User's Guide
SLLU270–May 2019

TCAN45xx Software User's Guide

The purpose of this document is to give an overview and some basic examples of the TI TCAN45xx SPI to CAN FD bridge device to assist with the development of a custom application. This document refers to the device data sheet for register descriptions.

Contents

List of Figures

List of Tables

Trademarks

All trademarks are the property of their respective owners.

1 Introduction

The TCAN45xx family of SBCs support classic CAN and CAN FD communication with SPI interface. This allows users to incorporate CAN or CAN FD into their system without a deep understanding of the CAN/CAN FD protocol.

For developers that already have software built around the common Bosch M_CAN controller, it is quick to port over to the TCAN45xx. Only the low-level register read and write functions must be changed to be a SPI-based read and write. The register set is the same. A CAN Controller is integrated into the microcontroller/processor and an external transceiver is required to communicate on a CAN bus, as shown in Figure 1.

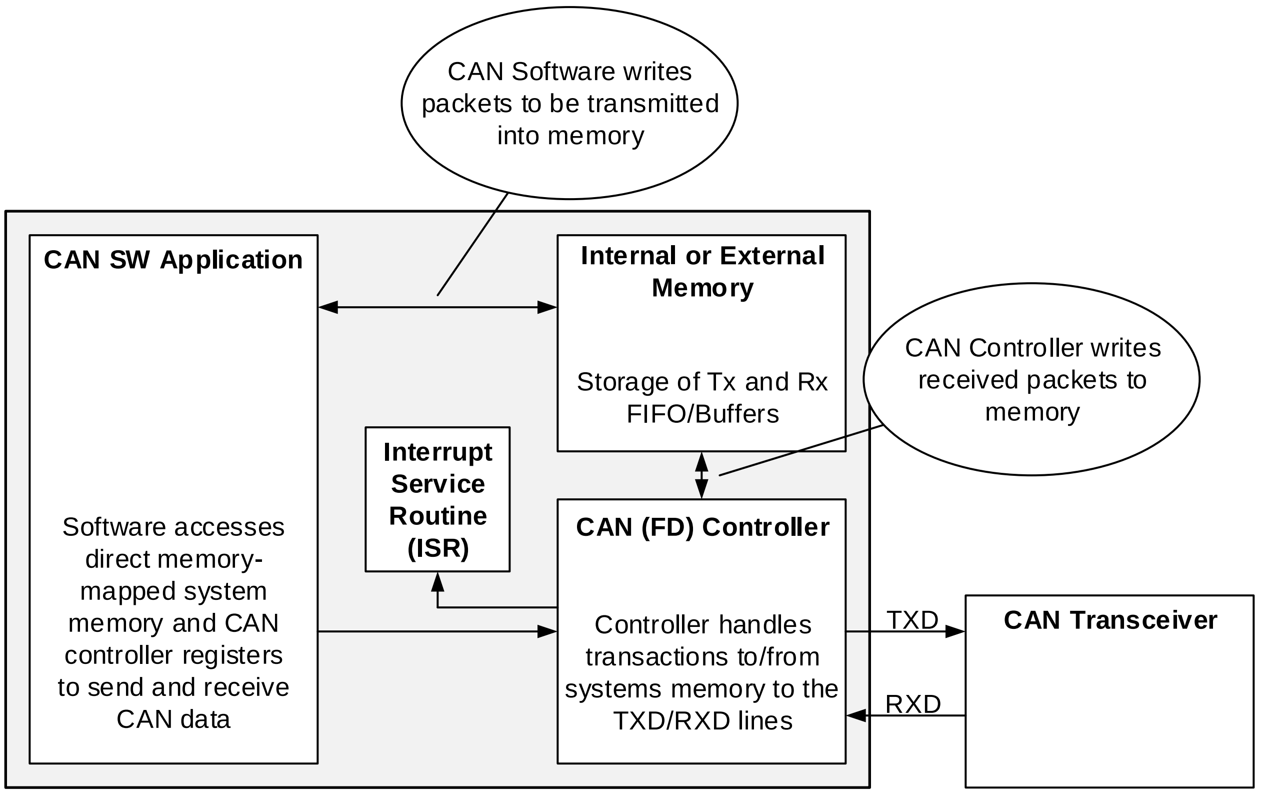
CAN Software writes packets to be transmitted into memory
CAN SW Application
Software accesses direct memory-mapped system memory and CAN controller registers to send and receive CAN data
Internal or External Memory
Storage of Tx and Rx FIFO/Buffers
CAN Controller writes received packets to memory
Interrupt Service Routine (ISR)
CAN (FD) Controller
Controller handles transactions to/from systems memory to the TXD/RXD lines
TXD
RXD
CAN Transceiver

Figure 1. Integrated CAN Application

For users who wish to add CAN FD functionality (or additional CAN busses), but their selected microcontroller has no built in CAN controller, the TCAN45xx includes the memory, CAN FD Controller and transceiver in a single package. Figure 2 shows how an existing solution can be ported to an external controller by changing the code that writes to registers to use a SPI read or write instead.

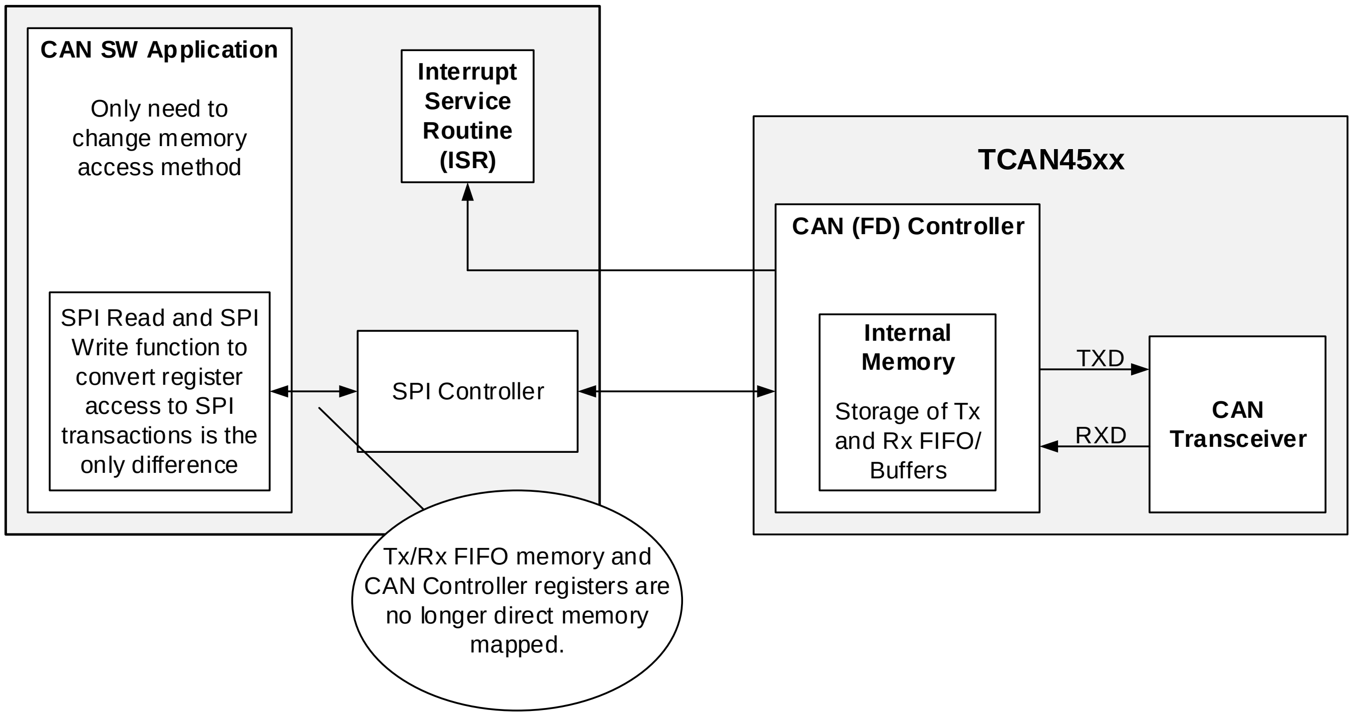
CAN SW Application
Only need to change memory access method
SPI Read and SPI Write function to convert register access to SPI transactions is the only difference
Interrupt Service Routine (ISR)
SPI Controller
Tx/Rx FIFO memory and CAN Controller registers are no longer direct memory mapped.
TCAN45xx
CAN (FD) Controller
Internal Memory
Storage of Tx and Rx FIFO/ Buffers
TXD
RXD
CAN Transceiver

Figure 2. CAN Application with TCAN45xx

2 TCAN45xx SPI to CAN Basics

The TCAN45xx gives CAN FD functionality to applications without requiring anything more than a SPI bus. The TCAN45xx handles all CAN and CAN FD traffic, and includes the transceiver in the package. This makes adding CAN functionality to a system quick and easy, since the only requirement on the processor is to have a SPI interface. There are many features of the TCAN45xx family, which include CAN and non-CAN related functions, such as a watchdog timer, CAN transceiver and controller with CAN FD support, on-device memory for storing and sending CAN messages, partial networking and more. This document provides several configuration and use case examples to aid in quick development.

3 Device Features

This portion of the document pertains to device features which are not directly CAN related.

3.1 *Watchdog Timer (TCAN4550 Only)*

The watchdog timer (WDT) is a feature which must be reset by the host (via SPI write to the WD_BIT 18 of the Modes of Operation and Pin Configuration register, or via a general purpose input pin) within the timer window; otherwise, a watchdog action occurs. This feature is available on the TCAN4550. There are 3 options for watchdog actions (selectable via the WD_ACTION bits in the Modes of Operation and Pin Configuration register):

1. Set an interrupt flag, if an output is configured to reflect WDT output, then the pin shows a low (default)
2. Pulse the inhibit (INH) pin with a low-high-low pattern of ≈ 300 ms, and place TCAN4550 into standby mode
3. Pulse watchdog output reset pin with a low-high-low pattern of ≈ 300 ms

These watchdog action options allow the user flexibility in their use of the timer, as well as behavior when it times out. It is possible to automatically recover/reset a host processor if it does not respond within a certain amount of time to reset the WDT, allowing for automatic recovery in the event of a system failure.

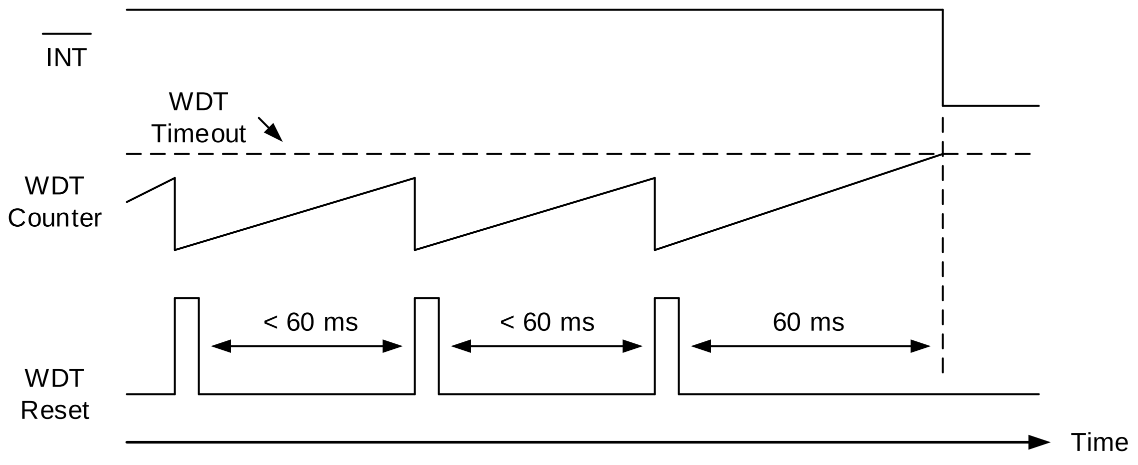

Figure 3. Example Watchdog Timer Behavior (60 ms and $\overline{\text{INT}}$ output)

An example watchdog timer situation is shown in Figure 3. The WDT is configured for 60 ms time out value. Three reset writes are performed before the timer overflows. In this example, the processor does not reset the watchdog timer after the 3rd reset, and the WDT counter hits the timeout value. Once the WDT hits the time out value, it performs the watchdog action. In this example, the action is set to the default interrupt flag, and the output is configured to reflect the WDT output. Figure 4 shows what the INH pin behavior is when the watchdog behavior is configured to pulse the inhibit pin.

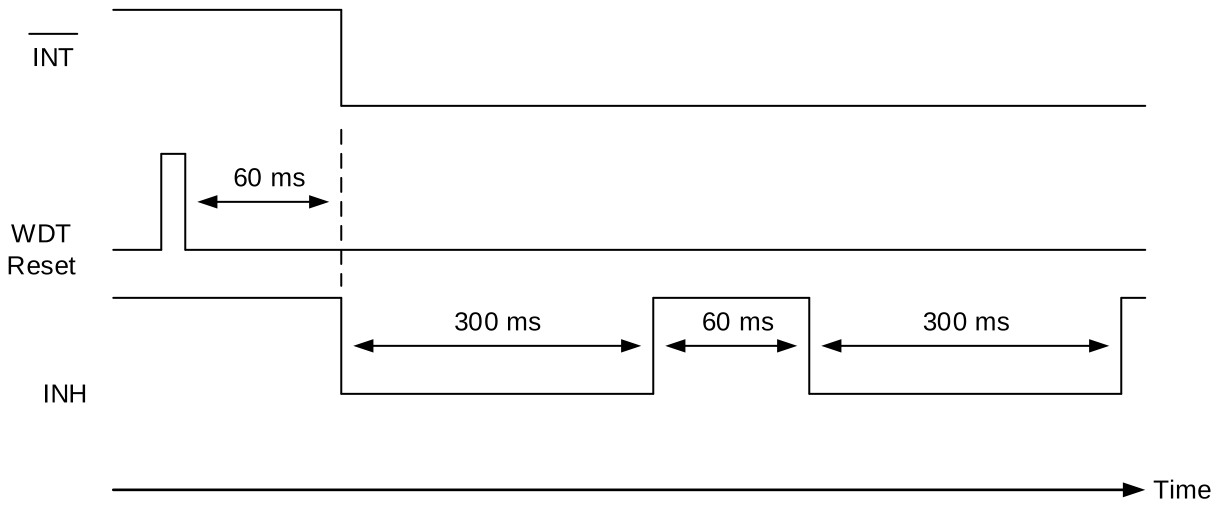

Figure 4. Example Watchdog Timer Behavior (60 ms and INH output)

3.1.1 Configuring the Watchdog Timer

To configure the watchdog timer, the designer must decide the following items:

Table 1. Design Requirements

Design Requirement	Available Options	Description
Watchdog timer timeout value	60 ms / 600 ms / 3 s / 6 s	The time out value, the amount of time the microcontroller has to reset the watchdog timer, otherwise a WDT interrupt occurs.
Watchdog timeout action	Interrupt flag / Inhibit pin and standby mode / Pulse watchdog output pin	Describes what the TCAN4550 does when the watchdog time out has occurred.
Watchdog clock reference	20 MHz / 40 MHz	Sets the appropriate clock divider based on the input oscillator frequency
Reset pulse method	SPI Write / SPI Write or GPIO Pulse	Sets how the timer is reset, either via SPI only, or enable a GPIO input pulse to reset the timer. The SPI method is always enabled

Once the parameters in question are selected, then the desired values may be written to the Modes of Operation and Pin Configuration Register (0x0800).

As an example, a watchdog timer timeout value of 600 ms, a timeout action of Inhibit pin, and standby mode is selected. The user must also make sure that the enable bit is set for the watchdog timer.

Table 2. Design Selections

Design Requirement	Selected Options	Register Bits	Value
Watchdog timer timeout value	600 ms	0x0800 [29:28]	2'b01
Watchdog timeout action	Inhibit pin and standby mode	0x0800 [17:16]	2'b10
Watchdog clock reference	40 MHz	0x0800 [27]	1'b1
WD_Enable	Enabled	0x0800 [3]	1'b1

The power up default of the Modes of Operation and Pin Configuration Register (0x0800) is 0xC8000468. The appropriate bits must be set in this register according to Table 2. This results in a value of 0xD8020468. The watchdog has been configured, but does not start until the first WD Reset pulse has occurred (from either the WD input pin if configured, or a SPI write to the WD_RESET bit). Once this pulse has occurred, the watchdog timer begins and the microcontroller must reset the watchdog counter within the configured timeout value window to prevent the watchdog action from occurring.

3.2 *Starting the Watchdog Timer*

The watchdog timer does not start until the initial write to the WD_BIT (0x0800 [18]) or a pulse to the GPIO pin that is configured for watchdog input (if enabled).

3.2.1 Resetting the Watchdog Timer

Once the timer has been started, the microcontroller must clear the watchdog counter before the watchdog timer has crossed the timeout value. This may be accomplished by writing to 0x0800 [18] (WD_BIT), which resets the timer's counter, and the process repeats.

3.2.2 Watchdog Timeout

If the microcontroller is unable to reset the timer before the timeout value, then a watchdog interrupt is set, and the TCAN4550 performs the action described by the WD_ACTION bits in 0x0800 [17:16].

4 CAN / CAN FD

This section of the document pertains to the CAN controller core, and the portions required to communicate on a CAN bus.

A CAN controller is a state machine that manages the protocol-specific details, leaving the microcontroller to manage the actual data being sent and received.

In a CAN system, there are a few critical variables that must be set correctly to ensure proper communication across the peripheral devices. Bit timing is the most critical.

4.1 *Bit Timing Setup*

In the CAN protocol, there are 4 sections to a nominal bit time: the sync segment, the prop segment, and the phase 1 and phase 2 segments. This is shown in Figure 5. Each CAN bit is over-sampled by the CAN controller. In the example below, each CAN bit is sampled 10 times (10 time quanta). The state of the bit is sampled between phase 1 and phase 2. This determines if the current CAN bit is a 1 or a 0. The sync segment is required to be a single time quanta (tq) while the other 3 segments are controllable to set the CAN data rate. In the example below, the propagation delay segment is 3 tq, phase 1 is 4 tq, and phase 2 is 2 tq. Therefore, the sum equals 10 tq. The sample point in this example is 80% to allow for enough time for the signal to propagate through a CAN bus. If the sample clock were 10 MHz (40 MHz input clock with a prescaler of 1:4), then the CAN data rate would be 1 Mbps (10 MHz / (10 tq/bit) = 1 Mbps).

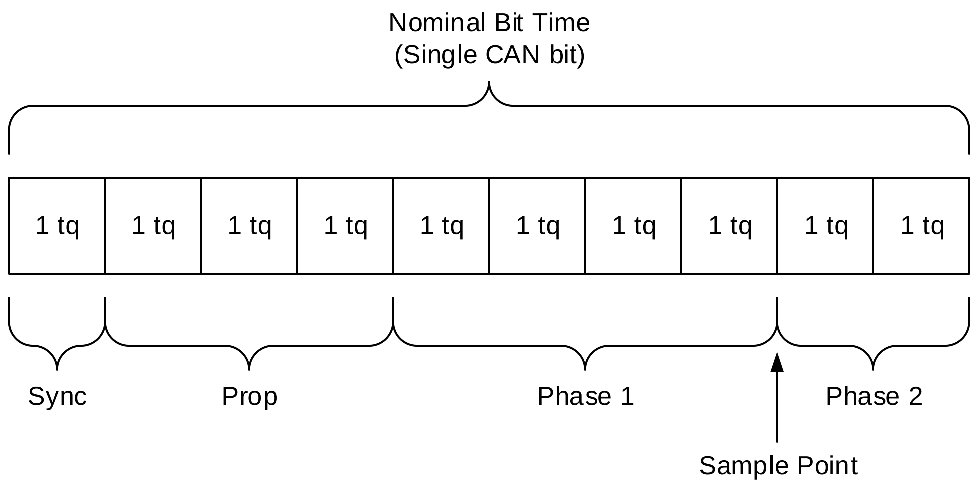

Figure 5. Nominal bit Time

The TCAN45xx takes the timing parameters in 2 single values: Prop + Phase 1, and Phase 2. This tells the CAN controller when to sample the bit, and how many tq are in a bit. Since the sync bit is required and always equal to a single time quanta, this is assumed by the TCAN45xx and does not require the microcontroller to input this data. Figure 6 shows how the TCAN45xx would interpret these values, and how a user can envision the single data bit.

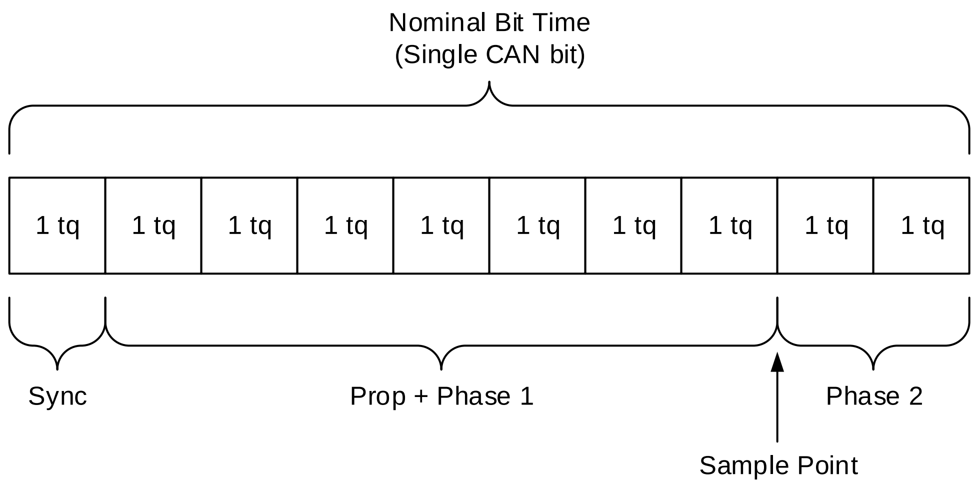

Figure 6. Nominal bit time for the TCAN45xx

The TCAN45xx registers for the CAN bit timing interpret values greater than the raw register value. For example, if a user inputs 0 into the phase 2 bits, then the TCAN45xx interprets it as a value of 1. This is because none of the prop or phase values may be 0 tq. So care must be taken when inputting the bit timing settings that the microcontroller writes 1 less than the desired value. Table 3 outlines the parameters required for this example.

Table 3. CAN Bit Timing Values

Segment	Raw value	Register value
Prescaler	4	3
Prop + Phase 1	7 tq	6
Phase 2	2 tq	1

The table above shows the raw values, and the values that would be entered into the register. The prescaler value is used to scale the input 40 MHz clock, in this example, to a 10 MHz clock, with 10 time quanta to achieve a nominal bit rate of 1 Mbps. There is 1 additional parameter called the sync jump width, which allows the CAN controller to compensate for a certain number of time quanta of error. This value is generally set equal to Phase 2. Refer to a CAN protocol manual for more information. As an example, if the microcontroller were to write these values to the TCAN45xx, it would write 0x02030601 to the NBTP (0x101C) register. In the previous write, the nominal resynchronization jump width is set to the same value as Phase 2 (note that NSJW is bits [31:25]).

It is also important to note that there is a separate register for CAN FD bit rate switching (BRS) enabled messages, which allows the CAN controller to switch to a faster speed for the data payload portion of the message. The process is the same, but the values are written to the DBTP (0x100C) register instead. In systems with CAN FD and bit rate switching enabled, there will frequently be an additional propagation delay offset needed to properly sample bits. This delay is called the transmitter delay compensation and has its own register called TDCR (0x1048). If this value is not set properly, high speed data payloads will likely interpret the data incorrectly, or go into an error state.

4.2 Message RAM

The Message RAM (MRAM) is a block of memory to be used by the TCAN45xx for sending and receiving CAN messages. The layout of this memory is up to the system designer. It is very important to note that the TCAN45xx does NOT perform any checks of the MRAM layout to ensure a valid configuration which is free from any overlapping sections. As such, it is critical that the MRAM be properly configured or unanticipated behavior may occur.

Overlapping memory sections also may occur without directly realizing it. A read or write to the MRAM address space wraps around to the start of the MRAM after the last address of the MRAM. For example, on the TCAN45xx, there is 2 kB of MRAM, occupying address 0x8000 to 0x87FF. If a user attempts to read or write to register 0x8800, this is the same as a read or write to 0x8000. So it becomes critical to make sure that the MRAM memory is not over-allocated, either.

Another feature of the MRAM is Error Correcting Code (ECC) functionality. This feature is able to correct a single bit error per word of memory, and detect and warn the M_CAN module if there are more errors. This is important to keep in mind while reading or writing to the MRAM, since the ECC is updated on a write to memory only. On a fresh power up of the system, the ECC values will not be valid for the data in MRAM, and attempting to perform an action that reads from the MRAM without first writing results in a Bit Error Uncorrectable error (BEU). Particularly, it is important that the user writes at least 8 bytes of payload data (in addition to the header) into a TX buffer, regardless of if the Data Length Code (DLC) of the packet is less than 8.

4.2.1 MRAM Sections

There are 7 sections that are available for use within the MRAM memory space. All of these are optional, and the order of sections does not matter.

Table 4. Message RAM

Start Address Location	Section Name	Description
SIDFC.FLSSA	11-bit ID Filter	11-bit ID Filter Elements, used to filter any incoming 11-bit ID CAN message. The system designer writes to this section.
XIDFC.FLESA	29-bit ID Filter	29-bit ID Filter Elements, used to filter any incoming 29-bit ID CAN message. The system designer writes to this section.
RXF0C.F0SA	Rx FIFO 0	Rx FIFO 0, a FIFO which stores incoming CAN messages. The system designer reads from this section.
RXF1C.F1SA	Rx FIFO 1	Rx FIFO 1, a FIFO which stores incoming CAN messages. The system designer reads from this section.
RXBC.RBSA	Rx Buffers	Rx Buffers is a series of buffers which specific CAN messages can be sent to. These are not a FIFO, and if new data is received into the buffer without the old data being read, the original data is lost. The system designer reads from this section.

Table 4. Message RAM (continued)

Start Address Location	Section Name	Description
TXEFC.EFSA	Transmit Event FIFO	A FIFO which stores CAN message transmit event messages. These elements are generated by the TCAN45xx when transmitting a message, and are for the microcontroller to read to see the status of a sent message. The system designer reads from this section.
TXBC.TBSA	Tx Buffers	These Tx Buffers store CAN messages to be transmitted. When sending a CAN message, data must first be loaded into the buffer, then a transmission request must be sent to the TCAN45xx from the microcontroller to start the transmission. The system designer writes to this section.

It is important to realize that each of these sections are optional. The system designer can select which sections they require, and how many elements of each section are needed. As mentioned earlier, the TCAN45xx does not check the configuration registers to ensure that the MRAM is laid out free of overlaps, so the designer needs to ensure that this is the case. There are no requirements on the order of these sections, or that they must be 'back to back'. The system designer is given full control over the use of the MRAM.

4.2.2 Example MRAM Configuration

As mentioned in Section 4.2.1, there are 7 sections available for use. This section will walk through the configuration of an example setup.

Table 5. Message RAM Design Example

Section Name	Number of Elements	Maximum Data Size
11-bit Filter (SID Filter)	2	-
29-bit Filter (XID Filter)	1	-
Rx FIFO 0	4 (Warn at 2)	48 Bytes
Rx FIFO 1	5 (Warn at 3)	64 Bytes
Rx Buffers	0	-
Tx Event FIFO	3	-
Tx Buffers	10 (Tx FIFO)	64 Bytes

Table 5 lists the number of elements the system designer of this example desires. Once the number of elements and the desired maximum data payload (of the CAN message) is known, then the system designer is able to start calculating the start addresses to ensure that there are no overlaps.

There are a few notes to keep in mind when calculating the start addresses. The lower 2 bits are ignored by the TCAN45xx. This means that every start address must be on a 4 byte boundary (0x00, 0x04, 0x08, 0x0C, etc). For the TCAN45xx family, the MRAM memory space starts at 0x8000. However, when writing the start address into the TCAN45xx start address registers, the leading bit is dropped. For example, if a section has a start address of 0x800C, a user would simply write 0x000C into the start address bits of the appropriate configuration register. When performing a SPI read or write to the memory location, the 0x800C address must be used.

Table 6 outlines the necessary calculations to determine the start address.

Table 6. Message RAM Design Example Continued

Section Name	Maximum Data Size	Bytes per Element	Number of Elements	Total Bytes per Section	Start Address	End Address	Start Address (Hex)
11-bit Filter (SID Filter)	-	4	2	8	0	7	0x8000
29-bit Filter (XID Filter)	-	8	1	8	8	15	0x8008
Rx FIFO 0	48 Bytes	8 + Max data size = 56	4 (Warn at 2)	224	16	239	0x8010
Rx FIFO 1	64 Bytes	8 + Max data size = 72	5 (Warn at 3)	360	240	599	0x80F0
Rx Buffers	-		0	0	-	-	-
Tx Event FIFO	-	8	3 (Warn at 2)	24	600	623	0x8258
Tx Buffers	64 Bytes	8 + Max data size = 72	10 (Tx FIFO)	720	624	1343	0x8270

In Table 6, the system designer determines how many bytes are in each element (from the data sheet), and then is able to determine how many bytes are used by each section. This allows the designer to determine the start address of each section while ensuring that no overlap occurs. In this example, the decimal values of the register ignore the 0x8000 prefix which is shown in the hex value of the address. When writing these start address into the appropriate configuration register, the 0x8000 prefix will be removed, but to read or write from these portions of memory, the 0x8000 prefix is required.

The next step in this process determines the actual register values needed for each of the elements.

Table 7. Message RAM Design Example Register Values

Register Name	Register Name	Register Address	Number of Elements	Section Start Address	Register Value
11-bit Filter (SID Filter)	SIDFC	0x1084	2	0x8000	0x00020000
29-bit Filter (XID Filter)	XIDFC	0x1088	1	0x8008	0x00010008
Rx FIFO 0	RXF0C	0x10A0	4 (Warn at 2)	0x8010	0x02040010
Rx FIFO 1	RXF1C	0x10B0	5 (Warn at 3)	0x80F0	0x030500F0
Rx Buffers	RXBC	0x10AC	0	-	0x00000000
Rx Element Size Config	RXESC	0x10BC	0 B (Rx Buffers), 64 B (FIFO 1), 48 B (FIFO 0)	-	0x00000076
Tx Event FIFO	TXEFC	0x10F0	3 (Warn at 2)	0x8258	0x02030258
Tx Buffers	TXBC	0x10C0	10 (Tx FIFO)	0x8270	0x0A000270
Tx Element Size Config	TXESC	0x10C8	64 B	-	0x00000007

The register values shown in Table 7 configures the TCAN45xx MRAM based on the design example. Each of the registers shown are protected write registers meaning that the user cannot write to them unless the CCE and INIT bits in the CCCR register are set. This ensures that no accidental device configuration changes may occur while the device is operating on a CAN bus. Setting the CCE and INIT bits puts the device into an initialization mode and prevents it from interfering with the CAN bus. At this point, the next step would be to write the desired ID filters to the appropriate sections and finish configuring the TCAN45xx. Figure 7 shows a visual representation of the example MRAM allocation. Not all of the memory is used but there is no requirement that it must be. The memory in this example is also tightly packed together to prevent gaps between the sections in memory.

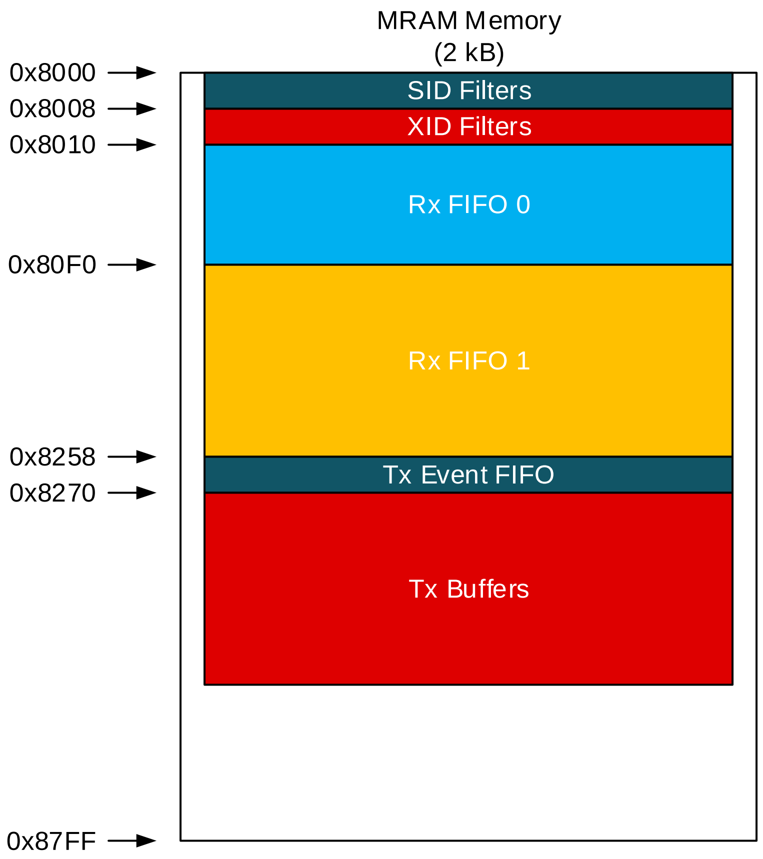

Figure 7. Visual Representation of MRAM Allocation

4.2.3 11-bit Filters (SID Filters)

The SID filter allows a system designer to either reject or accept CAN messages based on ID. When accepting a packet, there are options as to which of the 3 available receiving FIFOs / Buffers to send it to (RX FIFO 0/1 or RX Buffers). It is also possible to mark a message as 'high priority', and set another interrupt to alert the processor. The TCAN45xx will only compare an incoming message to the SID filters if the incoming message uses a standard ID (11 bits). If the incoming message contains an extended ID, then the XID filters will be used instead.

The filter must first match an ID based on the SFT value (and the values entered in SFID1 and SFID2). When an ID is matched, the action described in the Standard Filter Element Configuration (SFEC) is performed. If the behavior of a filter element is to accept the packet, the filter index will be added to the header of the CAN packet in the MRAM, so that the user may know which filter the ID matched with.

As stated earlier in Section 4.2, since MRAM values are unknown after power up and the ECC values are not valid. It is important that data must be written to each of the filter locations in MRAM that are configured for use in the SIDFC or XIDFC registers. Failure to do this results in a M_CAN BEU error, which puts the TCAN45xx device into initialization mode, and require user intervention before CAN communication can continue.

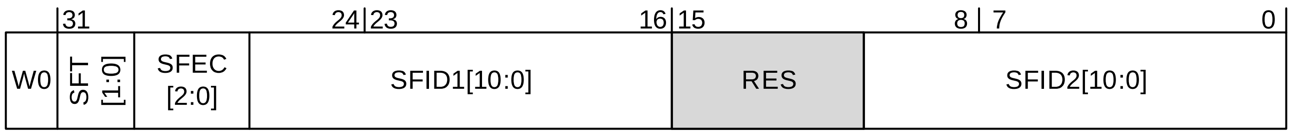

Figure 8. SID Filter Element

Table 8. SID Filter Element

Name	Description	Bits
SFT	Standard Filter Type	[31 : 30]
SFEC	Standard Filter Element Configuration	[29 : 27]
SFID1	Standard Filter ID 1	[26 : 16]
Reserved	Reserved	[15 : 11]
SFID2	Standard Filter ID 2	[10 : 0]

4.2.3.1 Standard Filter Type (SFT)

The standard filter type sets what type of filter this element is. This describes how to match this filter with an incoming CAN message

Table 9. SFT

Value (Binary)	Description
2b00	Range Filter: Accepts all IDs from SFID1 to SFID2
2b01	Dual ID Filter: Matches both SFID1 and SFID2 (ID must be exact match)
2b10	Classic Filter: SFID1 is the filter, SFID2 is the mask. 0 in the mask corresponds to "don't care". See below for more information
2b11	Filter Element Disabled: This filter will match nothing.

For the range and dual ID filter, the ID of the incoming CAN message must match (or be within the range) of the values SFID1 and SFID2. The classic filter uses a filter and mask setup, which allows a designer to match all IDs that have desired bits set in the ID. Table 10 shows a filter and mask example. A 0 in the mask corresponds to a "don't care" and a 1 signifies that the bit must match the value in the filter.

Table 10. Filter and Mask Example

	Example 1	Example 2	Example 3	Example 4	Example 5
Filter (SFID1)	11b001 0010 1111	11b001 0010 1111	11b001 0010 1111	11b000 0000 0000	11b000 0000 0000
Mask (SFID2)	11b000 1111 1111	11b000 1111 1111	11b000 1110 1111	11b000 0000 1000	11b000 0000 1000
Example ID	11b001 *0001* 1111	11b001 0010 1111	11b001 0011 1111	11b000 0000 *1*000	11b111 1111 0111
Result	Not matched	Matched	Matched	Not matched	Matched

4.2.3.2 Standard Filter Element Configuration (SFEC)

The standard filter element configuration describes the action to be performed if an incoming CAN message matches the filtering rule described SFT (and the values in SFID1 and SFID2). There are 8 options available for matching

Table 11. SFEC

Value (Binary)	Description
3b000	Disable filter element. Does nothing, matches nothing
3b001	Store in Rx FIFO 0
3b010	Store in Rx FIFO 1
3b011	Reject message. No interrupt will be set, and message is simply ignored.
3b100	Set as priority message (interrupt), message storage location depends on other filters or default behavior
3b101	Set as priority (interrupt) and store in FIFO 0
3b110	Set as priority (interrupt) and store in FIFO 1
3b111	Store into Rx Buffer or as debug message. If this is used, SFT is ignored and SFID1 is the filter. SFID2[10:9] describes where to store message, SFID2[5:0] describes which Rx Buffer to put the message (must be within the Rx Buffer configuration)

4.2.4 29-bit Filter (XID Filter)

An XID filter operates just like the SID filter, except that it is ran against extended IDs. Please read Section 4.2.3 for more information. The values and behavior are the same for XID, but with 29-bit filters and masks instead of 11-bit filters used by the SID filter. This means that each element consists of 2 words (8 bytes) instead of 1 word (4 bytes). Table 12 shows the XID Filter element.

As stated earlier in Section 4.2, since MRAM values are unknown after power up and the ECC values are not valid. It is important that data must be written to each of the filter locations in MRAM that are configured for use in the SIDFC or XIDFC registers. Failure to do this results in a M_CAN BEU error, which will put the TCAN45xx device into initialization mode, and require user intervention before CAN communication can continue.

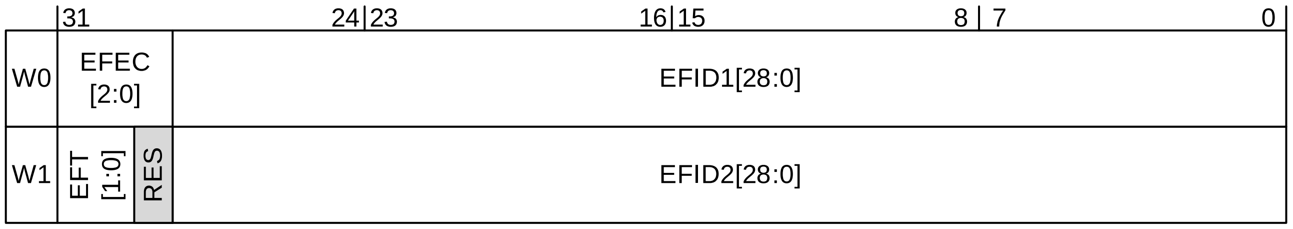

Figure 9. XID Filter Element

Table 12. XID Filter Element

Name	Description	Word	Bits
EFEC	Extended Filter Element Configuration	W0	[31 : 29]
EFID1	Extended Filter ID 1		[28 : 0]
EFT	Standard Filter ID 1	W1	[31 : 30]
Reserved	Reserved		[29]
EFID2	Extended Filter ID 2		[28 : 0]

4.2.5 Rx FIFO 0 / 1

The Rx FIFO gives the user 2 separate FIFOs, if so desired, to choose where to place incoming CAN messages. For example, a designer might want all 11-bit ID messages to go into Rx FIFO 0, but all 29-bit ID messages to go into Rx FIFO 1. Another example is all messages which are low priority (based on ID) go to Rx FIFO 1, and all high priority messages go to Rx FIFO 0. There is no restriction on how these FIFOs may be used, and there is no requirement that both must be configured. It is acceptable to have only Rx FIFO 0 or Rx FIFO 1 (as a note, the default behavior of the TCAN45xx is to accept all packets into Rx FIFO 0 unless the packet matches a filter, then the action described in the filter will be performed).

Each Rx FIFO element contains an 8 byte header, which contains the received ID, data length code (DLC), and many diagnostic flags, such as whether CAN FD was used, or if bit rate switching was enabled. A timestamp is also contained in this header, which allows the designer to know when the packet was received. After the 8 byte header, there is data from the incoming message, up to the maximum allowed data size as described in the Rx FIFO configuration register. If the DLC is greater than the maximum data size described in the configuration register, then the data is truncated and anything past the maximum data size is lost. If the DLC is less than the maximum data size allowed, then the designer must not assume that any data after the DLC-described size is valid. The TCAN45xx does NOT clear out the data fields. For example, if the Rx FIFO is configured to allow up to 8 bytes, but the DLC is 7 bytes, then the last byte of data in memory is not automatically cleared, and should not be used. It contains the last value that was written by a previous CAN message.

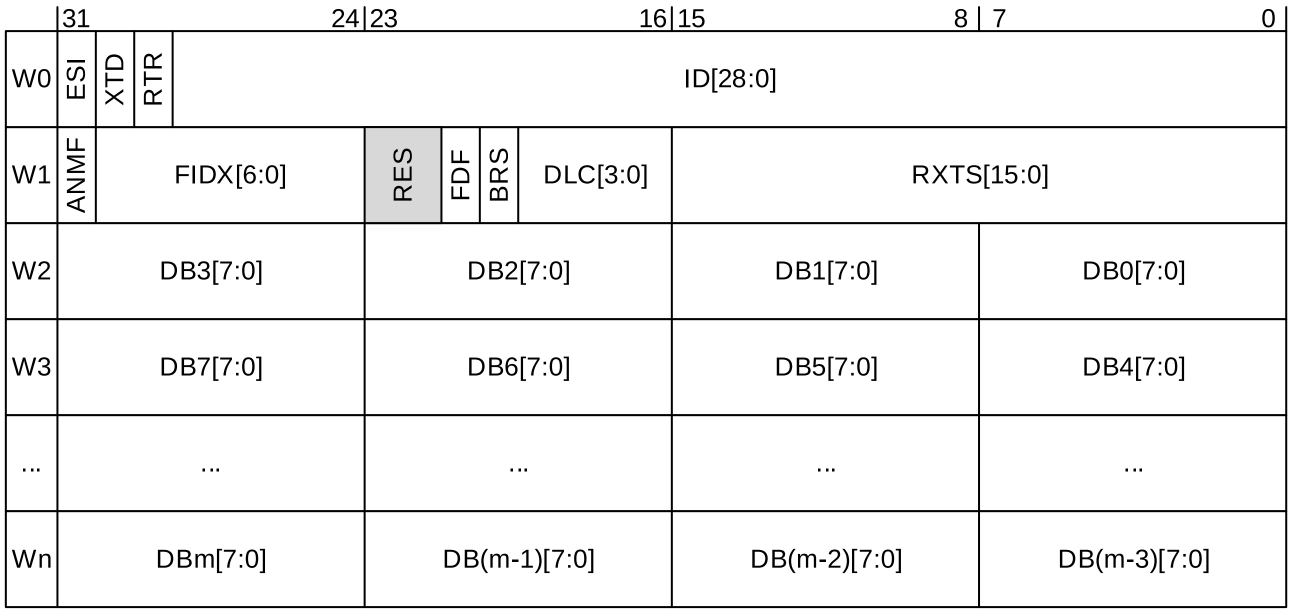

Figure 10. Rx FIFO / Buffer Element

Table 13. Rx FIFO / Buffer Element

Name	Description	Word	Bits
ESI	Error State Indicator	W0	[31]
XTD	Extended Identifier (When set to 0, ID[28 : 18] is used for standard ID)		[30]
RTR	Remote Transmission Request		[29]
ID	Identifier (When XTD is 0, ID[28 : 18] is used for standard ID)		[28 : 0]
ANMF	Accepted Non-matching Frame of Filter Element	W1	[31]
FIDX	Filter Index that Message Matched if ANMF = 0		[30 : 24]
Reserved	Reserved		[23 : 22]
FDF	FD Format		[21]
BRS	Bit Rate Switch		[20]
DLC	Data Length Code		[19 : 16]
RXTS	Rx Timestamp		[15 : 0]
DB3	Data Byte 3	W2	[31 : 24]
DB2	Data Byte 2		[23 : 16]
DB1	Data Byte 1		[15 : 8]
DB0	Data Byte 0		[7 : 0]
DBm	Data Byte m	Wn	[31 : 24]
DB(m-1)	Data Byte (m-1)		[23 : 16]
DB(m-2)	Data Byte (m-2)		[15 : 8]
DB(m-3)	Data Byte (m-3)		[7 : 0]

4.2.6 Rx Buffers

The Rx Buffers allow the designer to place certain CAN messages into a buffer. These buffers contain the same 8 byte header of the Rx FIFOs (see Section 4.2.5) and the data comes after the header. The only difference is that these buffers do not behave like a FIFO. A SID or XID filter element must explicitly tell the TCAN45xx which of the Rx Buffers an incoming message must be sent to. This also means that if a new packet is matched and a filter instructs it to be moved to a Rx Buffer, which already has unread data in it, that data in the buffer is over-written.

4.2.7 Tx Event FIFO

The Tx Event FIFO stores messages from the TCAN45xx when a request to transmit a CAN message from the Tx Buffer has been received. Each FIFO element is 8 bytes long and contains the ID that was transmitted, whether CAN FD or bit rate switching was used or not, as well as diagnostic flags such as a transmission time stamp, and the error state indicator bit. In order to have a Tx Event FIFO element written, when transmitting a message, the Event FIFO Control bit (EFC) must be set to 1 in the corresponding Tx Buffer header.

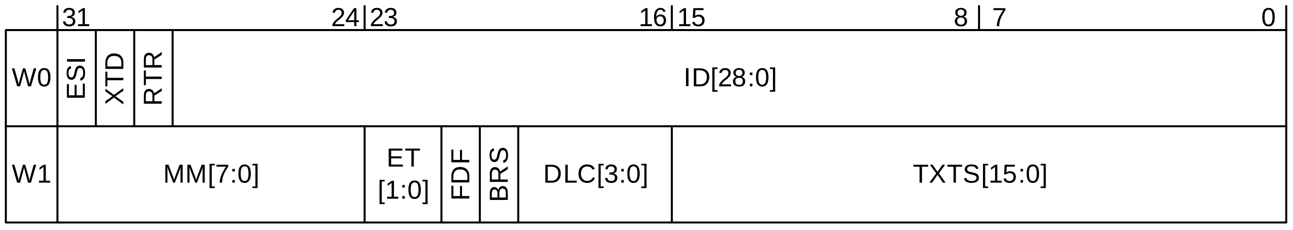

Figure 11. Tx Event FIFO Element

Table 14. Tx Event FIFO Element

Name	Description	Word	Bits
ESI	Error State Indicator	W0	[31]
XTD	Extended Identifier (When set to 0, ID[28 : 18] is used for standard ID)		[30]
RTR	Remote Transmission Request		[29]
ID	Identifier (When XTD is 0, ID[28 : 18] is used for standard ID)		[28 : 0]
MM	Message Marker	W1	[31 : 24]
ET	Event Type: 2b01 = Tx Event, 2b10 = Transmission in spite of cancellation		[23 : 22]
FDF	FD Format		[21]
BRS	Bit Rate Switch		[20]
DLC	Data Length Code		[19 : 16]
TXTS	Tx Timestamp		[15 : 0]

4.2.8 Tx Buffers

Tx Buffers are used as the buffer to load an outgoing CAN message details and data. The layout of each element is very similar to the Rx FIFO / Buffer (see Section 4.2.5) where there is an 8 byte header used, and the data payload to be sent comes after it. There 3 main types transmission topologies available:

1. Dedicated Tx Buffers: Each individual buffer is handled by the microcontroller, with the intent of each buffer having its own message ID
2. Tx FIFO: The TCAN45xx handles buffer management. The microcontroller reads the Tx FIFO put index to place new messages in the correct buffer. When requesting multiple messages be sent, the Tx FIFO get index is referenced and sends data based on what was first added to the FIFO
3. Tx Queue: Similar to a Tx FIFO except that consecutive messages do not need to be placed in consecutive buffers. When writing to a buffer, the destination buffer must not have a pending

transmission request. If multiple transmission requests are made at the same time via the TXBAR register, the TCAN45xx prioritizes messages with the lowest ID (highest priority). The priority is not based on the order which the transmission request was made or data was written into the queue.

It is possible to have both dedicated Tx Buffers and either a Tx FIFO or Tx Queue. In this situation, the dedicated buffers consist of the first buffers, and the FIFO or queue start after the end of the dedicated buffers.

As stated earlier in Section 4.2, since MRAM values are unknown after power up and the ECC values will not be valid, it is important that at least 2 words (8 bytes) of payload data be written into any TX buffer element, even if the DLC is less than 8. Failure to do this will result in a M_CAN BEU error, which puts the TCAN45xx device into initialization mode, and require user intervention before CAN communication can continue.

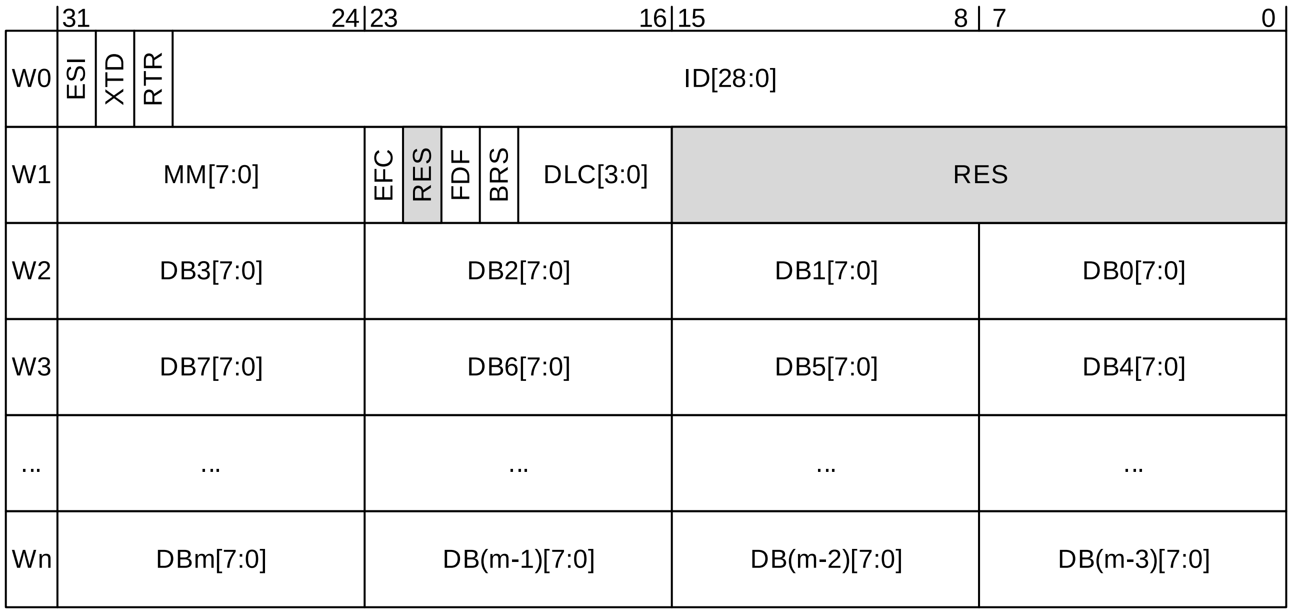

Figure 12. Tx FIFO / Buffer Element

Table 15. Tx FIFO / Buffer Element

Name	Description	Word	Bits
ESI	Error State Indicator	W0	[31]
XTD	Extended Identifier (When set to 0, ID[28 : 18] is used for standard ID)		[30]
RTR	Remote Transmission Request		[29]
ID	Identifier (When XTD is 0, ID[28 : 18] is used for standard ID)		[28 : 0]
MM	Message Marker, used when storing a Tx Event FIFO entry	W1	[31 : 24]
EFC	Event FIFO Control, stores a log in the Tx Event FIFO when set to 1		[23]
Reserved	Reserved		[22]
FDF	FD Format		[21]
BRS	Bit Rate Switch		[20]
DLC	Data Length Code		[19 : 16]
Reserved	Reserved		[15 : 0]

Table 15. Tx FIFO / Buffer Element (continued)

Name	Description	Word	Bits
DB3	Data Byte 3	W2	[31 : 24]
DB2	Data Byte 2		[23 : 16]
DB1	Data Byte 1		[15 : 8]
DB0	Data Byte 0		[7 : 0]
DBm	Data Byte m	Wn	[31 : 24]
DB(m-1)	Data Byte (m-1)		[23 : 16]
DB(m-2)	Data Byte (m-2)		[15 : 8]
DB(m-3)	Data Byte (m-3)		[7 : 0]

4.3 Sending and Receiving CAN Messages

In order to transmit a CAN message on the TCAN45xx, the following should be complete:

1. Ensure that the TCAN45xx is in standby mode (register 0x0800[7:6] = 0b01). This forces M_CAN into INIT mode.
2. Set the M_CAN CCCR register to allow for configuration. Set CCE and INIT bits if not already set. NOTE: The CSR bit reads back a 1 when in standby mode, but the user MUST write a 0 to this bit when doing a read-modify-write; otherwise, CAN communications fails.
3. If CAN FD and Bit Rate Switching (BRS) support is desired, it must be globally enabled via the FDF and BRS bits in the CCCR register during configuration. See the device datasheet for more information about this register
4. Any desired device features should be configured (see Section 3)
5. CAN timing information must be set (see Section 4.1)
6. The MRAM sections should be configured and initialized with any data (see Section 4.2)
7. Put the TCAN45xx device into "normal" mode (register 0x0800[7:6] = 0b10) to turn on the transceiver and enable the CAN core for transmission

Once these steps are complete, the microcontroller is able to transmit a message by writing to the Tx Buffer and then requesting the message be sent by writing to the TXBAR register.

4.3.1 Writing a CAN Message to the Tx Buffer

For this example CAN message, the example device configuration from Section 4.2.2 are used for Tx Buffer elements, maximum data size, and start address.

Table 16. Example CAN Message

Parameter	Value
ID	0x12345678
Message Marker	0x1
Message Format	CAN FD with Bit Rate Switching enabled[(1)]
Data	0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77

(1) CAN FD and BRS must be enabled globally in the CCCR register during configuration by setting the FDF and BRS flags.

With the desired CAN message information in Table 16, it the following procedure can take place

1. Check Tx FIFO/Queue Status register (TXFQS: 0x10C4) bits [5:0] to make sure the free level is greater than 0 (meaning that at least 1 buffer is open/free) and that TFQF bit is set to 0
2. Read TXFQS.TFQPI to get which index the message should be loaded into
3. Calculate the memory offset to determine the start address. For this example, it is assumed that the Put Index read back as 3, note that what your read may be different. Buffer address = Tx Buffer Start address + (Tx Buffer Element Size x Put Index). In this example, it becomes 0x8264 + (0x48 x 0x3) = 0x833C. Note that 0x48 is a hexadecimal value, corresponding to 72. The TX Buffer Element size is

the sum of the 2 words of header (8 bytes) and the configured element data size (64 bytes in this example).

4. Write the CAN message to memory
5. Send transmission request to the TCAN45xx

To achieve this procedure, the SPI writes are shown below in Table 17

Table 17. SPI Writes

Register	Address	Action	Value	Notes
TXFQS	0x10C4	Read	0x0003030A	Put index is 3, free level is 10, bit for full FIFO is NOT set
Tx Buffer[3]	0x833C	Write	0x52345678	Header word #1, XTD and ID set
Tx Buffer[3]	0x8340	Write	0x01B70000	Header word #2, Set message marker, enable event FIFO, CAN FD and BRS, and set the DLC to 7
Tx Buffer[3]	0x8344	Write	0x44332211	Write the first 4 bytes of data
Tx Buffer[3]	0x8348	Write	0x00776655	Write the last 3 bytes of data
TXBAR	0x10D0	Write	0x00000008	Request buffer 3 (bit 3) start transmission

After the SPI writes listed in Table 17, the TCAN45xx starts the transmission process for the data in buffer index 3. Since the event FIFO bit was set, an entry is added to the event FIFO, which the microcontroller may read.

4.3.2 Reading a CAN Message from a Rx FIFO

For this example CAN message, the example device configuration from Section 4.2.2 is used for Rx Buffer elements, maximum data size, and start address.

To read a message from the TCAN45xx FIFO, the process can be broken down into the following steps

1. Determine where the new message is (Rx FIFO 0, Rx FIFO 1, or Rx Buffer)
2. Based on the buffer location of the new message, determine the buffer index and then the start address to read from MRAM
3. Read the MRAM to retrieve the message
4. Acknowledge the new message is read to release the FIFO element for a new message

The process varies slightly depending on if the new message is in a Rx Buffer or a Rx FIFO; since they are fundamentally different. The FIFOs require the microcontroller to read a FIFO status register which tells the microcontroller how many new messages are in the FIFO, and what index to start reading at. The buffer requires the microcontroller to read the New Message Register, which tells the microcontroller which buffers have unread messages in them. At the end of each read, the microcontroller must let the TCAN45xx know that the new message has been received in order to release the FIFO element for reuse. The example below will give an example of responding to a new message interrupt in Rx FIFO 1, and that the message received was the message described in Section 4.3.1. The process of reading a message from Rx FIFO 0 is almost identical, except, instead of reading FIFO 1 registers, FIFO 0 registers are read instead. For example, instead of reading RXF1S to get the get index to calculate the start address, the microcontroller would read RXF0S. The IR register has separate bits for Rx FIFO 0 or Rx FIFO 1 new messages, to let the microcontroller know which set of registers should be read.

Table 18. SPI Reads

Register	Address	Action	Value	Notes
Device Interrupts	0x0820	Read	0x80000082	M_CAN_INT bit is set, so M_CAN has an interrupt
IR	0x1050	Read	0x00000010	New message in RX FIFO 1, need to read RX FIFO 1 status to get more information
IR	0x1050	Write	0x00000010	Clear the interrupt by writing the bit back to the IR register
RXF1S	0x10B4	Read	0x00040301	There is 1 unread message in the FIFO at index 3
-	0x81C8	-	-	Based on setup in Table 7, start address for index 3 is: hex(72 * 3) + 0x80F0 = 0x81C8

Table 18. SPI Reads (continued)

Register	Address	Action	Value	Notes
Rx FIFO 1 [3]	0x81C8[1]	Read	0x52345678	Header word #1, XTD is set and ID[28:0] = 0x12345678
Rx FIFO 1 [3]	0x81CC	Read	0x01B70000	Header word #2, 7 bytes of data sent with CAN FD and BRS enabled
Rx FIFO 1 [3]	0x81D0	Read	0x44332211	First 4 bytes of data, 0x11 is the first received byte
Rx FIFO 1 [3]	0x81D4	Read	0x00776655	Last 3 bytes of data, 0x77 was the last received byte
RXF1A	0x10B8	Write	0x00000003	Write the index of the FIFO index read to acknowledge that it has been read and clear it for use

(1) This start address is calculated with [(Data Size + 8) x Get Index] + Start Address. Data size comes from the element size configuration (RXESC) plus the 8 bytes of header. Get index is read from the RXF1S register. Start address is in the RXF1C register, but without the 0x8 prefix.

4.3.3 Reading a CAN Message from a Rx Buffer

Reading a message from the Rx Buffer is similar to the process of reading a message from a Rx FIFO (as shown in Section 4.3.2) but calculating the start address varies. The difference is that the Rx Buffer index to use for the address calculation comes from the New Data 1/2 registers instead of a Rx FIFO 0/1 Status register.

In the below example, it is assumed that the Rx Buffer element is configured with 5 elements, with a start address configured at 0x8100 and RXESC configured for Rx Buffer data size of 32 bytes (unrelated to any previous setup examples).

Table 19. SPI Reads

Register	Address	Action	Value	Notes
Device Interrupts	0x0820	Read	0x80000082	M_CAN_INT bit is set, so M_CAN has an interrupt
IR	0x1050	Read	0x00080000	New message sent to dedicated Rx Buffer
IR	0x1050	Write	0x00080000	Clear the interrupt by writing the bit back to the IR register
NDAT1	0x1098	Read	0x00000004	There is 1 new message in Rx Buffer index 2 (3rd element)
-	0x8150	-	-	Start address for index 2 is: hex(40 * 2) + 0x8100 = 0x8150
Rx Buffer [2]	0x8150[1]	Read	0x52345678	Header word #1, XTD is set and ID[28:0] = 0x12345678
Rx Buffer [2]	0x8154	Read	0x01B70000	Header word #2, 7 bytes of data sent with CAN FD and BRS enabled
Rx Buffer [2]	0x8158	Read	0x44332211	First 4 bytes of data, 0x11 is the first received byte
Rx Buffer [2]	0x815C	Read	0x00776655	Last 3 bytes of data, 0x77 was the last received byte
NDAT1	0x1098	Write	0x00000004	Write the bit corresponding to the buffer that was read

(1) This start address is calculated via with [(Data Size + 8) x Get Index] + Start Address. Data size comes from the element size configuration (RXESC) plus the 8 bytes of header. Get index is read from the NDAT1/NDAT2 register. Start address is in the RXBC register, but without the 0x8 prefix.

5 Software Optimization Tips

This section outlines common issues, and tips on how ensure maximum performance from the software-side.

5.1 SPI Transaction Idle Time

The most common issue that hurts CAN throughput is poor SPI throughput. The following are common causes with examples following the graphical representation in Figure 13.

1. Idle time between $\overline{\text{CS}}$ and start of data
2. Idle time between MCU's SPI Words
3. Idle time between end of data and $\overline{\text{CS}}$

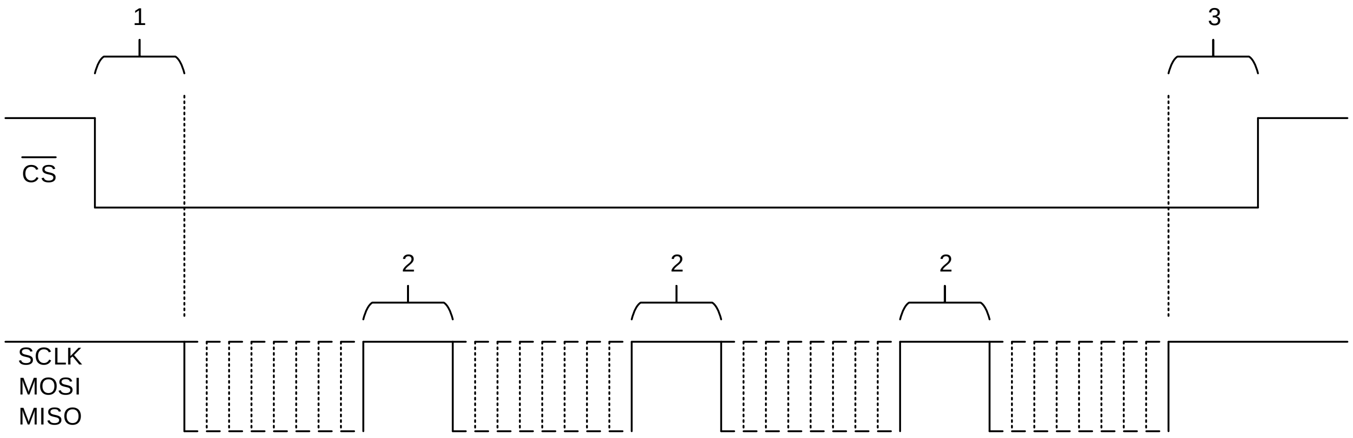

Figure 13. An Example Inefficient SPI Transfer

These issues add to the idle time spent doing nothing, which limits how much data can be passed to the TCAN45xx. To fix these issues, an oscilloscope is required. It will take time to optimize the SPI driver code to the point where there are no significant delays. While the steps to optimize the code is processor-specific, the following are things to look for to improve SPI performance:

- Use MCU's SPI hardware FIFOs: Most processors' SPI module support FIFO operation, which allows the user to queue up several words of data at once. The SPI hardware then automatically shifts data onto the bus with minimal between-word/byte delays (labeled at 2 in Figure 13).
- Use the MCU's SPI hardware chip select control: It is typically best to use the SPI module's chip select control logic instead of a GPIO that is controlled via software. The minimum spacing for #1 and #3 in the picture above is a limit of the SPI hardware in the MCU. Letting the hardware module control when the chip select normally ensures that these delays are kept to a minimum. This is not always possible, since some hardware SPI modules are incapable of holding chip select low during a large SPI transfer (8 bytes is the minimum transfer size of the TCAN45xx). This optimization may not be possible on all MCU's/processors. In these situations, software control of chip select is required.

The goal of the SPI transaction is to look like Figure 14, where no time is wasted during the SPI transaction. This maximizes SPI throughput, which allows the CPU to spend less time on the actual SPI transactions, and more time on the CAN messages.

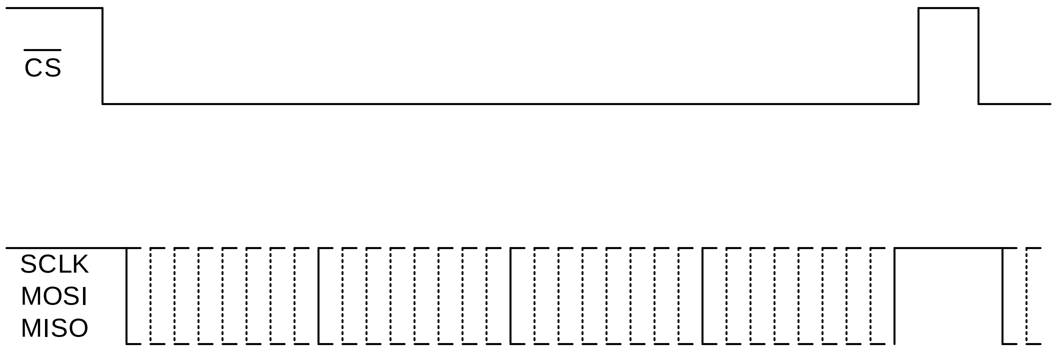

Figure 14. An Example Efficient SPI Transfer

5.2 Use Burst Reads and Writes

Since every SPI transaction requires a single word (4 bytes) header, doing single word data transfers is inefficient (only 50% of the SPI bus data is register data). By using larger SPI packets, throughput can be significantly increased by transferring large chunks of data when possible, such as reading or writing a CAN message from/to the TCAN45xx). The TCAN45xx SPI header has a word parameter, which tells it how many words (4 bytes each) of data will be transferred. Valid values are 0 to 255 words (0x00, 0xFF), where 0 is interpreted as 256 words.

In Figure 15, an example 4 words of continuos data are transferred with single word transactions only. It is easy to see how throughput is hurt since each transfer requires a new SPI header. Over large blocks of data, this inefficiency will hinder throughput.

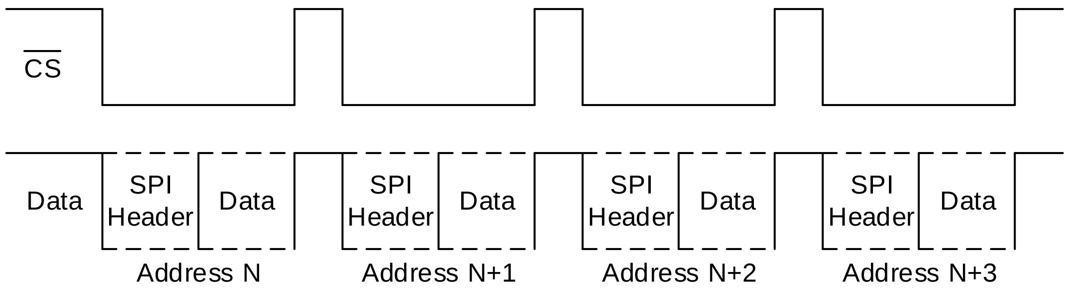

Figure 15. Inefficient Transfer of Large Data

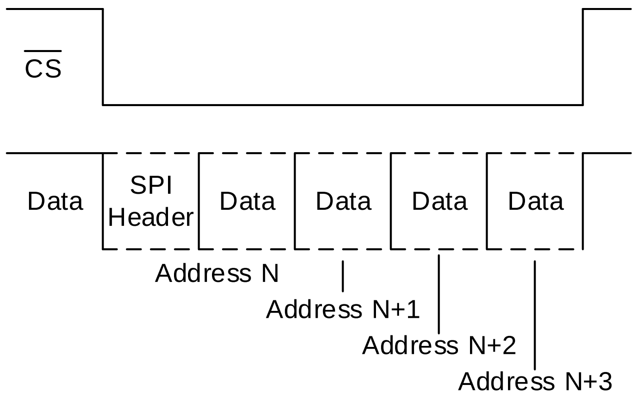

Figure 16. Efficient Transfer of Large Data

As shown in Figure 16, a burst transfer is used. In this specific example, 4 words of data are being transferred in a single SPI transaction. Using this burst mode whenever possible helps keep SPI throughput up. Some common tasks for the TCAN45xx that is able to take advantage of this feature are as follows:

- MRAM Reads or Writes: Since the MRAM is a large 2 KB block of memory that is read from and written to by the host processor, burst transfers are helpful when attempting to read or write CAN messages where many words of data are typically moved.
- M_CAN Configuration: Upon configuration of the TCAN45xx, many of the registers must be written to. It is possible to use burst transfers to configure the part is few SPI transactions. This is a start up benefit, rather than an operating benefit.

5.3 Bulk Reading Incoming CAN Messages

In many systems, there is a delay between a CAN message being received by the TCAN45xx and the host processor reading it. In some cases, this initial latency could be large enough that multiple CAN messages is received before the host intervenes. The process to read a CAN message is typically the following:

1. Read Interrupt Registers to see if new message interrupt is set.
2. Read the RXFIFO status register. This will tell the host which index to read, and how many unread messages there are in the FIFO.
3. Calculate the address to read from (see Section 4.3.2 for more information) and perform the read from MRAM.
4. Write the index to the acknowledge index to release it for accepting new messages.
5. Return to step 2 until the number of unread messages is 0.

This procedure is the regular and easier to follow method for reading a message from the MRAM, but it is not the most efficient. To improve on this procedure, extra reads to the RXFIFO status registers can be omitted, as well as writes to the acknowledge register. This tip only provides an advantage when there is more than 1 new message in the FIFO. An outline of this improved procedure is below.

1. Read Interrupt Registers to see if new message interrupt is set.
2. Read the RXFIFO status register. From this, the starting index and current fill level are both known.
3. Calculate the address to read from (see Section 4.3.2 for more information) and perform the read from MRAM. Repeat this step for the number of unread messages that was read in step 2. As a note, it is possible to optimize this step further and perform a single SPI transaction to pull multiple FIFO elements of data at a time. NOTE: The FIFOs on this device are circular, and the data does NOT move. This means that care must be taken when reading messages that loop around the end of the FIFO, since the host has to start a new SPI transaction for index 0.
4. Once all messages were read from the MRAM, write only the index of the last read message to the acknowledge register. The TCAN45xx will assume that all messages from the first to the supplied index have been read and releases them all.
5. It is recommended to go back to step 2 and see if any new messages came in during this burst read procedure. If not, then exit the read routine.

WARNING

There is a concern regarding the depth of the FIFO for this tip. Care must be taken to ensure that the FIFO does NOT overflow while a read is in progress. Since acknowledging of the FIFO doesn't occur until the end of reading ALL new messages, if the host does not perform this process fast enough while many new messages are coming into the TCAN45xx, it is possible that messages could be lost. This concern exists for the standard method of reading messages, but in certain situations where there are several messages in the FIFO, and not many elements free for new messages, delaying the release of the registers could more easily cause data loss.

When burst reads are being performed, care must be taken to ensure that a read over flow of the MRAM FIFO location does not occur. For example, if a FIFO has 10 elements, and the initial status read says the get index is 8 and there are 3 messages to be read, the software will need to perform a read to indexes 8, 9, 0. Trying to read index 10 (the 11th element) of a 10 element FIFO will result in an out of bounds issue and the data will not be expected CAN message.

6 Software Libraries

6.1 AutoSAR

AutoSAR (Automotive Open Systems Architecture) is a partnership of automotive-interested parties that have worked towards developing a specification for automotive systems. It provides specifications that describe the software modules which communicate with hardware and builds a common methodology of application development. The advantage of such specifications is that drivers for different hardware modules can be written in a way that are easily adaptable for many different processors or microcontrollers, since they are not built for a specific piece of hardware, but rather rely on lower-layer function calls which are defined by AutoSAR.

Figure 17 shows the different layers and how a user's application can communicate with hardware through the AutoSAR layer. There are 3 main types of AutoSAR blocks which play into the system.

- MCAL: The Microcontroller Abstraction Layer (MCAL) provides standardized function calls to peripherals of a microcontroller such as the IOs, bus ports (SPI or I2C), or even memory. By abstracting these peripherals to common function calls, it makes it possible to simply swap MCAL drivers for a different processor when migrating, but all the code above should behave the same with very little to no modifications. These drivers are processor/vendor specific since they deal with the specific code required to perform a function
- ECU Abstraction Layer: This layer provides a common set of functions that an electronic control unit (ECU) would use such as CAN communication or GPIOs for sensing buttons or controlling lights. These modules are designed to be processor independent and call on the MCAL layer to handle whatever functions are required. In the case of the TCAN45xx, the MCAL driver for SPI is used to communicate to the TCAN45xx.
- Services Layer: This layer provides background services to the application such as network services and bus communication services. An example for the TCAN45xx is that a beacon bus communication may be desired (a periodic bus ping to check status of devices on the bus) and this service communicates to the TCAN45xx ECU abstraction driver in order to actually send and receive the CAN messages.

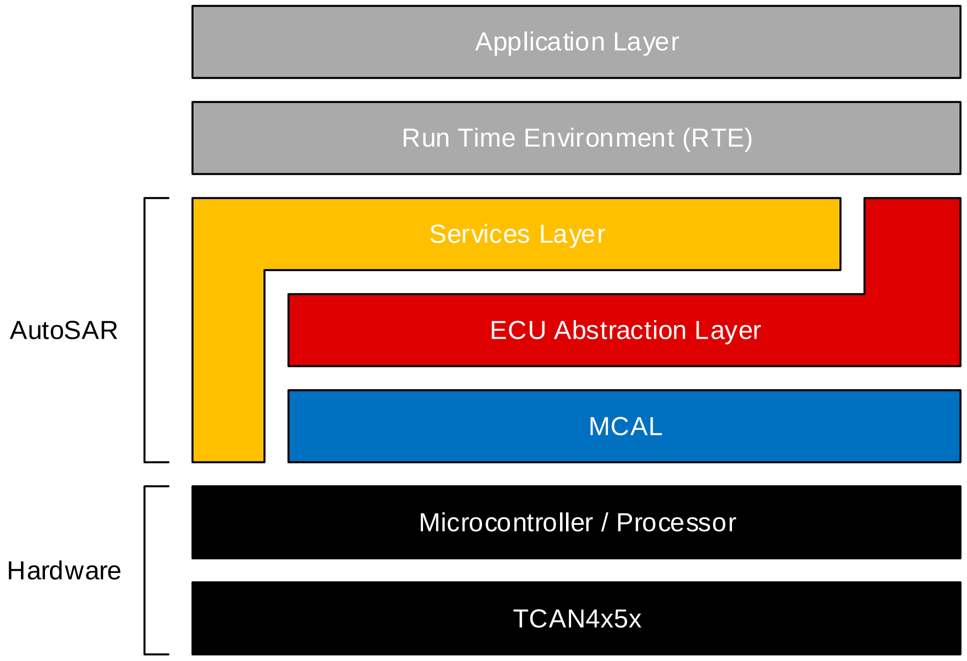

Figure 17. AutoSAR Abstraction Layers

Texas Instruments provides the necessary ECU abstraction layer driver source code necessary to integrate the TCAN45xx into an AutoSAR environment. Contact TI to get the source code.

6.2 *Microcontroller Abstraction*

A lower-level API is available for controlling various TCAN45xx functions and sending CAN messages. It does not have as significant of an overhead as AutoSAR, but is not abstracted as much. These libraries are provided to help a developer add the TCAN45xx to their system. This API abstracts TCAN45xx function calls to call upon a SPI abstraction layer function, allowing the user to quickly change code to a different microcontroller by only changing the code which controls the SPI peripheral. There are 3 main layers in this abstraction

- SPI Abstraction Layer: Responsible for handling the processor-specific SPI function calls to perform register reads and writes to the TCAN45xx
- TCAN45xx API Layer: Provides a set of functions for performing TCAN45xx actions. For example,

functions for reading a CAN message, or sending a CAN message

- Application Layer: The end-user's code which calls upon the API to communicate with the TCAN45xx easily and without much overhead

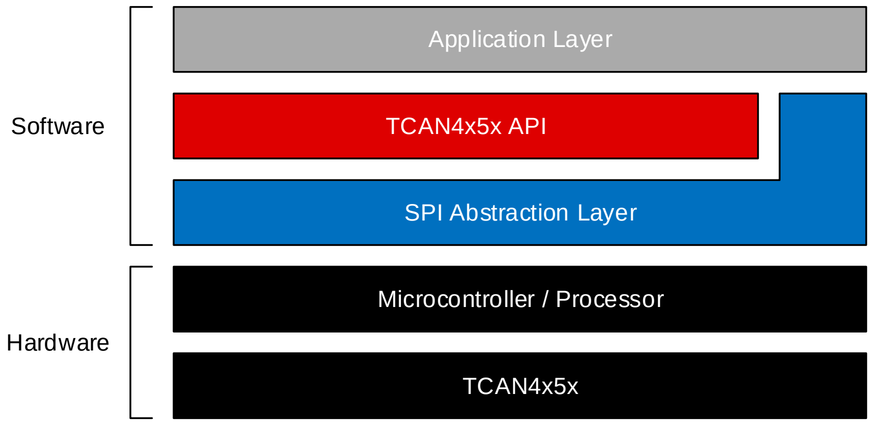

Figure 18. Microcontroller Abstraction

Texas Instruments provides Microcontroller Abstraction source code but it is important to note that the SPI Abstraction Layer driver is developed for a MSP430FR6989. The appropriate SPI drivers will need to be written for other processors.

6.2.1 Updating the SPI Abstraction Layer for Other Microcontrollers

All of the code in the TCAN45xx API software layer relies on the SPI Abstraction Layer. There are 8 functions that reside in this layer. The first 2 listed below are the fixed-length single-word read and write functions. The next 6 are the multi-word read and write functions

- uint32_t AHB_READ_32 (uint16_t address) : Single-register 32-bit word read
- void AHB_WRITE_32 (uint16_t address, uint32_t data) : Single-register 32-bit word write
- void AHB_READ_BURST_START (uint16_t address, uint8_t words) : Send the SPI header for a multi-word read, providing the starting register address and how many words are read
- uint32_t AHB_READ_BURST_READ ( ) : Returns a 32-bit word of data that is read, without toggling the $\overline{\text{CS}}$ pin
- void AHB_READ_BURST_END ( ) : At the end of a multi-register read, this function ends a SPI transaction by pulling the $\overline{\text{CS}}$ pin high
- void AHB_WRITE_BURST_START (uint16_t address, uint8_t words) : Send the SPI header for a multi-word write, providing the starting register address and how many words are written
- void AHB_WRITE_BURST_WRITE (uint32_t data) : Writes a 32-bit word of data, without toggling the $\overline{\text{CS}}$ pin
- void AHB_WRITE_BURST_END ( ) : At the end of a multi-register write, this function ends a SPI transaction by pulling the $\overline{\text{CS}}$ pin high

For the multi-register read and write functions, there are 3 individual function calls to perform the read or write. The start of the SPI transaction transmits the correct command code, register address, and number of words that is transmitted but does not start any data transfer. The READ/WRITE functions do the actual data transfer to make it easier to put into a loop to handle each word of reading and writing. The END function will pull the $\overline{\text{CS}}$ pin high to signal to the TCAN45xx that the SPI transaction is complete.

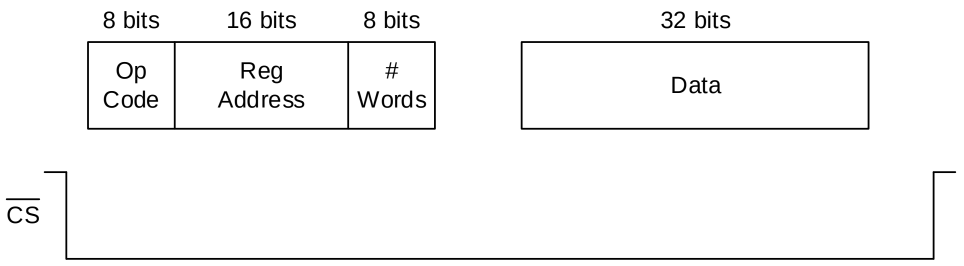

Figure 19. 32-bit SPI Read or Write Example

In Figure 19, a single 32-bit (1 word) SPI read or write example is shown. The first word contains the SPI header, which tells the TCAN45xx what action to perform (read or write), what register address to start at, and how many words of data to read/write.

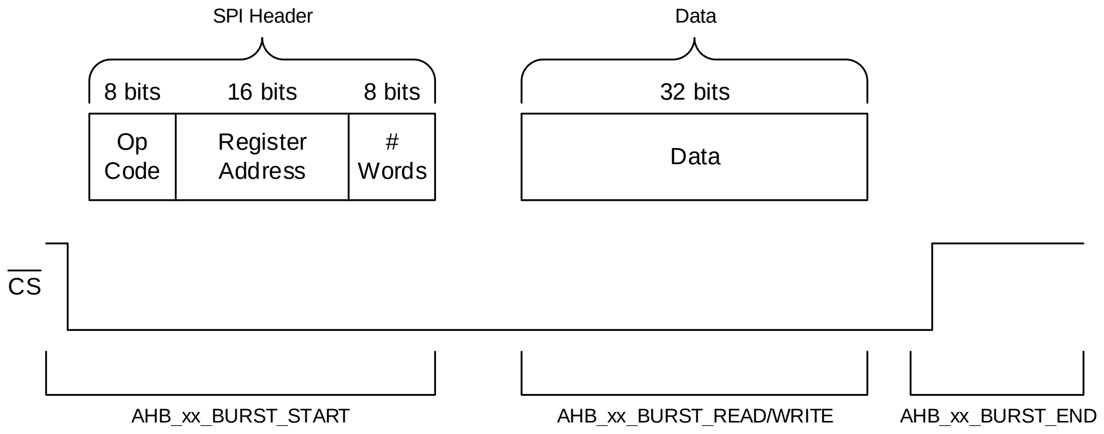

Figure 20. SPI Packet Breakdown

Figure 20 shows how a single word read or write can be broken up into 3 functions. The START function pulls $\overline{\text{CS}}$ low and sends the SPI header. The READ/WRITE function is responsible for reading or writing a single word at a time, for however many words were stated in the SPI header. The END function is responsible for pulling $\overline{\text{CS}}$ high, to signal the end of the transfer.

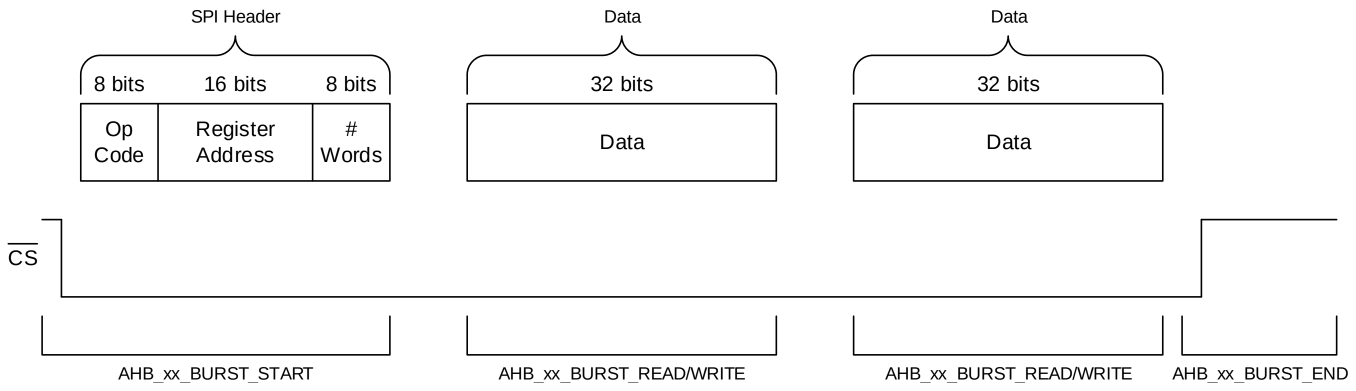

Figure 21. Multi-word SPI Packet Example

Figure 21 shows a 2 word SPI transfer example, and how the READ/WRITE function is called twice to do each individual word transfer. This is helpful for cutting down on SPI overhead when transferring a CAM message to or from the TCAN45xx, and minimizes the maximum SPI frequency required.