

## AN52701

**Author:** Anup Mohan

**Associated Project:** No

**Associated Part Family:** CY8C3866AXI, CY8C5588AXI

**Software Version:** PSoC<sup>®</sup> Creator™

**Associated Application Notes:** None

### Application Note Abstract

This application note describes the implementation of a controller area network (CAN) bus communication between two PSoC<sup>®</sup> 3 and PSoC 5 devices. It explains how to send and receive CAN messages and handle error messages.

### Introduction

The CAN peripheral is a fully functional Controller Area Network, supporting communication baud rates up to 1 Mbps at 8 MHz. It is CAN2.0A and CAN2.0B compliant according to the ISO-11898 specification. The CAN protocol was originally designed for automotive applications with a focus on high level fault detection and recovery. This ensures high communication reliability at low cost. Because of its success in automotive applications, CAN was adopted as a standard communication protocol for motion oriented embedded control applications (CANOpen) and factory automation applications (DeviceNet).

### Types Of CAN Messages

Based on the identifier, CAN messages are classified into two types:

- Standard
- Extended

A standard CAN message has an 11 bit identifier; extended CAN messages have a 29 bit identifier. The formats for standard and extended CAN messages are shown in [Appendix A](#). CAN also supports the transmission and reception of remote transmit requests (RTR). RTR messages are sent by the requesting node to the node owning the message.

### Overview

The CAN component in PSoC Creator sets up the CAN controller to transmit or receive standard and extended messages in two ways: Basic CAN implementation and Full CAN implementation. In basic CAN implementation, the message parameters are specified through API. In full CAN implementation, the CAN message parameters are specified in the component schematic configuration itself.

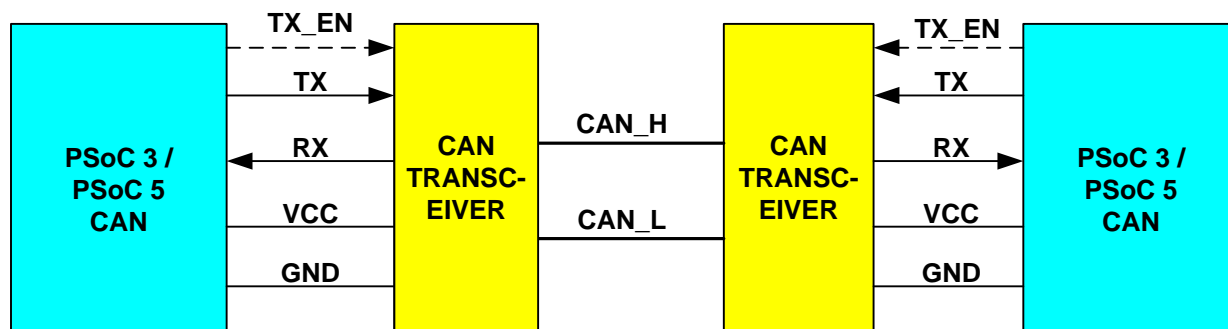
This application note explains the transmission and reception of different types of CAN messages between two PSoC 3 and PSoC 5 devices at a baud rate of 500 Ksps. It explains the following with example code:

- How to transmit and receive a CAN message with a standard identifier using the Basic CAN method.
- How to transmit CAN messages and receive messages with standard and extended identifiers using the Full CAN method.
- How to transmit, receive, and auto reply to RTR frames.

### Circuit Setup

A CAN transceiver is required to convert the Tx and Rx signals of the PSoC 3 and PSoC 5 CAN controller to CAN\_H (high) and CAN\_L (low) logic levels. The block diagram of the setup is shown in [Figure 1](#). The TX\_EN shown in dotted lines is optional. Only some CAN transceivers require an enable signal.

Figure 1. Circuit Setup

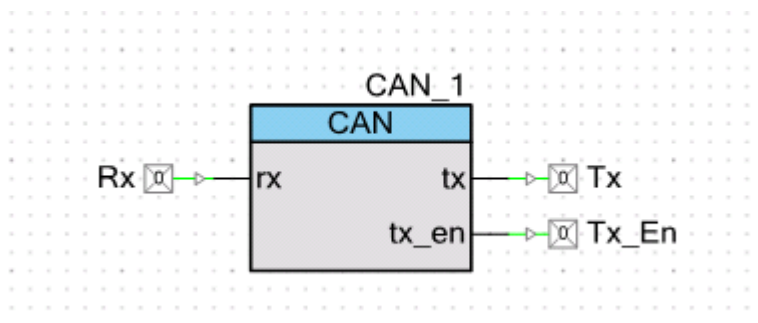


A sample diagram showing the necessary circuit connection for the CAN Transceiver is given in [Appendix B](#).

## Transmitting Basic CAN Frame

To transmit a basic CAN frame, the schematic in PSoC Creator appears as follows.

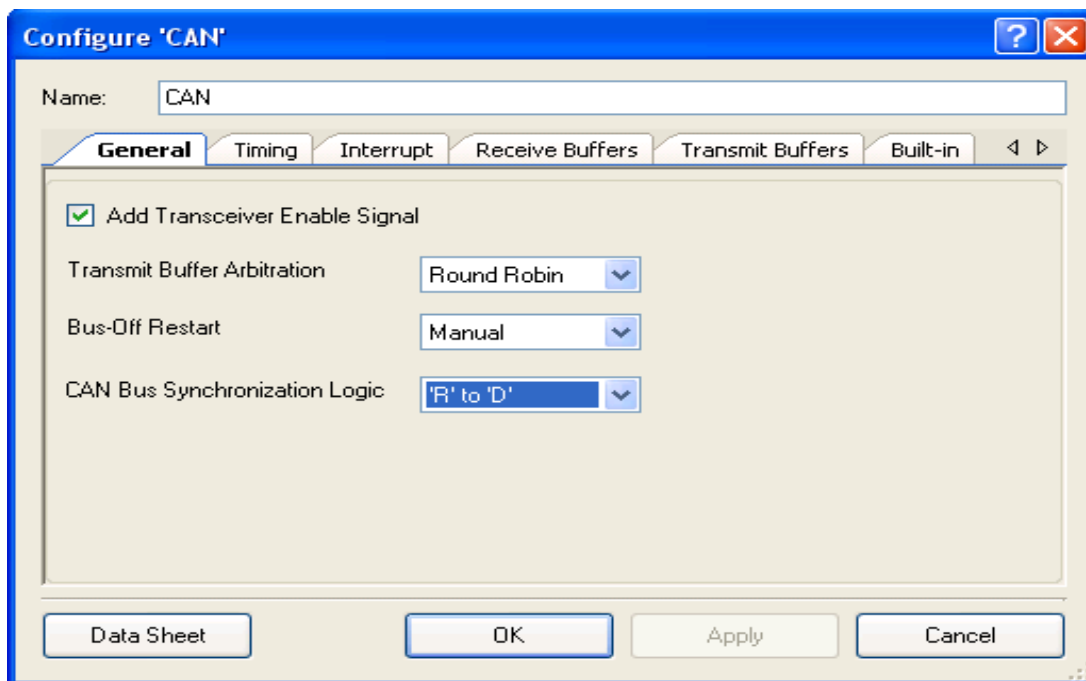
Figure 2. Basic CAN Frame Schematic



The CAN Tx and Rx lines can be routed to any digital port pin. The digital port pin is selected in the design wide resources (.cydwr) file. The transceiver enable signal is an output signal used to enable the transceiver in the circuit. It is an optional setting and can be disabled if not required. The configuration settings of the CAN component are explained in the following sections. Refer the CAN component data sheet for more details.

## General Configuration

Figure 3. General Configuration Settings



In this document, the component is named as CAN; the default name is CAN\_1.

The CAN component supports two and three wire interface to the external transceiver. The “Add Transceiver Enable Signal” checkbox is selected if the transceiver enable signal is required. The options include:

**Transmit Buffer Arbitration:** Round Robin is selected to give all buffers the same probability to transmit a message. The other setting is Fixed Priority, which is used when more than one buffer is transmitting messages and they need to be prioritized. For simple applications similar to the one explained here, the default value is chosen.

**Bus-Off Restart:** Manual is selected to restart the bus manually when Bus Off occurs. The other setting is Automatic. The default value is chosen here.

**CAN Bus Synchronization Logic:** ‘R’ (Recessive) to ‘D’ (Dominant) edge is used for bus synchronization. The other selection is to synchronize the CAN bus on Both Edges. The default value is chosen as ‘R’ to ‘D’.

## Timing Configuration

The bit time is defined as the reciprocal of number of bits transmitted on a CAN bus per second. Bit time is divided into three segments as shown in Figure 4. Each segment is represented as fixed units of time called Time Quanta (TQ), which is derived from the oscillator clock.

$$\text{BitTime} = (1 + (\text{tseg1} + 1) + (\text{tseg2} + 1)) \times \text{TQ}$$

**Tseg1, Tseg2:** These segments compensate for the edge phase shift errors. The tseg1 also takes in the propagation time which includes any delays in the network.

**Sample Point:** At this point, the state of the bus is read and the bit is interpreted. It is located at the end of tseg1. Sampling point is ideally 60 to 80% of the bit time. It can also be selected to use three sampling points, and the majority decision is used.

**Synchronization Jump Width(SJW):** By resynchronization, the tseg1 is lengthened or tseg2 is shortened by the SJW time quanta selected. SJW puts a limit to this resynchronization. The length of tseg2 must be greater than the Synchronization Jump Width.

**Bit Rate:** it is the Bit Rate Prescaler (BRP) value used to calculate the Time Quantum.

Figure 4. Bit Time Segments

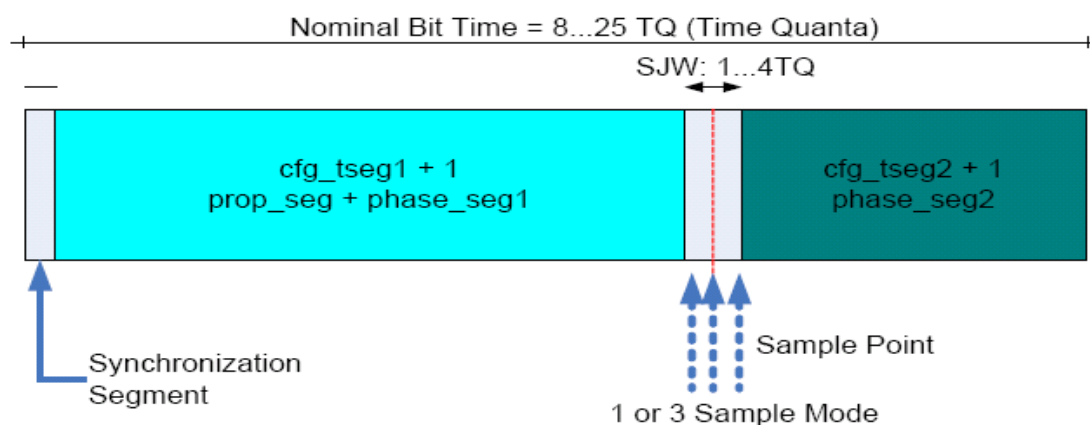
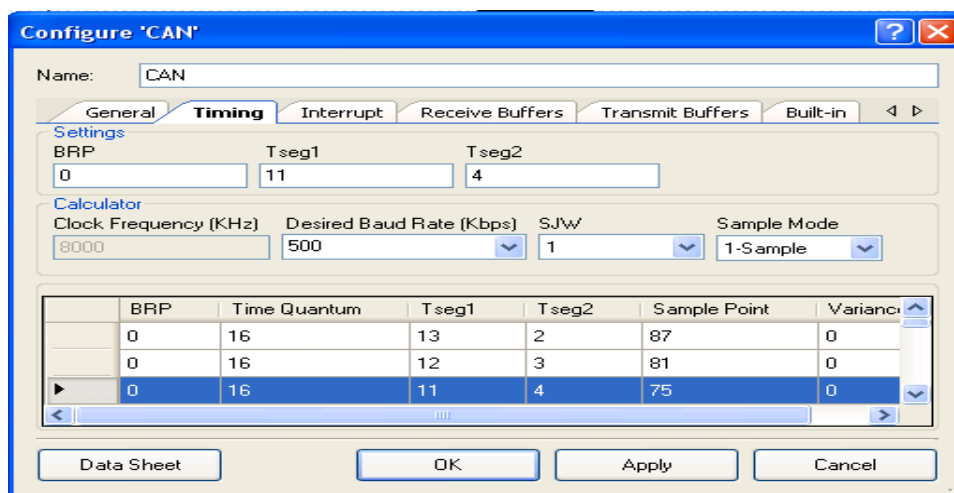


Figure 5. Timing Configuration Settings



CAN controller uses BUS\_CLK frequency. Minimum clock frequency that supports all baud rates is 8 MHz. For more details, refer the *PSoC Creator Technical Reference Manual*.

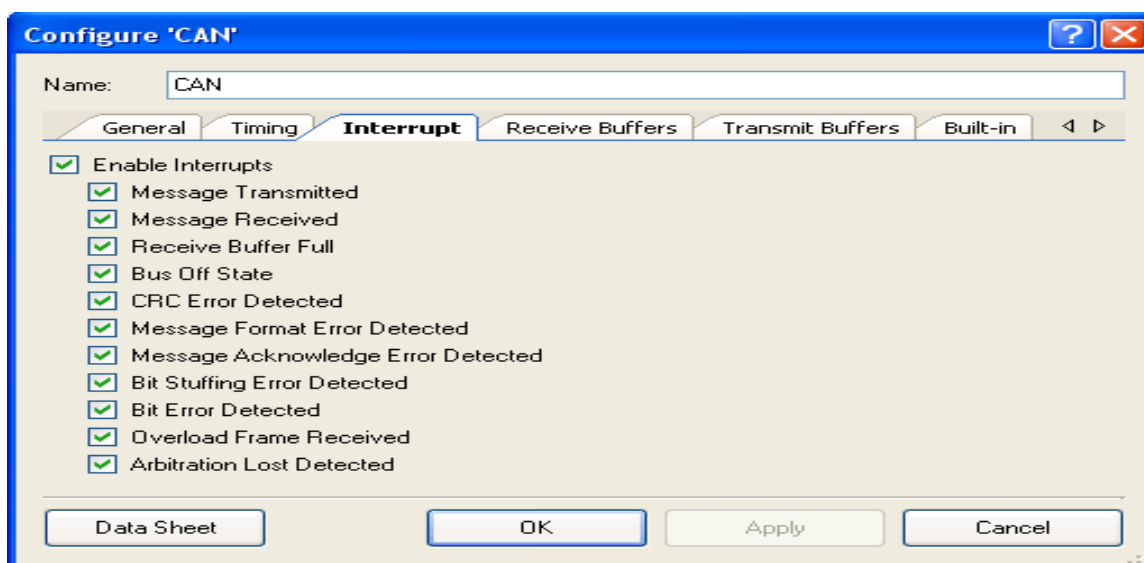
When the calculator settings of Clock Frequency, Baud Rate, SJW, and Sample Mode are done, the calculator generates suitable values of BRP, Tseg1, and Tseg2 as shown in Figure 5. Appropriate values for BRP, Tseg1, and Tseg2 are selected from the table by double clicking on the corresponding row.

## Interrupt Configuration

The Interrupt tab configurations are shown in Figure 6.

The required interrupts are enabled by selecting the corresponding checkbox.

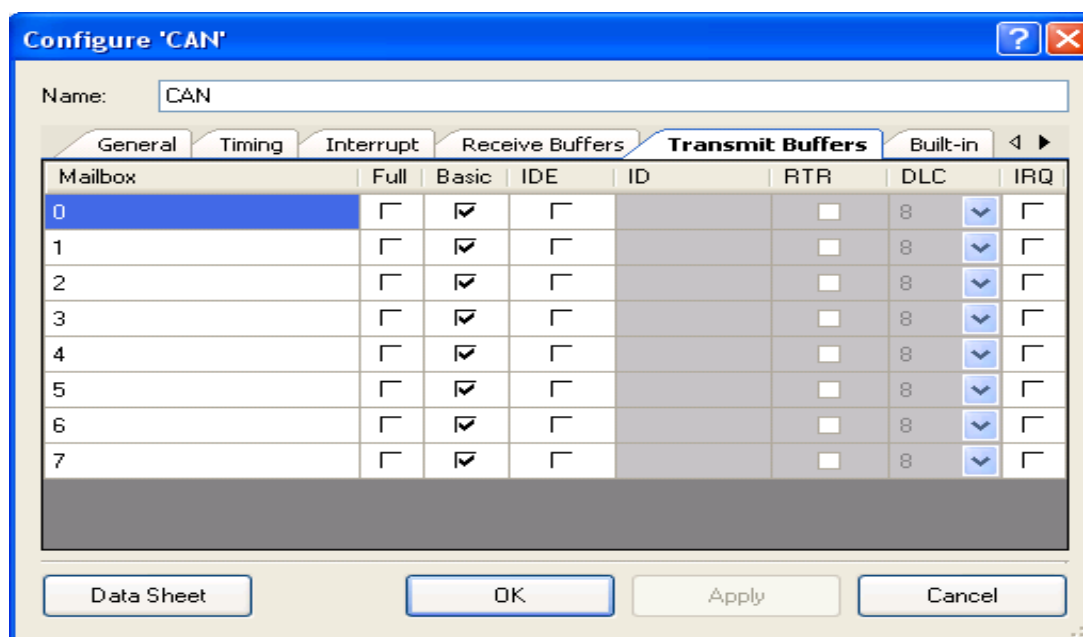
Figure 6. Interrupt Configuration Settings



## Transmit Buffer

CAN has eight transmit buffers. The configuration for the transmit buffers to transmit a basic CAN message is illustrated in Figure 7.

Figure 7. Transmit Buffer



The previous screenshot shows all the transmit buffers being configured in the “Basic” mode.

## Firmware for Sending a Basic CAN Message

To transmit basic messages in CAN standard or extended, a common API is provided in *CAN.c*.

```
CAN_SendMsg(CANTXMsg *message)
```

Where, CANTXMsg is a generic structure used to define all parameters required for a CAN message. The structure is as follows.

```
typedef struct _CANTXMsg
{
    uint32 id;
    uint8 rtr;
    uint8 ide;
    uint8 dlc;
    uint8 irq;
    DATA_BYTES_MSG *msg;
} CANTXMsg;
```

DATA\_BYTES\_MSG is a structure used to represent eight bytes of data. The structure is as follows.

```
typedef struct _DATA_BYTES_MSG
{
    uint8 byte[8];
} DATA_BYTES_MSG;
```

For a basic CAN message, the identifier, rtr bit, ide bit, dlc, interrupt request bit, and the pointer to the structure of 8 bytes that represents data, are passed through CANTXMsg structure. The structures are available in the *CAN.h* file.

The use of structures to continuously transmit a standard CAN message is explained with an example.

## Example Code

```
/* Declare Msg_Ptr as structure variable of type CANTXMsg */
CANTXMsg CAN_message;

/* Declare Data as structure variable of type DATA_BYTES_MSG */
DATA_BYTES_MSG Data;

/* Size of data to be transmitted */
#define DATALENGTH 0x08

void main()
{
    uint8 index;

    /* Define 8 bytes of data */
    uint8 CAN_data[]={0x00,0x11,0x00,0x11,0x00,0x11,0x00,0x11};

    /*
    Using Standard Identifier. CAN ID is 0x001
    */
    CAN_message.id = 0x001;
    /* RTR frame is not required */

    CAN_message.rtr = 0;
    /* Extended ID is not used */

    CAN_message.ide = 0;
    /* Specify the no:of data bytes to be sent */

    CAN_message.dlc = DATALENGTH;
    /* Enable interrupt for the CAN message */

    CAN_message.irq = 1;

    /* Write address of structure that represents data*/
    CAN_message.msg = &Data;
```

```

/* Enable global interrupts */
CYGlobalIntEnable;

/* Move data to the structure */
for(index=0;index<DATALENGTH;index++)
{
    Data.byte[index] = CAN_data[index];
}

/* Initialize and start CAN node */
CAN_Init();
CAN_Start();

while (1)
{

    /* Transmit the CAN message */
    CAN_SendMsg(&Msg_Ptr);

}
}

```

## Message Transmitted ISR

The message transmitted interrupt routine is available in the *CAN.c* file. The ISR format is as follows.

```

void CAN_MsgTXIsr(void)
{
    /* `#START MESSAGE_TRANSMITTED_ISR` */

    /* place user code here */

    /* `#END` */

    /* Clear Transtmit Message flag */
    CAN_INT_SR.byte[1] |= CAN_TX_MSG_MASK;
}

```

The same procedure is followed to transmit an extended CAN frame. Check the IDE box in the transmit buffer tab and write a value of '1' to the 'ide' in the CANTXMsg structure. The identifier specified is 29 bits long for the extended CAN message.

## Receiving Basic CAN Frame

The schematic and other component settings are similar to that of transmitting messages. The receive buffer settings are shown in Figure 8.

All sixteen receive message buffers are configured as basic CAN buffers and the corresponding interrupts are enabled.

Figure 8. Receive Buffer Settings

The screenshot shows the 'Configure CAN' dialog box with the 'Receive Buffers' tab selected. The 'Name' field is set to 'CAN'. The dialog contains a table with 16 mailboxes, each with checkboxes for 'Full', 'Basic', 'IDE', 'ID', 'RTR', 'RTRreply', 'IRQ', and 'Linking'. All 'Basic' and 'IRQ' checkboxes are checked, while 'Full', 'IDE', 'ID', 'RTR', 'RTRreply', and 'Linking' are unchecked.

Mailbox	Full	Basic	IDE	ID	RTR	RTRreply	IRQ	Linking
0	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
1	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
2	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
3	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
4	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
5	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
6	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
7	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
8	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
9	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
10	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
11	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
12	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
13	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
14	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
15	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>

Buttons at the bottom: Data Sheet, OK, Apply, Cancel.



## Firmware for Receiving a Basic CAN Message

After the message is received in a buffer, the message received interrupt is triggered if the IRQ bit of that buffer is set. The message received interrupt is selected in the Interrupt tab of the customizer. The receive message ISR (CAN\_MsgRXIsr) automatically calls the basic CAN receive message function. This ISR is available in the *CAN.c* file.

The CAN\_ReceiveMsg() in the file *CAN\_TX\_RX\_func.c* is used to retrieve the basic CAN messages. The format of the API is as shown.

```
void CAN_ReceiveMsg(uint8 rxreg)
{
    if (CAN_RX[rxreg].rxcmd.byte[0] &
    CAN_RX_ACK_MSG)
    {
        /* `#START MESSAGE_BASIC_RECEIVED` */

        /* place user code here */

        /* `#END` */
        CAN_RX[rxreg].rxcmd.byte[0] |=
        CAN_RX_ACK_MSG;
    }
}
```

The argument for the function is the mailbox number (name of the receive buffer, see [Figure 8](#) on page 8) in which the message is received and is automatically obtained in the CAN receive message ISR.

CAN\_RX is the define for the generic structure CANRXstruct from which all CAN message parameters are obtained. It is the define to the base address of the

CAN receive buffers. The CANRXstruct matches the order in which the CAN registers are available in PSoC 3 and PSoC 5. Hence, the CAN\_RX define can be used to access the CAN receive registers directly. The CANRXstruct is available in the *CAN.h* file. The format of the structure is as follows.

```
typedef struct _CANRXstruct
{
    CAN_reg32 rxcmd;
    CAN_reg32 rxid;
    DATA_BYTES rxdata;
    CAN_reg32 rxamr;
    CAN_reg32 rxacr;
    CAN_reg32 rxamrd;
    CAN_reg32 rxacrd;
} CANRXstruct;
```

Where, DATA\_BYTES is a structure that represents eight bytes of data. The structure is as follows.

```
typedef struct _DATA_BYTES
{
    reg8 byte[8];
} DATA_BYTES;
```

[Figure 9](#) shows a register map with the CAN receive registers. For more details on CAN receive registers refer the specific section for CAN in the Technical Reference Manual.

Figure 9. CAN Receive Register Map

REGISTERS															
COMMAND REGISTER (CAN.Rxn.CMD)	RSVD <31:24>	WPN2 <23>	RSVD1 <22>	RTR <21>	IDE <20>	DLC <19:16>	RSVD <15:8>	WPNL <7>	LINK FLAG <6>	Rx INT ENBL <5>	RTR REPLY <4>	BUFF ENBL <3>	RTR ABORT <2>	RTR REPLY PNDG <1>	MSGAV <0>
IDENTIFIER (CAN.Rxn.ID)	ID <31:4>														RSVD <3:0>
DATA REGISTER (CAN.Rxn.DH)	D0 <63:56>				D1 <55:48>				D2 <47:40>				D3 <39:32>		
DATA REGISTER (CAN.Rxn.DL)	D4 <31:24>				D5 <23:16>				D6 <15:8>				D7 <7:0>		
AMR REGISTER (CAN.Rxn.AMR)	ID <31:3>												IDE <2>	RTR <1>	RSVD <0>
ACR REGISTER (CAN.Rxn.ACR)	ID <31:3>												IDE <2>	RTR <1>	RSVD <0>
AMRD REGISTER (CAN.Rxn.AMRD)	RSVD <15:0>								DATA_LSB <15:0>						
ACRD REGISTER (CAN.Rxn.ACRD)	RSVD <15:0>								DATA_LSB <15:0>						

n = 0,1,...,15

## Setting Acceptance Filter

Each receive buffer has its own acceptance filter that is used to filter incoming messages. An acceptance filter consists of the Acceptance Mask register (AMR) and the Acceptance Code register (ACR). A corresponding bit in the AMR decides whether the respective incoming bit is checked against the respective bit in the ACR. If a bit in the AMR is '0', then the corresponding bit in the ACR is checked against the corresponding incoming bit. If the bit in AMR is '1', then the corresponding bit in ACR is not checked against the incoming bit. The format of AMR and ACR registers are as shown in [Figure 9](#). For standard CAN messages, the 11 bits to the extreme left of AMR and ACR are used as identifiers.

The AMR and ACR is set using the CAN\_RXRegisterInit() API available in CAN.c. The format of the API is as follows.

```
CAN_RXRegisterInit(uint32 *reg, uint32
configuration);
```

Where, \*reg is the pointer to CAN receive register and configuration is the value to be written to that register.

For example, the acceptance filter settings to receive messages with ID ranging from 0x100 to 0x1FF, IDE=0 and RTR=0 is as follows.

```
uint32 amr_value,acr_value = 0;

/* To pass messages with ID ranging from 0x100 to 0x1ff */
amr_value = (0x1fffffff9);

/* Message with ID 0x100 */
acr_value = (0x20000000);

/* Set the AMR value */
CAN_RXRegisterInit((uint32)&CAN_RX[1].rxamr, amr_value);

/* Set the ACR value */
CAN_RXRegisterInit((uint32)&CAN_RX[1].rxacr, acr_value);
```

The function has a return value of '1' if the register was written and verified, otherwise a '0'.

## Example Code

An example code showing the use of API for receiving a Standard CAN message follows.

### MAIN.C

```
#include <device.h>

/* Array to save received data */
uint8 Receivedata[8];

void main()
{
    uint32 amr_value,acr_value = 0;

    /* Enable global interrupts */
    CYGlobalIntEnable;

    /* To pass messages with ID ranging from 0x10 to 0x1f */
    amr_value = (0x1fffffff9);

    /* Message with ID 0x100 */
    acr_value = (0x20000000);

    /* Set the AMR value for RX buffer 1*/
    CAN_RXRegisterInit((uint32)&CAN_RX[1].rxamr, amr_value);

    /* Set the ACR value for RX buffer 1 */
    CAN_RXRegisterInit((uint32)&CAN_RX[1].rxacr, acr_value);

    /* Initialize CAN node */
    CAN_Init();
```

```

    /* Start CAN node */
    CAN_Start();

    /* Enable all CAN interrupts */
    CAN_GlobalIntEnable();

    for(;;)
    {
        }
    }
}

```

#### CAN\_TX\_RX\_func.c

```

#include "CAN.h"
/* `#START TX_RX_FUNCTION` */

extern uint8 receiveData[8];
uint8 index;
uint8 length;

/* `#END` */

void CAN_ReceiveMsg(uint8 rxreg)
{
    if (CAN_RX[rxreg].rxcmd.byte[0] & CAN_RX_ACK_MSG)
    {
        /* `#START MESSAGE_BASIC_RECEIVED` */

        /* To retrieve DLC of the received data*/
        length = CAN_GET_DLC(rxreg);

        for (index=0;index<length;index++)
        {
            /* Save the received data bytes to Receivedata[] array */
            receiveData[bIndex] = CAN_RX_DATA_BYTE(rxreg,index);
        }

        /* `#END` */

        CAN_RX[rxreg].rxcmd.byte[0] |= CAN_RX_ACK_MSG;
    }
}

```

To receive an extended CAN message, the same procedure is followed. For an extended CAN message, the identifier received is 29 bits long. Hence, the acceptance filter settings should be for 29 bits instead of 11 bits.

## Transmitting Full CAN Message

To transmit a full CAN message, the transmit buffer settings is as shown.

Figure 10. Transmit Buffer Settings

Mailbox	Full	Basic	IDE	ID	RTR	DLC	IRQ
0	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	0x001	<input type="checkbox"/>	1	<input checked="" type="checkbox"/>
1	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	0x012	<input type="checkbox"/>	8	<input checked="" type="checkbox"/>
2	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	0x123	<input type="checkbox"/>	8	<input checked="" type="checkbox"/>
3	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	0x00001234	<input type="checkbox"/>	4	<input checked="" type="checkbox"/>
4	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	0x00012345	<input type="checkbox"/>	8	<input checked="" type="checkbox"/>
5	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	0x00123456	<input type="checkbox"/>	6	<input checked="" type="checkbox"/>
6	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	0x01234567	<input type="checkbox"/>	8	<input checked="" type="checkbox"/>
7	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	0x12345678	<input type="checkbox"/>	8	<input checked="" type="checkbox"/>

To transmit a standard CAN message, the IDE field is left unchecked, and the 11 bit identifier is specified in the ID field as shown in the first three mailbox settings. To transmit an extended CAN message, the IDE field is checked and the 29 bit identifier is specified in the ID field. DLC is specified in the corresponding field. Values from 1 to 8 are allowed.

Other configuration settings are similar to basic CAN configuration settings.

## Firmware for Sending a Full CAN Message

Both standard and extended CAN messages can be transmitted as full CAN messages using the API `CAN_SendMsgX()`, where 'X' stands for the mailbox numbers configured as 'FULL'. You can also type a string to replace the number and the API reflects the mailbox name. The important macros available in `CAN.h` which are used to send a full CAN message are as follows.

### CAN.h

```
#define CAN_TX_DATA_BYTE(mailbox,i)  /* Macro to load data to the Full transmitt
buffer using a for loop */

/* Macros for access to TX DATA for mailbox(i) */
#define CAN_TX_DATA_BYTE1(i)         CAN_TX[i].txdata.byte[3]
#define CAN_TX_DATA_BYTE2(i)         CAN_TX[i].txdata.byte[2]
#define CAN_TX_DATA_BYTE3(i)         CAN_TX[i].txdata.byte[1]
#define CAN_TX_DATA_BYTE4(i)         CAN_TX[i].txdata.byte[0]
#define CAN_TX_DATA_BYTE5(i)         CAN_TX[i].txdata.byte[7]
#define CAN_TX_DATA_BYTE6(i)         CAN_TX[i].txdata.byte[6]
#define CAN_TX_DATA_BYTE7(i)         CAN_TX[i].txdata.byte[5]
#define CAN_TX_DATA_BYTE8(i)         CAN_TX[i].txdata.byte[4]

/* These Macros can be used to load individual bytes of data for mailbox(i) */
```

Data is specified in the file `CAN_TX_RX_func.c`. In the file, `CAN_SendMsg0 ()` to `CAN_SendMsg7 ()` are defined. When a transmit buffer is configured as Full, the corresponding API `CAN_SendMsgX()` is available to transmit the message.

The structure of CAN\_SendMsgX () is as follows.

```
uint8 CAN_SendMsg0(void)
{
    uint8 result = CYRET_SUCCESS;
    if (CAN_TX[0].txcmd.byte[0] & CAN_TX_REQUEST_PENDING)
    {
        result = FAIL;
    }
    else
    {
        /* `#START MESSAGE_0_TRANSMITTED` */

        /* `#END` */
        CY_SET_REG32((reg32 *) &CAN_TX[0].txcmd, CAN_SEND_MESSAGE);
    }

    return result;
}
```

CAN\_TX is the define for the generic structure CANTXstruct where all message parameters are specified. It is the define to the base address of the CAN transmit buffers. The CANTXstruct matches the order in which the CAN registers are available in the PSoC 3 and PSoC 5. Therefore, the CAN\_TX define is used to access the CAN transmit registers directly. The CANTXstruct is available in the *CAN.h* file. The format of the structure is as follows.

```
typedef struct _CANTXstruct
{
    CAN_reg32 txcmd;
    CAN_reg32 txid;
    DATA_BYTES txdata;
} CANTXstruct;
```

A register map with the CAN transmit registers is shown in [Figure 11](#). For more details on CAN transmit registers, refer the section on CAN in the Technical Reference Manual.

Figure 11. CAN Transmit Register Map

REGISTERS											
COMMAND REGISTER (CAN.Txn.CMD)	RSVD <31:24>	WPN2 <23>	RSVD1 <22>	RTR <21>	IDE <20>	DLC <19:16>	RSVD <15:4>	WPN1 <3>	Tx INT ENBL <2>	Tx ABORT <1>	Tx REQ <0>
IDENTIFIER (CAN.Txn.ID)	ID <31:4>									RSVD <3:0>	
DATA REGISTER (CAN.Txn.DH)	D0 <63:56>		D1 <55:48>			D2 <47:40>			D3 <39:32>		
DATA REGISTER (CAN.Txn.DL)	D4 <31:24>		D5 <23:16>			D6 <15:8>			D7 <7:0>		

n = 0,1,...,7

An example code to specify eight bytes of data using CAN\_SendMsg0 () in CAN\_TX\_RX\_func.c is given.

```
#include "CAN.h"
/* `#START TX_RX_FUNCTION` */

extern uint8 CAN_message[8];
uint8 index;

/* `#END` */

uint8 CAN_SendMsg0(void)
{
    uint8 result = CYRET_SUCCESS;
    if (CAN_TX[0].txcmd.byte[0] & CAN_TX_REQUEST_PENDING)
    {
        result = FAIL;
    }
    else
    {
        /* `#START MESSAGE_0_TRANSMITTED` */
        /* To transmit message from mailbox 0 */
        for (index=0;index<8;index++)
        {
            CAN_TX_DATA_BYTE(0,index) = CAN_message[index];
        }

        /* `#END` */
        CY_SET_REG32((reg32 *) &CAN_TX[0].txcmd, CAN_SEND_MESSAGE);
    }

    return result;
}
```

In the 'main' code, the API CAN\_SendMsgX () must be called to transmit the message from the corresponding message buffer.

## Receiving FULL CAN Frame

To receive a full can frame, the receiving buffer settings is as shown in Figure 12. The receive message buffers can be configured as 'FULL' as shown. The necessary fields like ID, IDE, and IRQ are also specified.

Figure 12. Receive Buffer Settings

Mailbox	Full	Basic	IDE	ID	RTR	RTRreply	IRQ	Linking
0	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	0x001	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
1	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	0x012	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
2	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	0x123	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
3	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	0x00001234	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
4	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	0x00012345	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
5	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	0x00123456	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
6	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	0x01234567	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
7	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	0x12345678	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
8	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
9	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
10	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

## Firmware for Receiving a Full CAN Message

The important macros available in *CAN.h*, which are used to receive a full CAN message are as follows.

### CAN.h

```
#define CAN_GET_RX_IDE(i)      /* Macro to access IDE of the received message for
mailbox(i) */
#define CAN_GET_RX_ID(i)      /* Macro to access identifier of the received message
for mailbox(i) */
#define CAN_GET_DLC(i) /* Macro to access DLC of the received message for
mailbox(i) */

#define CAN_RX_DATA_BYTE(mailbox,i) /* Macros to retrieve RX DATA for mailbox(i)
using a for loop */

/* Macros for access to RX DATA for mailbox(i) */
#define CAN_RX_DATA_BYTE1(i)      CAN_RX[i].rxdata.byte[3]
#define CAN_RX_DATA_BYTE2(i)      CAN_RX[i].rxdata.byte[2]
#define CAN_RX_DATA_BYTE3(i)      CAN_RX[i].rxdata.byte[1]
#define CAN_RX_DATA_BYTE4(i)      CAN_RX[i].rxdata.byte[0]
#define CAN_RX_DATA_BYTE5(i)      CAN_RX[i].rxdata.byte[7]
#define CAN_RX_DATA_BYTE6(i)      CAN_RX[i].rxdata.byte[6]
#define CAN_RX_DATA_BYTE7(i)      CAN_RX[i].rxdata.byte[5]
#define CAN_RX_DATA_BYTE8(i)      CAN_RX[i].rxdata.byte[4]

/* Macros for receiving and accessing individual data bytes mailbox(i) */
```

When the message is received in a buffer, the message received interrupt is triggered if the IRQ bit of that buffer is set. The receive message ISR (CAN\_MsgRXIsr) automatically calls the corresponding CAN receive message function as shown.

```
void CAN_MsgRXIsr(void)
{
...
/* RX Full mailboxes handler */
switch(i)
{
    case 0 : CAN_ReceiveMsg0();
    break;

    case 2 : CAN_ReceiveMsg2();
    break;

    .
    .
    .
    default:
    break;
}
...
}
```

CAN\_ReceiveMsgX () API in CAN\_TX\_RX\_func.c is used to retrieve data from the received message frame. 'X' corresponds to the message buffer configured as 'Full'. The structure of the API is as follows.

```
void CAN_ReceiveMsg0(void)
{
    /* `#START MESSAGE_0_RECEIVED` */

    /* `#END` */

    CAN_RX[10].rxcmd.byte[0] |= CAN_RX_ACK_MSG;
}
}
```

CAN\_RX is the define for the generic structure CANRXstruct from which all CAN message parameters are obtained. The format of the structure is given in the section, [Firmware for Receiving a Basic CAN Message](#).

To retrieve data using the API CAN\_ReceiveMsgX () in CAN\_TX\_RX\_func.c, write the appropriate code inside the function. The use of CAN\_ReceiveMsg0 () API to retrieve eight bytes of data is shown in the following example.

```
void CAN_ReceiveMsg0(void)
{
    /* `#START MESSAGE_0_RECEIVED` */
    /* Receive message on mailbox 0 */
    for(index=0;index<=8;index++)
    {
        ReceiveData[index]= CAN_RX_DATA_BYTE(0,index);
    }
    /* `#END` */
    CAN_RX[0].rxcmd.byte[0] |= CAN_RX_ACK_MSG;
}
```

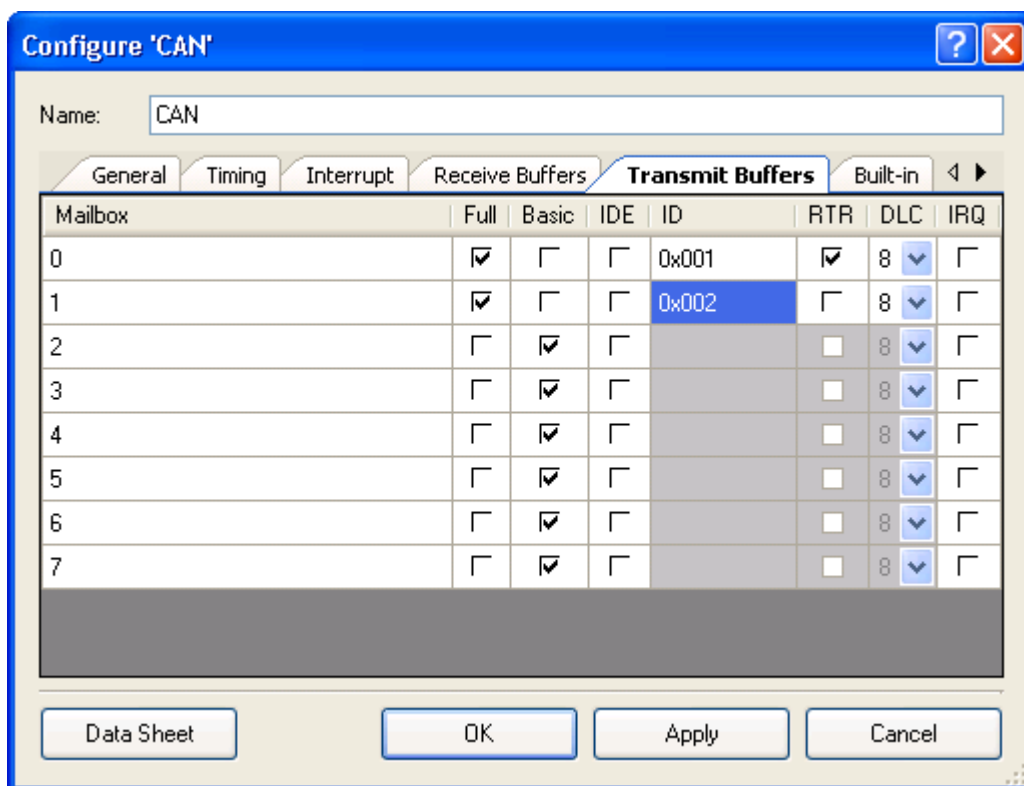
Both standard and extended messages can be received as full CAN messages. The *MAIN.C* code and the AMR and ACR settings are as explained in the [Firmware for Receiving a Basic CAN Message](#) section.



## Transmitting RTR Message

Remote frames are used to initiate transmission between two nodes and the node acting as a receiver sends the remote frame. A remote frame can use standard and extended formats. A remote frame is different from a data frame in that the RTR bit is always equal to '1' and the data field is absent. Although no data is transmitted in an RTR frame, the DLC field must match the expected data length of the response to an RTR frame. To transmit an RTR frame, the transmit buffer settings is shown in Figure 13.

Figure 13. Transmit Buffer Settings



**Configure 'CAN'**

Name:

General Timing Interrupt Receive Buffers **Transmit Buffers** Built-in

Mailbox	Full	Basic	IDE	ID	RTR	DLC	IRQ
0	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	0x001	<input checked="" type="checkbox"/>	8	<input type="checkbox"/>
1	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	0x002	<input type="checkbox"/>	8	<input type="checkbox"/>
2	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>		<input type="checkbox"/>	8	<input type="checkbox"/>
3	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>		<input type="checkbox"/>	8	<input type="checkbox"/>
4	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>		<input type="checkbox"/>	8	<input type="checkbox"/>
5	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>		<input type="checkbox"/>	8	<input type="checkbox"/>
6	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>		<input type="checkbox"/>	8	<input type="checkbox"/>
7	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>		<input type="checkbox"/>	8	<input type="checkbox"/>

Data Sheet OK Apply Cancel

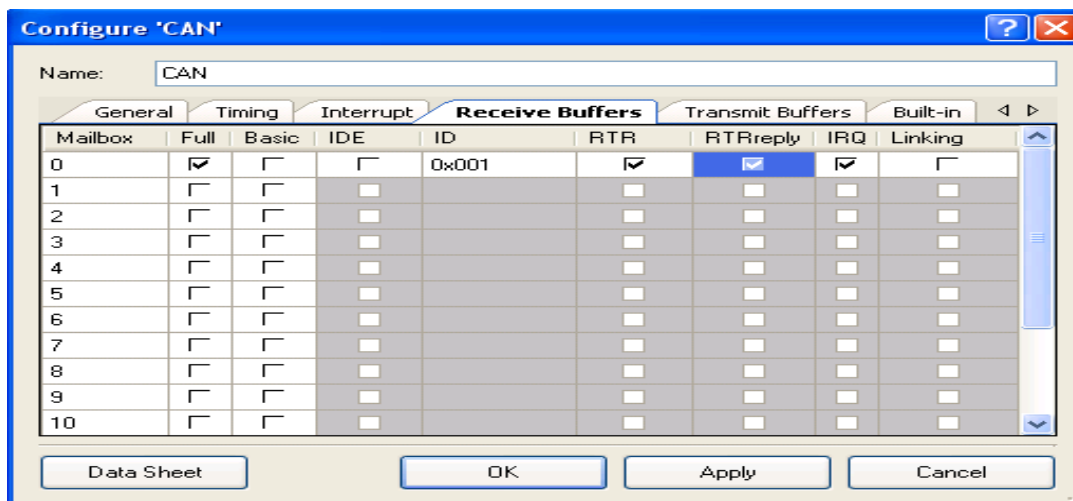
As seen in the figure, for mailbox 0, the RTR bit is selected indicating that it is an RTR message. Mailbox 1 can be used to send normal Full CAN message.

The software code to transmit an RTR message is similar to that of transmitting full CAN message. The API `CAN_SendMsgX()` is used to transmit the RTR message. RTR messages usually do not contain data; therefore, data need not be specified in the `CAN_TX_RX_func.c` file.

## Receiving and Auto Replying an RTR Message

The CAN module supports automatic answering of RTR message requests. All sixteen receive buffers support this feature. If an RTR message is accepted in a receive buffer where the RTR REPLY option is selected, then this buffer automatically replies to this message with the content of the receive buffer.

Figure 14. Receive Buffer Settings for Auto Replying to RTR Message



The node which transmits the RTR message must receive the RTR reply message. The receive buffer settings of the node should not have RTR reply option selected. The receive buffer settings must be the same as that of receiving a full CAN message.

An example code at the node, which initiates the RTR message follows. In the code, the node transmits the RTR message and waits for the RTR reply message to send a particular message.

### MAIN.C

```
#include <device.h>

uint8 RTRreceive;

void main()
{
    /* Enable global interrupts*/
    CYGlobalIntEnable;

    /* Initialize and start CAN node*/
    CAN_Init();
    CAN_Start();

    /* Enable all CAN interrupts*/
    CAN_GlobalIntEnable();

    /* Send RTR message*/
    CAN_SendMsg0();
}
```

```
for(;;)
{
    if(RTRreceive)
    {
        /* Send another message if RTR is received*/
        CAN_SendMsg1();
        /* Clear the flag */
        RTRreceive = 0;
    }
}
```

### CAN\_TX\_RX\_func.c

```
#include "CAN.h"
/* `#START TX_RX_FUNCTION` */
extern uint8 bRTRreceive;
/* `#END` */

void CAN_ReceiveMsg0(void)
{
    /* `#START MESSAGE_0_RECEIVED` */
    /* Set the flag on RTR
    reception */
    RTRreceive = 1;
    /* `#END` */

    CAN_RX[10].rxcmd.byte[0] |=
    CAN_RX_ACK_MSG;
}
```

## Error Management

The CAN component handles five types of errors:

- Bit error
- Form error
- Stuff error
- CRC error
- Acknowledge error

A transmitter detects bit and acknowledgment errors, while a receiving node detects form, stuff, and CRC errors.

If a message fails any one of these error detection methods, it is not accepted and an error frame is generated from the receiving node. This forces the transmitting node to resend the message until it is received correctly. However, if a faulty node hangs up a bus by continuously repeating an error, its transmit capability is removed by the controller after reaching the error limit.

Each node that detects an error, increments the error counter (transmit or receive). The CAN module can be in three different states depending on the error counter values.

**Error Active State:** A node is in 'error active' state if the transmit or receive error counters are less than or equal to 127 decimal. An error active node can take part in normal bus communication.

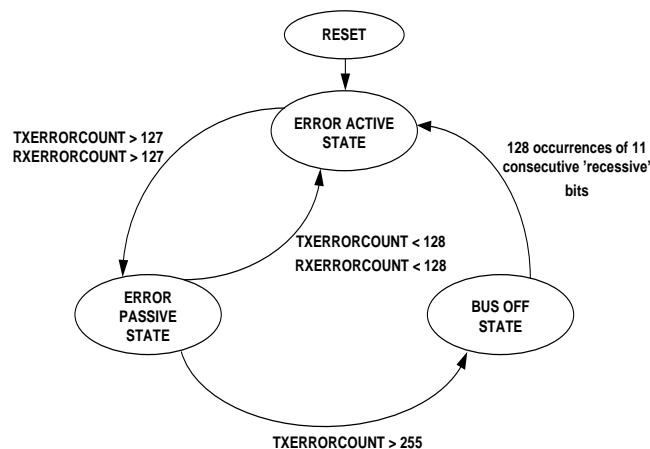
**Error Passive State:** A node is in 'error passive' state if the transmit or receive error counter value exceeds or equals 128 decimal. An error passive node takes part in bus communication.

**Bus Off State:** A node is in 'Bus Off' state if the transmit error counter exceeds or equals the value of 256 decimal. A node that is in Bus Off does not take part in any bus communication. It has no effect on the bus.

The restart from Bus Off state can be made manual or automatic as explained in the [General Configuration](#) section. When automatic is selected, the bus becomes active after 128 occurrences of 11 consecutive 'recessive' bits are monitored on the bus.

The error states in CAN are explained using a state diagram as shown in [Figure 15](#).

Figure 15. Error States



Each error causes the generation of an interrupt if the corresponding interrupt is enabled in the Interrupt Configuration tab.

For example, the bit error ISR in CAN.c is as follows:

```

void CAN_BitErrorIsr(void)
{
    /* `#START BIT_ERROR_ISR` */

    /* `#END` */

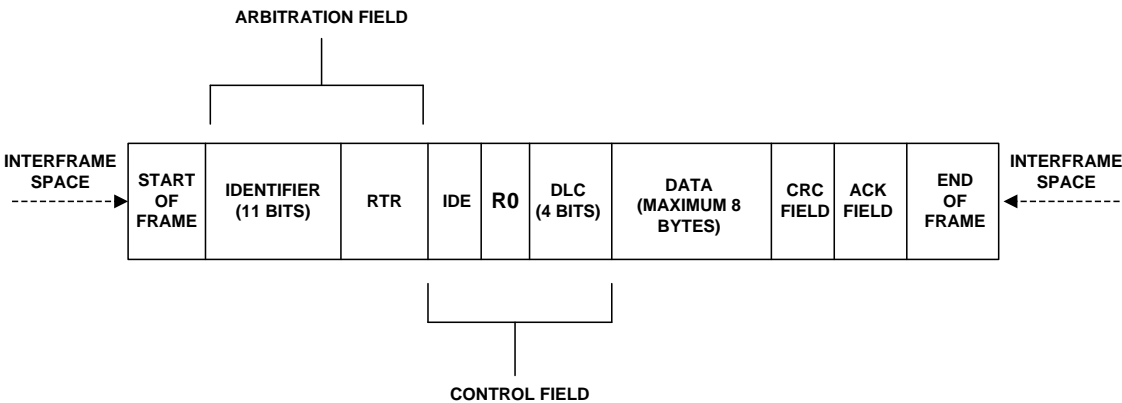
    /* Clear Bit Error flag */
    CAN_INT_SR.byte[0] |=
    CAN_BIT_ERROR_MASK;
}
  
```

## Summary

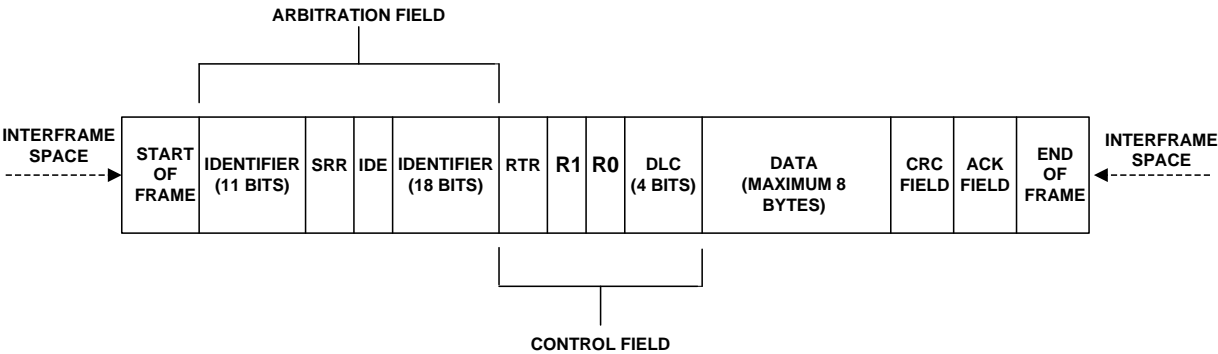
PSoc's flexibility combined with PSoc Creator™ IDE enables easy implementation of efficient CAN bus communication. This application note demonstrates how to transmit and receive CAN messages between two PSoc 3 and PSoc 5 devices.

# Appendix A

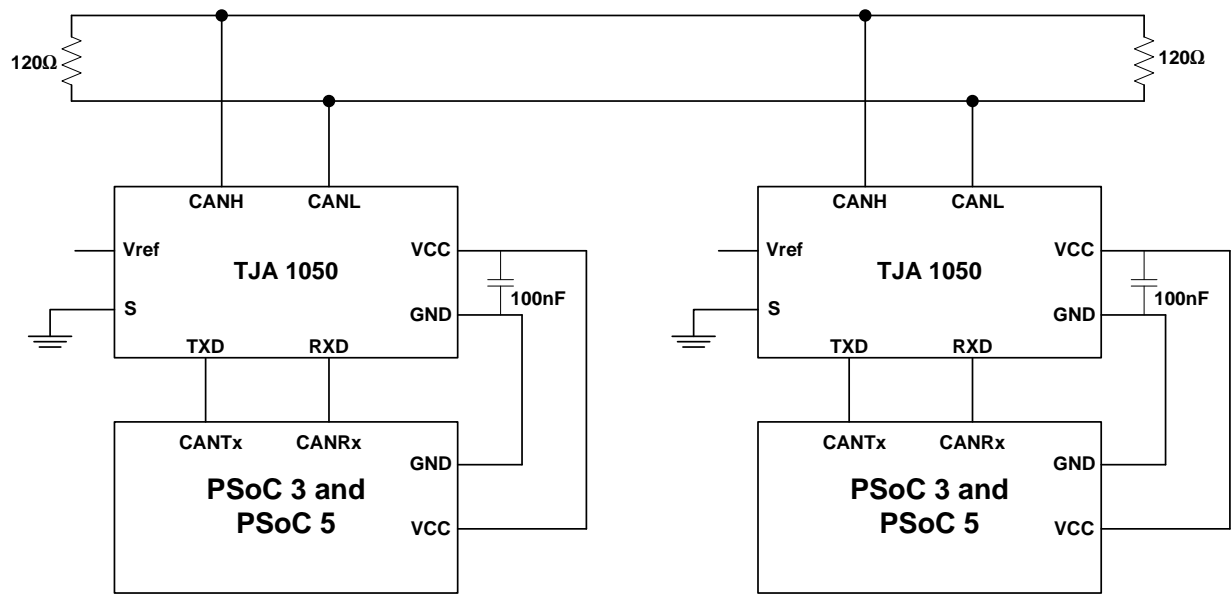
## STANDARD DATA FRAME



## EXTENDED DATA FRAME



# Appendix B



## Document History

**Document Title:** Implementing CAN Bus Communication using PSoC® 3 and PSoC 5

**Document Number:** 001-52701

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	2710279	ANUP	05/22/09	New application note
*A	2763879	ANUP	09/15/09	Updated Figure 2: Basic CAN Frame Schematic
*B	2947435	ANUP	06/08/10	Changed document title. Updated to PSoC Creator Beta 4.1 and made the projects PSoC 5 compatible

PSoC is a registered trademark and PSoC Designer is a trademark of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are the property of their respective owners.

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

© Cypress Semiconductor Corporation, 2009-2010. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

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

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

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