Enhanced Controller Area Network (CAN)

HIGHLIGHTS

This section of the manual contains the following major topics:

Note: This family reference manual section is meant to serve as a complement to device data sheets. Depending on the device variant, this manual section may not apply to all dsPIC33E/PIC24E devices.

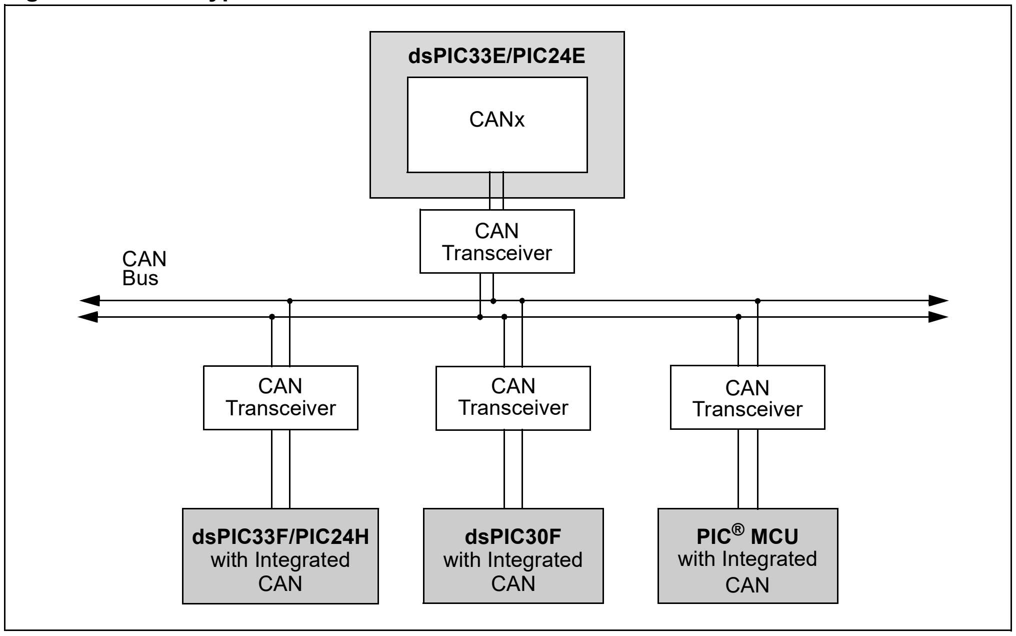
Please consult the note at the beginning of the **"Enhanced Controller Area Network (CAN)"** chapter in the current device data sheet to check whether this document supports the device you are using.

Device data sheets and family reference manual sections are available for download from the Microchip Worldwide Website at: http://www.microchip.com.

1.0 INTRODUCTION

The dsPIC33E/PIC24E Enhanced Controller Area Network (CAN) module implements the CAN Specification 2.0B, which is used primarily in industrial and automotive applications. This asynchronous serial data communication protocol provides reliable communications in an electrically noisy environment. Figure 1-1 illustrates a typical CAN bus topology.

Figure 1-1: Typical CAN Bus Network

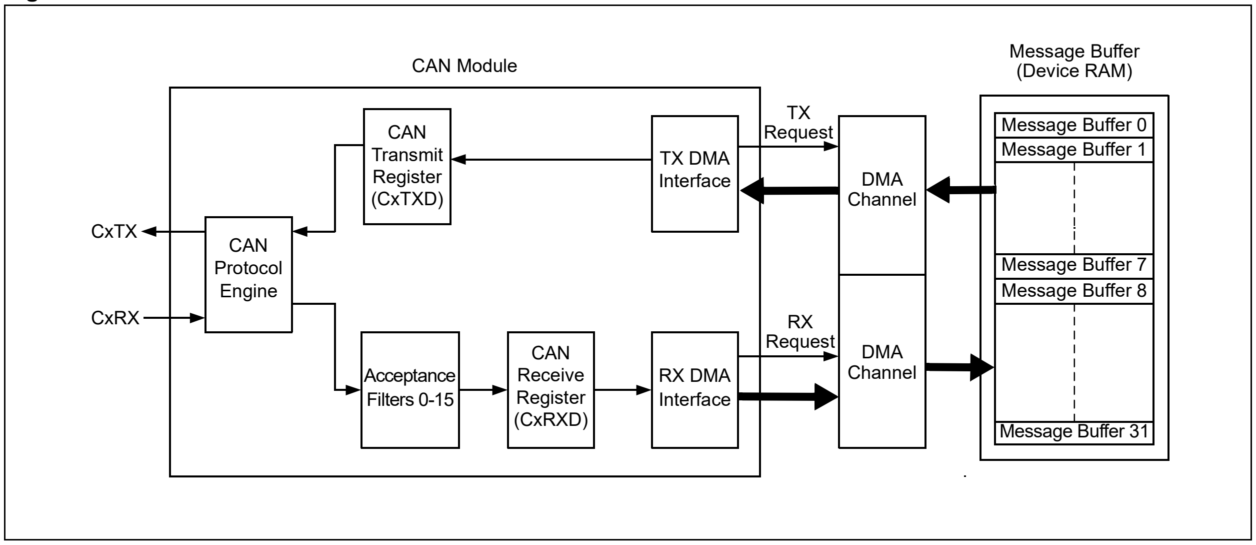

dsPIC33E/PIC24E
CANx
CAN Transceiver
CAN Bus
CAN Transceiver
CAN Transceiver
CAN Transceiver
dsPIC33F/PIC24H with Integrated CAN
dsPIC30F with Integrated CAN
PIC® MCU with Integrated CAN

The CAN module supports the following key features:

- Standards Compliance:
  - Full CAN 2.0B compliance
  - Programmable bit rate up to 1 Mbps
- Message Reception:
  - 32 message buffers – all of which can be used for reception
  - 16 acceptance filters for message filtering
  - Three acceptance filter mask registers for message filtering
  - Automatic response to Remote Frame
  - Up to 32-message deep First-In First-Out (FIFO) buffer
  - DeviceNet™ addressing support
  - Direct Memory Access (DMA) interface for message reception
- Message Transmission:
  - Eight message buffers configurable for message transmission
  - User-defined priority levels for message buffers used for transmission
  - DMA interface for message transmission
- Other Features:
  - Loopback, Listen All Messages and Listen-Only modes for self-test, system diagnostics and bus monitoring
  - Low-power operating modes

Figure 1-2 illustrates the general structure of the CAN module and its interaction with the DMA Controller and device RAM.

Figure 1-2: CAN Interaction with DMA



1.1 CAN Module

The CAN module consists of a protocol engine, message acceptance filters, and separate transmit and receive DMA interfaces. The protocol engine transmits and receives messages to and from the CAN bus (as per the CAN Specification 2.0B protocol). The user-configurable acceptance filters are used by the CAN module to examine the received message and determine if it should be stored in the DMA message buffer or discarded.

For received messages, the receive DMA interface generates a receive data interrupt to initiate a DMA cycle. The receive DMA channel reads data from the CxRXD register and writes them into the message buffer.

For transmit messages, the transmit DMA interface generates a transmit data interrupt to start a DMA cycle. The transmit DMA channel reads from the message buffer and writes to the CxTXD register for message transmission.

1.2 Message Buffers

The CAN module supports up to 32 message buffers for storing data transmitted or received on the CAN bus. These buffers can be located anywhere in device RAM (start address of the buffer may be needed to be aligned on an address boundary). Message Buffers 0-7 can be configured for either transmit or receive operation. Message Buffers 8-31 are receive-only buffers and cannot be used for message transmission.

1.3 DMA Controller

The DMA Controller acts as an interface between the message buffers and CAN to transfer data back and forth without CPU intervention. The DMA Controller supports up to 15 channels for transferring data between the device RAM and the dsPIC33E/PIC24E device peripherals. Two separate DMA channels are required to support the CAN message transmission and the CAN message reception.

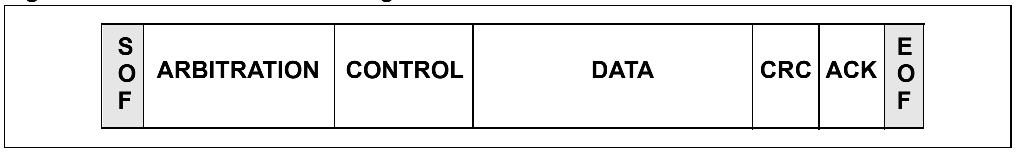
Each DMA channel has a DMA Request register (DMAxREQ), which is used by the user application to assign an interrupt event to trigger a DMA-based message transfer.

2.0 CAN MESSAGE FORMATS

The CAN bus protocol uses asynchronous communication. Information is passed from the transmitters to receivers in data frames, which are composed of byte fields that define the contents of the data frame as illustrated in Figure 2-1.

Each frame begins with a Start-of-Frame (SOF) bit and terminates with an End-of-Frame (EOF) bit field. The SOF is followed by the Arbitration and Control fields, which identify the message type, format, length and priority. This information allows each node on the CAN bus to respond appropriately to the message. The Data field conveys the message content and is of variable length, ranging from 0 bytes to eight bytes. Error protection is provided by the Cyclic Redundancy Check (CRC) and Acknowledgment (ACK) fields.

Figure 2-1: CAN Bus Message Frame



The CAN bus protocol supports four frame types:

- **Data Frame** – carries data from transmitter to the receivers
- **Remote Frame** – transmitted by a node on the bus, to request transmission of a data frame with the same identifier from another node
- **Error Frame** – transmitted by any node when it detects an error
- **Overload Frame** – provides an extra delay between successive data or remote frames

Data frames and remote frames are separated from preceding frames by an Interframe Space. The CAN Specification 2.0B defines two additional data formats:

- **Standard Data Frame** – intended for standard messages that use 11 identifier bits
- **Extended Data Frame** – intended for extended messages that use 29 identifier bits

There are three CAN Specification versions:

- **2.0A** – considers 29-bit identifier as error
- **2.0B Passive** – ignores 29-bit identifier messages
- **2.0B Active** – handles both 11-bit and 29-bit identifiers

The dsPIC33E/PIC24E CAN module is compliant with the CAN Specification 2.0B, while providing enhanced message filtering capabilities.

Note: For detailed information on the CAN bus protocol, refer to the Bosch CAN Specification.

2.1 Standard Data Frame

The standard data frame message begins with a SOF bit followed by a 12-bit Arbitration field, as shown in Figure 2-2. The Arbitration field contains an 11-bit identifier and RTR bit. The identifier defines the type of information contained in the message and is used by each receiving node to determine if the message is of interest to it. The RTR bit distinguishes a data frame from a remote frame. For a standard data frame, the RTR bit is clear.

Following the Arbitration field is a 6-bit Control field, which provides more information about the contents of the message. The first bit in the Control field is an Identifier Extension (IDE) bit, which distinguishes the message as either a standard data frame or extended data frame. A standard data frame is indicated by a Dominant state (logic level ‘0’) during transmission of the IDE bit. The second bit in the Control field is a reserved (RB0) bit, which is in the Dominant state (logic level ‘0’). The last four bits in the Control field represent the Data Length Code (DLC), which specifies the number of data bytes present in the message.

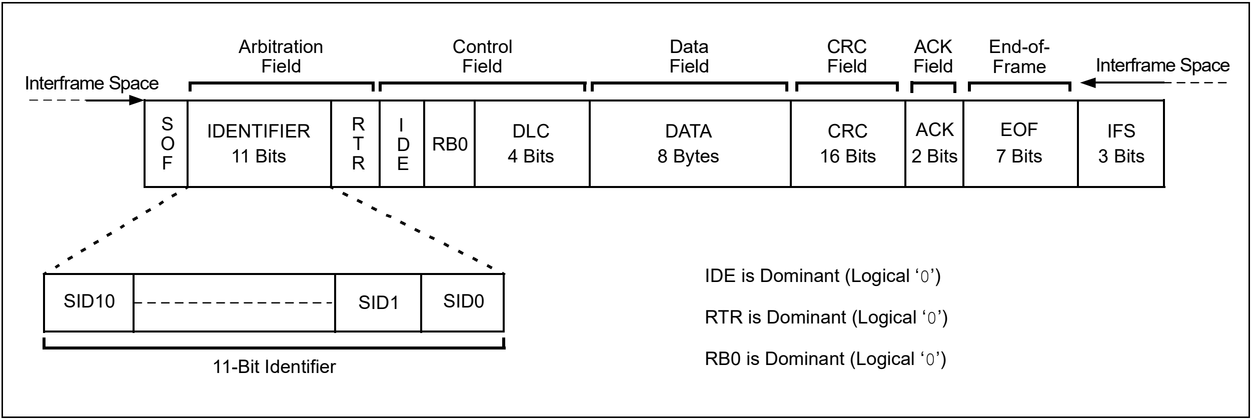
The Data field follows the Control field. This field carries the message data – the actual payload of the data frame. This field is of variable length, ranging from 0 byte to eight bytes. The number of bytes is user-selectable.

The Data field is followed by the CRC field, which is a 16-bit CRC sequence with one delimiter bit.

The Acknowledgment (ACK) field is sent as a recessive bit (logic level ‘1’), and is overwritten as a dominant bit by any receiver that has received the data correctly. The message is Acknowledged by the receiver regardless of the result of the acceptance filter comparison.

The last field is the EOF field, which consists of seven recessive bits that indicate the end of message.

Figure 2-2: Format of the Standard Data Frame



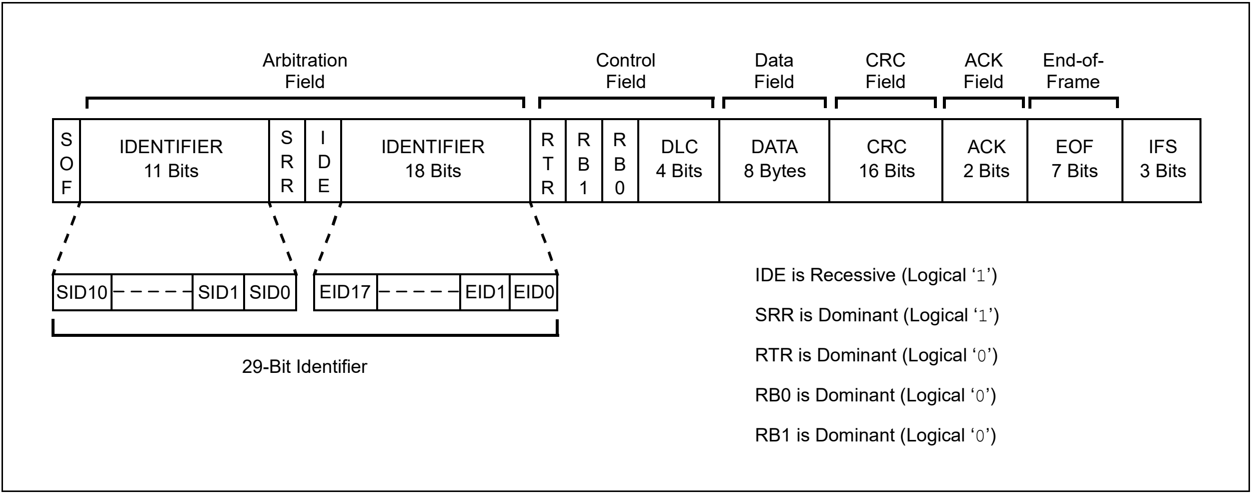
2.2 Extended Data Frame

The extended data frame begins with a SOF bit followed by a 31-bit Arbitration field, as illustrated in Figure 2-3. The Arbitration field in an extended data frame contains a 29-bit identifier in two fields separated by a Substitute Remote Request bit (SRR) and an IDE bit. The SRR bit determines if the message is a remote frame (SRR = 1 for extended data frames). The IDE bit indicates the data frame type. For the extended data frame, IDE = 1.

The extended data frame Control field consists of seven bits. The first bit is the RTR. For the extended data frame, RTR = 0. The next two bits, RB1, and RB0, are reserved bits that are in the Dominant state (logic level ‘0’). The last four bits in the Control field are the DLC, which specifies the number of data bytes present in the message.

The remaining fields in an extended data frame are identical to a standard data frame.

Figure 2-3: Format of the Extended Data Frame



2.3 Remote Frame

A node expecting to receive data from another node can initiate transmission of the respective data by the source node, by sending a remote frame. A remote frame can be in the standard format (Figure 2-4) or the extended format (Figure 2-5).

A remote frame is similar to a data frame, with the following exceptions:

- The RTR bit is recessive (RTR = `1`)
- There is no Data field
- The value of the DLC bits is 0 ≤ DLC ≤ 8

Figure 2-4: Format of the Standard Remote Frame

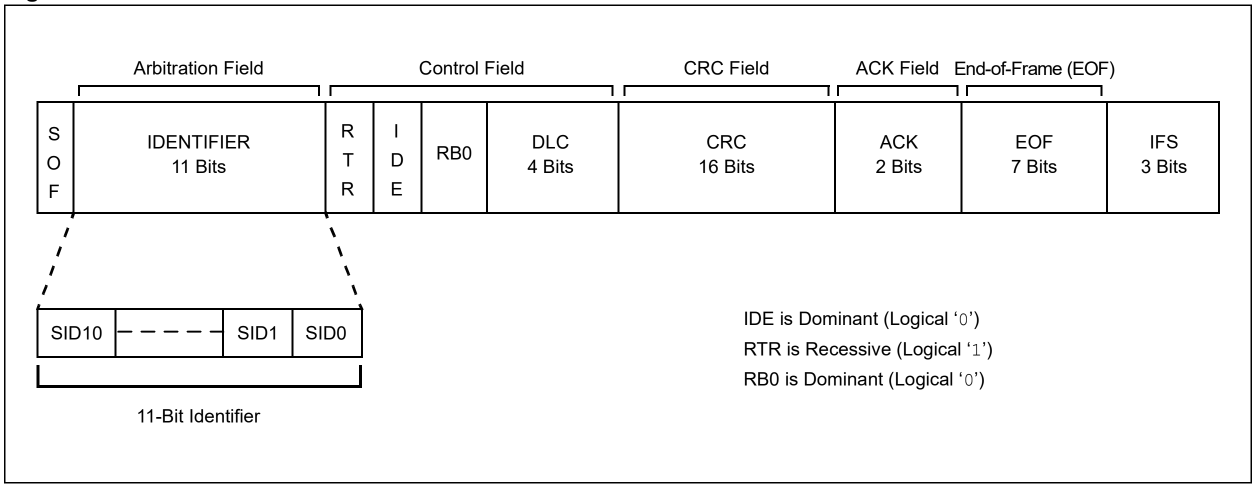
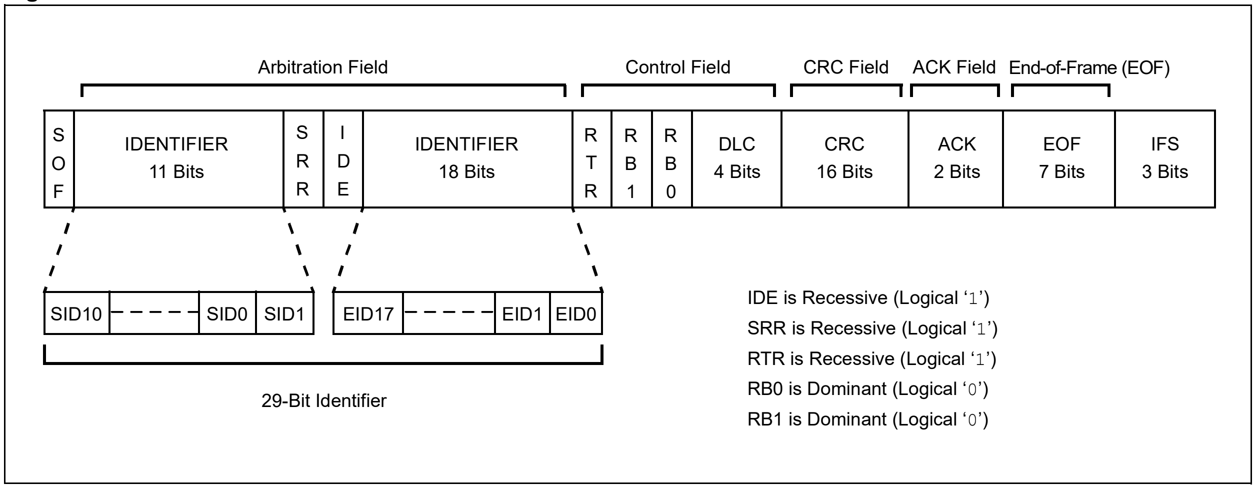


Figure 2-5: Format of the Extended Remote Frame



2.4 Error Frame

An error frame is generated by any node that detects a bus error. An error frame consists of an Error Flag field followed by an Error Delimiter field. The Error Delimiter consists of eight recessive bits and allows the bus nodes to restart communication correctly after an error has occurred. There are two types of Error Flag fields, depending on the error status of the node that detects the error:

- **Error Active Flag** – contains six consecutive dominant bits, which forces all other nodes on the network to generate error echo flags, thereby resulting in a series of 6-12 dominant bits on the bus.
- **Error Passive Flag** – contains six consecutive recessive bits, with the result that unless the bus error is detected by the transmitting node, the transmission of an Error Passive flag will not affect the communications of any other node on the network.

2.5 Overload Frame

An overload frame can be generated by a node, either when a dominant bit is detected during Interframe Space, or when a node is not ready to receive the next message (for example, if it is still reading the previous received message). An overload frame has the same format as an error frame with an active error flag, but can only be generated during Interframe Space. It consists of an Overload Flag field with six dominant bits, followed by an Overload Delimiter field with eight recessive bits. A node can generate a maximum of two sequential overload frames to delay the start of the next message.

2.6 Interframe Space

The Interframe Space separates the successive frames being transmitted on the CAN bus. It consists of at least three recessive bits, referred to as Intermission. The Interframe Space allows nodes time to internally process the previously received message before the start of the next frame. If the transmitting node is in the Error Passive state, an additional eight recessive bits will be inserted in the Interframe Space before any other message is transmitted by the node. The additional eight recessive bits (also referred to as the Suspend Transmit field) allow time for other transmitting nodes to take control of the bus.

3.0 REGISTER MAPS

Table 3-1 through Table 3-3 map the bit functions for the CAN Module registers.

Table 3-1: CAN Register Map When C1CTRL1.WIN = 0 or 1(1)

File Name	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	All Resets
CxCTRL1	—	—	CSIDL	ABAT	CANCKS	REQOP[2:0]			OPMODE[2:0]			—	CANCAP	—	—	WIN	0480
CxCTRL2	—	—	—	—	—	—	—	—	—	—	—	DNCNT[4:0]					0000
CxVEC	—	—	—	FILHIT[4:0]					—	ICODE[6:0]							0040
CxFCTRL	DMABS[2:0]			—	—	—	—	—	—	—	—	FSA[4:0]					0000
CxFIFO	—	—	FBP[5:0]						—	—	FNRB[5:0]						0000
CxINTF	—	—	TXBO	TXBP	RXBP	TXWAR	RXWAR	EWARN	IVRIF	WAKIF	ERRIF	—	FIFOIF	RBOVIF	RBIF	TBIF	0000
CxINTE	—	—	—	—	—	—	—	—	IVRIE	WAKIE	ERRIE	—	FIFOIE	RBOVIE	RBIE	TBIE	0000
CxEC	TERRCNT[7:0]								RERRCNT[7:0]								0000
CxCFG1	—	—	—	—	—	—	—	—	SJW[1:0]		BRP[5:0]						0000
CxCFG2	—	WAKFIL	—	—	—	SEG2PH[2:0]			SEG2PHTS	SAM	SEG1PH[2:0]			PRSEG[2:0]			0000
CxFEN1	FLTEN[15:0]																FFFF
CxFMSKSEL1	F7MSK[1:0]		F6MSK[1:0]		F5MSK[1:0]		F4MSK[1:0]		F3MSK[1:0]		F2MSK[1:0]		F1MSK[1:0]		F0MSK[1:0]		0000
CxFMSKSEL2	F15MSK[1:0]		F14MSK[1:0]		F13MSK[1:0]		F12MSK[1:0]		F11MSK[1:0]		F10MSK[1:0]		F9MSK[1:0]		F8MSK[1:0]		0000

Legend: — = unimplemented, read as '0'. Reset values are shown in hexadecimal.
Note 1: Not all bits are available for all devices. Please refer to the **"Enhanced Controller Area Network (CAN)"** chapter in the specific device data sheet for more details.

Table 3-2: CAN Register Map When C1CTRL1.WIN = 0(1)

File Name	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	All Resets
See definition when WIN = x																	
CxRXFUL1	RXFUL[15:0]																0000
CxRXFUL2	RXFUL[31:16]																0000
CxRXOVF1	RXOVF[15:0]																0000
CxRXOVF2	RXOVF[31:16]																0000
CxTR01CON	TXEN1	TXABT1	TXLARB1	TXERR1	TXREQ1	RTREN1	TX1PRI[1:0]		TXEN0	TXABT0	TXLARB0	TXERR0	TXREQ0	RTREN0	TX0PRI[1:0]		0000
Cx1TR23CON	TXEN3	TXABT3	TXLARB3	TXERR3	TXREQ3	RTREN3	TX3PRI[1:0]		TXEN2	TXABT2	TXLARB2	TXERR2	TXREQ2	RTREN2	TX2PRI[1:0]		0000
CxTR45CON	TXEN5	TXABT5	TXLARB5	TXERR5	TXREQ5	RTREN5	TX5PRI[1:0]		TXEN4	TXABT4	TXLARB4	TXERR4	TXREQ4	RTREN4	TX4PRI[1:0]		0000
CxTR67CON	TXEN7	TXABT7	TXLARB7	TXERR7	TXREQ7	RTREN7	TX7PRI[1:0]		TXEN6	TXABT6	TXLARB6	TXERR6	TXREQ6	RTREN6	TX6PRI[1:0]		xxxx
CxRXD	Received Data Word																xxxx
CxTXD	Transmit Data Word																xxxx

Legend: x = unknown value on Reset; — = unimplemented, read as '0'. Reset values are shown in hexadecimal.
Note 1: Not all bits are available for all devices. Please refer to the **"Enhanced Controller Area Network (CAN)"** chapter in the specific device data sheet for more details.

Table 3-3: CAN Register Map When C1CTRL1.WIN = 1[1]

File Name	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	All Resets
See definition when WIN = x																	
CxBUFPNT1	F3BP[3:0]				F2BP[3:0]				F1BP[3:0]				F0BP[3:0]				0000
CxBUFPNT2	F7BP[3:0]				F6BP[3:0]				F5BP[3:0]				F4BP[3:0]				0000
CxBUFPNT3	F11BP[3:0]				F10BP[3:0]				F9BP[3:0]				F8BP[3:0]				0000
CxBUFPNT4	F15BP[3:0]				F14BP[3:0]				F13BP[3:0]				F12BP[3:0]				0000
CxRXM0SID	SID[10:3]								SID[2:0]			—	MIDE	—	EID[17:16]		xxxx
CxRXM0EID	EID[15:8]								EID[7:0]								xxxx
CxRXM1SID	SID[10:3]								SID[2:0]			—	MIDE	—	EID[17:16]		xxxx
CxRXM1EID	EID[15:8]								EID[7:0]								xxxx
CxRXM2SID	SID[10:3]								SID[2:0]			—	MIDE	—	EID[17:16]		xxxx
CxRXM2EID	EID[15:8]								EID[7:0]								xxxx
CxRXF0SID	SID[10:3]								SID[2:0]			—	EXIDE	—	EID[17:16]		xxxx
CxRXF0EID	EID[15:8]								EID[7:0]								xxxx
CxRXF1SID	SID[10:3]								SID[2:0]			—	EXIDE	—	EID[17:16]		xxxx
CxRXF1EID	EID[15:8]								EID[7:0]								xxxx
CxRXF2SID	SID[10:3]								SID[2:0]			—	EXIDE	—	EID[17:16]		xxxx
CxRXF2EID	EID[15:8]								EID[7:0]								xxxx
CxRXF3SID	SID[10:3]								SID[2:0]			—	EXIDE	—	EID[17:16]		xxxx
CxRXF3EID	EID[15:8]								EID[7:0]								xxxx
CxRXF4SID	SID[10:3]								SID[2:0]			—	EXIDE	—	EID[17:16]		xxxx
CxRXF4EID	EID[15:8]								EID[7:0]								xxxx
CxRXF5SID	SID[10:3]								SID[2:0]			—	EXIDE	—	EID[17:16]		xxxx
CxRXF5EID	EID[15:8]								EID[7:0]								xxxx
CxRXF6SID	SID[10:3]								SID[2:0]			—	EXIDE	—	EID[17:16]		xxxx
CxRXF6EID	EID[15:8]								EID[7:0]								xxxx
CxRXF7SID	SID[10:3]								SID[2:0]			—	EXIDE	—	EID[17:16]		xxxx
CxRXF7EID	EID[15:8]								EID[7:0]								xxxx
CxRXF8SID	SID[10:3]								SID[2:0]			—	EXIDE	—	EID[17:16]		xxxx
CxRXF8EID	EID[15:8]								EID[7:0]								xxxx
CxRXF9SID	SID[10:3]								SID[2:0]			—	EXIDE	—	EID[17:16]		xxxx
CxRXF9EID	EID[15:8]								EID[7:0]								xxxx
CxRXF10SID	SID[10:3]								SID[2:0]			—	EXIDE	—	EID[17:16]		xxxx
CxRXF10EID	EID[15:8]								EID[7:0]								xxxx
CxRXF11SID	SID[10:3]								SID[2:0]			—	EXIDE	—	EID[17:16]		xxxx

Legend: x = unknown value on Reset; — = unimplemented, read as '0'. Reset values are shown in hexadecimal.

Note 1: Not all bits are available for all devices. Please refer to the **"Enhanced Controller Area Network (CAN)"** chapter in the specific device data sheet for more details.

Table 3-3: CAN Register Map When C1CTRL1.WIN = 1[1] (Continued)

File Name	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	All Resets
CxRXF11EID	EID[15:8]								EID[7:0]								xxxx
CxRXF12SID	SID[10:3]								SID[2:0]			—	EXIDE	—	EID[17:16]		xxxx
CxRXF12EID	EID[15:8]								EID[7:0]								xxxx
CxRXF13SID	SID[10:3]								SID[2:0]			—	EXIDE	—	EID[17:16]		xxxx
CxRXF13EID	EID[15:8]								EID[7:0]								xxxx
CxRXF14SID	SID[10:3]								SID[2:0]			—	EXIDE	—	EID[17:16]		xxxx
CxRXF14EID	EID[15:8]								EID[7:0]								xxxx
CxRXF15SID	SID[10:3]								SID[2:0]			—	EXIDE	—	EID[17:16]		xxxx
CxRXF15EID	EID[15:8]								EID[7:0]								xxxx

Legend: x = unknown value on Reset; — = unimplemented, read as ‘0’. Reset values are shown in hexadecimal.

Note 1: Not all bits are available for all devices. Please refer to the **“Enhanced Controller Area Network (CAN)”** chapter in the specific device data sheet for more details.

4.0 CAN REGISTERS

The CAN module has a large number of Special Function Registers (SFRs) that are used to configure the message acceptance filters and the message buffers. To enable effective use of data RAM space, multiple sets of SFRs are mapped onto the same set of memory addresses. The SFR Map Window Select bit (WIN) in CAN Control Register 1 (CxCTRL1[0]) is used to selectively access one of these sets of SFRs.

If CxCTRL1[0] = `1`, the message acceptance filters, masks and Filter Buffer Pointer registers are accessed by the user application.

If CxCTRL1[0] = `0`, the buffer control and status registers, and the transmit and receive data registers are accessed by the user application.

Register 4-1: CxCFG1: CAN Baud Rate Configuration Register 1

U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
—	—	—	—	—	—	—	—
bit 15							bit 8

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
SJW[1:0]		BRP[5:0]					
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-8	**Unimplemented:** Read as ‘`0`’
bit 7-6	**SJW[1:0]:** Synchronization Jump Width bits `11` = Length is 4 x TQ `10` = Length is 3 x TQ `01` = Length is 2 x TQ `00` = Length is 1 x TQ
bit 5-0	**BRP[5:0]:** Baud Rate Prescaler bits `11 1111` = TQ = 2 x 64 x 1/FCAN[1] • • • `00 0010` = TQ = 2 x 3 x 1/FCAN[1] `00 0001` = TQ = 2 x 2 x 1/FCAN[1] `00 0000` = TQ = 2 x 1 x 1/FCAN[1]

Note 1: FCAN is either FP or twice FP depending on the CANCKS bit (CxCTRL[11]) selection.

Register 4-2: CxCFG2: CAN Baud Rate Configuration Register 2

U-0	R/W-x	U-0	U-0	U-0	R/W-x	R/W-x	R/W-x
—	WAKFIL	—	—	—	SEG2PH[2:0]		
bit 15							bit 8

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
SEG2PHTS	SAM	SEG1PH[2:0]			PRSEG[2:0]		
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 15 **Unimplemented:** Read as '0'

bit 14 **WAKFIL:** Select CAN Bus Line Filter for Wake-up bit
1 = Uses CAN bus line filter for wake-up
0 = CAN bus line filter is not used for wake-up

bit 13-11 **Unimplemented:** Read as '0'

bit 10-8 **SEG2PH[2:0]:** Phase Segment 2 bits
111 = Length is 8 x TQ
•
•
•
000 = Length is 1 x TQ

bit 7 **SEG2PHTS:** Phase Segment 2 Time Select bit
1 = Freely programmable
0 = Maximum of SEG1PHx bits or Information Processing Time (IPT), whichever is greater

bit 6 **SAM:** Sample CAN Bus Line bit
1 = Bus line is sampled three times at the sample point
0 = Bus line is sampled once at the sample point

bit 5-3 **SEG1PH[2:0]:** Phase Segment 1 bits
111 = Length is 8 x TQ
•
•
•
000 = Length is 1 x TQ

bit 2-0 **PRSEG[2:0]:** Propagation Time Segment bits
111 = Length is 8 x TQ
•
•
•
000 = Length is 1 x TQ

Register 4-3: CxFEN1: CAN Acceptance Filter Enable Register

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
FLTEN[15:8]							
bit 15							bit 8

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
FLTEN[7:0]							
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-0	**FLTEN[15:0]:** Enable Filter x bits (x = 0-15) 1 = Enables filter x to accept messages 0 = Disables filter x

Register 4-4: CxRXFnSID: CAN Acceptance Filter Standard Identifier Register n (n = 0-15)

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
SID[10:3]							
bit 15							bit 8

R/W-x	R/W-x	R/W-x	U-0	R/W-x	U-0	R/W-x	R/W-x
SID[2:0]			—	EXIDE[1]	—	EID[17:16]	
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-5	**SID[10:0]:** Standard Identifier bits 1 = Message address bit, SIDx, must be ‘1’ to match filter 0 = Message address bit, SIDx, must be ‘0’ to match filter
bit 4	**Unimplemented:** Read as ‘0’
bit 3	**EXIDE:** Extended Identifier Enable bit[1] If MIDE = 1: 1 = Matches only messages with Extended Identifier addresses 0 = Matches only messages with Standard Identifier addresses If MIDE = 0: Ignores EXIDE bit.
bit 2	**Unimplemented:** Read as ‘0’
bit 1-0	**EID[17:16]:** Extended Identifier bits 1 = Message address bit, EIDx, must be ‘1’ to match filter 0 = Message address bit, EIDx, must be ‘0’ to match filter

Note 1: If no mask is applied to a filter, the following occurs: The filter accepts only standard frames. The filter does not accept extended frames even if the EXIDE bit is set to ‘1’.

Register 4-5: CxRXFnEID: CAN Acceptance Filter Extended Identifier Register n (n = 0-15)

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
EID[15:8]							
bit 15							bit 8

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
EID[7:0]							
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-0	**EID[15:0]:** Extended Identifier bits 1 = Message address bit, EIDx, must be ‘1’ to match filter 0 = Message address bit, EIDx, must be ‘0’ to match filter

Register 4-6: CxRXMnSID: CAN Acceptance Filter Mask Standard Identifier Register n (n = 0-2)

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
SID[10:3]							
bit 15							bit 8

R/W-x	R/W-x	R/W-x	U-0	R/W-x	U-0	R/W-x	R/W-x
SID[2:0]			—	MIDE	—	EID[17:16]	
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-5	**SID[10:0]:** Standard Identifier bits 1 = Includes bit, SIDx, in filter comparison 0 = Bit, SIDx, is a “don’t care” in filter comparison
bit 4	**Unimplemented:** Read as ‘0’
bit 3	**MIDE:** Identifier Receive Mode bit 1 = Matches only message types (standard or extended address) that correspond to the EXIDE bit in filter 0 = Matches either standard or extended address message if filters match (that is, if (Filter SID) = (Message SID), or if (Filter SID/EID) = (Message SID/EID))
bit 2	**Unimplemented:** Read as ‘0’
bit 1-0	**EID[17:16]:** Extended Identifier bits 1 = Includes bit, EIDx, in filter comparison 0 = Bit, EIDx, is a “don’t care” in filter comparison

Register 4-7: CxRXMnEID: CAN Acceptance Filter Mask Extended Identifier Register n (n = 0-2)

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
EID[15:8]							
bit 15							bit 8

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
EID[7:0]							
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-0	**EID[15:0]:** Extended Identifier bits 1 = Includes bit, EIDx, in filter comparison 0 = Bit, EIDx, is a “don’t care” in filter comparison

Register 4-8: CxFMSKSEL1: CAN Filter 7-0 Mask Selection Register 1

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
F7MSK[1:0]		F6MSK[1:0]		F5MSK[1:0]		F4MSK[1:0]	
bit 15							bit 8

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
F3MSK[1:0]		F2MSK[1:0]		F1MSK[1:0]		F0MSK[1:0]	
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-14	**F7MSK[1:0]:** Mask Source for Filter 7 bits 11 = Reserved; do not use 10 = Acceptance Mask 2 registers contain mask 01 = Acceptance Mask 1 registers contain mask 00 = Acceptance Mask 0 registers contain mask
bit 13-12	**F6MSK[1:0]:** Mask Source for Filter 6 bits (same values as bits 15-14)
bit 11-10	**F5MSK[1:0]:** Mask Source for Filter 5 bits (same values as bits 15-14)
bit 9-8	**F4MSK[1:0]:** Mask Source for Filter 4 bits (same values as bits 15-14)
bit 7-6	**F3MSK[1:0]:** Mask Source for Filter 3 bits (same values as bits 15-14)
bit 5-4	**F2MSK[1:0]:** Mask Source for Filter 2 bits (same values as bits 15-14)
bit 3-2	**F1MSK[1:0]:** Mask Source for Filter 1 bits (same values as bits 15-14)
bit 1-0	**F0MSK[1:0]:** Mask Source for Filter 0 bits (same values as bits 15-14)

Register 4-9: CxFMSKSEL2: CAN Filter 15-8 Mask Selection Register 2

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
F15MSK[1:0]		F14MSK[1:0]		F13MSK[1:0]		F12MSK[1:0]	
bit 15							bit 8

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
F11MSK[1:0]		F10MSK[1:0]		F9MSK[1:0]		F8MSK[1:0]	
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-14	**F15MSK[1:0]:** Mask Source for Filter 15 bits `11` = Reserved; do not use `10` = Acceptance Mask 2 registers contain mask `01` = Acceptance Mask 1 registers contain mask `00` = Acceptance Mask 0 registers contain mask
bit 13-12	**F14MSK[1:0]:** Mask Source for Filter 14 bits (same values as bits 15-14)
bit 11-10	**F13MSK[1:0]:** Mask Source for Filter 13 bits (same values as bits 15-14)
bit 9-8	**F12MSK[1:0]:** Mask Source for Filter 12 bits (same values as bits 15-14)
bit 7-6	**F11MSK[1:0]:** Mask Source for Filter 11 bits (same values as bits 15-14)
bit 5-4	**F10MSK[1:0]:** Mask Source for Filter 10 bits (same values as bits 15-14)
bit 3-2	**F9MSK[1:0]:** Mask Source for Filter 9 bits (same values as bits 15-14)
bit 1-0	**F8MSK[1:0]:** Mask Source for Filter 8 bits (same values as bits 15-14)

Register 4-10: CxBUFPNT1: CAN Filter 0-3 Buffer Pointer Register 1

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
F3BP[3:0]				F2BP[3:0]			
bit 15							bit 8

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
F1BP[3:0]				F0BP[3:0]			
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-12 **F3BP[3:0]:** RX Buffer Mask for Filter 3 bits
`1111` = Filter hits received in RX FIFO buffer
`1110` = Filter hits received in RX Buffer 14
•
•
•
`0001` = Filter hits received in RX Buffer 1
`0000` = Filter hits received in RX Buffer 0

bit 11-8 **F2BP[3:0]:** RX Buffer mask for Filter 2 bits (same values as bits 15-12)

bit 7-4 **F1BP[3:0]:** RX Buffer mask for Filter 1 bits (same values as bits 15-12)

bit 3-0 **F0BP[3:0]:** RX Buffer mask for Filter 0 bits (same values as bits 15-12)

Register 4-11: CxBUFPNT2: CAN Filter 4-7 Buffer Pointer Register 2

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
F7BP[3:0]				F6BP[3:0]			
bit 15							bit 8

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
F5BP[3:0]				F4BP[3:0]			
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-12 **F7BP[3:0]:** RX Buffer Mask for Filter 7 bits
`1111` = Filter hits received in RX FIFO buffer
`1110` = Filter hits received in RX Buffer 14
•
•
•
`0001` = Filter hits received in RX Buffer 1
`0000` = Filter hits received in RX Buffer 0

bit 11-8 **F6BP[3:0]:** RX Buffer mask for Filter 6 bits (same values as bits 15-12)

bit 7-4 **F5BP[3:0]:** RX Buffer mask for Filter 5 bits (same values as bits 15-12)

bit 3-0 **F4BP[3:0]:** RX Buffer mask for Filter 4 bits (same values as bits 15-12)

Register 4-12: CxBUFPNT3: CAN Filter 8-11 Buffer Pointer Register 3

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
F11BP[3:0]				F10BP[3:0]			
bit 15							bit 8

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
F9BP[3:0]				F8BP[3:0]			
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-12 **F11BP[3:0]:** RX Buffer Mask for Filter 11
`1111` = Filter hits received in RX FIFO buffer
`1110` = Filter hits received in RX Buffer 14
•
•
•
`0001` = Filter hits received in RX Buffer 1
`0000` = Filter hits received in RX Buffer 0

bit 11-8 **F10BP[3:0]:** RX Buffer mask for Filter 10 (same values as bit 15-12)

bit 7-4 **F9BP[3:0]:** RX Buffer mask for Filter 9 (same values as bit 15-12)

bit 3-0 **F8BP[3:0]:** RX Buffer mask for Filter 8 (same values as bit 15-12)

Register 4-13: CxBUFPNT4: CAN Filter 12-15 Buffer Pointer Register 4

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
F15BP[3:0]				F14BP[3:0]			
bit 15							bit 8

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
F13BP[3:0]				F12BP[3:0]			
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-12 **F15BP[3:0]:** RX Buffer Mask for Filter 15 bits
`1111` = Filter hits received in RX FIFO buffer
`1110` = Filter hits received in RX Buffer 14
•
•
•
`0001` = Filter hits received in RX Buffer 1
`0000` = Filter hits received in RX Buffer 0

bit 11-8 **F14BP[3:0]:** RX Buffer mask for Filter 14 bits (same values as bits 15-12)

bit 7-4 **F13BP[3:0]:** RX Buffer mask for Filter 13 bits (same values as bits 15-12)

bit 3-0 **F12BP[3:0]:** RX Buffer mask for Filter 12 bits (same values as bits 15-12)

Register 4-14: CxRXFUL1: CAN Receive Buffer Full Register 1

R/C-0	R/C-0	R/C-0	R/C-0	R/C-0	R/C-0	R/C-0	R/C-0
RXFUL[15:8]							
bit 15							bit 8

R/C-0	R/C-0	R/C-0	R/C-0	R/C-0	R/C-0	R/C-0	R/C-0
RXFUL[7:0]							
bit 7							bit 0

Legend:			
R = Readable bit	C = Writable bit, but only ‘0’ can be written to clear the bit		
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-0 **RXFUL[15:0]:** Receive Buffer x Full bits
1 = Buffer is full (set by module)
0 = Buffer is empty (cleared by software)

Register 4-15: CxRXFUL2: CAN Receive Buffer Full Register 2

R/C-0	R/C-0	R/C-0	R/C-0	R/C-0	R/C-0	R/C-0	R/C-0
RXFUL[31:24]							
bit 15							bit 8

R/C-0	R/C-0	R/C-0	R/C-0	R/C-0	R/C-0	R/C-0	R/C-0
RXFUL[23:16]							
bit 7							bit 0

Legend:			
R = Readable bit	C = Writable bit, but only ‘0’ can be written to clear the bit		
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-0 **RXFUL[31:16]:** Receive Buffer x Full bits
1 = Buffer is full (set by module)
0 = Buffer is empty (cleared by user software)

Register 4-16: CxRXOVF1: CAN Receive Buffer Overflow Register 1

R/C-0	R/C-0	R/C-0	R/C-0	R/C-0	R/C-0	R/C-0	R/C-0
RXOVF[15:8]							
bit 15							bit 8

R/C-0	R/C-0	R/C-0	R/C-0	R/C-0	R/C-0	R/C-0	R/C-0
RXOVF[7:0]							
bit 7							bit 0

Legend:			
R = Readable bit	C = Writable bit, but only ‘0’ can be written to clear the bit		
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-0 **RXOVF[15:0]:** Receive Buffer n Overflow bits
1 = Module attempted to write to a full buffer (set by module)
0 = No overflow condition (cleared by user software)

Register 4-17: CxRXOVF2: CAN Receive Buffer Overflow Register 2

R/C-0	R/C-0	R/C-0	R/C-0	R/C-0	R/C-0	R/C-0	R/C-0
RXOVF[31:24]							
bit 15							bit 8

R/C-0	R/C-0	R/C-0	R/C-0	R/C-0	R/C-0	R/C-0	R/C-0
RXOVF[23:16]							
bit 7							bit 0

Legend:			
R = Readable bit	C = Writable bit, but only ‘0’ can be written to clear the bit		
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-0 **RXOVF[31:16]:** Receive Buffer n Overflow bits
1 = Module attempted to write to a full buffer (set by module)
0 = No overflow condition (cleared by user software)

Register 4-18: CxFCTRL: CAN FIFO Control Register

R/W-0	R/W-0	R/W-0	U-0	U-0	U-0	U-0	U-0
DMABS[2:0]			—	—	—	—	—
bit 15							bit 8

U-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	—	FSA[4:0]				
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-13 **DMABS[2:0]:** Message Buffer Size bits

`111` = Reserved; do not use
`110` = 32 buffers in device RAM
`101` = 24 buffers in device RAM
`100` = 16 buffers in device RAM
`011` = 12 buffers in device RAM
`010` = 8 buffers in device RAM
`001` = 6 buffers in device RAM
`000` = 4 buffers in device RAM

bit 12-5 **Unimplemented:** Read as ‘`0`’

bit 4-0 **FSA[4:0]:** FIFO Start Area bits

`11111` = Reads buffer RB31
`11110` = Reads buffer RB30
•
•
•
`00010` = TX/RX buffer, TRB2
`00001` = TX/RX buffer, TRB1
`00000` = TX/RX buffer, TRB0

Register 4-19: CxFIFO: CAN FIFO Status Register

U-0	U-0	R-0	R-0	R-0	R-0	R-0	R-0
—	—	FBP[5:0]					
bit 15							bit 8

U-0	U-0	R-0	R-0	R-0	R-0	R-0	R-0
—	—	FNRB[5:0]					
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-14 **Unimplemented:** Read as ‘0’

bit 13-8 **FBP[5:0]:** FIFO Buffer Pointer bits

`011111` = RB31 buffer
`011110` = RB30 buffer
•
•
•
`000001` = TRB1 buffer
`000000` = TRB0 buffer

bit 7-6 **Unimplemented:** Read as ‘0’

bit 5-0 **FNRB[5:0]:** FIFO Next Read Buffer Pointer bits

`011111` = RB31 buffer
`011110` = RB30 buffer
•
•
•
`000001` = TRB1 buffer
`000000` = TRB0 buffer

Register 4-20: CxINTF: CAN Interrupt Flag Register

U-0	U-0	R-0	R-0	R-0	R-0	R-0	R-0
—	—	TXBO	TXBP	RXBP	TXWAR	RXWAR	EWARN
bit 15							bit 8

R/C-0	R/C-0	R/C-0	U-0	R/C-0	R/C-0	R/C-0	R/C-0
IVRIF	WAKIF	ERRIF	—	FIFOIF	RBOVIF	RBIF	TBIF
bit 7							bit 0

Legend:	C = Writable bit, but only ‘0’ can be written to clear the bit		
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-14 **Unimplemented:** Read as ‘0’

bit 13 **TXBO:** Transmitter in Error State Bus Off bit
1 = Transmitter is in Bus Off state
0 = Transmitter is not in Bus Off state

bit 12 **TXBP:** Transmitter in Error State Bus Passive bit
1 = Transmitter is in Bus Passive state
0 = Transmitter is not in Bus Passive state

bit 11 **RXBP:** Receiver in Error State Bus Passive bit
1 = Receiver is in Bus Passive state
0 = Receiver is not in Bus Passive state

bit 10 **TXWAR:** Transmitter in Error State Warning bit
1 = Transmitter is in Error Warning state
0 = Transmitter is not in Error Warning state

bit 9 **RXWAR:** Receiver in Error State Warning bit
1 = Receiver is in Error Warning state
0 = Receiver is not in Error Warning state

bit 8 **EWARN:** Transmitter or Receiver in Error State Warning bit
1 = Transmitter or receiver is in Error State Warning state
0 = Transmitter or receiver is not in Error State Warning state

bit 7 **IVRIF:** Invalid Message Interrupt Flag bit
1 = Interrupt request has occurred
0 = Interrupt request has not occurred

bit 6 **WAKIF:** Bus Wake-up Activity Interrupt Flag bit
1 = Interrupt request has occurred
0 = Interrupt request has not occurred

bit 5 **ERRIF:** Error Interrupt Flag bit (multiple sources in CxINTF[13:8] bits)
1 = Interrupt request has occurred
0 = Interrupt request has not occurred

bit 4 **Unimplemented:** Read as ‘0’

bit 3 **FIFOIF:** FIFO Almost Full Interrupt Flag bit
1 = Interrupt request has occurred
0 = Interrupt request has not occurred

bit 2 **RBOVIF:** RX Buffer Overflow Interrupt Flag bit
1 = Interrupt request has occurred
0 = Interrupt request has not occurred

bit 1 **RBIF:** RX Buffer Interrupt Flag bit
1 = Interrupt request has occurred
0 = Interrupt request has not occurred

bit 0 **TBIF:** TX Buffer Interrupt Flag bit
1 = Interrupt request has occurred
0 = Interrupt request has not occurred

Register 4-21: CxINTE: CAN Interrupt Enable Register(1)

U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
—	—	—	—	—	—	—	—
bit 15							bit 8

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
IVRIE	WAKIE	ERRIE	—	FIFOIE	RBOVIE	RBIE	TBIE
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-8 **Unimplemented:** Read as ‘0’

bit 7 **IVRIE:** Invalid Message Interrupt Enable bit
1 = Interrupt request enabled
0 = Interrupt request not enabled

bit 6 **WAKIE:** Bus Wake-up Activity Interrupt Flag bit
1 = Interrupt request enabled
0 = Interrupt request not enabled

bit 5 **ERRIE:** Error Interrupt Enable bit
1 = Interrupt request enabled
0 = Interrupt request not enabled

bit 4 **Unimplemented:** Read as ‘0’

bit 3 **FIFOIE:** FIFO Almost Full Interrupt Enable bit
1 = Interrupt request enabled
0 = Interrupt request not enabled

bit 2 **RBOVIE:** RX Buffer Overflow Interrupt Enable bit
1 = Interrupt request enabled
0 = Interrupt request not enabled

bit 1 **RBIE:** RX Buffer Interrupt Enable bit
1 = Interrupt request enabled
0 = Interrupt request not enabled

bit 0 **TBIE:** TX Buffer Interrupt Enable bit
1 = Interrupt request enabled
0 = Interrupt request not enabled

Note 1: Setting a bit in the CxINTE register only enables the corresponding interrupt source. To generate a Cx interrupt, the CxIE bit must be set in the Interrupt module.

Register 4-22: CxVEC: CAN Interrupt Code Register

U-0	U-0	U-0	R-0	R-0	R-0	R-0	R-0
—	—	—	FILHIT[4:0]				
bit 15							bit 8

U-0	R-1	R-0	R-0	R-0	R-0	R-0	R-0
—	ICODE[6:0](1,2)						
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-13 **Unimplemented:** Read as ‘`0`’

bit 12-8 **FILHIT[4:0]:** Filter Hit Number bits

`10000-11111` = Reserved; do not use
`01111` = Filter 15
•
•
•
`00001` = Filter 1
`00000` = Filter 0

bit 7 **Unimplemented:** Read as ‘`0`’

bit 6-0 **ICODE[6:0]:** Interrupt Flag Code bits(1,2)

`1000101-1111111` = Reserved; do not use
`1000100` = FIFO almost full interrupt
`1000011` = Receiver overflow interrupt
`1000010` = Wake-up interrupt
`1000001` = Error interrupt
`1000000` = No interrupt

`0100000-0111111` = Reserved; do not use
`0011111` = RB31 buffer interrupt
`0011110` = RB30 buffer interrupt
•
•
•
`0001001` = RB9 buffer interrupt
`0001000` = RB8 buffer interrupt
`0000111` = TRB7 buffer interrupt
`0000110` = TRB6 buffer interrupt
`0000101` = TRB5 buffer interrupt
`0000100` = TRB4 buffer interrupt
`0000011` = TRB3 buffer interrupt
`0000010` = TRB2 buffer interrupt
`0000001` = TRB1 buffer interrupt
`0000000` = TRB0 Buffer interrupt

Note 1: The ICODE[6:0] bits are cleared when the corresponding interrupts flag bits in the CxINTF register are cleared.

2: The ICODE[6:0] bits only reflect the status of enabled interrupt sources. The corresponding bits in the CxINTE register must be set.

Register 4-23: CxCTRL1: CAN Control Register 1

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-1	R/W-0	R/W-0
—	—	CSIDL	ABAT	CANCKS	REQOP[2:0]		
bit 15							bit 8

R-1	R-0	R-0	U-0	R/W-0	U-0	U-0	R/W-0
OPMODE[2:0]			—	CANCAP	—	—	WIN
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-14	**Unimplemented:** Read as ‘0’
bit 13	**CSIDL:** Stop in Idle Mode bit 1 = Discontinues module operation when device enters Idle mode 0 = Continues module operation in Idle mode
bit 12	**ABAT:** Abort All Pending Transmissions bit 1 = Signals all transmit buffers to abort transmission 0 = Module will clear this bit when all transmissions are aborted
bit 11	**CANCKS:** CAN Module Clock (FCAN) Source Select bit 1 = FCAN is equal to 2 * FP 0 = FCAN is equal to FP
bit 10-8	**REQOP[2:0]:** Request Operation Mode bits 111 = Sets Listen All Messages mode 110 = Reserved; do not use 101 = Reserved; do not use 100 = Sets Configuration mode 011 = Sets Listen-Only mode 010 = Sets Loopback mode 001 = Sets Disable mode 000 = Sets Normal Operation mode
bit 7-5	**OPMODE[2:0]:** Operation Mode bits 111 = Module is in Listen All Messages mode 110 = Reserved; do not use 101 = Reserved; do not use 100 = Module is in Configuration mode 011 = Module is in Listen-Only mode 010 = Module is in Loopback mode 001 = Module is in Disable mode 000 = Module is in Normal Operation mode
bit 4	**Unimplemented:** Read as ‘0’
bit 3	**CANCAP:** CAN Message Receive Timer Capture Event Enable bit 1 = Enables input capture based on CAN message receive 0 = Disables CAN capture
bit 2-1	**Unimplemented:** Read as ‘0’
bit 0	**WIN:** SFR Map Window Select bit 1 = Uses filter window 0 = Uses buffer window

Register 4-24: CxCTRL2: CAN Control Register 2

U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
—	—	—	—	—	—	—	—
bit 15							bit 8

U-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	—	DNCNT[4:0]				
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-5	**Unimplemented:** Read as ‘0’
bit 4-0	**DNCNT[4:0]:** DeviceNet™ Filter Bit Number bits 10011-11111 = Invalid selection 10010 = Compare bits [7:0] of byte 0 and bits [7:0] of byte 1 and bits [7:6] of byte 2 with EID[17:0] 10001 = Compare bits [7:0] of byte 0 and bits [7:0] of byte 1 and bit 7 of byte 2 with EID[17:1] 10000 = Compare bits [7:0] of data byte 0 and bits [7:0] of data byte 1 with EID[17:2] 01111 = Compare bits [7:0] of data byte 0 and bits [7:1] of data byte 1 with EID[17:3] 01110 = Compare bits [7:0] of data byte 0 and bits [7:2] of data byte 1 with EID[17:4] 01101 = Compare bits [7:0] of data byte 0 and bits [7:3] of data byte 1 with EID[17:5] 01100 = Compare bits [7:0] of data byte 0 and bits [7:4] of data byte 1 with EID[17:6] 01011 = Compare bits [7:0] of data byte 0 and bits [7:5] of data byte 1 with EID[17:7] 01010 = Compare bits [7:0] of data byte 0 and bits [7:6] of data byte 1 with EID[17:8] 01001 = Compare bits [7:0] of data byte 0 and bit [7] of data byte 1 with EID[17:9] 01000 = Compare bits [7:0] of data byte 0 with EID[17:10] 00111 = Compare bits [7:1] of data byte 0 with EID[17:11] 00110 = Compare bits [7:2] of data byte 0 with EID[17:12] 00101 = Compare bits [7:3] of data byte 0 with EID[17:13] 00100 = Compare bits [7:4] of data byte 0 with EID[17:14] 00011 = Compare bits [7:5] of data byte 0 with EID[17:15] 00010 = Compare bits [7:6] of data byte 0 with EID[17:16] 00001 = Compare bit 7 of data byte 0 with EID[17] 00000 = Does not compare data bytes

Register 4-25: CxTRmnCON: CAN TX/RX Buffer m Control Register (m = 0,2,4,6; n = 1,3,5,7)

R/W-0	R-0	R-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0
TXENn	TXABTn	TXLARBn	TXERRn	TXREQn	RTRENn	TXnPRI[1:0]	
bit 15							bit 8

R/W-0	R-0	R-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0
TXENm	TXABTm[1]	TXLARBm[1]	TXERRm[1]	TXREQm	RTRENm	TXmPRI[1:0]	
bit 7							bit 0

Legend:

R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-8 See definition for bits 7-0, Controls Buffer n

bit 7 **TXENm:** TX/RX Buffer Selection bit
1 = Buffer TRBn is a transmit buffer
0 = Buffer TRBn is a receive buffer

bit 6 **TXABTm:** Message Aborted bit[1]
1 = Message was aborted
0 = Message completed transmission successfully

bit 5 **TXLARBm:** Message Lost Arbitration bit[1]
1 = Message lost arbitration while being sent
0 = Message did not lose arbitration while being sent

bit 4 **TXERRm:** Error Detected During Transmission bit[1]
1 = A bus error occurred while the message was being sent
0 = A bus error did not occur while the message was being sent

bit 3 **TXREQm:** Message Send Request bit
1 = Requests that a message be sent; once the message is successfully sent, the bit is automatically cleared
0 = Setting this bit to ‘0’ while a message is being sent aborts the message transmission

bit 2 **RTRENm:** Auto-Remote Transmit Enable bit
1 = When a remote frame is received, TXREQ bit will automatically set
0 = When a remote frame is received, TXREQ bit will be unaffected

bit 1-0 **TXmPRI[1:0]:** Message Transmission Priority bits
11 = Highest message priority
10 = High intermediate message priority
01 = Low intermediate message priority
00 = Lowest message priority

Note 1: This bit is cleared when the TXREQ bit is set.

Register 4-26: CxEC: CAN Transmit/Receive Error Count Register

R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0
TERRCNT[7:0]							
bit 15							bit 8

R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0
RERRCNT[7:0]							
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-8	**TERRCNT[7:0]:** Transmit Error Count bits
bit 7-0	**RERRCNT[7:0]:** Receive Error Count bits

Register 4-27: CxRXD: CAN Receive Data Register

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
DATA[15:8]							
bit 15							bit 8

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
DATA[7:0]							
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-0	**DATA[15:0]:** Receive Data bits

Register 4-28: CxTXD: CAN Transmit Data Register

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
DATA[15:8]							
bit 15							bit 8

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
DATA[7:0]							
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-0	**DATA[15:0]:** Transmit Data bits

5.0 CAN MESSAGE BUFFERS

CAN message buffers are located in device RAM. They are not CAN SFRs. The user application must directly write into the device RAM area that is configured for CAN message buffers. The location and size of the buffer area is defined by the user application.

This section provides information on how the message buffer words are organized for transmission and reception. (See also **Section 2.0 "CAN Message Formats"** for message buffer layout details and **Section 9.0 "DMA Controller Configuration"** for details on how to configure CAN message buffers in device RAM.)

Buffer 5-1: CAN Message Buffer Word 0

U-x	U-x	U-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
—	—	—	SID[10:6]				
bit 15							bit 8

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
SID[5:0]						SRR	IDE
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 15-13 **Unimplemented:** Read as '0'

bit 12-2 **SID[10:0]:** Standard Identifier bits

bit 1 **SRR:** Substitute Remote Request bit

When IDE = 0:

1 = In case of transmission, this will cause a remote frame to be transmitted; in case of a received message, indicates a remote frame was received

0 = Normal message

When IDE = 1:

The SRR bit must be set to '1'

bit 0 **IDE:** Extended Identifier bit

1 = Message will transmit Extended Identifier or received message has an Extended Identifier

0 = Message will transmit Standard Identifier or received message has a Standard Identifier

Buffer 5-2: CAN Message Buffer Word 1

U-x	U-x	U-x	U-x	R/W-x	R/W-x	R/W-x	R/W-x
—	—	—	—	EID[17:14]			
bit 15							bit 8

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
EID[13:6]							
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 15-12 **Unimplemented:** Read as '0'

bit 11-0 **EID[17:6]:** Extended Identifier bits

Buffer 5-3: CAN Message Buffer Word 2

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
EID[5:0]						RTR	RB1
bit 15							bit 8

U-x	U-x	U-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
—	—	—	RB0	DLC[3:0]			
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-10	**EID[5:0]:** Extended Identifier bits
bit 9	**RTR:** Remote Transmission Request bit When IDE = 1: 1 = In case of transmission, this will cause a remote frame to be transmitted; in case of received message, indicates a remote frame was received 0 = Normal message When IDE = 0: The RTR bit is ignored.
bit 8	**RB1:** Reserved Bit 1 User application must set this bit to ‘0’ per CAN Specification.
bit 7-5	**Unimplemented:** Read as ‘0’
bit 4	**RB0:** Reserved Bit 0 User application must set this bit to ‘0’ per CAN Specification.
bit 3-0	**DLC[3:0]:** Data Length Code bits

Buffer 5-4: CAN Message Buffer Word 3

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
Byte 1							
bit 15							bit 8

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
Byte 0							
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-8	CAN Message Byte 1
bit 7-0	CAN Message Byte 0

Buffer 5-5: CAN Message Buffer Word 4

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
Byte 3							
bit 15							bit 8

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
Byte 2							
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-8	CAN Message Byte 3
bit 7-0	CAN Message Byte 2

Buffer 5-6: CAN Message Buffer Word 5

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
Byte 5							
bit 15							bit 8

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
Byte 4							
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-8	CAN Message Byte 5
bit 7-0	CAN Message Byte 4

Buffer 5-7: CAN Message Buffer Word 6

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
Byte 7							
bit 15							bit 8

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
Byte 6							
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-8	CAN Message Byte 7
bit 7-0	CAN Message Byte 6

Buffer 5-8: CAN Message Buffer Word 7

U-0	U-0	U-0	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
—	—	—	FILHIT[4:0]				
bit 15							bit 8

U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
—	—	—	—	—	—	—	—
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as ‘0’	
-n = Value at POR	‘1’ = Bit is set	‘0’ = Bit is cleared	x = Bit is unknown

bit 15-13	**Unimplemented:** Read as ‘0’
bit 12-8	**FILHIT[4:0]:** Filter Hit Code bits Encodes the number of the filter that resulted in writing this buffer (only written by module for receive buffers, unused for transmit buffers).
bit 7-0	**Unimplemented:** Read as ‘0’

6.0 CAN OPERATING MODES

The CAN module can operate in one of the several modes selected by the user application. These modes include:

- Configuration mode
- Normal Operation mode
- Listen-Only mode
- Listen All Messages mode
- Loopback mode
- Disable mode

The operating modes are requested by the user application that is writing to the Request Operation Mode bits (REQOP[2:0]) in the CAN Control Register 1 (CxCTRL1[10:8]). The CAN module Acknowledges entry into the requested mode by the OPMODE[2:0] bits (CxCTRL1[7:5]). Mode transition is performed in synchronization with the CAN network. That is, the CAN module waits until it detects a bus Idle sequence (11 recessive bits) before it changes mode.

6.1 Configuration Mode

After a hardware Reset, the CAN module is in Configuration mode (OPMODE[2:0] = `100`). The error counters are cleared and all registers contain the Reset values. In order to modify the CAN Baud Rate Configuration registers (CxCFG1 and CxCFG2), the CAN module must be in the Configuration mode.

6.2 Normal Operation Mode

In the Normal Operation mode, the CAN module can transmit and receive the CAN messages. Normal Operation mode is requested after initialization by programming the REQOP[2:0] bits (CxCTRL1[10:8]) to ‘`000`’. When OPMODE[2:0] = `000`, the module proceeds with normal operation.

6.3 Listen-Only Mode

The Listen-Only mode is used mainly for bus monitoring without participating in the transmission process. The node in Listen-Only mode does not generate an Acknowledge or error frames – one of the other nodes must do it. The Listen-Only mode can be used for detecting the baud rate on the CAN bus.

6.4 Listen All Messages Mode

The Listen All Messages mode is used for system debugging. Basically, all messages are received regardless of their identifier, even when there is an error. If the Listen All Messages mode is activated, the transmission and reception operate the same as Normal Operation mode, except that if a message is received with an error, it is still transferred to the message buffer.

6.5 Loopback Mode

The Loopback mode is used for self-test to allow the CAN module to receive its own message. In this mode, the CAN transmit path is connected to the receive path internally. A “dummy” Acknowledgment is provided, thereby eliminating the need for another node to provide the Acknowledge bit.

6.6 Disable Mode

The Disable mode is used to ensure a safe shutdown before putting the device in Sleep or Idle mode. That is, the CAN module waits until it detects a bus Idle sequence (11 recessive bits) before it changes the mode. When the module is in Disable mode, it stops its own clocks, having no effect on the CPU or other modules. The module wakes up when the bus activity occurs or when the CPU sets the OPMODE[2:0] bits to ‘`000`’.

The CxTX pin stays in the Recessive state while the module is in Disable mode.

7.0 TRANSMITTING CAN MESSAGES

A node originating a message is a transmitter of that message. The node remains a transmitter until the bus becomes Idle or the unit loses arbitration. Figure 7-1 illustrates a typical CAN transmission process.

Message Buffers 0-7 (located in device RAM) are configured to transmit or receive CAN messages using the TX/RX Buffer Selection bit (TXENn) in the corresponding CAN TX/RX Buffer m Control register (CxTRmnCON[7]). If the TXENn bit is set, the message buffer is configured for transmission. For the layout of standard and extended frames in the message buffer, and the states of the IDE, SRR, RTR, RB0 and RB1 bits for Standard Data, Extended Data, Standard Remote, or Extended Remote frames as per the CAN Specification; refer to **Section 2.0 "CAN Message Formats"**.

7.1 Message Transmission Flow

To transmit a message over the CAN bus, the user application must perform these tasks:

1. Configure a message buffer for transmission and assign a priority to the buffer.
2. Write the CAN message in the message buffer located in device RAM.
3. Set the transmit request bit for the buffer to initiate message transmission.

The message transmission is initiated by setting the Message Send Request bit (TXREQm) in the CAN TX/RX Buffer m Control register (CxTRmnCON[3]). The TXREQm bit is cleared automatically after the message is transmitted. Before the SOF is sent, all the buffers ready for transmission are examined to determine which buffer has the highest priority. The transmit buffer with the highest priority is sent first.

Note: Setting the TXREQm bit when the TXENn bit is '0' will result in unpredictable module behavior.

Each of the transmit message buffers can be assigned to any of the four user application-defined priority levels using the TXnPRI[1:0] bits (CxTRmnCON[1:0]).

TXnPRI[1:0] message transmission priority selections are:

- `11` = The transmit message has the highest priority
- `10` = The transmit message has intermediate high priority
- `01` = The transmit message has intermediate low priority
- `00` = The transmit message has the lowest priority

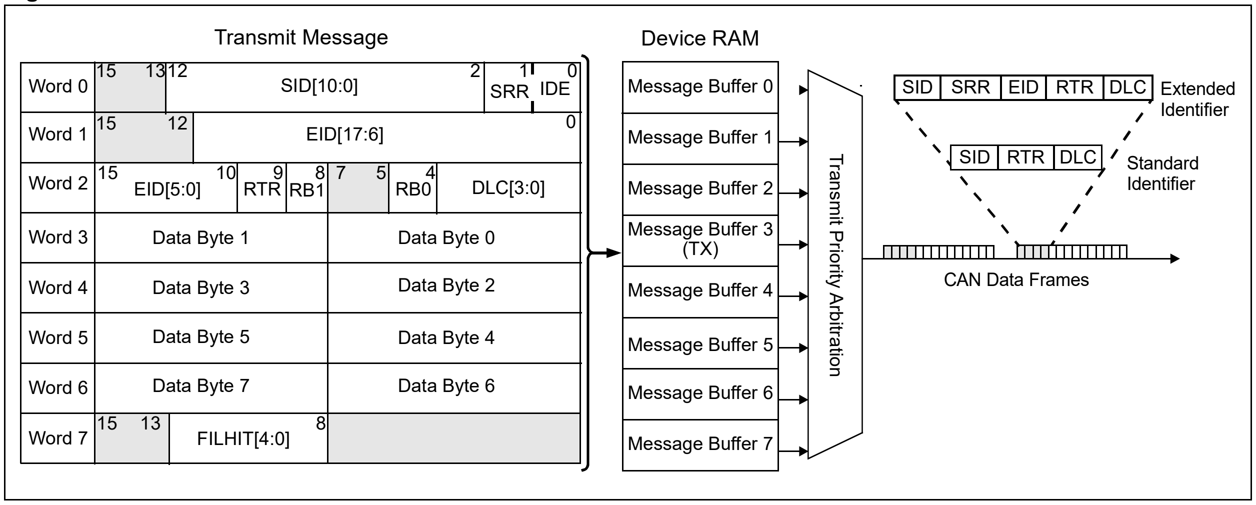
There is a natural order of priority for message buffers that are assigned to the same user application-defined priority level. Message Buffer 7 has the highest natural order of priority. The user application-defined priority levels override the natural order of priority.

If the message fails to transmit, one of the other condition flags will be set and the TXREQ bit will remain set, indicating that the message is still pending for transmission. If the message tries to transmit but encounters an error condition, the TXERR bit will be set. If the message tries to transmit but loses arbitration, the TXLARB bit will be set. If the retransmitted message gets transmitted successfully, then TXREQ will be cleared, but TXLARB would still be set. TXLARB gets cleared when TXREQ is set again.

TABLE 7-1: INTERPRETING THE MESSAGE TRANSMISSION WITH RESPECT TO TXLARB AND TXREQ STATUS

TXLARB	TXREQ	Description
0	0	Message transmission is successful. No loss of arbitration.
0	1	Message transmission is pending.
1	0	Message retransmitted successfully after loss of arbitration.
1	1	The message was not retransmitted after loss of arbitration.

Figure 7-1: CAN Transmission



A code example to transmit a standard frame using Message Buffer 0 is shown in Example 7-1.

Example 7-1: Code Example for Standard Data Frame Transmission

```
#include <p33Exxxx.h>

/*  This code example demonstrates a method to configure the Allie#2009
CAN module to transmit
    Standard ID CAN messages. */

/*  Include fuse configuration code here. Optionally the fuse configuration can be specified
    via MPLAB IDE Menu operations. */

FUSE_CONFIGURATION_MACROS_COME_HERE

#define NUM_OF_CAN_BUFFERS 32

/*  This is the CAN message buffer declaration. Note the buffer alignment. */

        unsigned int can1MsgBuf[NUM_OF_CAN_BUFFERS][8]
        __attribute__((aligned(NUM_OF_CAN_BUFFERS * 16)));

int main(void)
{
    unsigned long address;

    /*  Place code to set device speed here. For this example the device speed should be set at
        40 MHz (i.e., the device is operating at 40 MIPS). */

    ConfigureDeviceClockFor40MIPS();

    /*  The dsPIC33E device features I/O remap. This I/O remap configuration for the CAN
        module can be performed here. */

    SetIORemapForCANModule();

    /*  Set up the CAN1 module to operate at 250 kbps. The CAN module should be first placed
        in configuration mode. */

    C1CTRL1bits.REQOP = 4;
    while(C1CTRL1bits.OPMODE != 4);

    C1CTRL1bits.WIN = 0;

    /*  Set up the CAN module for 250kbps speed with 10 Tq per bit. */

    C1CFG1 = 0x47;         // BRP = 8 SJW = 2 Tq
    C1CFG2 = 0x2D2;
    C1FCTRL = 0xC01F;      // No FIFO, 32 Buffers

/*  Assign 32x8word Message Buffers for CAN1 in device RAM. This example uses DMA0 for TX.
    Refer to Section 9.1 "DMA Operation for Transmitting Data" for details on DMA channel
    configuration for CAN transmit. */

    DMA0CONbits.SIZE = 0x0;
    DMA0CONbits.DIR = 0x1;
    DMA0CONbits.AMODE = 0x2;
    DMA0CONbits.MODE = 0x0;
    DMA0REQ = 70;
    DMA0CNT = 7;
    DMA0PAD = (volatile unsigned int)&C1TXD;
    DMA0STAL = (unsigned int) &can1msgBuf;
    DMA0STAH = 0;
    DMA0CONbits.CHEN = 0x1;
```

Example 7-1: Code Example for Standard Data Frame Transmission (Continued)

```
     /* Configure Message Buffer 0 for Transmission and assign priority */

    C1TR01CONbits.TXEN0 = 0x1;
    C1TR01CONbits.TX0PRI = 0x3;

    /*  At this point the CAN1 module is ready to transmit a message. Place the CAN module in
        Normal mode. */

    C1CTRL1bits.REQOP = 0;
    while(C1CTRL1bits.OPMODE != 0);

    /* Write to message buffer 0 */
    /* CiTRBnSID = 0bxxx1 0010 0011 1100
       IDE = 0b0
       SRR = 0b0
       SID<10:0>= 0b100 1000 1111 */

    can1MsgBuf[0][0] = 0x123C;

    /* CiTRBnEID = 0bxxxx 0000 0000 0000
       EID<17:6> = 0b0000 0000 0000 */

    can1MsgBuf[0][1] = 0x0000;
/* CiTRBnDLC = 0b0000 0000 xxx0 1111
       EID<17:6> = 0b000000
       RTR = 0b0
       RB1 = 0b0
       RB0 = 0b0
       DLC = 0b1111 */

    can1MsgBuf[0][2] = 0x0008;
/* Write message data bytes */
    can1MsgBuf[0][3] = 0xabcd;
    can1MsgBuf[0][4] = 0xabcd;
    can1MsgBuf[0][5] = 0xabcd;
    can1MsgBuf[0][6] = 0xabcd;

    /* Request message buffer 0 transmission */
    C1TR01CONbits.TXREQ0 = 0x1;

/*  The following shows an example of how the TXREQ bit can be polled to check if transmission
    is complete. */

    while(C1TR01CONbits.TXREQ0 == 1);

    /* Message was placed successfully on the bus */
    while(1);
}
```

A code example to transmit an extended frame using Message Buffer 2 is shown in Example 7-2.

Example 7-2: Code Example for Extended Data Frame Transmission

```
#include <p33Exxxx.h>

/*  This code example demonstrates a method to configure the CAN module to transmit Extended
    ID CAN messages. */

/*  Include fuse configuration code here. Optionally the fuse configuration can be specified via
    MPLAB IDE Menu operations. */

FUSE_CONFIGURATION_MACROS_COME_HERE

#define NUM_OF_CAN_BUFFERS 32

/*  This is the CAN message buffer declaration. Note the buffer alignment. */

        unsigned int can1MsgBuf[NUM_OF_CAN_BUFFERS][8]
        __attribute__((aligned(NUM_OF_CAN_BUFFERS * 16)));

int main(void)
{
    unsigned long address;

    /* Place code to set device speed here. For this example the device speed should be set at
       40 MHz (i.e., the device is operating at 40 MIPS). */

ConfigureDeviceClockFor40MIPS();

    /* The dsPIC33E device features I/O remap. This I/O remap configuration for the CAN module
       can be performed here. */

    SetIORemapForCANModule();

/*  Set up the CAN1 module to operate at 250 kbps. The CAN module should be first placed in
    configuration mode. */
    C1CTRL1bits.REQOP = 4;
    while(C1CTRL1bits.OPMODE != 4);

    C1CTRL1bits.WIN = 0;

    /* Set up the CAN module for 250kbps speed with 10 Tq per bit. */

    C1CFG1 = 0x47;     // BRP = 8 SJW = 2 Tq
    C1CFG2 = 0x2D2;
    C1FCTRL = 0xC01F;  // No FIFO, 32 Buffers

/*  Assign 32x8 word Message Buffers for CAN1 in device RAM. This example uses DMA0 for TX.
    Refer to Section 9.1 "DMA Operation for Transmitting Data" for details on DMA channel
    configuration for CAN transmit. */

    DMA0CONbits.SIZE = 0x0;
    DMA0CONbits.DIR = 0x1;
    DMA0CONbits.AMODE = 0x2;
    DMA0CONbits.MODE = 0x0;
    DMA0REQ = 70;
    DMA0CNT = 7;
    DMA0PAD = (volatile unsigned int)&C1TXD;
    DMA0STAL = (unsigned int) &can1msgBuf;
    DMA0STAH = 0;
    DMA0CONbits.CHEN = 0x1;
```

Example 7-2: Code Example for Extended Data Frame Transmission (Continued)

```
    /* Configure Message Buffer 2 for Transmission and assign priority */

    C1TR23CONbits.TXEN2 = 0x1;
    C1TR23CONbits.TX2PRI = 0x3;

    /*  At this point the CAN1 module is ready to transmit a message. Place the CAN module in
        Normal mode. */

    C1CTRL1bits.REQOP = 0;
    while(C1CTRL1bits.OPMODE != 0);

    /* Write to message buffer 2*/
    /* CiTRBnSID = 0bxxx1 0010 0011 1101
       IDE = 0b1
       SRR = 0b1
       SID<10:0> : 0b100 1000 1111 */

    can1MsgBuf[2][0] = 0x123D;

    /* CiTRBnEID = 0bxxxx 1111 0000 0000
       EID<17:6> = 0b1111 0000 0000 */

    can1MsgBuf[2][1] = 0x0F00;

/* CiTRBnDLC = 0b0000 1100 xxx0 1111
       EID<17:6> = 0b000011
       RTR = 0b0
       RB1 = 0b0
       RB0 = 0b0
       DLC = 0b1000 */

    can1MsgBuf[2][2] = 0x0C08;

/* Write message data bytes */
    can1MsgBuf[2][3] = 0xabcd;
    can1MsgBuf[2][4] = 0xabcd;
    can1MsgBuf[2][5] = 0xabcd;
    can1MsgBuf[2][6] = 0xabcd;

/* Request message buffer 2 transmission */
    C1TR23CONbits.TXREQ2 = 0x1;

/*  The following shows an example of how the TXREQ bit can be polled to check if transmission
    is complete. */

    while(C1TR23CONbits.TXREQ2 == 1);

    /* Message was placed successfully on the bus */
    while(1);
}
```

7.2 Aborting a Transmit Message

Setting the Abort All Pending Transmissions bit (ABAT) in the CAN Control Register 1 (CxCTRL1[12]) requests an abort of all pending messages. To abort a specific message, the Message Send Request (TXREQm) bit (CxTRmnCON[3]) associated with that message buffer must be cleared. In either case, the message is only aborted if the CAN module has not started transmitting the message on the bus or the message has been started, but it is interrupted by loss of arbitration or an error.

7.3 Transmitting and Responding Remote Frames

7.3.1 TRANSMIT A REMOTE FRAME

A node expecting to receive a data frame with a specific identifier value can initiate the transmission of the respective data by another node by sending the remote frame. The remote frame can be either in the standard format or extended format.

A remote frame is similar to a data frame, with the following exceptions:

- The RTR bit is recessive (RTR = `1`)
- There is no Data field
- The value of the DLCx bits is $0 \le DLC \le 8$

To transmit a remote frame, the user application must perform these tasks:

1. Configure the message buffer for transmission and assign a priority to the buffer.
2. Write the remote frame in the appropriate message buffer. The transmitted identifier must be identical to the identifier of the data frame to be received.
3. Set the transmit request bit for the buffer to initiate transmission of the remote frame.

7.3.2 RESPOND TO A REMOTE FRAME

The node acting as the source to respond to the remote frame request needs to configure an acceptance filter to match the identifier of the remote frame. Message Buffers 0-7 can respond to remote frames, so the Acceptance Filter Buffer Pointer (FnBP) should point to one of the eight message buffers. The TX/RX Buffer Selection (TXENn) and Auto-Remote Transmit Enable (RTRENm) bits in the CAN TX/RX Buffer m Control register (CxTRMNCON[7] and CxTRmnCON[2]) must be set to respond to the remote frame.

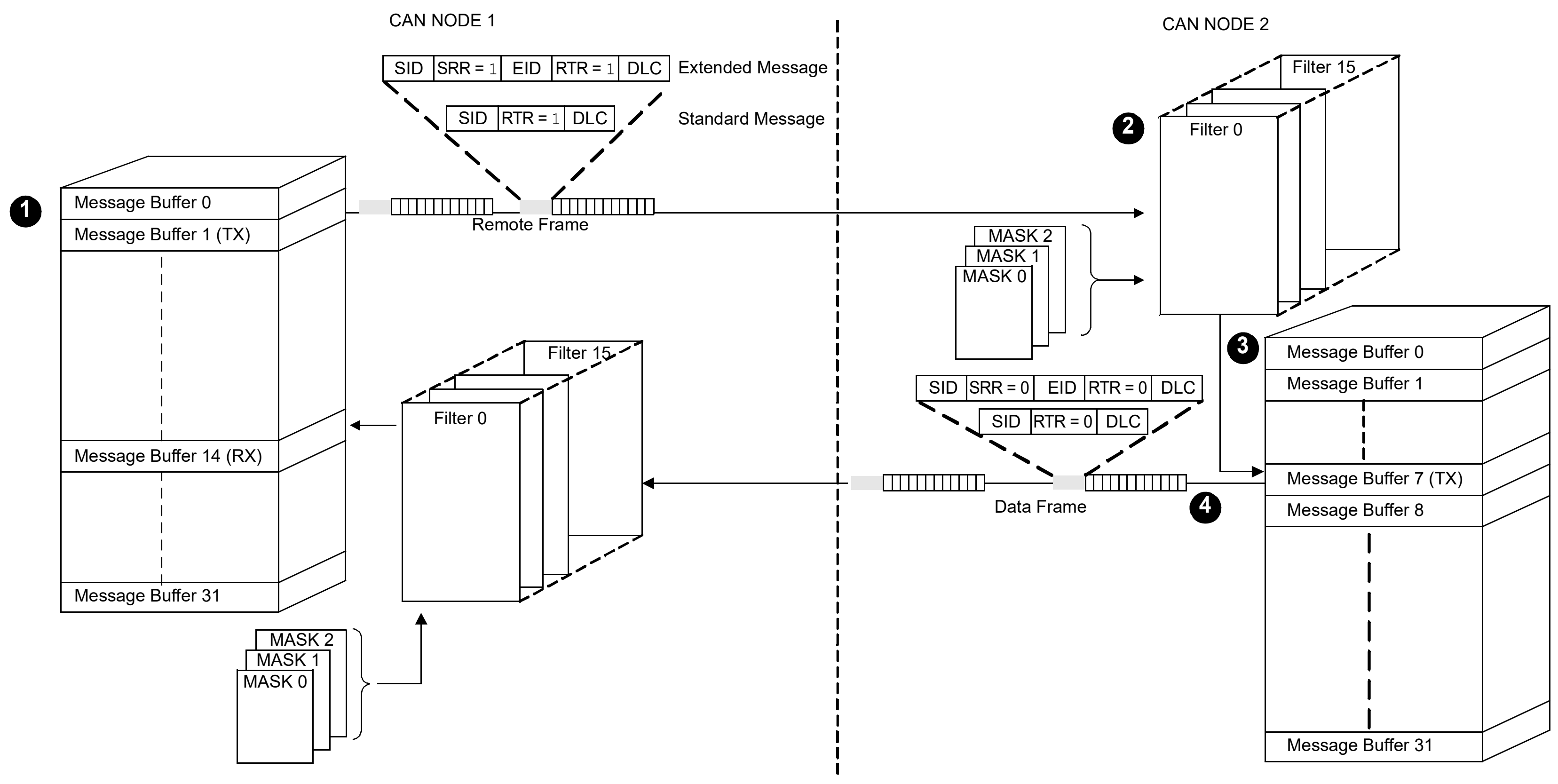
This is the only case where the Acceptance Filter Buffer Pointer (FnBP) points to a message buffer that is configured for transmission (TXENn = `1`).

Figure 7-2 illustrates the remote frame handling process:

1. CAN Node 1 sends an RTR (using Message Buffer 1).
2. CAN Node 2 receives the request and responds by sending the data frame (using Message Buffer 7).
3. The data frame is received by CAN Node 1.
4. The data frame is stored in Message Buffer 14 of CAN Node 1.

Note: When configured for automatic response to remote frames (RTREN = `1`), the CAN module ignores the value of the DLCx bits in the incoming RTR message. If the application needs to transmit a data payload size specified by the DLCx bits in the received RTR message, it should not enable automatic RTR response. The application should process the RTR message like any other received message. Check if the RTR bit is set and then transmit a message whose payload size is equal to the DLCx bits in the receive RTR message.

Figure 7-2: Remote Frame Transmit and Response



Note 1: The node transmitting the remote frame must have a transmit buffer from which to send the remote frame, and one receive buffer to receive the data frame.

2: The node receiving the remote frame must have a transmit buffer from which to transmit a data frame in response to the received remote frame.

3: The FnBPx bits (CxBUFPNTm) should be pointing to a transmit buffer in case of remote transmission.

4: The RTREN bits (CxTRmnCON) should be set so that when a remote transmit is received, the TXREQ bits (CxTRmnCON) will be set automatically.

A code example to transmit an extended remote frame using Message Buffer 2 is shown in Example 7-3.

Example 7-3: Code Example for Transmitting Extended Remote Frame

```
#include <p33Exxxx.h>

/*  This code example demonstrates a method to configure the CAN module to transmit Extended ID
    CAN Remote frames */

/*  Include fuse configuration code here. Optionally, the fuse configuration can be specified
    via MPLAB IDE Menu operations. */

FUSE_CONFIGURATION_MACROS_COME_HERE

#define NUM_OF_CAN_BUFFERS 32

/*  This is the CAN message buffer declaration. Note the buffer alignment. */

        unsigned int can1MsgBuf[NUM_OF_CAN_BUFFERS][8]
        __attribute__((aligned(NUM_OF_CAN_BUFFERS * 16)));

int main(void)
{
    unsigned long address;

    /* Place code to set device speed here. For this example the device speed should be set at
    40 MHz (i.e., the device is operating at 40 MIPS). */

    ConfigureDeviceClockFor40MIPS();

/*  The dsPIC33E device features I/O remap. This I/O remap configuration for the CAN module can
    be performed here. */

    SetIORemapForCANModule();

    /*  Set up the CAN1 module to operate at 250 kbps. The CAN module should be first placed
        in configuration mode. */

    C1CTRL1bits.REQOP = 4;
    while(C1CTRL1bits.OPMODE != 4);

    C1CTRL1bits.WIN = 0;

    /* Set up the CAN module for 250kbps speed with 10 Tq per bit. */
    C1CFG1 = 0x47;     // BRP = 8 SJW = 2 Tq
    C1CFG2 = 0x2D2;
    C1FCTRL = 0xC01F;  // No FIFO, 32 Buffers

    /*  Assign 32x8word Message Buffers for CAN1 in device RAM. This example uses DMA0 for TX.
        Refer to Section 9.1 "DMA Operation for Transmitting Data" for details on DMA channel
        configuration for CAN transmit. */

    DMA0CONbits.SIZE = 0x0;
    DMA0CONbits.DIR = 0x1;
    DMA0CONbits.AMODE = 0x2;
    DMA0CONbits.MODE = 0x0;
    DMA0REQ = 70;
    DMA0CNT = 7;
    DMA0PAD = (volatile unsigned int)&C1TXD;
    DMA0STAL = (unsigned int) &can1msgBuf;
    DMA0STAH = 0;
    DMA0CONbits.CHEN = 0x1;
```

Example 7-3: Code Example for Transmitting Extended Remote Frame (Continued)

```
/* Configure Message Buffer 2 for Transmission and assign priority */

    C1TR23CONbits.TXEN2 = 0x1;
    C1TR23CONbits.TX2PRI = 0x3;

    /*  At this point the CAN1 module is ready to transmit a message. Place the CAN module
        in Normal mode. */

    C1CTRL1bits.REQOP = 0;
    while(C1CTRL1bits.OPMODE != 0);

    /* Write to message buffer 2*/
    /* CiTRBnSID = 0bxxx1 0010 0011 1101
       IDE = 0b1
       SRR = 0b1
       SID<10:0> : 0b100 1000 1111 */

    can1MsgBuf[2][0] = 0x123F;

    /* CiTRBnEID = 0bxxxx 1111 0000 0000
       EID<17:6> = 0b1111 0000 0000 */

    can1MsgBuf[2][1] = 0x0F00;

    /* CiTRBnDLC = 0b0000 1100 xxx0 1111
       EID<17:6> = 0b000011
       RTR = 0b1
       RB1 = 0b0
       RB0 = 0b0
       DLC = 0b1000 */

    /* RTR bit is set */
    can1MsgBuf[2][2] = 0x0E00;

    /* An RTR message does not have a data payload. */

/* Request message buffer 2 transmission */
    C1TR23CONbits.TXREQ2 = 0x1;

    /* The following shows an example of how the TXREQ bit can be polled to check if
        transmission is complete. */

    while(C1TR23CONbits.TXREQ2 == 1);

    /* Message was placed successfully on the bus */
    while(1);
}
```

8.0 RECEIVING CAN MESSAGES

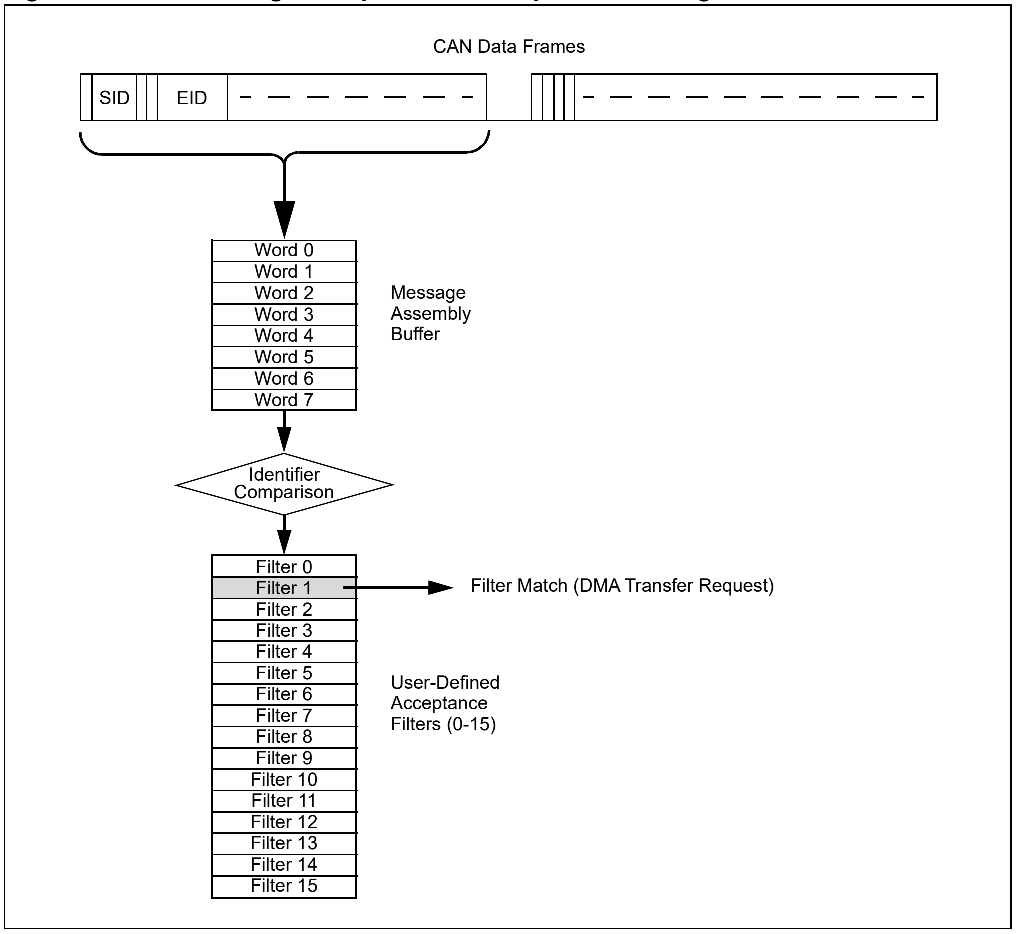
The CAN module can receive both standard and extended frames on the CAN bus node. This module has the additional capability of automatically transferring the received messages to the user-defined buffers in device RAM, thereby eliminating the need for the user application to explicitly copy messages from hardware registers to the user-defined buffers. The storage format of each message within the DMA buffer is identical to that of transmit buffers, with each message (including the associated status register) occupying eight words in device RAM.

The two main stages that constitute the CAN reception process are described below, with a simplified reception process example illustrated in Figure 8-1 and Figure 8-4.

8.1 Message Reception and Acceptance Filtering

As illustrated in Figure 8-1, every incoming message on the bus is received into a Message Assembly Buffer, and its identifier field is compared with a set of 16 user-defined acceptance filters. Each received standard data frame contains an 11-bit SID, and each extended data frame contains an 11-bit SID and an 18-bit EID. If all bits in the incoming identifier completely match the corresponding bits in any of the acceptance filters, the CAN module generates a DMA transfer request to the DMA Controller so that the message can be received into the appropriate buffer in device RAM.

Figure 8-1: Message Reception and Acceptance Filtering 7



8.1.1 ACCEPTANCE FILTERS

Figure 8-2 illustrates the incoming message identifier being compared with the filter/mask bits for standard frames. Figure 8-3 illustrates the incoming message identifier being compared with the filter/mask bits for extended frames.

Figure 8-2: Acceptance Filtering for a Standard Message

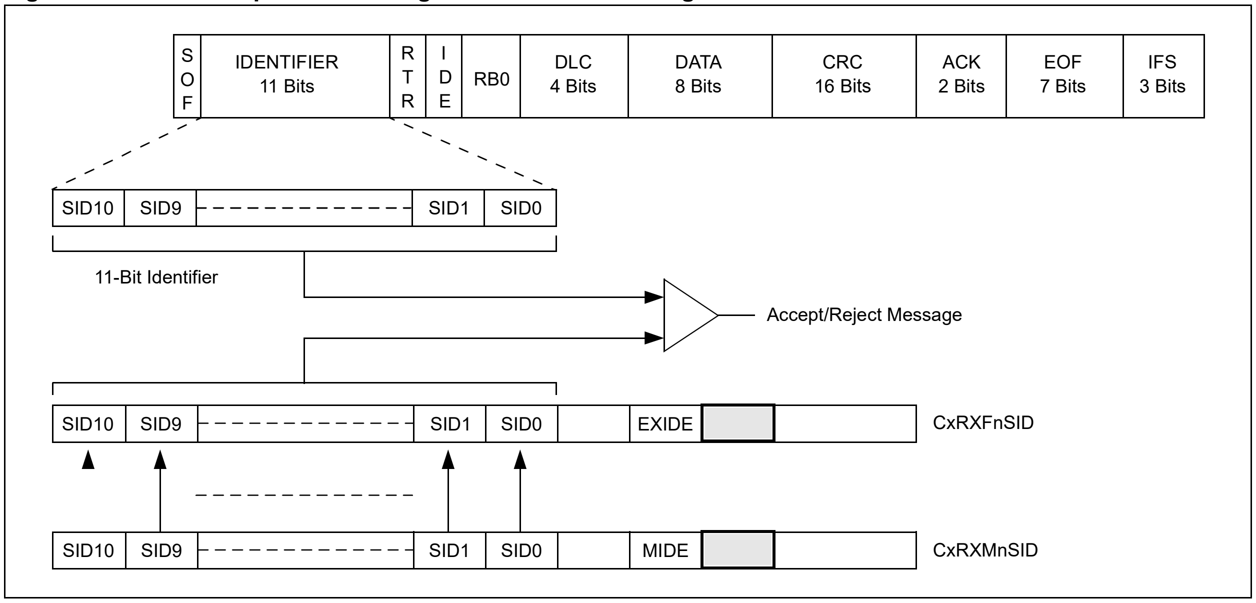
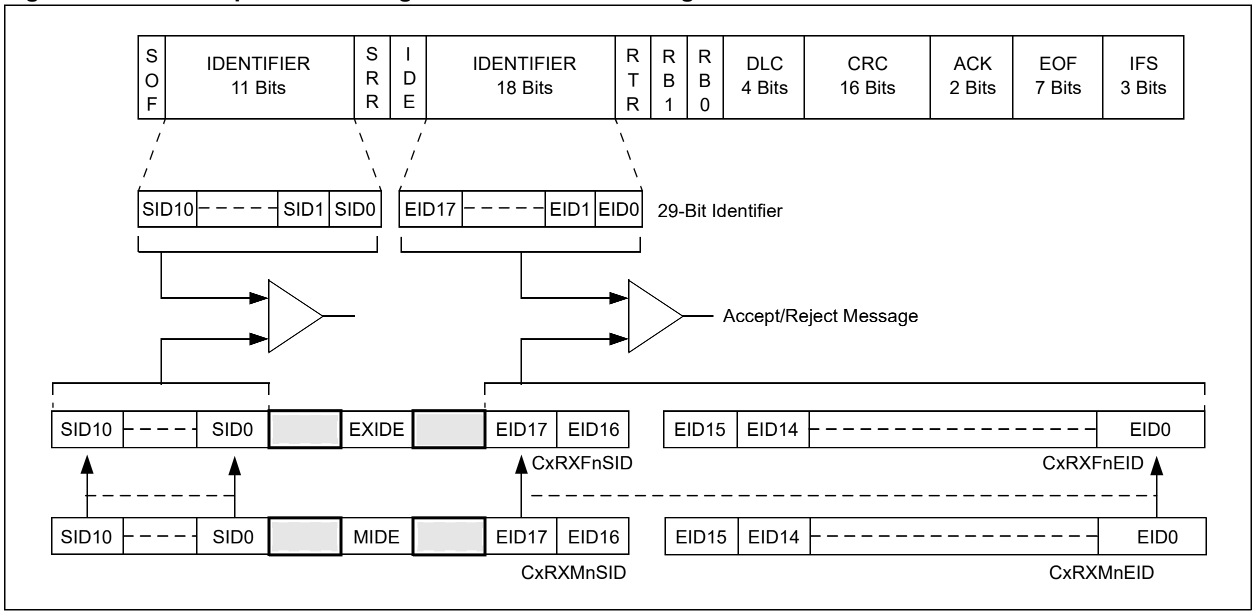


Figure 8-3: Acceptance Filtering for an Extended Message



The Acceptance Filters 0-15 can be individually enabled or disabled using the Enable Filter bits (FLTENx) in the CAN Acceptance Filter Enable register (CxFEN1[15:0]). The value of 'x' signifies the register bit and corresponds to the index of the acceptance filter.

The Acceptance Filters 0-15 specify the identifiers that must be contained in an incoming message for its contents to be passed to a receive buffer. Each of these filters consists of two registers – one for the SIDs and the other for EIDs. These registers are identified as:

- **CxRXFnSID: CAN Acceptance Filter Standard Identifier Register n (n = 0-15)**
- **CxRXFnEID: CAN Acceptance Filter Extended Identifier Register n (n = 0-15)**

8.1.2 ACCEPTANCE FILTER MASKS

As shown in Figure 8-2 and Figure 8-3, an acceptance filter mask determines which bits in the incoming message identifiers are examined with the acceptance filters.

The acceptance filters optionally select one of the acceptance filter masks using the Mask Source Select (FnMSK[1:0]) bits in the CxFMSKSEL1 and CxFMSKSEL2 registers:

- **CxFMSKSEL1: CAN Filter 7-0 Mask Selection Register 1**
- **CxFMSKSEL2: CAN Filter 15-8 Mask Selection Register 2**

The selection values for the FnMSK[1:0] bits are:

- 11 = Reserved
- 10 = Select Acceptance Filter Mask 2
- 01 = Select Acceptance Filter Mask 1
- 00 = Select Acceptance Filter Mask 0

Table 8-1 is a truth table that indicates how each bit in the identifier is compared to the masks and filters to determine if the message should be accepted or rejected. The mask bit essentially determines which bits to apply the filter to. If any mask bit is set to ‘0’, that bit is automatically accepted, regardless of the filter bit.

Table 8-1: Acceptance Filter/Mask Truth Table

Mask (SIDx/EIDx)	Filter (SIDx/EIDx)	Message (SIDx/EIDx)	Accept or Reject Bit x
0	x	x	Accept
1	0	0	Accept
1	0	1	Reject
1	1	0	Reject
1	1	1	Accept

8.1.3 MESSAGE TYPE SELECTION

The Extended Identifier Enable bit (EXIDE) in the CAN Acceptance Filter Standard Identifier Register n (CxRXFnSID[3]) enables reception of either SID or EID messages. The Identifier Receive Mode bit (MIDE) in the CAN Acceptance Filter Mask Standard Identifier Register n (CxRXMnSID[3]) enables the EXIDE bit. If the MIDE bit is set, only the type of message selected by the EXIDE bit is accepted. If the MIDE bit is clear, the EXIDE bit is ignored and all messages that match the filter are accepted.

Table 8-2: Message Type Selections

EXIDE	MIDE	Selection
0	1	Acceptance filter to check for SID
1	1	Acceptance filter to check for EID
x	0	Acceptance filter to check for SID/EID

8.1.4 ACCEPTANCE FILTER CONFIGURATION

A sample code used to configure Acceptance Filter 0 to receive the SID messages using the Acceptance Filter Mask register to mask the SID[2:0] bits is shown in Example 8-1.

Example 8-1: Code Example for Filtering Standard Data Frame

```
#include <p33Exxxx.h>

/*  This code example demonstrates a method to configure the CAN module to receive Standard ID
    CAN messages. SID Messages with SID range 0x1D0-0x1D7 will be accepted. */

/*  Include fuse configuration code here. Optionally the fuse configuration can be specified via
    MPLAB IDE Menu operations. */

FUSE_CONFIGURATION_MACROS_COME_HERE

#define NUM_OF_CAN_BUFFERS 32

/*  This is the CAN message buffer declaration. Note the buffer alignment. */

    unsigned int can1MsgBuf[NUM_OF_CAN_BUFFERS][8]
    __attribute__((aligned(NUM_OF_CAN_BUFFERS * 16)));

int main(void)
{
    unsigned long address;

    /*  Place code to set device speed here. For this example the device speed should be set at
        40 MHz (i.e., the device is operating at 40 MIPS). */

    ConfigureDeviceClockFor40MIPS();

    /*  The dsPIC33E device features I/O remap. This I/O remap configuration for the CAN module
        can be performed here. */

    SetIORemapForCANModule();


    /*  Set up the CAN1 module to operate at 250 kbps. The CAN module should be first placed
        in configuration mode. */

    C1CTRL1bits.REQOP = 4;
    while(C1CTRL1bits.OPMODE != 4);

    C1CTRL1bits.WIN = 1;

    /* Set up the CAN module for 250kbps speed with 10 Tq per bit. */

    C1CFG1 = 0x47;     // BRP = 8 SJW = 2 Tq
    C1CFG2 = 0x2D2;
    C1FCTRL = 0xC01F;  // No FIFO, 32 Buffers

    /*  Assign 32x8word Message Buffers for CAN1 in device RAM. This example uses DMA1 for RX.
        Refer to Section 9.1 "DMA Operation for Transmitting Data" for details on DMA channel
        configuration for CAN transmit. */

    DMA1CONbits.SIZE = 0x0;
    DMA1CONbits.DIR = 0x0;
    DMA1CONbits.AMODE = 0x2;
    DMA1CONbits.MODE = 0x0;
    DMA1REQ = 34;
    DMA1CNT = 7;
    DMA1PAD = (volatile unsigned int)&C1RXD;
    DMA1STAL = (unsigned int) &can1msgBuf;
    DMA1STAH = 0;
    DMA1CONbits.CHEN = 0x1;
```

Example 8-1: Code Example for Filtering Standard Data Frame (Continued)

```
    /* Select Acceptance Filter Mask 0 for Acceptance Filter 0 */

    C1FMSKSEL1bits.F0MSK=0x0;

    /* Configure Acceptance Filter Mask 0 register to mask SID<2:0>
     * Mask Bits (11-bits) : 0b111 1111 1000 */

    C1RXM0SIDbits.SID = 0x7F8;

    /*  Configure Acceptance Filter 0 to match standard identifier
        Filter Bits (11-bits): 0b011 1010 xxx with the mask setting, message with SID
        range 0x1D0-0x1D7 will be accepted by the CAN module. */

    C1RXF0SIDbits.SID = 0x01D0;

    /* Acceptance Filter 0 to check for Standard Identifier */

    C1RXM0SIDbits.MIDE = 0x1;
    C1RXF0SIDbits.EXIDE= 0x0;

    /* Acceptance Filter 0 to use Message Buffer 10 to store message */

    C1BUFPNT1bits.F0BP = 0xA;

    /* Filter 0 enabled for Identifier match with incoming message */

    C1FEN1bits.FLTEN0=0x1;

    /* Clear Window Bit to Access CAN
     * Control Registers */

    C1CTRL1bits.WIN=0x0;

    /* Place the CAN module in normal
     * mode. */

    C1CTRL1bits.REQOP = 0;
    while(C1CTRL1bits.OPMODE != 0);

    /*  The following code shows one example of how the application can wait
        for a message to be received in message buffer 10 */

    while(1)
    {
        /* Message was received. */
        while (C1RXFUL1bits.RXFUL10 == 0);
        C1RXFUL1bits.RXFUL10 = 0;

    }
}
```

A code example used to configure Acceptance Filter 2 to receive EID messages using the Acceptance Filter Mask register to mask the EID[5:0] bits is shown in Example 8-2.

Example 8-2: Code Example for Filtering Extended Data Frame

```
#include <p33Exxxx.h>

/*  This code example demonstrates a method to configure the CAN module to receive Extended
    ID CAN messages. EID Messages with EID range 0x3FFC0-0x3FFFF will be accepted. */

/*  Include fuse configuration code here. Optionally the fuse configuration can be specified
    via MPLAB IDE Menu operations. */

FUSE_CONFIGURATION_MACROS_COME_HERE

#define NUM_OF_CAN_BUFFERS 32

/*  This is the CAN message buffer declaration. Note the buffer alignment. */

    unsigned int can1MsgBuf[NUM_OF_CAN_BUFFERS][8]
    __attribute__((aligned(NUM_OF_CAN_BUFFERS * 16)));

int main(void)
{
    unsigned long address;

/*  Place code to set device speed here. For this example the device speed should be set at
    40 MHz (i.e., the device is operating at 40 MIPS). */

    ConfigureDeviceClockFor40MIPS();

    /*  The dsPIC33E device features I/O remap. This I/O remap configuration for the CAN
        module can be performed here. */

    SetIORemapForCANModule();

    /*  Set up the CAN1 module to operate at 250 kbps. The CAN module should be first placed
        in configuration mode. */

    C1CTRL1bits.REQOP = 4;
    while(C1CTRL1bits.OPMODE != 4);

    C1CTRL1bits.WIN = 1;

    /* Set up the CAN module for 250 kbps speed with 10 Tq per bit. */

    C1CFG1 = 0x47;     // BRP = 8 SJW = 2 Tq
    C1CFG2 = 0x2D2;
    C1FCTRL = 0xC01F;  // No FIFO, 32 Buffers

/* Configure Acceptance Filter Mask 1
     * register to mask EID<5:0>
     * Mask Bits (29-bits) : 0b1 1111 1111 1111 1111 1111 1100 0000
     * SID<10:0> : 0b11111111111 ..SID<10:0> or EID<28:18>
     * EID<17:16> : 0b11 ..EID<17:16>
     * EID<15:0> : 0b1111111111000000 ..EID<15:0> */

    C1RXM1SID = 0xFFEB;
    C1RXM1EID = 0xFFC0;

     /* Configure Acceptance Filter 2 to match extended identifier
     * Filter Bits (29-bits) : 0b0 0000 0000 0011 1111 1111 11xx xxxx
     * SID<10:0> : 0b00000000000 ..SID<10:0> or EID<28:18>
     * EID<17:16> : 0b11 ..EID<17:16>
     * EID<15:0> : 0b1111111111xxxxxx ..EID<15:0> */

    C1RXF2SID = 0xB;
    C1RXF2EID = 0xFFFF;

     /* Acceptance Filter 2 to use Message Buffer 6 to store message */

    C1BUFPNT1bits.F2BP = 0x6;

    /* Filter 2 enabled for Identifier match with incoming message */

    C1FEN1 = 0;
    C1FEN1bits.FLTEN2 = 0x1;
```

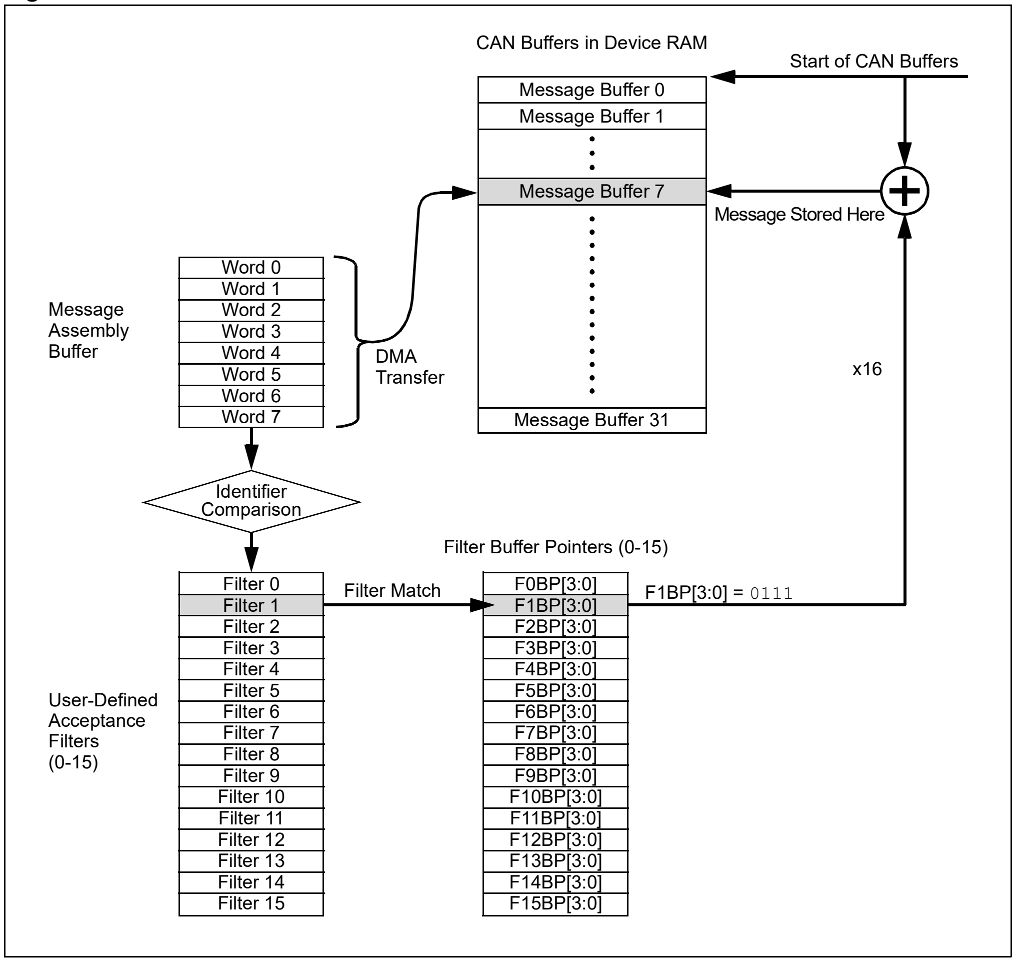
Example 8-2: Code Example for Filtering Extended Data Frame (Continued)

```
/*  Assign 32x8word Message Buffers for CAN1 in device RAM. This example uses DMA1 for RX.
    Refer to Section 9.1 “DMA Operation for Transmitting Data” for details on DMA channel
    configuration for CAN transmit. */

    DMA1CONbits.SIZE = 0x0;
    DMA1CONbits.DIR = 0x0;
    DMA1CONbits.AMODE = 0x2;
    DMA1CONbits.MODE = 0x0;
    DMA1REQ = 34;
    DMA1CNT = 7;
    DMA1PAD = (volatile unsigned int)&C1RXD;
    DMA1STAL = (unsigned int) &can1msgBuf;
    DMA1STAH = 0;
    DMA1CONbits.CHEN = 0x1;

    /* Select Acceptance Filter Mask 1 for Acceptance Filter 2 */

    C1FMSKSEL1bits.F2MSK=0x1;

    /* Clear Window Bit to Access CAN Control Registers */

    C1CTRL1bits.WIN = 0x0;

    C1CTRL1bits.REQOP = 0;
    while(C1CTRL1bits.OPMODE != 0);

/*  The following code shows one example of how the application can wait for a message to be
    received in message buffer 6 */

    while(1)
    {
        /* Message was received. */
        while (C1RXFUL1bits.RXFUL6 == 0);
        C1RXFUL1bits.RXFUL6 = 0;
    }
}
```

8.2 Buffer Selection and DMA Transfer

As illustrated in Figure 8-4, if a filter match occurs, a DMA transfer request is generated by the CAN module to the DMA Controller to automatically copy the received message into the appropriate message buffer in a user-defined device RAM area. The CAN module supports up to 32 message buffers. The user application can use the DMA Message Buffer Size bits (DMABS[2:0]) in the CAN FIFO Control register (CxFCTRL[15:13]) to select either 4, 6, 8, 12, 16, 24 or 32 message buffers. The selection of the receive buffer index (and therefore, the device RAM addresses in which a message is written by the DMA Controller) is dependent on which filter matched the incoming identifier, and is configurable by the user application. The DMA Controller moves the data into the appropriate addresses in the device RAM area and generates a DMA interrupt after the user-specified number of words are transferred. For more details on DMA channel configuration for CAN data transfers, refer to **Section 9.0 "DMA Controller Configuration"**.

Figure 8-4: Buffer Selection and DMA Transfer



8.2.1 BUFFER SELECTION

There are four Acceptance Filter Buffer Pointer registers that select which message buffer the received message is stored into for Acceptance Filters 0-15.

- **CxBUFPNT1: CAN Filter 0-3 Buffer Pointer Register 1**
- **CxBUFPNT2: CAN Filter 4-7 Buffer Pointer Register 2**
- **CxBUFPNT3: CAN Filter 8-11 Buffer Pointer Register 3**
- **CxBUFPNT4: CAN Filter 12-15 Buffer Pointer Register 4**

When the incoming message identifier is matched by one of the acceptance filters, the internal logic looks up the Buffer Pointer (FnBP[3:0]) and uses that as an address for the corresponding message buffer. The address is provided to the DMA channel by the peripheral. Therefore, the DMA channel must be configured in the Peripheral Indirect mode.

The values for FnBP[3:0] are interpreted as follows:

- `1111` = Message is received in Receive FIFO Buffer
- `1110` = Message is received in Message Buffer 14

  •
  •
  •
- `0001` = Message is received in Message Buffer 1
- `0000` = Message is received in Message Buffer 0

Note: Multi-message buffering can be implemented by a user application by configuring multiple acceptance filters with the same value. In this case, a received message may match multiple filters, and the CAN module will assign the message to the lowest numbered matching filter pointing to an empty buffer.

8.2.2 RECEIVING MESSAGES INTO MESSAGE BUFFERS 0-7

Message Buffers 0-7 can be configured to transmit or receive the CAN messages using the TX/RX Buffer Selection bit (TXENm) in the CAN TX/RX Buffer m Control register (CxTRmnCON[7]). The Acceptance Filter Buffer Pointer (FnBPx) selects one of the message buffers to store the received message, provided it is configured as a receive buffer (TXENm = `0`).

If a message buffer is set up as a transmitter with the RTRENm bit (CxTRmnCON[2]) set, and an acceptance filter pointing to that message buffer detects a message, the message buffer will handle the RTR instead of storing the message. This is the only case where the Acceptance Filter Buffer Pointer (FnBPx) points to a message buffer that is configured for transmission (TXENn = `1`).

Note: The user application should not set the TXREQ bits (CxTRmnCON) when the buffer is configured for receive operation (TXEN = `0`). This could result in unpredictable module behavior.

8.2.3 RECEIVING MESSAGES INTO MESSAGE BUFFERS 8-14

Message Buffers 8-14 are receive buffers. The Acceptance Filter Buffer Pointer (FnBPx) determines which message buffer to use.

8.2.4 RECEIVING MESSAGES INTO MESSAGE BUFFERS 15-31

Message Buffers 15-31 are receive buffers and are only usable as FIFO buffers because the Acceptance Filter Buffer Pointer bits (FnBP[3:0]) can only directly address 16 entities. When FnBP[3:0] = `1111`, the results of a hit on that filter will write to the next available buffer location within the FIFO.

8.2.5 RECEIVE BUFFER STATUS BITS

The receive buffers contain two status bits per message buffer: the Receive Buffer Full flag (RXFULx), and the Receive Buffer Overflow flag (RXOVFx). These status bits are grouped into registers for buffer full status and buffer overflow status.

- **CxRXFUL1: CAN Receive Buffer Full Register 1**
- **CxRXFUL2: CAN Receive Buffer Full Register 2**
- **CxRXOVF1: CAN Receive Buffer Overflow Register 1**
- **CxRXOVF2: CAN Receive Buffer Overflow Register 2**

When a received message is stored into a message buffer, the respective Receive Buffer Full flag (RXFULx) is set in the CAN Receive Buffer Full register, and a received buffer interrupt (CxINTF[1]) is generated. If an incoming message caused a filter match, and the message buffer assigned to the matching filter is full (that is, the RXFULx bit associated with that buffer is already '1'), the corresponding RXOVFx bit (where 'x' is the number of the message buffer associated with that buffer) is set and a receive buffer overflow interrupt is generated (CxINTF[2]). The message is lost.

Note: If multiple filters match the identifier of the incoming message, and all the message buffers assigned to all the matching filters are full, the RXOVFx bit corresponding to the lowest numbered matching filter is set.

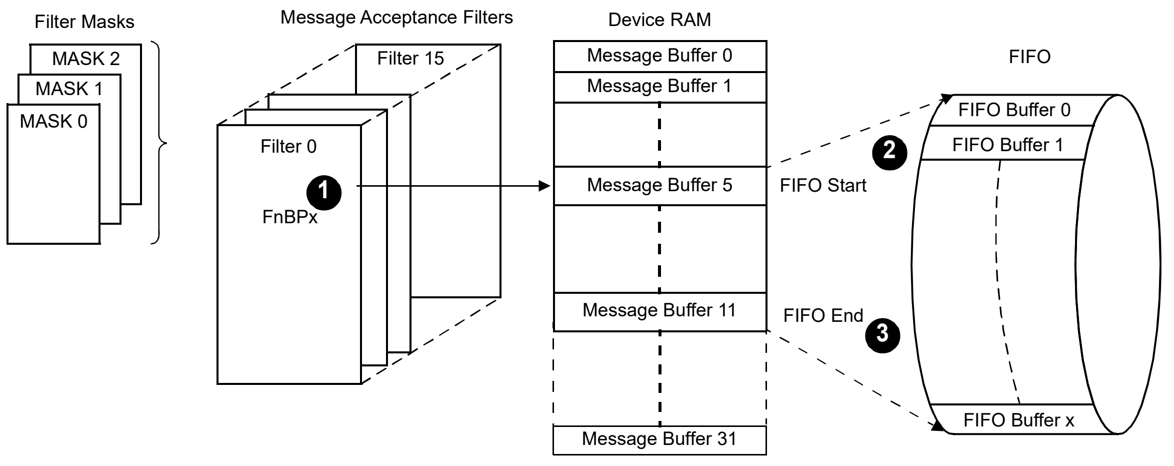
8.3 FIFO Buffer Operation

The CAN module supports up to 32 message buffers. The user application can employ the DMA Message Buffer Size bits (DMABS[2:0]) in the CAN FIFO Control register (CxFCTRL[15:13]) to specify the 4, 6, 8, 12, 16, 24 or 32 message buffers. The FIFO Start Area (FSA[4:0]) bits (CxFCTRL[4:0]) are used to specify the start of the FIFO within the buffer area. The end of FIFO is based on the number of message buffers defined in the DMABS[2:0] bits.

The user application should not allocate a FIFO area that contains the transmit buffers. Should this condition occur, the module will attempt to point to the transmit buffer, but when a message is received for that buffer, an overflow condition will cause the message contents to be lost.

Figure 8-5 illustrates that one of the message acceptance filters is set to store a received message in FIFO (FnBPx = `1111`). The start of FIFO is set to Message Buffer 5 (CxFCTRL[4:0] = `00101`) and the end of FIFO is set to Message Buffer 11 (CxFCTRL[15:13] = `011`) by allocating 12 message buffers.

Figure 8-5: Receiving Messages in FIFO



Note 1: The Acceptance Filter Buffer Pointer (FnBPx) should be '`1111`' to store the received message in FIFO.
2: The starting address of the FIFO is specified by the FSA[4:0] bits (CxFCTRL[4:0]). In the figure, FSA [4:0] = `00101`.
3: The end address of the FIFO is specified by DMABS[2:0] = `011` (CxFCTRL[15:13]).

8.3.1 RECEIVING MESSAGES INTO FIFO AREA

The acceptance filter stores the received message in the FIFO area when FnBP[3:0] = `1111`. It uses a simple Buffer Pointer, beginning with the start of the FIFO as defined above, and incrementing sequentially through the set of buffers within the FIFO area. When the end of the buffer is reached, the counter wraps and points to the start of the FIFO area.

The Write Pointer value is accessible and is readable only by the user software in the FBP[5:0] bits (CxFIFO[13:8]). When the message is stored in the buffer, the RXFULx bit associated with the buffer is set and the FIFO buffer counter increments.

If the FBP[5:0] value points to a buffer, and the RXFULx bit associated with that buffer is already '`1`' at the time of the filter hit, and before writing of the message contents, the RXOVLx bit associated with that buffer is set and the message is lost. After the message is lost, the FBP[5:0] value increments normally.

If the FBP[5:0] value points to a transmit/receive buffer that is selected as a transmit buffer at the time of the filter hit, and before writing of the message contents, the RXOVLx bit associated with that buffer is set and the message is lost. After the message is lost, the FBP[5:0] value increments normally.

The user software unloads the FIFO by reading the contents of a buffer. Once the buffer location is read, the user software clears the RXFULx bit corresponding to that buffer. When an RXFULx bit is cleared, the number of that corresponding buffer, plus one, is written to the FNRB[5:0] bits (CxFIFO[5:0]) by the module. Only the user software can read this value; left shift it by four bits and use it as an address offset for the next buffer to be read. The application software should read the buffers and clear the corresponding RXFULx bit sequentially. If not, and an RXFULx flag is set within a section of the FIFO intended to be available for writing, the RXOVLx bit associated with that buffer is set and the message is lost. After the message is lost, the FBP[5:0] value will increment, but FNRB[5:0] will not get incremented.

The module generates an interrupt condition if the FIFO is about to be full. This condition is computed as shown in Equation 8-1.

Equation 8-1: FIFO Interrupt Calculation

$$FNRB - FBP = 1$$

or

$$(FNRB = START) \text{ and } (FBP = END)$$

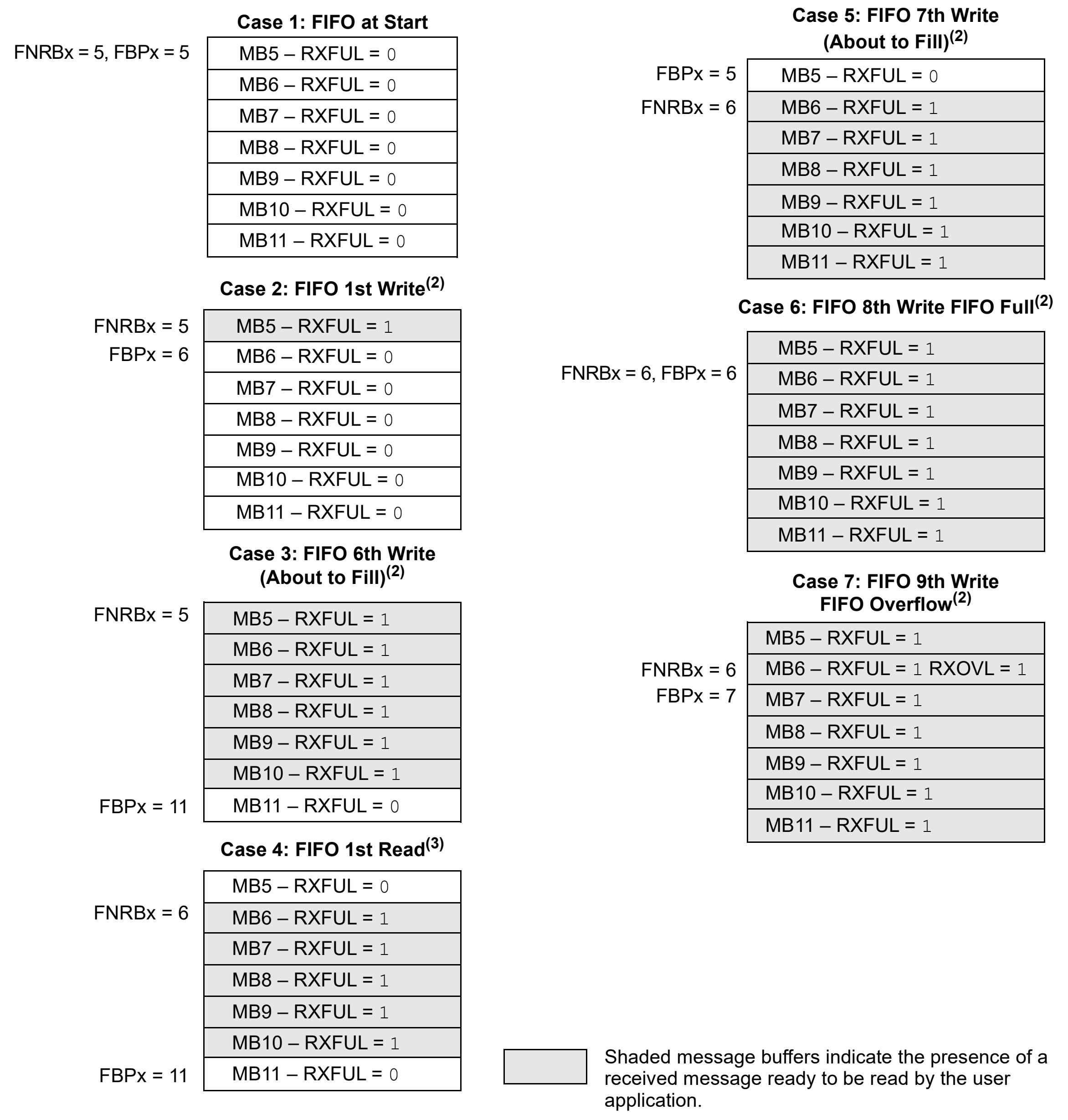
The interrupt is generated as the RXFULx bit is set for the buffer that was just written to, and after the FBPx bit has been updated. The computation uses the updated FBPx value.

8.4 FIFO Example

Figure 8-6 illustrates seven case examples of FIFO operation. The cases illustrated assume that the start of FIFO is set to Message Buffer 5 (CxFCTRL[4:0] = `101`) and the end of FIFO is set to Message Buffer 11 (CxFCTRL[15:13] = `011`).

- **Case 1** – is the initialized case of the FIFO before any messages are received. The FIFO Buffer Pointer points to Message Buffer 5 (FRBx = 5) and the FIFO Next Read Buffer Pointer points to Message Buffer 6 (FNRBx = 5).
- **Case 2** – shows the FIFO after one message is received and transferred to Message Buffer 5. The FIFO Buffer Pointer is incremented (FBPx = 6) and the RXFULx status bit for Message Buffer 5 is set (RXFULx = `1`).
- **Case 3** – shows the FIFO after the sixth received message. The FIFO Buffer Pointer points to the last location in the FIFO area (FBPx = 5 + 6 = 11) and the FIFO Next Read Pointer points to the start of the FIFO (FNRBx = 5). In this case, the FIFO is almost full and generates a FIFO interrupt.
- **Case 4** – shows the FIFO after the user software reads the first received message. When the user software clears the RXFUL status bit for message buffer 5, the module writes the FIFO Next Read Buffer Pointer with message buffer 5 plus 1 (FNRB = 5 + 1 = 6).
- **Case 5** – shows the FIFO after the seventh message is received and written to Message Buffer 11. The RXFULx status bit for Message Buffer 11 is set (RXFULx = `1`). Instead of incrementing, the FIFO Buffer Pointer is reloaded with the FIFO start address (FBPx = FSAx = 5). Note that FBPx is now mathematically one less than FNRBx, which is the condition that generates the FIFO interrupt at the time the RXFULx status bit is set for Message Buffer 11.
- **Case 6** – shows the FIFO after the eighth message is received and written to Message Buffer 5. Now the FIFO is full. There is no interrupt signaled for this condition.
- **Case 7** – shows the FIFO after the ninth received message. Now the FIFO has overflowed. The module sets the RXOVLx bit for the buffer intended for writing. The message is lost. The module generates a receive overflow interrupt.

Figure 8-6: Example of FIFO Operation[1]



Note 1: MBn represents Message Buffers 5-11.

2: **‘Write’** signifies that a message is stored in the FIFO message buffer and the RXFULx flag associated with that buffer is set.

3: **‘Read’** signifies that the user software unloads the FIFO message buffer by reading the contents of that buffer. Once the buffer location is read, the user software clears the RXFULx bit corresponding to that buffer.

8.5 DeviceNet™ Filtering

The DeviceNet filtering feature is based on the CAN Specification 2.0A protocol, in which up to 18 bits of the Data field can be compared with the EID of the message acceptance filter in addition to the SID.

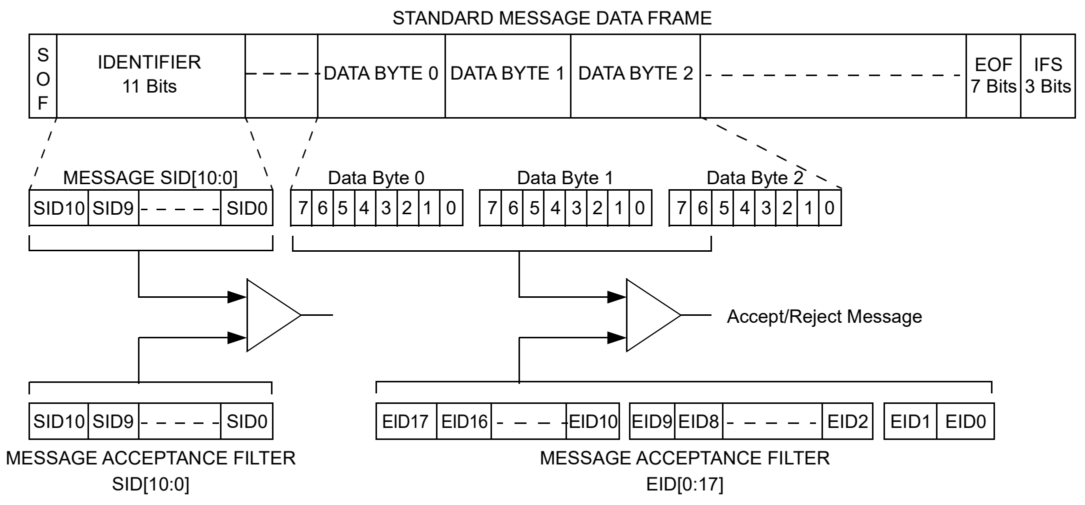
The DeviceNet feature is enabled or disabled by the DeviceNet Filter Bit Number (DNCNT[4:0]) bits in CAN Control Register 2 (CxCTRL2[4:0]). The value specified in the DNCNTx field determines the number of data bits to be used for comparison with the EID bits of the message acceptance filter. If the DNCNT[4:0] bits (CxCTRL2[4:0]) are cleared, the DeviceNet feature is disabled.

For a message to be accepted, the 11-bit SID must match the SID[10:0] bits in the message acceptance filter and the first 'n' data bits in the message should match the EID[17:0] bits in the message acceptance filter. For example, as illustrated in Figure 8-7, the first 18 data bits of the received message are compared with the corresponding identifier bits (EID[17:0]) of the message acceptance filter.

Note: The DeviceNet filtering feature will function only when all of the following are true:

- The IDE bit (CAN Message Buffer Word 0[0]) = `0`, which means the message is a Standard ID message
- The EXIDE bit (CxRXFn[3]) = `0`
- The MIDE bit (CxRXMnSID[3] = `1`
- The value of the DNCNT[4:0] bits (CxCTRL2[4:0]) is non-zero

Figure 8-7: CAN Operation with DeviceNet™ Filtering(1,2)



Note 1: The DeviceNet™ filtering configuration shown for the EIDx bits is DNCNT[4:0] = `10010`.

2: Any message acceptance filter used for DeviceNet filtering must have its MIDE bit set (CxRXMnSID[3] = `1`) and EXIDE bit cleared (CxRXFnSID[3] = `0`).

8.5.1 FILTER COMPARISONS

Table 8-1 shows the filter comparisons configured by the DNCNT[4:0] control bits (CxCTRL2[4:0]). For example, if DNCNT[4:0] = `00011`, a message in which only the 11-bit SID matches the SID acceptance filter (SID[10:0]), and bits 7, 6 and 5 of Data Byte 0 match the EID filter (EID[2:0]), the configuration is accepted.

Table 8-1: DeviceNet™ Filter Bit Configurations

DeviceNet™ Filter Configuration (DNCNT[4:0])	Received Message Data Bits to be Compared (Byte[bits])	EID Bits Used for Acceptance Filter
`00000`	No Comparison	No comparison
`00001`	Data Byte 0[7]	EID[17]
`00010`	Data Byte 0[7:6]	EID[17:16]
`00011`	Data Byte 0[7:5]	EID[17:15]
`00100`	Data Byte 0[7:4]	EID[17:14]
`00101`	Data Byte 0[7:3]	EID[17:13]
`00110`	Data Byte 0[7:2]	EID[17:12]
`00111`	Data Byte 0[7:1]	EID[17:11]
`01000`	Data Byte 0[7:0]	EID[17:10]
`01001`	Data Byte 0[7:0] and Data Byte 1[7]	EID[17:9]
`01010`	Data Byte 0[7:0] and Data Byte 1[7:6]	EID[17:8]
`01011`	Data Byte 0[7:0] and Data Byte 1[7:5]	EID[17:7]
`01100`	Data Byte 0[7:0] and Data Byte 1[7:4]	EID[17:6]
`01101`	Data Byte 0[7:0] and Data Byte 1[7:3]	EID[17:5]
`01110`	Data Byte 0[7:0] and Data Byte 1[7:2]	EID[17:4]
`01111`	Data Byte 0[7:0] and Data Byte 1[7:1]	EID[17:3]
`10000`	Data Byte 0[7:0] and Data Byte 1[7:0]	EID[17:2]
`10001`	Byte 0[7:0] and Byte 1[7:0] and Byte 2[7]	EID[17:1]
`10010`	Byte 0[7:0] and Byte 1[7:0] and Byte 2[7:6]	EID[17:0]
`10011` • • • `11111`	Invalid Selection	Invalid Selection

8.5.2 SPECIAL CASES

There may be special cases when the message contains fewer data bits than are called for by the DeviceNet filter configuration:

- **Case 1 –** If the DNCNT[4:0] bits are greater than 18, indicating that the user application selected a number of bits greater than the total number of EID bits, the filter comparison terminates with the 18th bit of the data (bit 6 of Data Byte 2). If the SID and all 18 data bits match, the message is accepted.
- **Case 2 –** If the DNCNT[4:0] bits are greater than 16, and the received message DLC is 2 (indicating a payload of two data bytes), the filter comparison terminates with the 16th bit of data (bit 0 of Data Byte 1). If the SID and all 16 bits match, the message is accepted.
- **Case 3 –** If the DNCNT[4:0] bits are greater than 8, and the received message has DLC = `1` (indicating a payload of one data byte), the filter comparison terminates with the 8th bit of data (bit 0 of Data Byte 0). If the SID and all 8 bits match, the message is accepted.
- **Case 4 –** If the DNCNT[4:0] bits are greater than 0, and the received message has DLC = `0`, indicating no data payload, the filter comparison terminates with the SID. If the SID matches, the message is accepted.

9.0 DMA CONTROLLER CONFIGURATION

The CAN module uses device RAM for the message buffers to support both transmission and reception of CAN messages. The number of message buffers to be used by the CAN module is specified by the DMA Message Buffer Size bits (DMABS[2:0]) in the CAN FIFO Control register (CxFCTRL[15:13]). The DMAxSTAL and DMAxSTAH registers in the DMA Controller define the start of the CAN buffer area. The DMA Controller moves data between CAN and the message buffers (placed in device RAM) without CPU intervention.

Note: For more information on configuring and using the DMA module, refer to **"Direct Memory Access (DMA)"** (DS70348) of the *"dsPIC33/PIC24 Family Reference Manual"*, which is available from the Microchip website (www.microchip.com).

9.1 DMA Operation for Transmitting Data

The user application selects a message for transmission by setting the Message Send Request bit (TXREQ) in the CAN TX/RX Buffer m or n Control register (CxTRmnCON[3]). The CAN controller uses DMA to read the message from the message buffer and transmit the message. The CAN module generates a transmit data interrupt to start a DMA cycle. In response to the interrupt, the DMA channel that is configured for CAN message transmission reads from the message buffer in device RAM and stores the message in the CAN Transmit Data register (CxTXD). Eight words are transferred for every message transmitted by the CAN controller. The detailed layout of the transmit message buffer is provided in **Section 2.0 "CAN Message Formats"**. A sample code for configuring the DMA channel for CAN1 transmission is shown in Example 9-1.

Example 9-1: DMA Channel 0 Configuration for CAN1 Transmission

```
/* This code snippet shows an example of configuring a DMA channel for CAN
transmission. Refer to Example 7-1 for an application example */

/* Data Transfer Size: Word Transfer Mode */
DMA0CONbits.SIZE = 0x0;

/* Data Transfer Direction: device RAM to Peripheral */
DMA0CONbits.DIR = 0x1;

/* DMA Addressing Mode: Peripheral Indirect Addressing mode */
DMA0CONbits.AMODE = 0x2;

/* Operating Mode: Continuous, Ping-Pong modes disabled */
DMA0CONbits.MODE = 0x0;

/* Assign CAN1 Transmit event for DMA Channel 0 */
DMA0REQ = 70;

/* Set Number of DMA Transfer per CAN message to 8 words */
DMA0CNT = 7;

/* Peripheral Address: CAN1 Transmit Register */
DMA0PAD = (volatile unsigned int); & CITxD;

unsigned long address;

DMA0STAL = (unsigned int) &can1msgBuf;
DMA0STAH = 0;

/* Channel Enable: Enable DMA Channel 0 */
DMA0CONbits.CHEN = 0x1;

/* Channel Interrupt Enable: Enable DMA Channel 0 Interrupt */
IEC0bits DMA0IE = 1;
```

9.2 DMA Operation for Receiving Data

When the CAN controller completes a received message (eight words), the completed message is transferred to a message buffer in device RAM by the DMA module. The CAN module generates a receive data interrupt to start a DMA cycle. In response to this interrupt, the DMA channel that is configured for CAN message reception reads from the CAN Receive Data register (CxRXD) and stores the message in the device RAM buffer. Eight words are transferred for every message that is received by the CAN controller. The detailed layout of the received message is provided in **Section 2.0 "CAN Message Formats"**. A sample code for configuring the DMA channel for CAN1 reception is shown in Example 9-2.

Example 9-2: DMA Channel 1 Configuration for CAN1 Reception

```
/* This code snippet shows an example of configuring a DMA channel for the
CAN message reception. Refer to Example 8-1 for an application example */

/* Data Transfer Size: Word Transfer Mode */
DMA1CONbits.SIZE = 0x0;

/* Data Transfer Direction: Peripheral to device RAM */
DMA1CONbits.DIR = 0x0;

/* DMA Addressing Mode: Peripheral Indirect Addressing mode */
DMA1CONbits.AMODE = 0x2;

/* Operating Mode: Continuous, Ping-Pong modes disabled */
DMA1CONbits.MODE = 0x0;

/* Assign CAN1 Receive event for DMA Channel 0 */
DMA1REQ = 34;

/* Set Number of DMA Transfer per CAN message to 8 words */
DMA1CNT = 7;

/* Peripheral Address: CAN1 Receive Register */
DMA1PAD = (volatile unsigned int) &C1RXD;
.
.
.
.
unsigned long address;

DMA1STAL = (unsigned int) &can1msgBuf;
DMA1STAH = 0;

/* Channel Enable: Enable DMA Channel 1 */
DMA1CONbits.CHEN = 0x1;

/* Channel Interrupt Enable: Enable DMA Channel 1 Interrupt */
IEC0bits.DMA1IE = 1;
```

Note: Please refer to **"Direct Memory Access (DMA)"** (www.microchip.com/DS70348) in the *"dsPIC33/PIC24 Family Reference Manual"* for more information for configuring the DMA Controller.

10.0 BIT TIMING

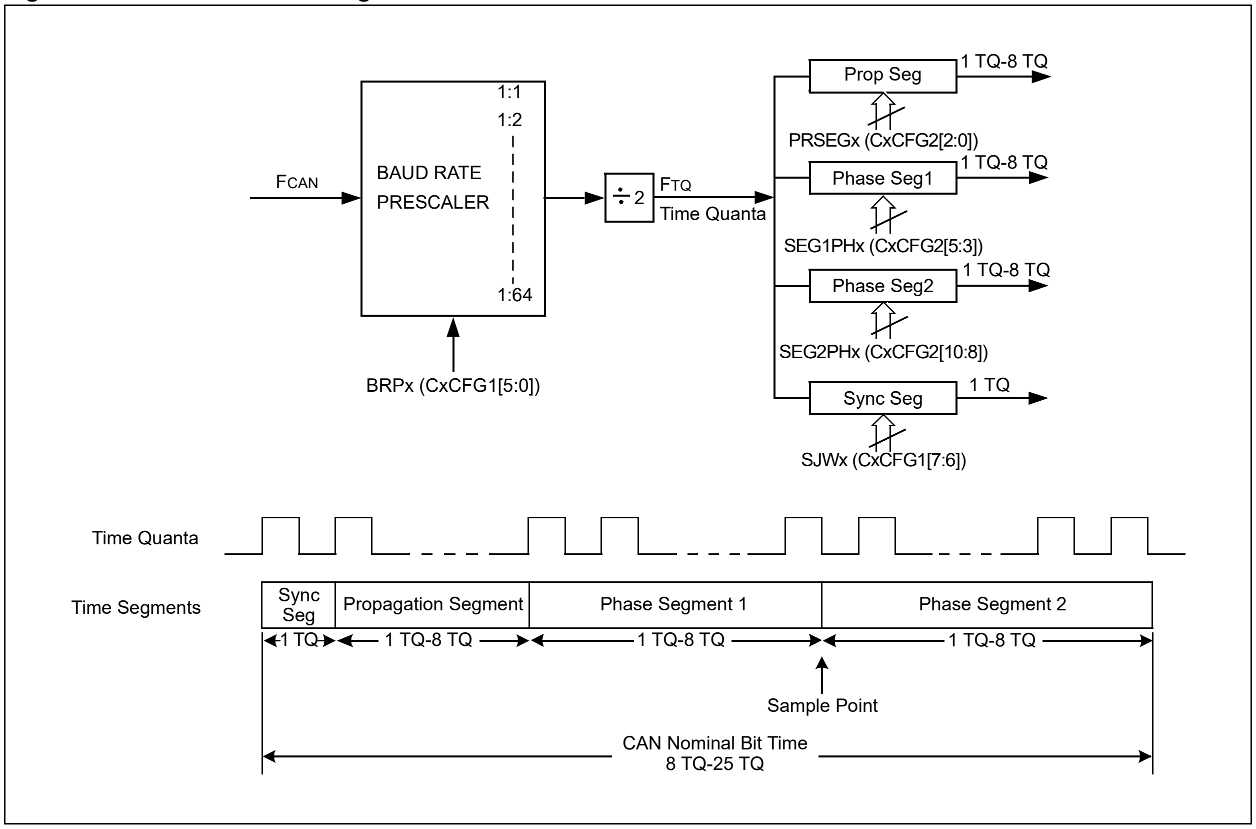
The nominal bit rate is the number of bits per second transmitted on the CAN bus.

Nominal Bit Time = 1 ÷ Nominal Bit Rate

There are four time segments in a bit time to compensate for any phase shifts due to oscillator drifts or propagation delays. These time segments do not overlap each other and are represented in terms of Time Quanta (TQ). One TQ is a fixed unit of time derived from the oscillator clock. The total number of Time Quanta in a nominal bit time must be programmed between 8 TQ and 25 TQ.

Figure 10-1 illustrates how the Time Quantum Frequency (FTQ) is obtained from the system clock and also how the different time segments are programmed.

Figure 10-1: CAN Bit Timing



10.1 Bit Segments

Each bit transmission time consists of four time segments:

- **Synchronization Segment** – This time segment synchronizes the different nodes connected on the CAN bus. A bit edge is expected to be within this segment. Based on CAN protocol, the Synchronization Segment is assumed to be 1 TQ.
- **Propagation Segment** – This time segment compensates for any time delay that may occur due to the bus line or due to the various transceivers connected on that bus.
- **Phase Segment 1** – This time segment compensates for errors that may occur due to phase shift in the edges. The time segment may be lengthened during resynchronization to compensate for the phase shift.
- **Phase Segment 2** – This time segment compensates for errors that may occur due to phase shift in the edges. The time segment may be shortened during resynchronization to compensate for the phase shift. The Phase Segment 2 time can be configured to be either programmable or specified by the Phase Segment 1 time.

10.2 Sample Point

The sample point is the point in a CAN bit time interval where the sample is taken and the bus state is read and interpreted. It is situated between Phase Segment 1 and Phase Segment 2. The CAN bus can be sampled once or three times at the sample point, as configured by the Sample CAN Bus Line bit (SAM) in the CAN Baud Rate Configuration register (CxCFG2[6]).

- If CxCFG2[6] = `1`, the CAN bus is sampled three times at the sample point. The most common of the three samples determines the bit value.
- If CxCFG2[6] = `0`, the CAN bus is sampled only once at the sample point.

10.3 Synchronization

Two types of synchronization are used: Hard Synchronization and Resynchronization. A Hard Synchronization occurs once at the SOF. Resynchronization occurs inside a frame.

- **Hard Synchronization** takes place on the recessive-to-dominant transition of the Start bit. The bit time is restarted from that edge.
- **Resynchronization** takes place when a bit edge does not occur within the Synchronization Segment in a message. One of the Phase Segments is shortened or lengthened by an amount that depends on the phase error in the signal. The maximum amount that can be used is determined by the Synchronization Jump Width (SJW) parameter (CxCFG1[7:6]).

The length of Phase Segment 1 and Phase Segment 2 can be changed depending on oscillator tolerances of the transmitting and receiving node. Resynchronization compensates for any phase shifts that may occur due to the different oscillators used by the transmitting and receiving nodes.

- **Bit Lengthening** – If the transmitting node in CAN has a slower oscillator than the receiving node, the next falling edge, and therefore, the sample point can be delayed by lengthening Phase Segment 1 in the bit time.
- **Bit Shortening** – If the transmitting node in CAN has a faster oscillator than the receiving node, the next falling edge, and therefore, the sample point of the next bit can be reduced by shortening Phase Segment 2 in the bit time.
- **Synchronization Jump Width (SJW)** – The SJW[1:0] bits (CxCFG1[7:6]) determine the synchronization jump width by limiting the amount of lengthening or shortening that can be applied to Phase Segment 1 and Phase Segment 2 time intervals. This segment should not be longer than Phase Segment 2 time. The width can be 1 TQ-4 TQ.

10.4 CAN Bit Time Calculations

The steps that must be performed by the user application to configure the bit timing for the CAN module are described below along with examples. It is assumed that the CANCKS bit (CxCTRL[11]) is cleared resulting in FCAN = FP.

10.4.1 STEP 1: CALCULATE THE TIME QUANTUM FREQUENCY (FTQ)

- Select the Baud Rate (FBAUD) for the CAN bus.
- Select the number of Time Quanta in a bit time, based on your system requirements.

Equation 10-1 shows the formula for computing FTQ.

Equation 10-1: Time Quantum Frequency (FTQ)

$$F_{TQ} = N \times F_{BAUD}$$

Note 1: The total number of Time Quanta in a nominal bit time must be programmed between 8 TQ and 25 TQ. Therefore, the FTQ is between 8 to 25 times the baud rate (FBAUD).

2: Ensure that FTQ is an integer multiple of FBAUD to get the precise bit time. Otherwise, the oscillator input frequency or the FBAUD may need to be changed.

10.4.2 STEP 2: CALCULATE THE BAUD RATE PRESCALER (BRP[5:0] (CXCFG1[5:0]))

Equation 10-2 shows the formula for computing the baud rate prescaler.

Equation 10-2: Baud Rate Prescaler

$$BRP<5:0>\ (CiCFG1<5:0>) = \frac{F_{CAN}}{(2 \times F_{TQ})} - 1$$

10.4.2.1 STEP 3: SELECT THE INDIVIDUAL BIT TIME SEGMENTS

The individual Bit Time Segments are selected using the CxCFG2 register:

Bit Time = Sync Segment + Propagation Segment + Phase Segment 1 + Phase Segment 2

Note 1: (Propagation Segment + Phase Segment 1) must be greater than or equal to the length of Phase Segment 2.

2: Phase Segment 2 must be greater than the Synchronous Jump Width.

Example 10-1 shows the procedure to calculate the F_{TQ} to obtain a CAN bus speed of 1 Mbps when the dsPIC33E/PIC24E device is operating at 40 MHz.

Example 10-1: CAN Bit Timing Calculation Example

Step 1: Calculate the F_{TQ}.

If F_{BAUD} = 1 Mbps and the number of Time Quanta ‘N’ = 20, then F_{TQ} = 20 MHz

Step 2: Calculate the baud rate prescaler.

BRP[5:0] (CxCFG1[5:0]) = [40000000/(2 * F_{TQ})] – 1 = 1 – 1 = 0

Step 3: Select the individual Bit Time Segments.

Synchronization Segment = 1 TQ (constant)

Based on system characteristics, if the Propagation Delay = 5 TQ and if the sample point is to be at 70% of Nominal Bit Time, then:

Phase Segment 2 = 30% of Nominal Bit Time = 6 TQ

Phase Segment 1 = 20 TQ – (1 TQ + 5 TQ + 6 TQ) = 8 TQ

A code example for configuring the CAN bit timing parameters is shown in Example 10-2.

Example 10-2: Code Example for Configuring CAN Bit Timing Parameters

```
/*  This code snippet shows an example of configuring the CAN module to
    operate at 1 Mbps while the device is operating at 40 MHz */

/* Set the Operating Frequency of the device to be 40 MHz */

#define FP 40000000

/*  Set the CAN module for Configuration Mode before writing into the Baud
    Rate Control Registers */

C1CTRL1bits.REQOP = 4;

/* Wait for the CAN module to enter into Configuration Mode */

while(C1CTRL1bits.OPMODE! = 4);

/* FCAN is selected to be FP by clearing the CANCKS bit
/* FCAN = FP = 40 MHz */

C1CTRL1bits.CANCKS = 0x0;

/* Phase Segment 1 time is 8 TQ */
C1CFG2bits.SEG1PH = 0x7;

/* Phase Segment 2 time is set to be programmable */

C1CFG2bits.SEG2PHTS = 0x1;

/* Phase Segment 2 time is 6 TQ */

C1CFG2bits.SEG2PH = 0x5;

/* Propagation Segment time is 5 TQ */

C1CFG2bits.PRSEG = 0x4;

/* Bus line is sampled three times at the sample point */

C1CFG2bits.SAM = 0x1;

/* Synchronization Jump Width set to 4 TQ */

C1CFG1bits.SJW = 0x3;

/* Baud Rate Prescaler bits set to 1:1, (i.e., TQ = (2*1*1)/FCAN) */

C1CFG1bits.BRP = 0x0;

/* Put the CAN Module into Normal Operating Mode */

C1CTRL1bits.REQOP = 0;

/* Wait for the CAN module to enter into Normal Operating Mode */

while(C1CTRL1bits.OPMODE! = 0);
```

11.0 CAN ERROR MANAGEMENT

11.1 CAN Bus Errors

The CAN specification defines five different ways of detecting errors:

- Bit Error
- Acknowledge Error
- Form Error
- Stuffing Error
- CRC Error

The bit error and the Acknowledge error occur at the bit level; the other three errors occur at the message level.

11.1.1 BIT ERROR

A node that is sending a bit on the bus also monitors the bus. A bit error is detected when the bit value that is monitored is different from the bit value that is sent. An exception is when a recessive bit is sent during the stuffed bit stream of the Arbitration field or during the ACK slot. In this case, no bit error occurs when a dominant bit is monitored. A transmitter sending a passive error frame and detecting a dominant bit does not interpret this as a bit error.

11.1.2 ACKNOWLEDGE ERROR

In the Acknowledge field of a message, the transmitter checks if the Acknowledge slot (which it has sent out as a recessive bit) contains a dominant bit. If not, this implies that no other node has received the frame correctly. An Acknowledge error has occurred and as a result, the message must be repeated.

11.1.3 FORM ERROR

A form error is detected when a fixed form bit field (EOF, Inter-frame Space, Acknowledge Delimiter or CRC Delimiter) contains one or more illegal bits. For a receiver, a dominant bit during the last bit of EOF is not treated as a form error.

11.1.4 STUFFING ERROR

A stuffing error is detected at the bit time of the sixth consecutive equal bit level in a message field that should be coded by the method of bit stuffing.

11.1.5 CRC ERROR

The node transmitting a message computes and transmits the CRC corresponding to the transmitted message. Every receiver on the bus performs the same CRC calculation as the transmitter. A CRC error is detected if the calculated result is not the same as the CRC value obtained from the received message.

11.2 Fault Confinement

Every CAN controller on a bus tries to detect the errors outlined above within each message. If an error is found, the discovering node transmits an error frame, thus destroying the bus traffic. The other nodes detect the error caused by the error frame (if they have not already detected the original error) and take appropriate action (that is, discard the current message).

The CAN module maintains two error counters:

- Transmit Error Counter (CxEC[15:8])
- Receive Error Counter (CxEC[7:0])

There are several rules governing how these counters are incremented and/or decremented. That is, a transmitter detecting a Fault increments its transmit error counter faster than the listening nodes will increment their receive error counter. This is because there is a good chance that it is the transmitter that is at Fault.

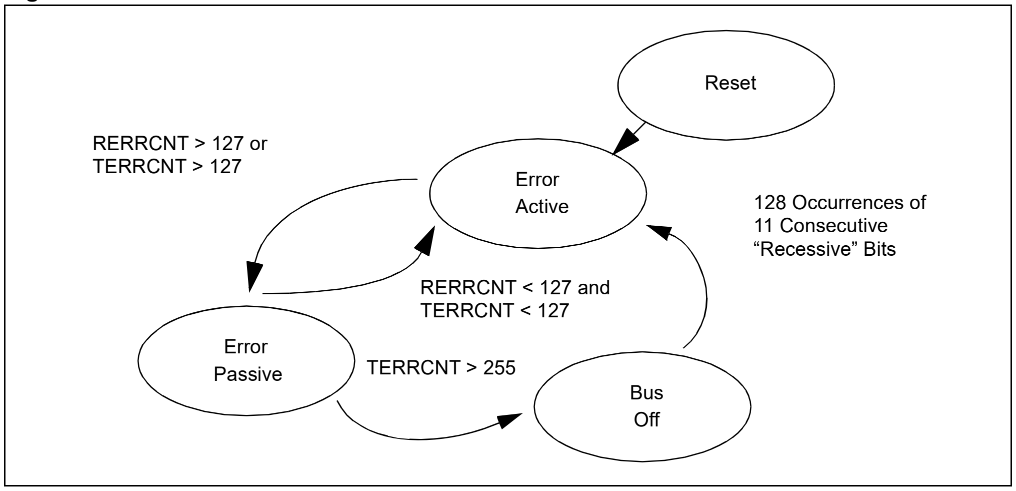
Note:	The error counters are modified according to the CAN Specification 2.0B.

A node starts out in the Error Active mode. When any one of the two error counters equals or exceeds a value of 127, the node enters a state known as Error Passive. When the transmit error counter exceeds a value of 255, the node enters the Bus Off state.

- An Error Active node transmits an Active Error frame when it detects errors
- An Error Passive node transmits a Passive Error frame when it detects errors
- A node that is in the Bus Off state transmits nothing on the bus

In addition, the CAN module employs an error warning feature that warns the user application (when the transmit error counter equals or exceeds 96) before the node enters the Error Passive state, as illustrated in Figure 11-1.

Figure 11-1: Error Modes



11.2.1 TRANSMITTER IN ERROR PASSIVE STATE

The Transmitter in Error State Bus Passive bit, TXBP (CxINTF[12]), is set when the transmit error counter equals or exceeds 128 and generates an error interrupt (CxINTF[5]) upon entry into the Error Passive state. The Transmit Error Passive flag is cleared automatically by the hardware if the transmit error counter becomes less than 128 or greater than 255.

11.2.2 RECEIVER IN ERROR PASSIVE STATE

The Receiver in Error Bus Passive bit, RXBP (CxINTF[11]), is set when the receive error counter equals or exceeds 128 and generates an error interrupt (CxINTF[5]) upon entry into the Error Passive state. The Receive Error Passive flag is cleared automatically by the hardware if the receive error counter becomes less than 128 or greater than 255.

11.2.3 TRANSMITTER IN BUS OFF STATE

The Transmitter in Error State Bus Off bit, TXBO (CxINTF[13]), is set when the transmit error counter equals or exceeds 256 and generates an error interrupt (CxINTF[5])

11.2.4 TRANSMITTER IN ERROR WARNING STATE

The Transmitter in Error State Warning bit, TXWAR (CxINTF[10]), is set when the transmit error counter is in the range of 96 and 127 (inclusive), and generates an error interrupt (CxINTF[5]) upon entry into the Error Warning state. The Transmit in Error State Warning flag is cleared automatically by the hardware if the transmit error counter becomes less than 96 or greater than 127.

11.2.5 RECEIVER IN ERROR WARNING STATE

The Receiver in Error State Warning bit, RXWAR (CxINTF[9]), is set when the receive error counter is in the range of 96 and 127 (inclusive), and generates an error interrupt (CxINTF[5]) upon entry into the Error Warning state. The Receiver in Error State Warning flag is cleared automatically by the hardware if the receive error counter becomes less than 96.

Additionally, there is an Error State Warning flag, EWARN (CxINTF[8]), which is set if at least one of the error counters equals or exceeds the error warning limit of 96. EWARN is reset if both error counters are less than the error warning limit.

12.0 CAN INTERRUPTS

The CAN module generates three different interrupts, each with its own interrupt vector, interrupt enable control bit, interrupt status flag and interrupt priority control bit. These interrupts are:

- **CxTX** – CAN Transmit Data Request
- **CxRX** – CAN Receive Data Ready
- **Cx** – CAN Event Interrupt

12.1 CAN Transmit Data Request Interrupt

The transmit data request interrupt represents the transmission of a single word in a CAN message through the CAN Transmit Data register (CxTXD). The user application needs to assign the CAN transmit data request interrupt to a DMA channel to automatically transfer messages from the appropriate device RAM buffers to the CAN module (CxTXD register).

12.2 CAN Receive Data Ready Interrupt

The receive data request interrupt represents the reception of a single word of a CAN message through the CAN Receive Data register (CxRXD). The user application needs to assign the CAN receive data ready interrupt to a DMA channel to automatically transfer messages from the CAN module (CxRXD register) to the appropriate device RAM buffers.

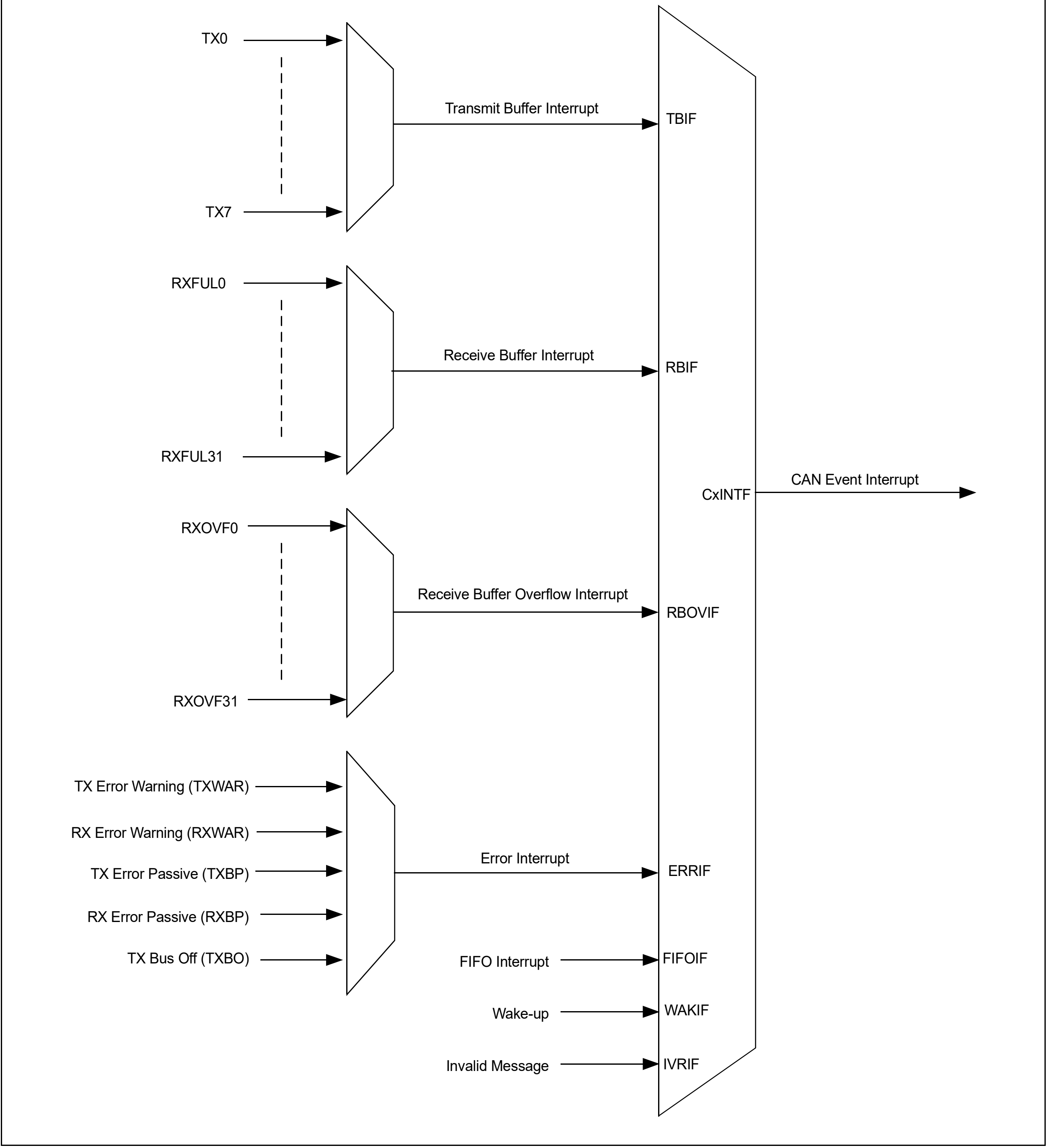
12.3 CAN Event Interrupt

The CAN event interrupt has seven main sources, each of which can be individually enabled. The CAN Interrupt Flag register (CxINTF) contains the interrupt flags, and the CAN Interrupt Enable register (CxINTE) contains the enable bits. The Interrupt Flag Code bits (ICODE[6:0]) in the CAN Interrupt Code register (CxVEC[6:0]) can be used in combination with a jump table for efficient handling of interrupts. All interrupts have one source, with the exception of the error interrupt. Any of five error interrupt sources (TX Error Warning, RX Error Warning, TX Error Passive, RX Error Passive and TX Bus Off) can set an Error Interrupt Flag (ERRIF). The source of the error interrupt is determined by reading the CxINTF register. To clear the interrupt, the ERRIF flag must be cleared. Clearing the ERRIF flag will not affect the status of the TX Error Warning, RX Error Warning, TX Error Passive, RX Error Passive and TX Bus Off flags. These bits are read-only bits and cannot be cleared by software.

Note: The ICODE[6:0] bits will reflect the highest priority CAN interrupt condition that is active. For this, the interrupt should be enabled (the IE bit in the CxINTE register should be set) and the interrupt condition should be active (the IF bit in the CxINTF register should be set).

Figure 12-1 illustrates the CAN event interrupt generation from various interrupt sources.

Figure 12-1: CAN Event Interrupts



12.3.1 TRANSMIT BUFFER INTERRUPT

The Message Buffers 0 to 7 that are configured for message transmission set the Transmit Buffer Interrupt (TBIF) bit (CxINTF[0]) after the CAN message is transmitted. The ICODE[6:0] bits (CxVEC[6:0]) indicate the specific message buffer that generated the transmit buffer interrupt. The transmit buffer interrupt must be cleared in the Interrupt Service Routine (ISR) by clearing the TBIF bit.

12.3.2 RECEIVE BUFFER INTERRUPT

When a message is successfully received and loaded into one of the receive buffers (Message Buffers 0 to 31), the receive buffer interrupt (CxINTF[1]) is activated after the module sets the RXFULx bits in the CxRXFULm registers. The ICODE[6:0] bits (CxVEC[6:0]) will indicate the particular buffer that generated the interrupt. The receive buffer interrupt must be cleared in the ISR by clearing the RBIF bit.

12.3.3 RECEIVE BUFFER OVERFLOW INTERRUPT

When a message is successfully received but the designated buffer is full, the receive overflow interrupt (CxINTF[2]) is activated after the module sets the RXOVFx bits in the CxRXOVFn registers. The ICODE[6:0] bits (CxVEC[6:0]) indicate which buffer generated the interrupt. The receive buffer overflow interrupt must be cleared in the ISR by clearing the RBOVIF bit (CxINTF[2]).

12.3.3.1 FIFO ALMOST FULL INTERRUPT

When the FIFO has only one remaining available buffer, the FIFO interrupt (CxINTF[3]) is activated after the module sets the RXFULx bits in the CxRXFULn registers for the next to last available buffer. The ICODE[6:0] bits (CxVEC[6:0]) indicate the FIFO overflow condition. The FIFO almost full interrupt must be cleared in the ISR by clearing the FIFOIF bit (CxINTF[3]).

12.3.4 ERROR INTERRUPT

The error interrupt (CxINTF[5]) is generated by five sources:

- TX Error Warning
- RX Error Warning
- TX Error Passive
- RX Error Passive
- TX Bus Off

The ICODE[6:0] bits (CxVEC[6:0]) indicate the error condition. The error interrupt must be cleared in the ISR by clearing the ERRIF bit (CxINTF[5]).

12.3.5 WAKE-UP INTERRUPT

In Sleep mode, the device monitors the CAN Receive pin (CxRX) for bus activity. A wake-up (CxINTF[6]) interrupt is generated when bus activity is detected. The ICODE[6:0] bits (CxVEC[6:0]) indicate the wake-up condition. The wake-up interrupt must be cleared in the ISR by clearing the WAKIF bit (CxINTF[6]).

12.3.6 INVALID MESSAGE INTERRUPT

The invalid message/transmission interrupt is generated for any other type of errors during message reception or transmission.

13.0 CAN LOW-POWER MODES

The CAN module can respond to the CPU `PWRSAV` instruction.

13.1 Sleep Mode

A CPU `PWRSAV,1` instruction stops the crystal oscillator and shuts down all system clocks. The user application must ensure that the module is not active when the CPU goes into Sleep mode. To protect the CAN bus system from fatal consequences due to violations of the above rule, the module drives the CxTX pin into the Recessive state while in Sleep mode. The recommended procedure is to bring the module into Disable mode before the CPU `PWRSAV,1` instruction is executed.

13.2 Idle Mode

A CPU `PWRSAV,0` instruction signals the module to optionally shut down clocks. The module powers down if the CAN Stop in Idle Mode bit (CSIDL) in the CAN Control Register 1 (CxCTRL1[13]) is '`1`'. The user application must ensure that the module is not active when the CPU goes into Idle mode. To protect the CAN bus system from fatal consequences due to violations of the above rule, the module drives the CxTX pin into the Recessive state while in Sleep mode. The recommended procedure is to bring the module into Disable mode before the CPU `PWRSAV,0` instruction is executed.

13.3 Wake-up Functions

The module monitors the RX line for activity while the device is in Sleep mode. If the WAKIE bit is set, the module generates an interrupt if bus activity is detected. Due to the delays in starting up the oscillator and CPU, the message activity that caused the wake-up is lost.

After the CPU wakes up from Sleep, the CPU executes the CAN event ISR (if interrupts are enabled); however, the CAN module itself would still be disabled. The CAN bus wake-up feature only wakes when the device is in Sleep mode.

The module features a low-pass filter on the CxRX input line, which should be enabled when the module is in CPU Sleep mode. This filter protects the module from wake-up due to short glitches on the CAN bus. The filter is enabled by setting the WAKFIL bit (CxCFG2[14]).

Note: The CAN wake-up filter must be enabled for the module to wake-up from Sleep mode on detecting CAN bus activity.

14.0 CAN TIME STAMPING USING INPUT CAPTURE

The CAN module generates a signal that can be sent to a timer capture input whenever a valid frame has been accepted. This is useful for time stamping and network synchronization. Because the CAN Specification defines a frame to be valid if no errors occurred before the EOF field has been transmitted successfully, the timer signal will be generated right after the EOF. A pulse of one bit time is generated. Time stamping is enabled by the CAN Message Receive Timer Capture Event Enable (CANCAP) control bit (CxCTRL1[3]). The IC2 capture input is used for time stamping.

Note: If the CAN capture is enabled, the IC2 pin becomes unusable as a general input capture pin. In this mode, the IC2 channel derives its input signal from the C1RX or C2RX pin instead of the IC2 pin.

15.0 RELATED APPLICATION NOTES

This section lists application notes that are related to this section of the manual. These application notes may not be written specifically for the dsPIC33E/PIC24E device family, but the concepts are pertinent and could be used with modification and possible limitations. The current application notes related to the Enhanced CAN Module module are:

Title	Application Note #
No application notes at this time.	N/A

Note:	Please visit the Microchip website (www.microchip.com) for additional Application Notes and code examples for the dsPIC33E/PIC24E family of devices.

16.0 REVISION HISTORY

Revision A (November 2008)

This is the initial released version of this document.

Revision B (March 2011)

This revision incorporates the following updates:

- Updated all DMA RAM references to device RAM
- Updated all time quanta clock (FTQ) references to time quantum frequency (FTQ)
- Updated all Remote Request Frame references to Remote Frame
- Updated all dsPIC33E references to dsPIC33E/PIC24E
- Buffers:
  - Updated the bit value description for bit 1 and bit 0 in **CAN Message Buffer Word 0**
  - Updated the bit value description for bit 9 in **CAN Message Buffer Word 2**
- Figures:
  - Updated Figure 1-1
  - Updated the label Message Buffer (DMA RAM) as Message Buffer (Device RAM) in Figure 1-2
  - Updated the logical value of SRR bit from '0' to '1' in Figure 2-3
  - Removed "Figure 21-17: CAN Message Buffer Memory Usage" in **Section 9.0 "DMA Controller Configuration"**
  - Updated the figure title for Figure 7-2
  - Changed Baud Rate Prescaler input FCY to FCAN in Figure 10-1
  - Updated Figure 11-1
- Examples:
  - Updated Example 7-1 through Example 9-2 and Example 10-2
- Equations:
  - Changed FCY to FCAN in Equation 10-2
- Notes:
  - Added a note on setting the TXREQm bit in **Section 7.1 "Message Transmission Flow"**
  - Deleted the following note in **Section 7.1 "Message Transmission Flow"**: Avoid setting the TXREQ bit for a buffer that is configured for RX. Doing so can result in erroneous operation
  - Added a note in **Section 7.3.2 "Respond to a Remote Frame"**
  - Added a note on **Section 8.2.2 "Receiving Messages Into Message Buffers 0-7"**
  - Updated the entire note in Figure 8-5
  - Added a note on DeviceNet Filtering feature in **Section 8.5 "DeviceNet™ Filtering"**
  - Added a note that provides additional details on the highest priority CAN interrupt condition in **Section 12.3 "CAN Event Interrupt"**
  - Updated the note on CAN wake-up filter in **Section 13.3 "Wake-up Functions"**
- Registers:
  - A general notes was added and FCY was changed to FCAN for the BRP[5:0] bit value definitions in Register 4-1
  - Updated the bit description for bit 15-13 in Register 4-18
  - Added the CANCKS bit in Register 4-23
  - Updated the bit value description for bit 2 in Register 4-25
- Sections:
  - Updated the Message Reception key feature in **Section 1.0 "Introduction"**
  - Removed the following frame type from the list of CAN bus protocol supports four frame types, in **Section 2.0 "CAN Message Formats"**: **Interframe Space** – provides a separation between successive frames

Revision B (March 2011) (Continued)

- Updated **Section 1.2 “Message Buffers”** and **Section 1.3 “DMA Controller”**

- Sections (Continued):
  - Updated the logical value of the SRR but from ‘0’ to ‘1’ in **Section 2.2 “Extended Data Frame”**
  - Updated the remote frame exceptions in **Section 2.3 “Remote Frame”**
  - Updated **Section 2.6 “Interframe Space”**
  - Updated the register description for CiCTRL1: CAN Control Register 1 and CiCTRL2: CAN Control Register 2 in **Section 3.6 “CAN Control and Error Counter Registers”**
  - Changed the section title for **Section 7.3 “Transmitting and Responding Remote Frames”**
  - Changed the sub section title for **Section 7.3.1 “Transmit a Remote Frame”** and **Section 7.3.2 “Respond to a Remote Frame”**
  - Updated the remote frame exceptions in **Section 7.3.1 “Transmit a Remote Frame”**
  - Updated **Section 9.0 “DMA Controller Configuration”**
  - Updated the first paragraph of **Section 10.4 “CAN Bit Time Calculations”**
  - Updated the transmit error counter range in **Section 11.2.4 “Transmitter in Error Warning State”**
  - Updated the receive error counter range in **Section 11.2.5 “Receiver in Error Warning State”**
  - Updated **Section 12.3.6 “Invalid Message Interrupt”**
  - Updated the term “TXCAN” to “CiTX” in **Section 13.1 “Sleep Mode”** and **Section 13.2 “Idle Mode”**
  - Updated **Section 13.3 “Wake-up Functions”**
- Tables:
  - Added the CANCKS bit to the CAN1 Register Map When C1CTRL1.WIN = 0 or 1 (see Table 3-1)
- Updated the Family Reference Manual name to dsPIC33E/PIC24E Family Reference Manual
- Additional minor corrections such as language and formatting updates were incorporated throughout the document

Revision C (December 2011)

This revision includes the following updates:

- Throughout the document, all occurrences of FCY were updated to: FP
- The following code examples were updated:
  - Code Example for Standard Data Frame Transmission (see Example 7-1)
  - Code Example for Extended Data Frame Transmission (see Example 7-2)
  - Code Example for Transmitting Extended Remote Frame (see Example 7-3)
  - Code Example for Filtering Standard Data Frame (see Example 8-1)
  - Code Example for Filtering Extended Data Frame (see Example 8-2)
  - DMA Channel 0 Configuration for CAN1 Transmission (see Example 9-1)
  - DMA Channel 1 Configuration for CAN1 Reception (see Example 9-2)
  - Code Example for Configuring CAN Bit Timing Parameters (see Example 10-2)
- Updated **Section 9.0 “DMA Controller Configuration”**
- All register maps were updated (CAN1 was changed to CAN and the File Names were updated) (see Table 3-1 through Table 3-3)
- Table 21-8, Table 21-9, and Table 21-10 were removed
- Additional minor corrections such as language and formatting updates were incorporated throughout the document

Revision D (September 2020)

This revision includes the following updates:

- Updated **Section 7.1 "Message Transmission Flow"**, **Section 7.2 "Aborting a Transmit Message"**, **Section 8.3.1 "Receiving Messages into FIFO Area"**, **Section 11.1.2 "Acknowledge Error"** and **Section 12.3 "CAN Event Interrupt"**.
- Updated Register 4-24.
- Updated Example 7-1, Example 7-2, Example 7-3, Example 8-1, Example 8-2, Example 9-1 and Example 9-2.
- Updated Table 3-1 and added Table 7-1.
- Reorganized **Section 3.0 "Register Maps"** and **Section 4.0 "CAN Registers"**.
- Updated to current family reference manual template format.
- Changed ECAN to CAN.
- Removed ™ from CAN.
- Additional minor updates to text and formatting have been incorporated throughout the document.

NOTES:

ISBN: 978-1-5224-6737-3

Worldwide Sales and Service

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
http://www.microchip.com/support
Web Address:
www.microchip.com

Atlanta
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

Austin, TX
Tel: 512-257-3370

Boston
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Dallas
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

Detroit
Novi, MI
Tel: 248-848-4000

Houston, TX
Tel: 281-894-5983

Indianapolis
Noblesville, IN
Tel: 317-773-8323
Fax: 317-773-5453
Tel: 317-536-2380

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608
Tel: 951-273-7800

Raleigh, NC
Tel: 919-844-7510

New York, NY
Tel: 631-435-6000

San Jose, CA
Tel: 408-735-9110
Tel: 408-436-4270

Canada - Toronto
Tel: 905-695-1980
Fax: 905-695-2078

ASIA/PACIFIC

Australia - Sydney
Tel: 61-2-9868-6733

China - Beijing
Tel: 86-10-8569-7000

China - Chengdu
Tel: 86-28-8665-5511

China - Chongqing
Tel: 86-23-8980-9588

China - Dongguan
Tel: 86-769-8702-9880

China - Guangzhou
Tel: 86-20-8755-8029

China - Hangzhou
Tel: 86-571-8792-8115

China - Hong Kong SAR
Tel: 852-2943-5100

China - Nanjing
Tel: 86-25-8473-2460

China - Qingdao
Tel: 86-532-8502-7355

China - Shanghai
Tel: 86-21-3326-8000

China - Shenyang
Tel: 86-24-2334-2829

China - Shenzhen
Tel: 86-755-8864-2200

China - Suzhou
Tel: 86-186-6233-1526

China - Wuhan
Tel: 86-27-5980-5300

China - Xian
Tel: 86-29-8833-7252

China - Xiamen
Tel: 86-592-2388138

China - Zhuhai
Tel: 86-756-3210040

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-3090-4444

India - New Delhi
Tel: 91-11-4160-8631

India - Pune
Tel: 91-20-4121-0141

Japan - Osaka
Tel: 81-6-6152-7160

Japan - Tokyo
Tel: 81-3-6880- 3770

Korea - Daegu
Tel: 82-53-744-4301

Korea - Seoul
Tel: 82-2-554-7200

Malaysia - Kuala Lumpur
Tel: 60-3-7651-7906

Malaysia - Penang
Tel: 60-4-227-8870

Philippines - Manila
Tel: 63-2-634-9065

Singapore
Tel: 65-6334-8870

Taiwan - Hsin Chu
Tel: 886-3-577-8366

Taiwan - Kaohsiung
Tel: 886-7-213-7830

Taiwan - Taipei
Tel: 886-2-2508-8600

Thailand - Bangkok
Tel: 66-2-694-1351

Vietnam - Ho Chi Minh
Tel: 84-28-5448-2100

EUROPE

Austria - Wels
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

Denmark - Copenhagen
Tel: 45-4485-5910
Fax: 45-4485-2829

Finland - Espoo
Tel: 358-9-4520-820

France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Garching
Tel: 49-8931-9700

Germany - Haan
Tel: 49-2129-3766400

Germany - Heilbronn
Tel: 49-7131-72400

Germany - Karlsruhe
Tel: 49-721-625370

Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Germany - Rosenheim
Tel: 49-8031-354-560

Israel - Ra'anana
Tel: 972-9-744-7705

Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

Italy - Padova
Tel: 39-049-7625286

Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

Norway - Trondheim
Tel: 47-7288-4388

Poland - Warsaw
Tel: 48-22-3325737

Romania - Bucharest
Tel: 40-21-407-87-50

Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

Sweden - Gothenberg
Tel: 46-31-704-60-40

Sweden - Stockholm
Tel: 46-8-5090-4654

UK - Wokingham
Tel: 44-118-921-5800
Fax: 44-118-921-5820

02/28/20