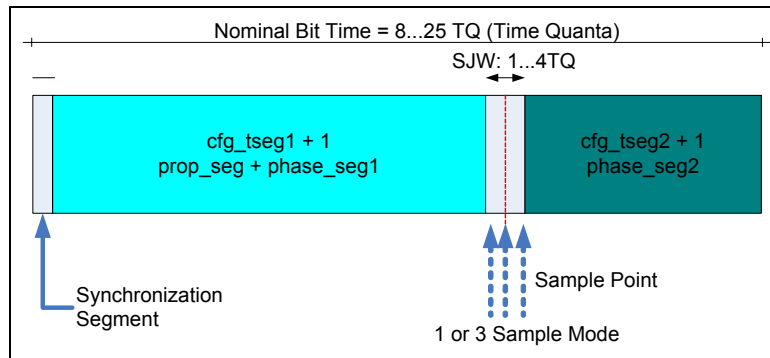
To guarantee a high level of fault tolerance, sub-bit timing configuration has been introduced with CAN to allow tighter control of determining the correct bus state for each CAN Bus.

A single CAN bit is represented by 4 segments:

- **Sync\_Seg:** Used to synchronize the various nodes on the bus.
- **Prop\_Seg:** Compensate for any physical delays (propagation delay on the physical bus and the internal CAN node).
- **Phase\_Seg1, Phase\_Seg2:** Used to compensate for phase edge errors. These segments are shortened or lengthened during resynchronization.

It is also common to find CAN controllers with only 3 segments, where the Prop\_Seg time is added to the Phase\_Seg1 time.

Figure 4 shows the bit timing representation, with all parameters necessary for its implementation:



**Figure 4 - Bit Timing Representation**

It is important to note that all CAN bit timing calculations are based on time quanta (TQ), defined as a fixed unit of time derived from the oscillator with a value between 8 and 25. In terms of time, a TQ is equivalent to as low as 1/25<sup>th</sup> bit or 40ns for a 1us bit length at 1Mbps bus speed.

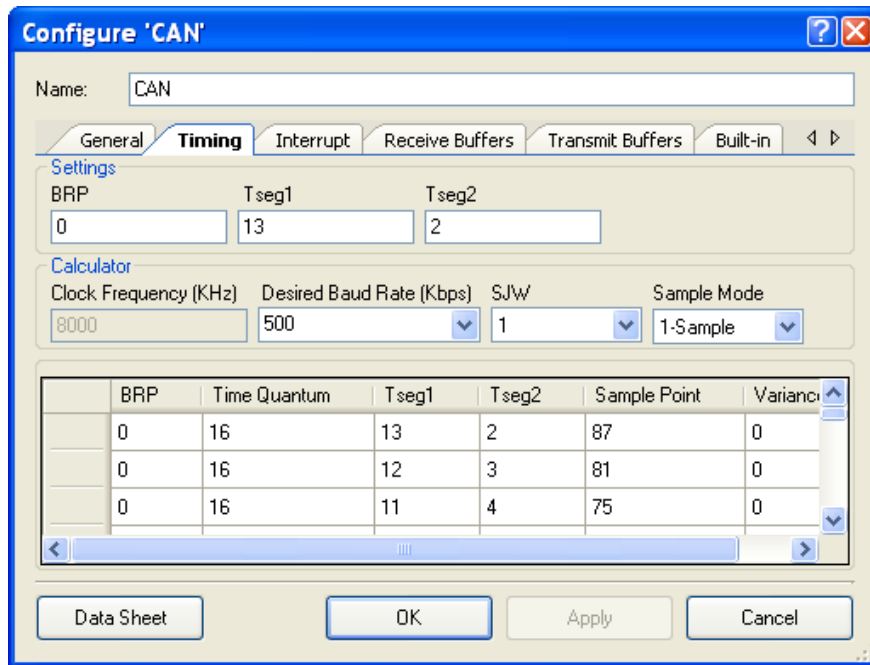
As an overly simplified rule of CAN bit timing, the list below could be setup to govern the values of the 4 bit time segments:

- Sync\_Seg = 1TQ
- $1TQ \leq \text{Prop\_Seg} \leq 8TQ$
- $1TQ \leq \text{Phase\_Seg1} \leq 8TQ$
- $2TQ \leq \text{Phase\_Seg2} \leq 8TQ$
- $1TQ \leq \text{SJW} \leq \text{MIN}(\text{Phase\_Seg1}, 4)$
- SJW – Synchronization Jump width, defined as the time length by which Phase\_Seg1 may be lengthened and Phase\_Seg2 may be shortened

The above relationships generate a large range of possible values for each of the parameters they involve, and selecting the right combination is crucial to the successful implementation of a robust CAN communication. Designers must not only account for oscillator accuracy, and propagation delays within the node in question, but also account for other system nodes with which communication must be established.

Based on the system clock, the baud rate required, and possibly the required sample point of the bit (i.e. CANOpen sample point required at 87.5%), configuring the bit timing for CAN becomes a challenge many designers are reluctant to undertake. This perceived complexity is causing many new embedded systems to reuse legacy MCUs and even software stacks rather than adopting new products which might address the overall system requirements a better way. Unfortunately, it puts CAN in the “It’s working, I don’t want to touch it” category of embedded peripherals.

Figure 5 shows a sample bit timing configuration for CAN, where creating new timing analysis or modifying a baud rate of an existing node configuration is no longer a risky task for an already operational CAN bus. By providing all specified and verified combinations of parameters to achieve a robust CAN timing implementation, designers can focus on the more complex tasks of the module’s main functionality.



	BRP	Time Quantum	Tseg1	Tseg2	Sample Point	Variance
	0	16	13	2	87	0
	0	16	12	3	81	0
	0	16	11	4	75	0

**Figure 5 - PSoC Creator - CAN Bit Timing Configuration**

Although available for decades now, CAN is still perceived as a complex peripheral in an embedded design, especially in research and development area (R&D). R&D activities involve multiple iterations of the system configuration, with CAN being a component in the overall system. CAN drivers and CAN stack development can be outsourced to specialized companies providing these services for a fee, some with limited post-delivery change and configurability. Lately, semiconductor companies (such as Cypress in the example above) are providing tools to bridge that gap and allow quick in-house development, and fast time to market for complex embedded systems.

Cypress Semiconductor  
198 Champion Court  
San Jose, CA 95134-1709  
Phone: 408-943-2600  
Fax: 408-943-4730  
<http://www.cypress.com>

© Cypress Semiconductor Corporation, 2007. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

PSoC Designer™, Programmable System-on-Chip™, and PSoC Express™ are trademarks and PSoC® is a registered trademark of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are property of the respective corporations.

This Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.