AN52701

PSoC® 3 – How to Enable CAN Bus Communication

Author: Anup Mohan
Associated Project: Yes
Associated Part Family: CY8C3866AXI
Software Version: PSoC® Creator™
Related Application Notes: None

Abstract

The controller area network (CAN) peripheral supports communication baud rates up to 1 Mbps at 8 MHz. It conforms to CAN2.0A and CAN2.0B according to the ISO-11898 specification. This application note describes how to use CAN bus communication between two PSoC® 3 devices, how to send and receive CAN messages, and how to manage error messages.

Contents

Introduction 1
Types Of CAN Messages 1
Overview 1
Circuit Setup 2
To Send a Basic CAN Frame 2
- General Configuration 3
- Timing Configuration 3
- Interrupt Configuration 4
- Transmit Buffer 5

Firmware to Send a Basic CAN Message 6
- Example Code 6
- Message Transmitted ISR 6

Receive Basic CAN Frame 7
Firmware Used to Receive a Basic CAN Message 7
Set the Acceptance Filter 8
- Example Code 9

Send a Full CAN Message 9
Firmware Used to Send a Full CAN Message 10
Receive a FULL CAN Frame 12
Firmware to Receive a Full CAN Message 12
Send RTR Message 13
- Receive and Automatically Send a Reply to an RTR Message 14

Error Management 15
Code Examples 15
CAN Clock Accuracy 16
Summary 17
Appendix A 18

Introduction

The CAN protocol was originally created for automotive applications with a focus on high-level faults detection and recovery. This ensures high communication reliability at low cost. Because of its success in automotive applications, CAN is now a standard communication protocol for motion - oriented, embedded control applications (CANOpen) and factory automation applications (DeviceNet).

Types Of CAN Messages

Based on the identifier, there are two types of CAN messages:

- Standard
- Extended

A standard CAN message has an 11-bit identifier. An extended CAN message 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 transmits requests (RTR). RTR messages are sent by the requesting node to the node that owns the message.

Overview

The CAN component in PSoC Creator sets up the CAN controller to send 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 an API. In

Full CAN implementation, the CAN message parameters are specified in the component configuration window schematic.

This document describes the transmission and reception of different types of CAN messages between two PSoC 3 devices at a baud rate of 500 ksps. This application note focuses on the following:

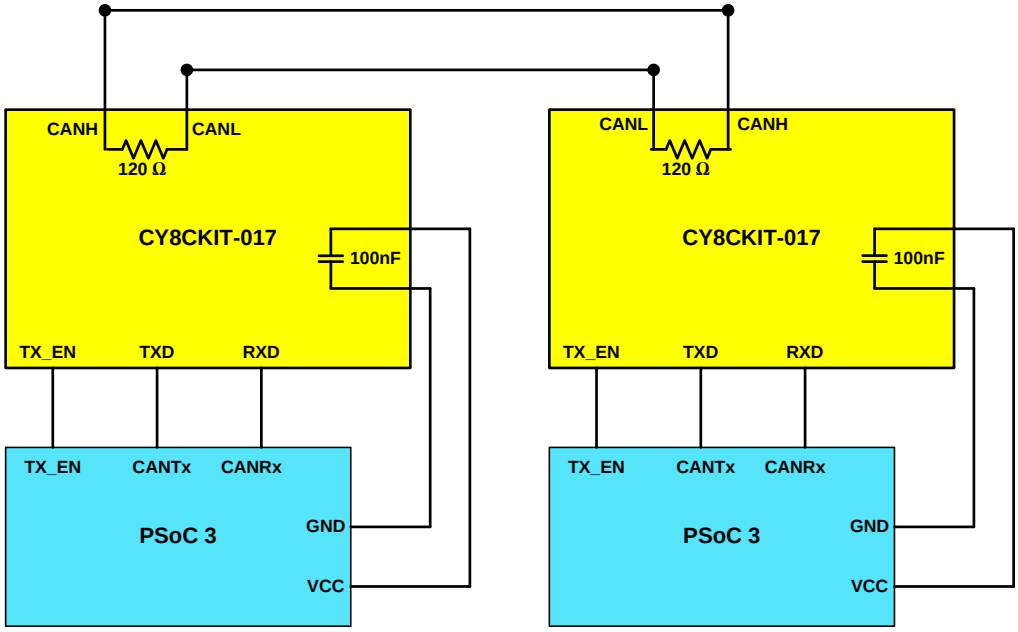
- How to use the Basic CAN method to send and receive CAN messages with a standard identifier.
- How to use the Full CAN method to send and receive CAN messages with standard and extended identifiers.
- How to send, receive, and automatically send replies to RTR frames.

Circuit Setup

A CAN transceiver must be used to convert the Tx and Rx signals of the PSoC 3 CAN controller to CAN_H (high) and CAN_L (low) logic levels. The CY8CKIT-017 CAN Expansion Board Kit includes a CAN transceiver circuit. Figure 1 shows a sample diagram with the necessary circuit connection for the CAN Transceiver.

Only some CAN transceivers have an enable signal (TX_EN).

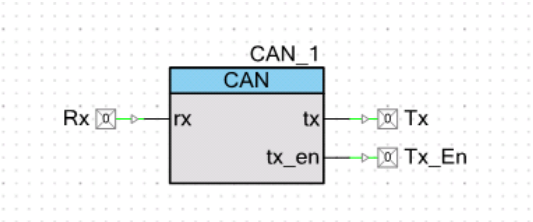
Figure 1. Circuit Setup



To Send a Basic CAN Frame

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

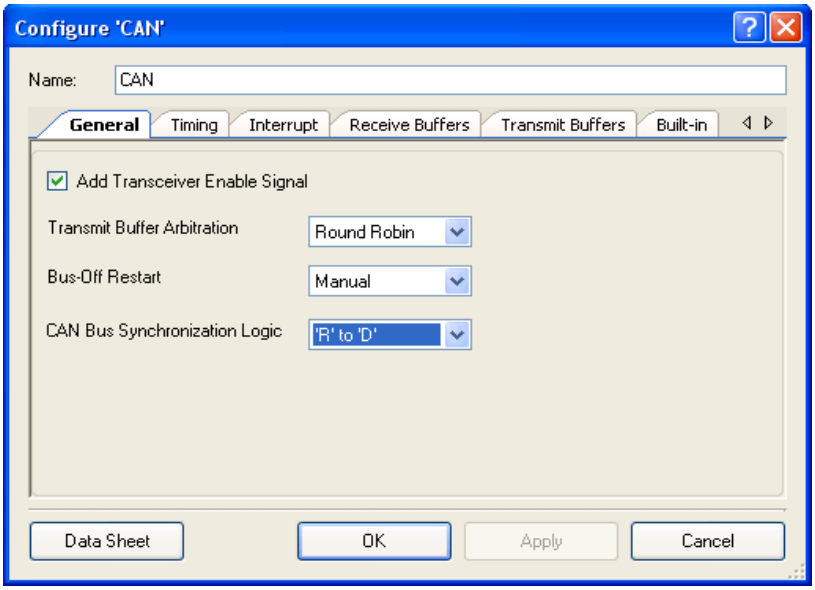
Figure 2. Basic CAN Frame Schematic



The CAN Tx and Rx lines can be sent 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 that enables 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 described in the following sections. For more information, see the CAN component datasheet.

General Configuration

Figure 3. General Configuration Settings



In this document, the component has the name CAN; the default name is CAN_1.

The CAN component supports two-wire and three-wire interfaces 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 send a message. The other setting is Fixed Priority, which is used when more than one buffer is sending messages and they must be given a priority. For simple applications similar to the one described here, the default value is used.
- **Bus-Off Restart:** Manual is selected to restart the bus manually when Bus-Off occurs. The other setting is Automatic. The default value is used 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 'R' to 'D' is used.

Timing Configuration

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

BitTime* = (1+(*tseg*1+1)+(*tseg*2+1))× *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. The sample point is at the end of tseg1 and is ideally 60 percent to 80 percent of the bit time. It can also be selected to use three sampling points, and the majority decision is used.

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

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

Figure 4. Bit Time Segments

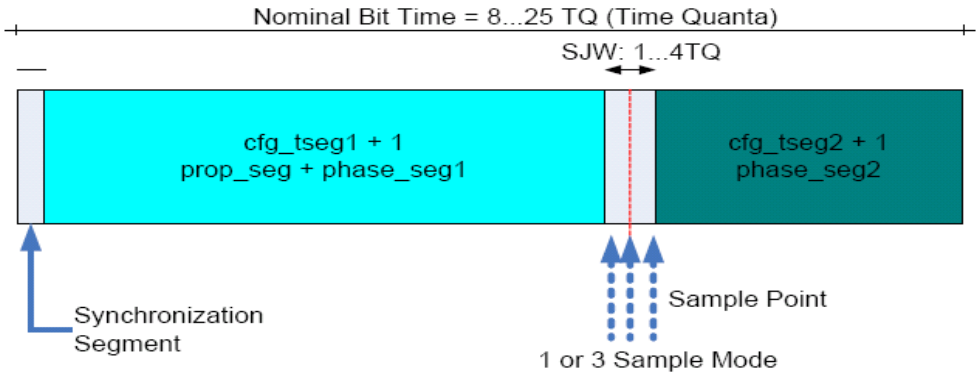
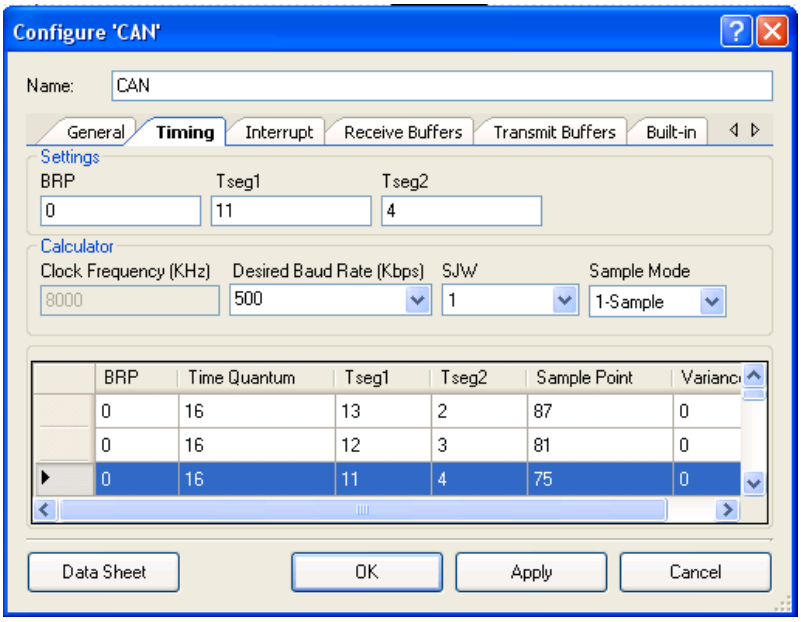


Figure 5. Timing Configuration Settings



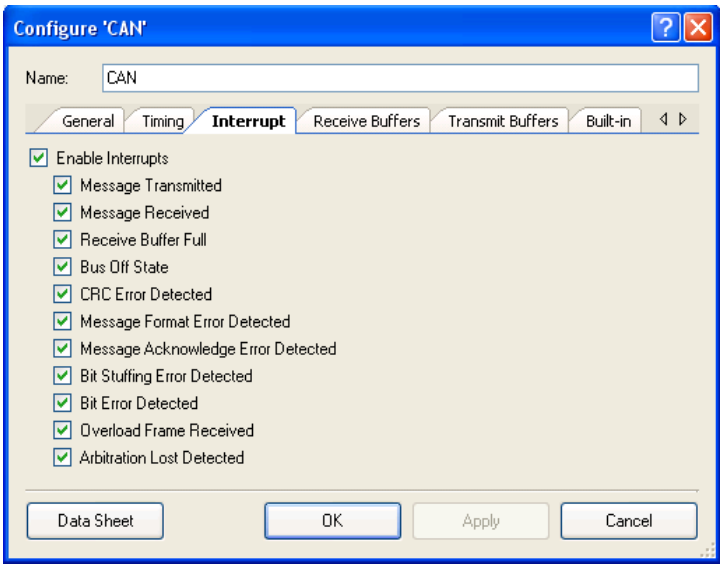
CAN controller uses BUS_CLK frequency. Minimum clock frequency that supports all baud rates is 8 MHz. For more details, see the *PSoC Creator Technical Reference Manual*.

When the calculator settings of Clock Frequency, Baud Rate, SJW, and Sample Mode are finished, the calculator generates suitable values of BRP, Tseg1, and Tseg2 as shown in Figure 5. To select appropriate values for BRP, Tseg1, and Tseg2 from the table, double-click the row with sample point in the range of 60 – 80 and variance 0.

Interrupt Configuration

Figure 6 shows the **Interrupt** tab configurations. Select the correct check box to enable the required interrupts.

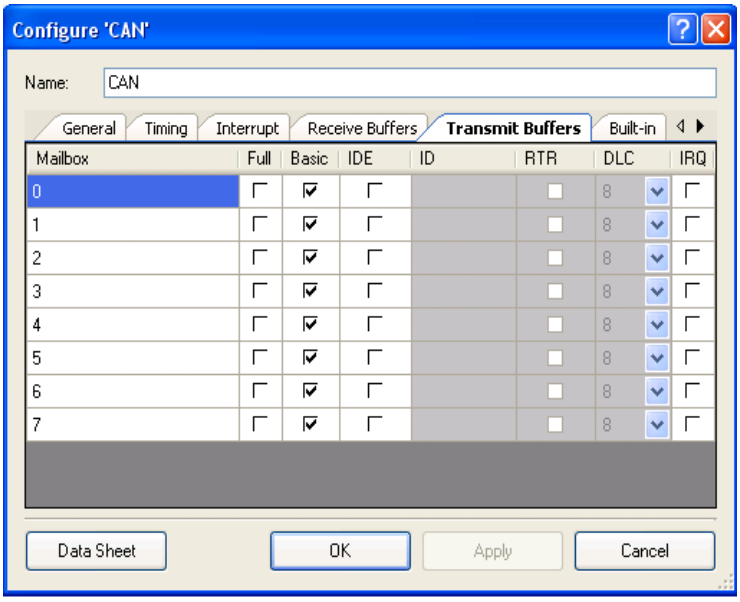
Figure 6. Interrupt Configuration Settings



Transmit Buffer

CAN has eight transmit buffers. Figure 7 shows the configuration the transmit buffers must have to send a basic CAN message.

Figure 7. Transmit Buffer



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

Firmware to Send a Basic CAN Message

A common API is included in *CAN.c* to send basic messages in CAN standard or extended:

```
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 that identifies 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 identifies data, are passed through the CANTXMsg structure. The structures are in the *CAN.h* file.

An example shows the use of structures to continuously send a standard CAN message.

Example Code

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

/* Declares 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;

   /* Defines 8 bytes of data */
   uint8
   CAN_data[]={0x00,0x11,0x00,0x11,0x00,0x
   11,0x00,0x11};

   /*
   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;
   /*  Enables  interrupt  for  the  CAN
   message */

   CAN_message.irq = 1;

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

   /* Enables global interrupts */
   CYGlobalIntEnable;

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

   /* Start CAN node */

   CAN_Start();

   while (1)
   {

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

   }
}
```

Message Transmitted ISR

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

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

   /* places user code here */

   /* `#END` */
```

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

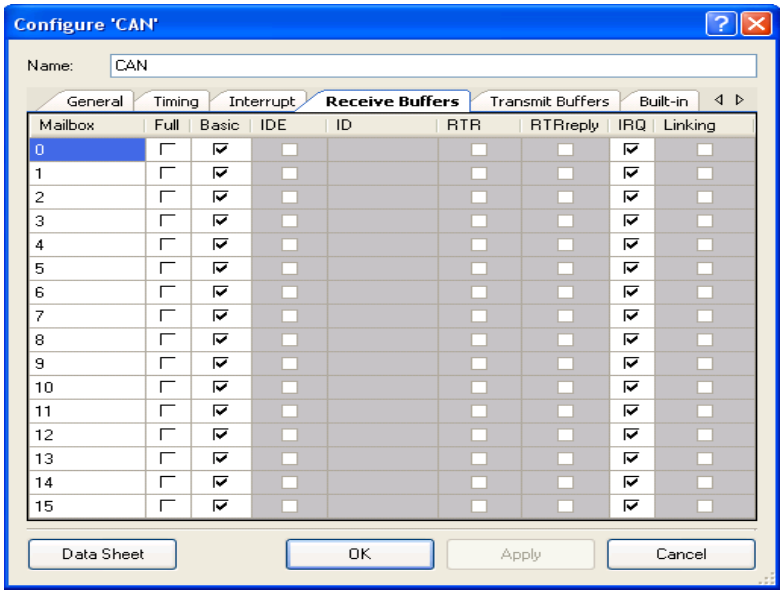
Receive Basic CAN Frame

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

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

All sixteen receive message buffers are configured as basic CAN buffers. The interrupts that correspond are enabled.

Figure 8. Receive Buffer Settings



Firmware Used to Receive a Basic CAN Message

After the message is received in a buffer, the message received interrupt is activated 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 in the *CAN.c* file.

The CAN_ReceiveMsg() API in the file *CAN_TX_RX_func.c* is used to get the basic CAN messages. The format of the API is as follows.

```
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 in Figure 8), in which the message is received. This is automatically received in the CAN receive message ISR.

CAN_RX is the define statement for the generic structure CANRXstruct from which all CAN message parameters are obtained. It is the define statement to the base address of the CAN receive buffers. The CANRXstruct matches the order in which the CAN registers are available in PSoC 3. For this reason, the CAN_RX define can be used to get access to 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 identifies 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, see 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

Set the Acceptance Filter

Each receive buffer has its own acceptance filter to filter messages that are received. When the acceptance filter is configured, only those incoming messages that satisfy the filter conditions are saved in the receive buffer. An acceptance filter includes the Acceptance Mask register (AMR) and the Acceptance Code register (ACR). A corresponding bit in the AMR defines whether the respective incoming bit is compared to the respective bit in the ACR. If a bit in the AMR is '0', then the corresponding bit in the ACR is compared to the corresponding bit that is received. If the bit in AMR is '1', then the corresponding bit in ACR is not compared to the bit that is received. The format of AMR and ACR registers is as shown in Figure 9. For standard CAN messages, the 11 bits to the extreme left of AMR and ACR are used as identifiers.

You use the CAN_RXRegisterInit() API available in *CAN.c* to set the AMR and ACR. 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 an ID that can have a range 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 = (0x1ffffff9);

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

/* Set the AMR value. The define used can
be found in file cyfitter.h */
```

```
CAN_RXRegisterInit((uint32)
CAN_CanIP__RX0_AMR, amr_value);

/* Set the ACR value. The define used can
be found in file cyfitter.h */
CAN_RXRegisterInit((uint32)
CAN_CanIP__RX0_ACR, acr_value);
```

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

Example Code

Example code that shows how to use the API that receives 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;

    /* Enables global interrupts */
    CYGlobalIntEnable;

    /* To pass messages with ID ranging
from 0x10 to 0x1f */
    amr_value = (0x1ffffff9);

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




    /* Starts CAN node */
    CAN_Start();

    /* Sets the AMR value for RX buffer 1*/

    CAN_RXRegisterInit((uint32)
CAN_CanIP__RX1_AMR, amr_value);

    /* Sets the ACR value for RX buffer 1
*/
    CAN_RXRegisterInit((uint32)
CAN_CanIP__RX1_ACR, acr_value);


    /* Enables all CAN interrupts */
    CAN_GlobalIntEnable();
        for(;;)
        {

        }
}
```

CAN_TX_RX_func.c

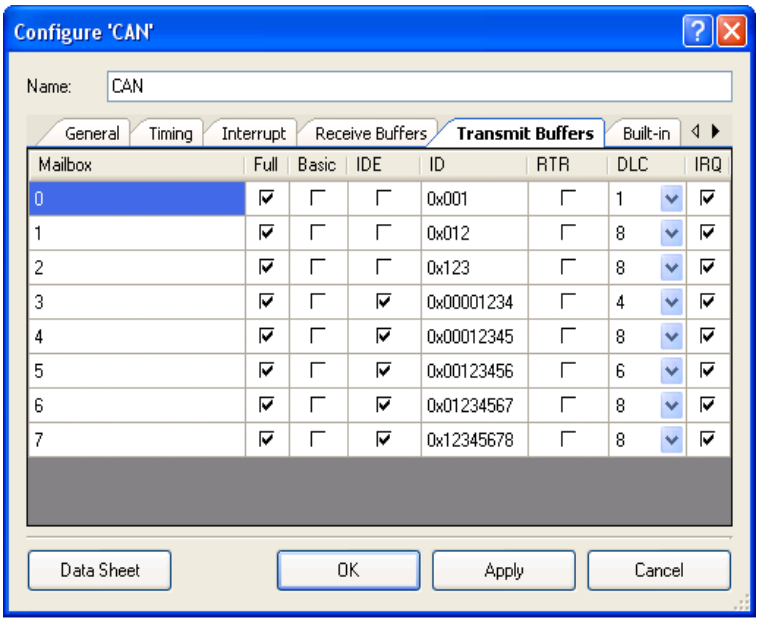
```
#include "CAN.h"
/* `#START TX_RX_FUNCTION` */

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++)
        {
            /*  Saves  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, use the same procedure. For an extended CAN message, the identifier received is 29 bits long. For this reason, the acceptance filter settings should be for 29 bits instead of 11 bits.

Send a Full CAN Message

To send a full CAN message, the transmit buffer settings are as shown in the following figure.

Figure 10. Transmit Buffer Settings



To send a standard CAN message, the IDE field is not selected, and the 11 bit identifier is identified in the ID field as shown in the first three mailbox settings. To send an extended CAN message, the IDE field is selected and the 29-bit identifier is identified 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 Used to Send a Full CAN Message

Both standard and extended CAN messages can be sent as full CAN messages with the API *CAN_SendMsgX()*, where 'X' is the mailbox numbers configured as 'FULL'. You can also type a string to replace the number and the API shows the mailbox name. The important macros available in *CAN.h* that 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 transmit
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() can send 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_TRASMITTED` */

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

        return result;
}
```

CAN_TX is the define statement for the generic structure CANTXstruct where all message parameters are specified. It is the define statement 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. Therefore, the CAN_TX define is used to get access to 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;
```

Figure 11 shows a register map with the CAN transmit registers. For more information on CAN transmit registers, see 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

Example code to use CAN_SendMsg0() in *CAN_TX_RX_func.c* to specify eight bytes of data is as follows.

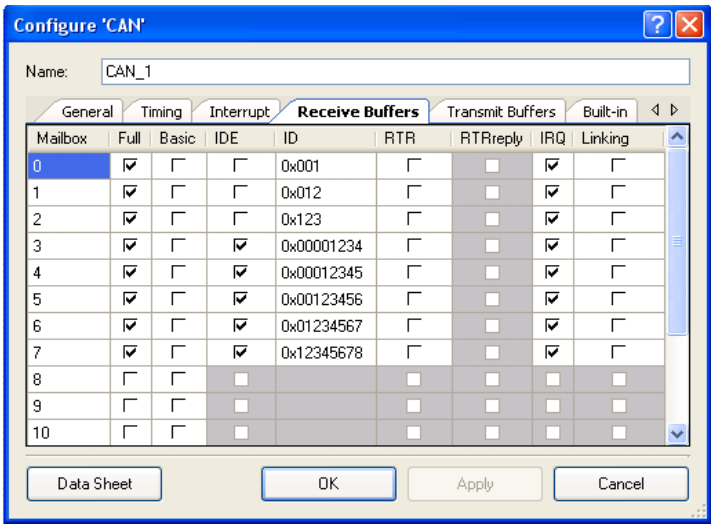
```
#include "CAN.h"
/* `#START TX_RX_FUNCTION` */

    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_TRASMITTED` */
        /* To send 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 send the message from the corresponding message buffer.

Receive a FULL CAN Frame

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

Figure 12. Receive Buffer Settings



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

#define CAN_RX_DATA_BYTE(mailbox,i)  /*
Macros to get 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 to receive and get access to
individual data bytes mailbox(i) */
```

When the message is received in a buffer, the message received interrupt is activated 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 get 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 statement for the generic structure CANRXstruct from which all CAN message parameters are obtained. The format of the structure is given in the Firmware Used to Receive a Basic CAN Message section.

To use the API CAN_ReceiveMsgX () in *CAN_TX_RX_func.c* to get data, write the appropriate code inside the function. The following example shows how to use the CAN_ReceiveMsg0 () API to get eight bytes of data.

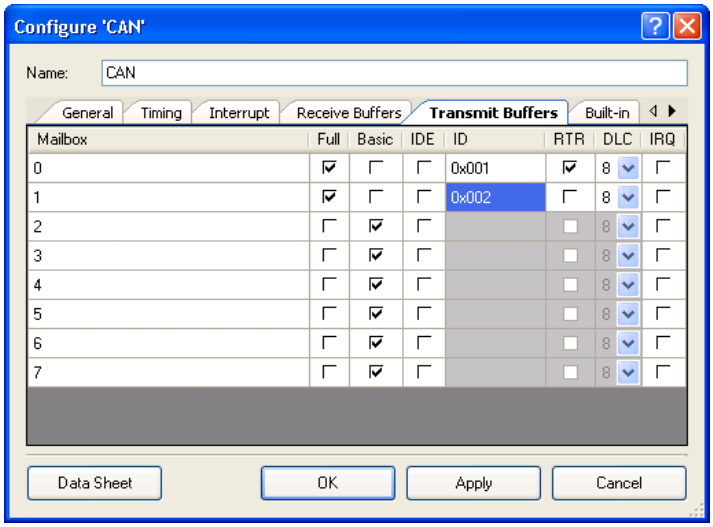
```
void CAN_ReceiveMsg0(void)
{
    /* `#START MESSAGE_0_RECEIVED` */
    /* Receive message on mailbox 0 */
    for(index=0;index<=8;index++)
    {

        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 described in the Firmware Used to Receive a Basic CAN Message section.

Send RTR Message

Remote frames are used to start transmission between two nodes. The node that has the receiver role 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 sent in an RTR frame, the DLC field must match the expected data length of the response to an RTR frame. To send an RTR frame, the transmit buffer settings is shown in Figure 13.

Figure 13. Transmit Buffer Settings



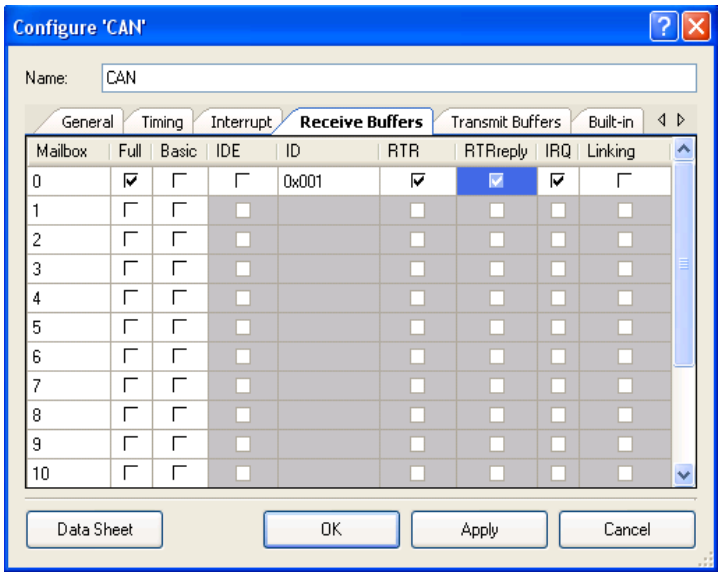
As you can see in the figure, for mailbox 0, the RTR bit is selected, which shows that it is an RTR message. Mailbox 1 can be used to send a normal Full CAN message.

The software code to send an RTR message is similar to that used to send a full CAN message. The API CAN_SendMsgX () is used to send the RTR message. RTR messages usually do not contain data; therefore, data can be ignored in the *CAN_TX_RX_func.c* file.

Receive and Automatically Send a Reply to 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, this buffer automatically sends a reply to this message with the content of the receive buffer.

Figure 14. Receive Buffer Settings for Automatic Reply to RTR Message



The node that sends the RTR message must receive the RTR reply message. The receive buffer settings of the node must not have the RTR reply option selected. The receive buffer settings must be the same as the settings used to receive a full CAN message.

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

MAIN.C

```
#include <device.h>
uint8 RTRreceive;
void main()
{
    /* Enables global interrupts*/
    CYGlobalIntEnable;

    /* Start CAN node*/
        CAN_Start();

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

    /* Sends RTR message*/
    CAN_SendMsg0();

        for(;;)
        {
            if(RTRreceive)
            {
/*  Sends  another  message  if  RTR  is
received*/
                CAN_SendMsg1();
                /* Clears 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 finds bit and acknowledgment errors. A receiving node finds 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 makes the transmitting node send the message again 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 the number of errors is equal to the error limit.

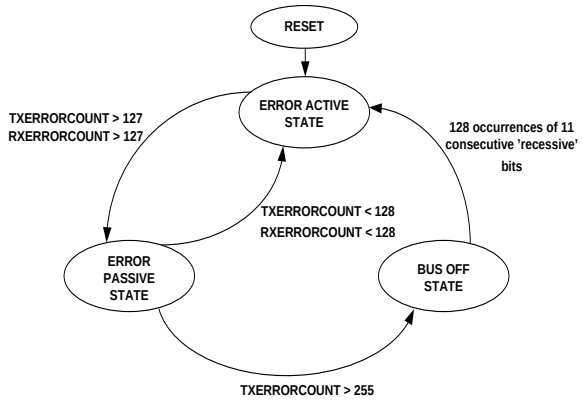
Each node that finds an error adds an increment to the error counter (transmit or receive). The CAN module can be in three different states, which are defined by the error counter values.

- **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 have normal bus communication.
- **Error Passive State:** A node is in 'error passive' state if the transmit or receive error counter value is greater than or equal to 128 decimal. An error passive node can have normal bus communication.
- **Bus Off State:** A node is in 'Bus Off' state if the transmit error counter is greater than or equal to 256 decimal. A node that is in Bus Off does not have any bus communication. It has no effect on the bus.

The restart from Bus Off state can be manual or automatic as the General Configuration section describes. When automatic is selected, the bus becomes active after 11 consecutive 'recessive' bits occur 128 times on the bus.

The state diagram shown in Figure 15 helps to describe the error states in CAN.

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` */

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

Code Examples

Two code examples, Example_FULLCAN_Sense and Example_FULLCAN_Control are attached to this application note. The code examples show basic information about how to send and receive a FULL CAN message.

To communicate between two PSoC 3 devices, load the Example_FULLCAN_Sense project on one PSoC 3 device and load the Example_FULLCAN_Control project on the other PSoC 3 device. Use two CAN transceivers to connect the two PSoC 3 devices as shown in Figure 1.

Example_FULLCAN_Sense

In Example_FULLCAN_Sense an 8-bit ADC is used to measure an analog input voltage. The ADC data is then transmitted through CAN bus to the CAN node at the other end of communication (Example_FULLCAN_Control). The destination CAN node would then send back a string that tells whether the ADC input voltage is in range (<2.5 V for V_{CC} = 5 V) or out of range. The received data (string) is the displayed on LCD.

Example_FULLCAN_Control

In Example_FULLCAN_Control, the CPU waits until data is received through the CAN bus. When a byte is received (8-bit ADC data send by CAN node: Example_FULLCAN_Sense) CPU sees whether the received data is greater than 0x80 (hex equivalent of 2.5 V (V_{CC} = 5 V). If it is, a string message "OverLoad" is sent to the CAN node on the other end of the communication (Example_FULLCAN_Sense). If the data is less than or equal to 0x80, the string message "In Range" is sent. The received and transmitted data are displayed on an LCD.

CAN Clock Accuracy

Accuracy of the CAN source clock (Bus clock) is an important measure for successful transmission and reception of CAN message. In the CAN bus specification, the accuracy of the clock is ≤0.5%. The clock accuracy depends on a lot of factors including the CAN bit timing and also external factors like delay of transceivers, bus line and receiver circuit.

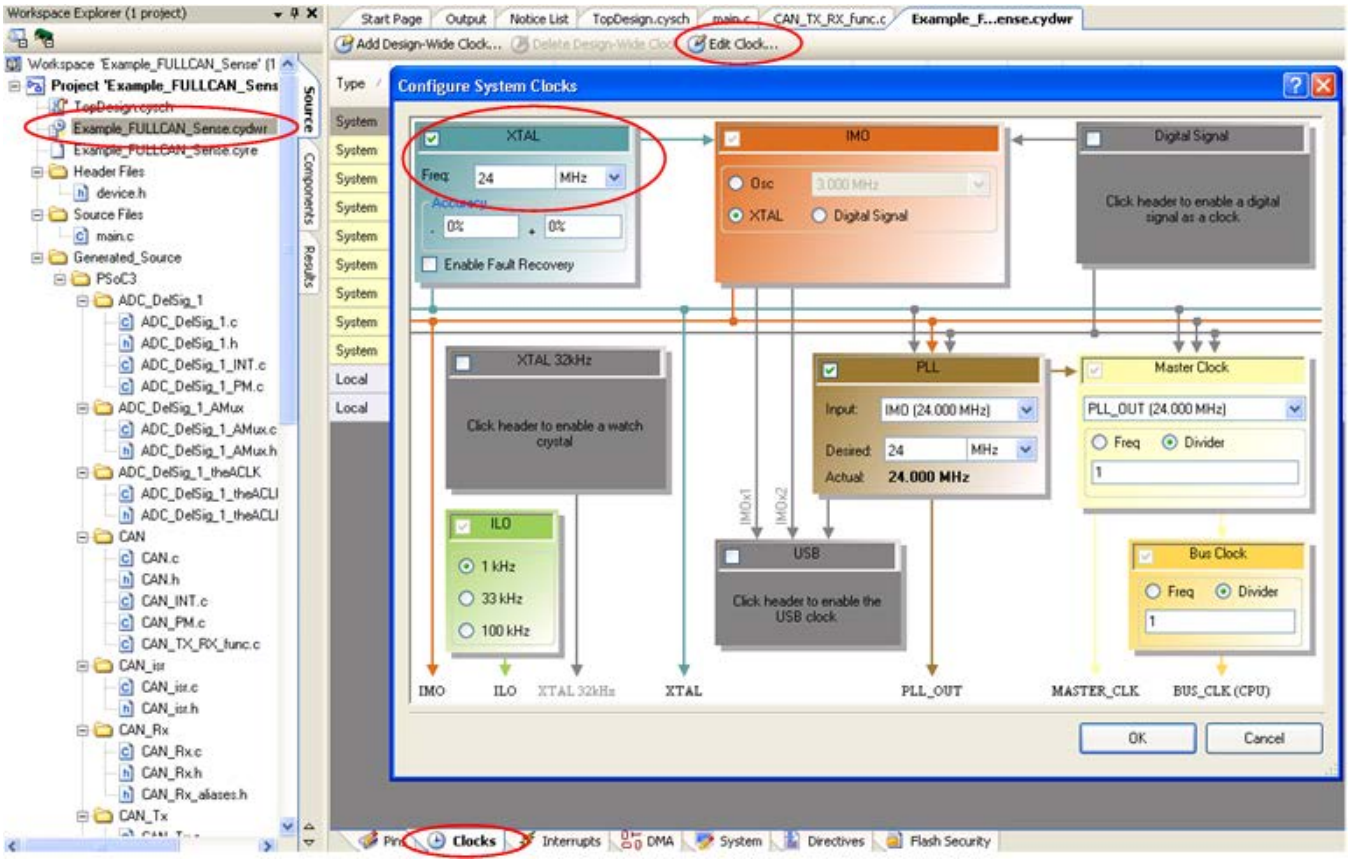
At baud rates greater than 125 ksps, you should get the CAN clock source (bus clock) from an external crystal (XTAL) with high accuracy (0.1%). To do this, configure the clock tree in the design wide resources file. This file has the extension *.cydwr*; for example, *Example_FULLCAN_Sense.cydwr*):

1. Open the design wide resources file. From the **Clocks** tab, click **Edit Clocks**. The clock tree opens.
2. Enable the XTAL: click the check box on top left of the clock tree.
3. For the source of IMO, select **XTAL**. Do not change the other settings.

If required, change the PLL frequency or master clock divider to a bus clock frequency other than the default 24 MHz.

Figure 16 shows an example of the Bus clock derived from XTAL.

Figure 16. Bus Clock Derivation from XTAL



Make sure that the XTAL is mounted on the pins P15[0] and P15[1] of the PSoC 3 chip.

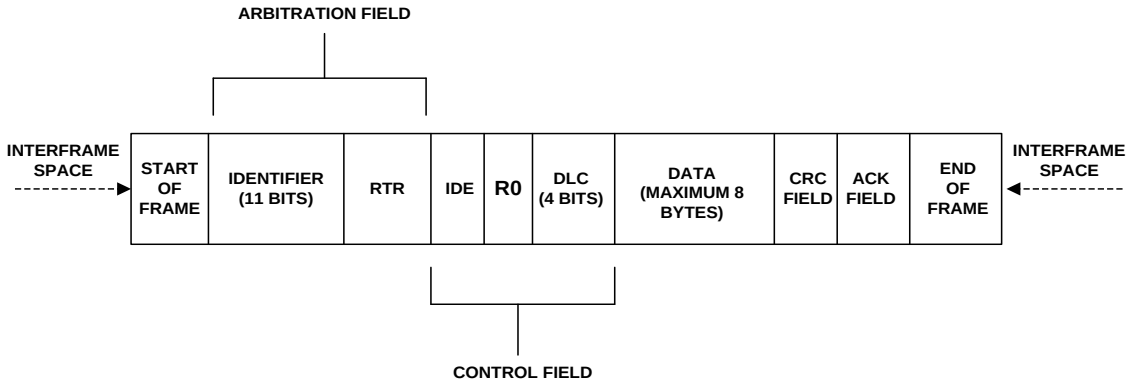
In the attached example projects, CAN is configured for a baud rate of 125 ksps. For this reason, the bus clock is not derived from an external crystal. To get the bus clock from other sources, see the code example Using External Clocks in PSoC 3 and PSoC 5.

Summary

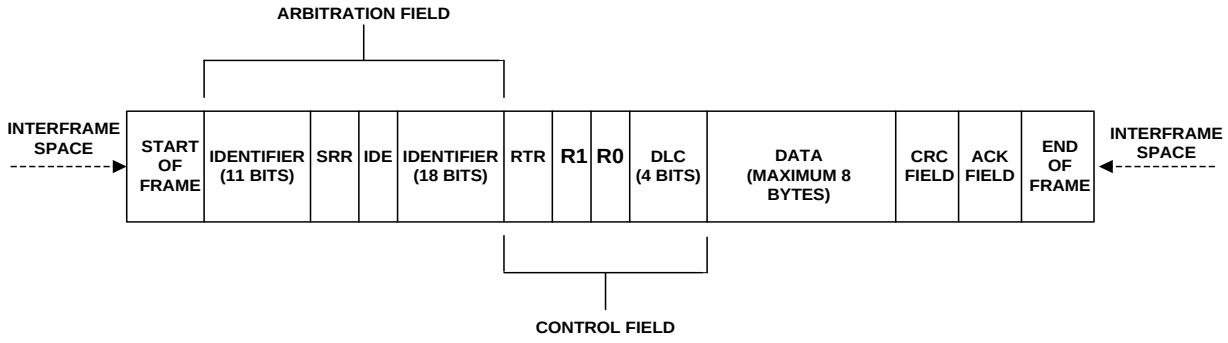
The flexibility of PSoC devices and PSoC Creator make efficient CAN bus communication easy to set up and use. This application note shows how to send and receive CAN messages between two PSoC 3 devices.

Appendix A

STANDARD DATA FRAME



EXTENDED DATA FRAME



Document History

Document Title: PSoC® 3 – How to Enable CAN Bus Communication – AN52701

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
*C	3032350	ANUP	09/17/10	Removed CAN_Init() from example code. Also moved CAN_RegisterInit() APIs after start API in page 10.
*D	3174976	ANUP	02/16/2011	Updated Setting Acceptance Filter. Added Code Example. Added CAN Clock Accuracy.
*E	3292004	LRDK	06/24/2011	Rewritten in Simplified English.
*F	3445166	DASG	11/22/2011	Project updated to PSoC Creator 2.0. Updated template.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

Automotive	cypress.com/go/automotive
Clocks & Buffers	cypress.com/go/clocks
Interface	cypress.com/go/interface
Lighting & Power Control	cypress.com/go/powerpsoc cypress.com/go/plc
Memory	cypress.com/go/memory
Optical Navigation Sensors	cypress.com/go/ons
PSoC	cypress.com/go/psoc
Touch Sensing	cypress.com/go/touch
USB Controllers	cypress.com/go/usb
Wireless/RF	cypress.com/go/wireless

PSoC® Solutions

psoc.cypress.com/solutions

PSoC 1 | PSoC 3 | PSoC 5

Cypress Developer Community

Community | Forums | Blogs | Video | Training

PSoC is a registered trademark and PSoC Designer is a trademark of Cypress Semiconductor Corporation. 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
Website : www.cypress.com

© Cypress Semiconductor Corporation, 2009-2011. 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.