AN52701

PSoC 3 and PSoC 5LP – Getting Started with Controller Area Network (CAN)

Author: Ranjith M
Associated Project: Yes
Associated Part Family: All PSoC 3 and PSoC 5LP parts with CAN
Related Code Example: CE95282, CE95283
Software Version: PSoC Creator™ 4.1

More code examples? We heard you.

To access an ever-growing list of hundreds of PSoC code examples, please visit our code examples web page. You can also explore the PSoC video library here.

This application note introduces the basic concepts of Controller Area Network (CAN) and demonstrates how CAN bus communication is implemented using PSoC® 3 and PSoC 5LP.

Contents

1 Introduction....1
1.1 Why Use CAN?....2
1.2 Why Use PSoC?....2
2 CAN Basics....2
2.1 Physical Layer....2
2.2 Transfer Layer....3
2.3 Bus Arbitration....4
2.4 Error Management in CAN....5
3 CAN in PSoC....5
3.1 Hardware....5
3.2 The CAN Component....7
3.3 PSoC Creator Projects....7
3.4 Firmware....15
4 Implementing the Hardware....18
4.1 Working with a CAN Analyzer....18
5 Example Projects....19
5.1 Examples 1 and 2: Simplex Communication....19
5.2 Examples 3 and 4: RTR feature in CAN....21
6 Summary....23
7 Related Resources....23
Appendix A. Data Frames....24
Appendix B. Interfacing PSoC with TJA1050 CAN Transceiver....25
Document History....26
Worldwide Sales and Design Support....27

1 Introduction

Controller Area Network (CAN) is a serial communication protocol developed by Robert Bosch GmbH in the early 1980s. This protocol was initially developed for automotive applications, for communications between subsystems with no central control. CAN is also being adopted in areas such as embedded systems (CANOpen) and factory automation (DeviceNet). CAN was standardized by the ISO in 2003 (ISO 11898-1:2003).

This application note introduces the basic concepts of the CAN protocol and demonstrates how CAN bus communication can be implemented using PSoC® 3 and PSoC 5LP. Four examples are included with this application note. Examples 1 and 2 together illustrate a simplex communication between two PSoCs. Examples 3 and 4 together demonstrate the Remote Transmission Request (RTR) feature of CAN.

This application note assumes that you are familiar with developing applications using PSoC Creator for PSoC 3 or PSoC 5LP. If you are new to PSoC 3 or PSoC 5LP, introductions can be found in AN54181, Getting Started with PSoC 3 and AN77759, Getting Started with PSoC 5LP. If you are new to PSoC Creator, see the PSoC Creator home page.

Cypress is a leading supplier of automotive microcontrollers. If you are interested in using CAN in our FM4 family of microcontrollers, refer to our FM4 CAN code example. Cypress also provides Arduino-compatible CAN shield to be used with our PSoC 4 family of devices. Refer to the CY8CKIT-026 webpage for more details on the CAN shield. An example project that demonstrates PSoC 4 CAN communication using the CY8CKIT-026 shield is available in the PSoC 4 CAN example webpage.

1.1 Why Use CAN?

CAN networks are designed for short messages with data length not more than 8 bytes, and a bit rate up to 1 Mbps. The CAN protocol offers several advantages over other serial communication protocols:

- CAN is a message-based protocol; CAN network nodes are not assigned specific addresses. This provides flexibility to add or remove a node from the network without affecting the rest of the network. In addition, if one of the nodes fails, the others continue to work and communicate properly.
- CAN messages can be prioritized.
- CAN has five levels of error checking, to ensure reliable traffic and data integrity.
- The CAN network has system-wide data consistency, that is, if a message is corrupt at a receiving node, the message is not accepted by any of the other receiving nodes.
- Corrupted messages are automatically retransmitted as soon as the bus is idle again.

1.2 Why Use PSoC?

PSoC 3 and PSoC 5LP integrate CAN functionality along with configurable analog, programmable digital, memory, and a central processor on a single chip. PSoC Creator provides built-in application programming interfaces (APIs) to abstract common tasks.

Moreover, all PSoC Creator Components, including the CAN Component, can be easily configured using a GUI, rather than writing C code for them. See application note AN70630 - Event Data Recorder with Controller Area Network using PSoC 3 and nvSRAM for a system-level application of the CAN Component.

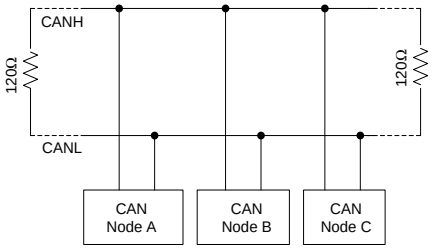
2 CAN Basics

This section explains the basics of the CAN protocol. A more detailed description protocol is available in the CAN specification. If you are familiar with CAN and want to see how it is implemented in PSoC 3 and PSoC 5LP, see the section CAN in PSoC.

2.1 Physical Layer

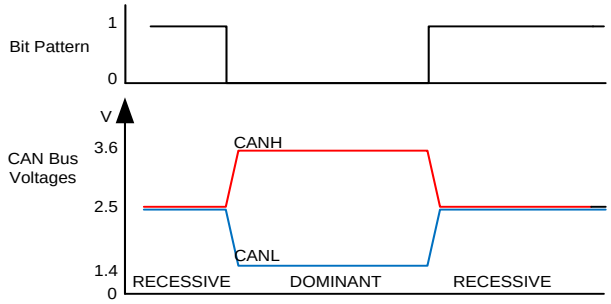
Figure 1 shows how devices connect to a CAN bus. The CAN bus consists of two physical lines, CANH and CANL. The CAN bus is typically terminated with a 120-Ω resistor at each end.

Figure 1. Nodes Connected to a CAN Network



The CAN bus carries differential signals, as Figure 2 shows. A '1' is represented by both lines at approximately the same voltage (typically 2.5 V). A '0' is represented by a 1.5 V to 3 V difference in voltage between the lines. A ‘1’ is called a recessive bit and a ‘0’ is called a dominant bit.

Figure 2. CAN Bus Voltages



The CAN protocol specifies that if recessive and dominant bits are simultaneously applied to the CAN bus by two different nodes, the bus must have the dominant bit. This can be related to a wired AND analogy—the bus does not have a recessive bit unless all nodes drive recessive bits. While in an idle state, the bus holds recessive bits.

2.2 Transfer Layer

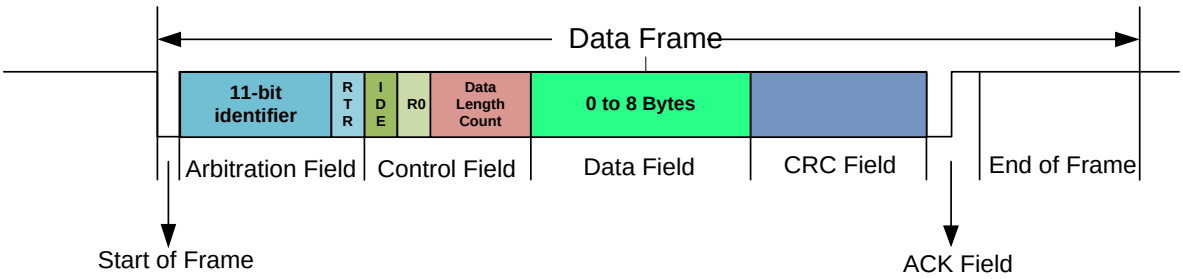
Any node can start transmitting a message when the CAN bus is idle. Messages are transmitted through the bus in a fixed format called a frame. CAN defines four frame types:

- Data Frame—carries data from a transmitter to a receiver
- Remote frame—transmitted by a CAN node to request transmission of a data frame
- Error Frame—transmitted by any unit on detecting an error on the bus
- Overload Frame—used to provide extra delay between the preceding and the succeeding data or remote frame

See Appendix A for format details for these frames.

A data frame is composed of seven-bit fields, as Figure 3 shows.

Figure 3. CAN Data Frame



Arbitration Field: The arbitration field of a data frame consists of two parts: an IDENTIFIER and a Remote Transmission Request (RTR) bit.

The identifier is used to describe the meaning of the data carried by the data frame. Each node on the bus checks the identifier and decides whether to accept the message.

Based on the length of the identifier field, two types of CAN messages are defined: standard and extended.

A standard CAN message has an 11-bit identifier and an extended CAN message has a 29-bit identifier. Thus, a standard CAN data frame can support 2^{11} different types of messages and an extended CAN data frame can support 2^{29} different messages.

The RTR bit is used to indicate whether the given frame is a data frame or a remote frame. The RTR bit is set ‘dominant’ in a data frame and ‘recessive’ in a remote frame.

Control Field: The control field of the data frame defines the number of data bytes in the data field. The control field is six bits long, with four bits for data length and two bits reserved for future expansion.

Data Field: The data field contains the data to be communicated.

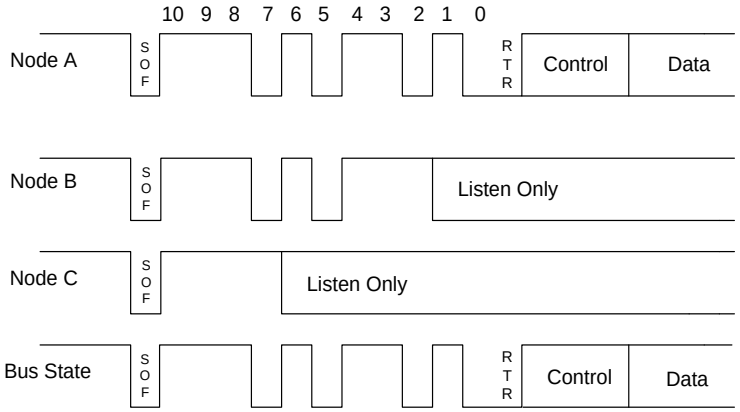
CRC Field: The Cyclic Redundancy Check (CRC) field is for error checking. This field contains a bit sequence, which is used to determine if the received frame contains any errors. The ACK field is used by the receiving nodes to acknowledge that the frame was correctly received.

A node acting as a receiver can request the transmission of a particular message from its source. This is achieved with the help of a remote frame. A remote frame is similar to a data frame except that it does not have a data field.

2.3 Bus Arbitration

If multiple nodes try to transmit at the same time, bus arbitration is done using identifier bits, as Figure 4 shows. Using the property of dominant bits described previously, after transmitting a bit on to the bus, each node checks if the bus state is the same as the bit that it transmitted. If they are the same, each node transmits the next bit. If they are different, the node stops transmitting and starts to receive the message on the bus. This is called the 'listen only' mode.

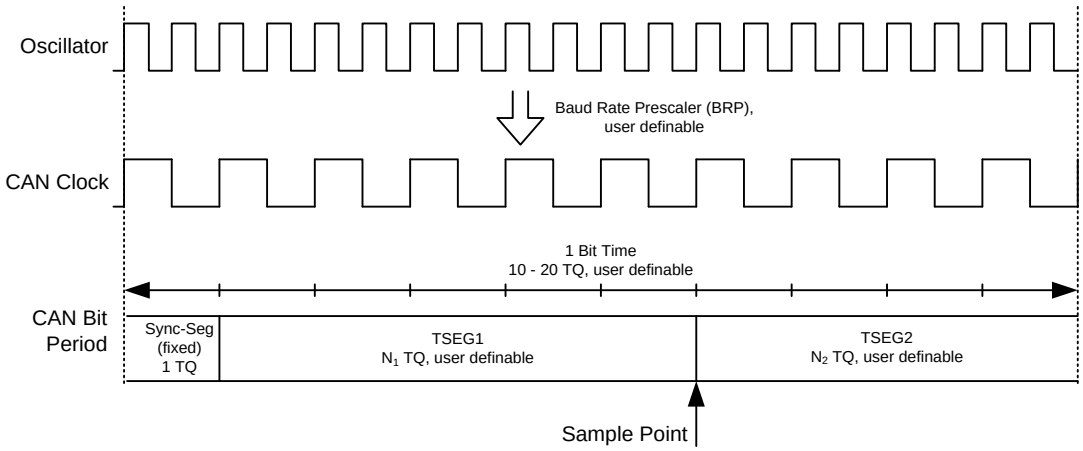
Figure 4. Bus Arbitration in CAN



Since each node reads back the bus after transmitting each bit, the bit duration must be larger than the largest propagation delay in the bus, to ensure that a collision does not go undetected. Therefore, the transmitted bit is read back after a delay to compensate for the propagation delay. The instant at which the bus is read back is called a "sample point", as Figure 5 shows.

The amount of time required to transmit a single bit is called the "bit time". In the CAN specification, bit time is expressed in terms of time quanta (TQ). A time quantum is a fixed unit of time derived from the oscillator period, as Figure 5 shows. The oscillator frequency is divided by a factor called Baud Rate Prescaler (BRP) to obtain the CAN clock frequency.

Figure 5. Derivation of Bit Time from Oscillator



At the start of the Sync Segment, or first TQ, the transmitter begins to drive the bit on to the bus. The transmitter continues to drive the bus throughout the bit time, and after a defined amount of time samples the bus for a collision. The time is determined by setting parameters TSEG1 and TSEG2; see Figure 18 on page 14. Generally, the sample point should be 60% to 80% of the bit time.

2.4 Error Management in CAN

A CAN node detects and handles five types of errors:

- Bit error—A bit error is detected by the transmitter when it finds that the bit transmitted and the bit on the bus are not the same.
- Form error—A form error is detected when there is any deviation in the message fixed format. A data length count (DLC) greater than 8 bytes is also considered a form error.
- Stuff error—Whenever a transmitter detects five consecutive bits of identical value in the bit stream to be transmitted, it automatically inserts a complementary bit into the stream. This is called bit stuffing. A stuff error is detected when six consecutive bits of identical value are detected.
- CRC error—A CRC error is detected when the value indicated by the CRC field of a frame does not match the frame's expected CRC value.
- Acknowledge error—An acknowledge error is detected by the transmitter if an acknowledgement (a ‘dominant’ bit) is not obtained during the acknowledge field of frame.

Bit errors and acknowledge errors are detected by the transmitter, and stuff errors, CRC errors, and form errors are detected by the receiver. If a message fails any of these error detection methods, the message is not accepted and the receiving node generates an error frame. The transmitting node, then, re-transmits the message until it is received correctly.

If a faulty node hangs the bus by continuously retransmitting an error frame, its transmit capability is removed after the number of errors equals the error limit. Each node maintains error counters for transmit and receive errors, which are incremented after detecting corresponding errors. Depending upon the value of the error counters, a CAN node can be in three different states:

- Error Active State—A node is in the "error active" state if the transmit or receive error counters are less than or equal to 127. An error active node can have normal bus communication.
- Error Passive State—A node is in the "error passive" state if the transmit or receive error counter value is greater than or equal to 128. An error passive node can have normal bus communication.
- Bus Off State—A node is in the "bus off" state if the transmit error counter is greater than or equal to 256. A node that is in bus off does not have any bus communication. It has no effect on the bus.

3 CAN in PSoC

3.1 Hardware

The CAN block in PSoC 3 and PSoC 5LP, shown in Figure 6 on page 6, is compliant with the CAN 2.0a and 2.0b standards. However, it requires an external transceiver to level shift the output voltages and to make it compatible with the CAN protocol. The TJA1050 from NXP or the SN65HVD1050-EP from TI can be used as an external transceiver. These devices translate between the bit pattern and bus voltages, as Figure 2 on page 3 shows.

With this application note, the Cypress kit CY8CKIT-017 uses the TJA1050 as the external transceiver, as Figure 7 on page 6 shows.

Figure 6. CAN Block Diagram

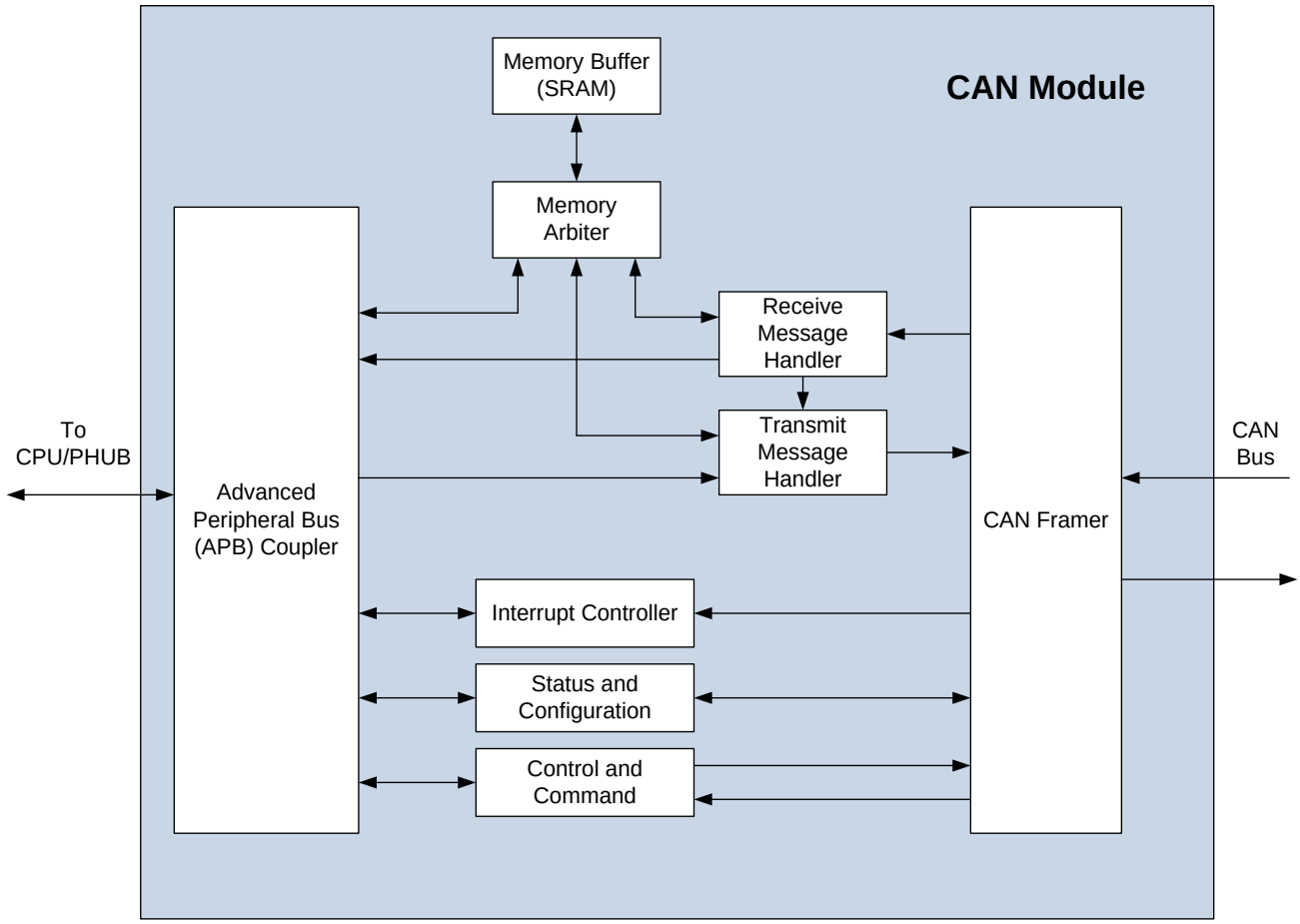
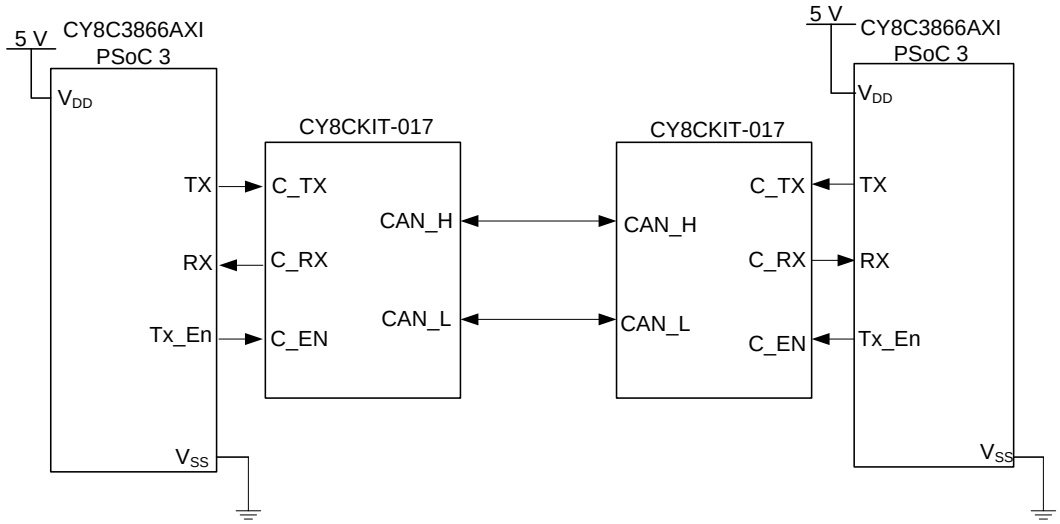


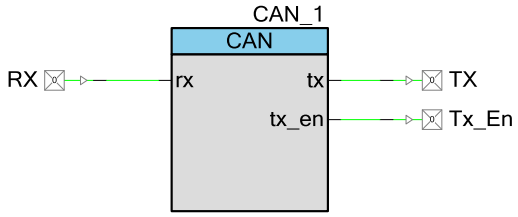
Figure 7. Hardware Connections for Implementing CAN Network



3.2 The CAN Component

Figure 8 shows how the CAN Component is displayed on the PSoC Creator schematic.

Figure 8. CAN Component in PSoC Creator



PSoC offers two different modes for CAN communication: Full CAN and Basic CAN. The following are the important differences between a full CAN message and a Basic CAN message:

- A Full CAN communication can be easily set up with the help of a GUI, with a very limited amount of programming involved. Basic CAN requires all of the parameters to be set in firmware.
- Full CAN uses hardware for message filtering. Basic CAN requires the CPU to be interrupted each time a message is received, to determine whether it is accepted or not.
- Full CAN can only receive a single type of message for each mailbox[1] whereas Basic CAN can accept messages with a range of identifiers for each mailbox.

The following section explains how to configure the CAN Component for a Full CAN communication. The steps below describe how to configure two PSoC devices, each in a separate development kit (DVK). The CAN Component in one PSoC is configured as the transmitter and the CAN Component in the other PSoC is configured as the receiver.

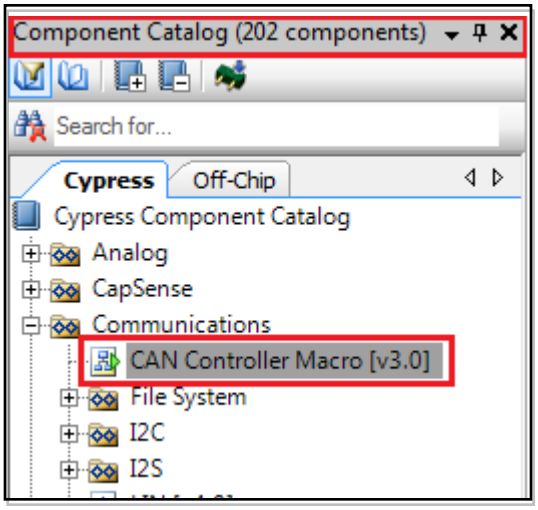
Note: Only the options relevant to this application note are described. See the CAN Component datasheet for a detailed description of all the options available with the CAN Component.

3.3 PSoC Creator Projects

Do the following steps to build projects to demonstrate basic CAN:

1. Open PSoC Creator and create a new project (**File** > **New** > **Project**). Name the project 'Receiver'.
2. Drag and drop the CAN Controller Macro from the Component Catalog to the TopDesign schematic, as Figure 9 shows.

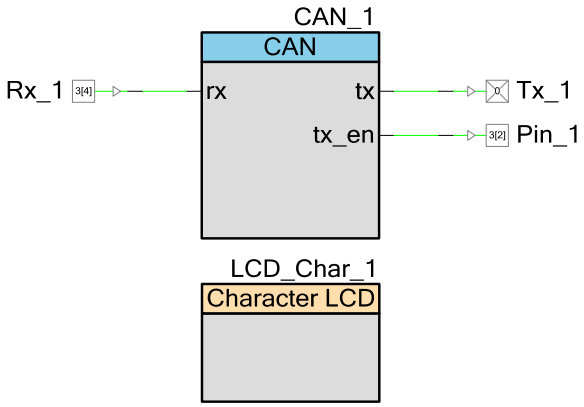
Figure 9. CAN Controller Macro in Component Catalog



[1] A mailbox is a set of input / output buffers for receiving and transmitting CAN messages.

3. Drag and drop the character LCD Component from the Component Catalog to the TopDesign schematic. This is available under the folder 'Display'. Figure 10 shows the complete schematic.

Figure 10. Completed TopDesign

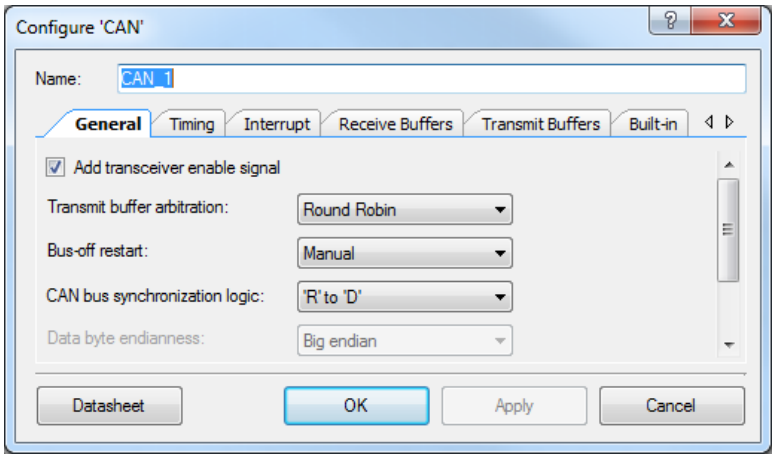


4. Double-click on the CAN Component in the TopDesign to open the configuration window.

3.3.1 CAN General Configuration

Figure 11 shows the general configuration tab for the PSoC Creator CAN Component.

Figure 11. General Settings Tab for CAN Component



5. Ensure that the Add Transceiver Signal checkbox is checked. This is enabled by default.

   The Add Transceiver Signal option in the General Configuration tab enables/disables the tx_en signal in the CAN Component. This signal is used to connect to the enable pin of the external transceiver.

6. Set the Transmit Buffer Arbitration to be ‘Round-Robin’.

   The Round Robin option ensures that all transmit mailboxes are given equal opportunity to transmit, whereas the fixed priority option assigns priorities to mailboxes according to which messages are transmitted.

7. Set the Bus-Off Restart method to be ‘Manual’.

   Bus-Off restart can be done either manually or automatically, monitoring the number of transmit errors.

8. Select the CAN Bus Synchronization Logic to be 'R' to 'D'.

   A clock signal is not transmitted in a CAN network. Instead, clocks of all the receiving nodes are synchronized at the falling edge of the start of frame (SOF) bit sent by the transmitter. Subsequent edges are also used for synchronizing the clocks, to accommodate for the small drifts in clocks between different nodes.

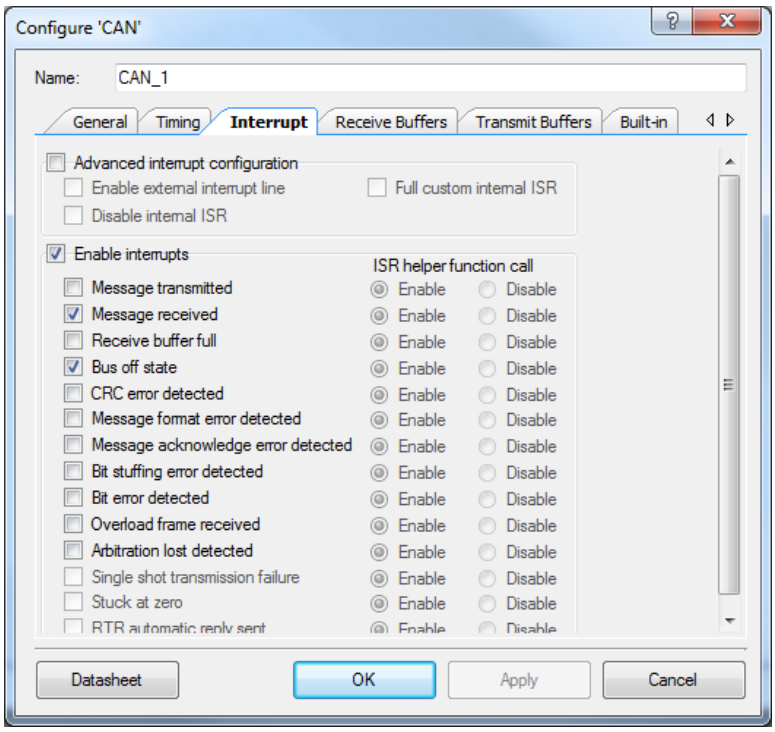
   The synchronization can be done on either recessive to dominant ('R' to 'D') transitions or on both edges (recessive to dominant and dominant to recessive).

3.3.2 CAN Interrupt Configuration

Figure 12 shows the interrupt settings tab for the CAN Component. Use this tab to enable or disable interrupts on a number of events. If an interrupt is enabled, PSoC executes an Interrupt Service Routine (ISR) when that event occurs.

The Message Received and Bus Off State interrupts are selected by default. The Message Received interrupt automatically calls the `ReceiveMsgx()` function in *CAN_TX_RX_func.c*.

Figure 12. Interrupt Settings Tab for CAN Component



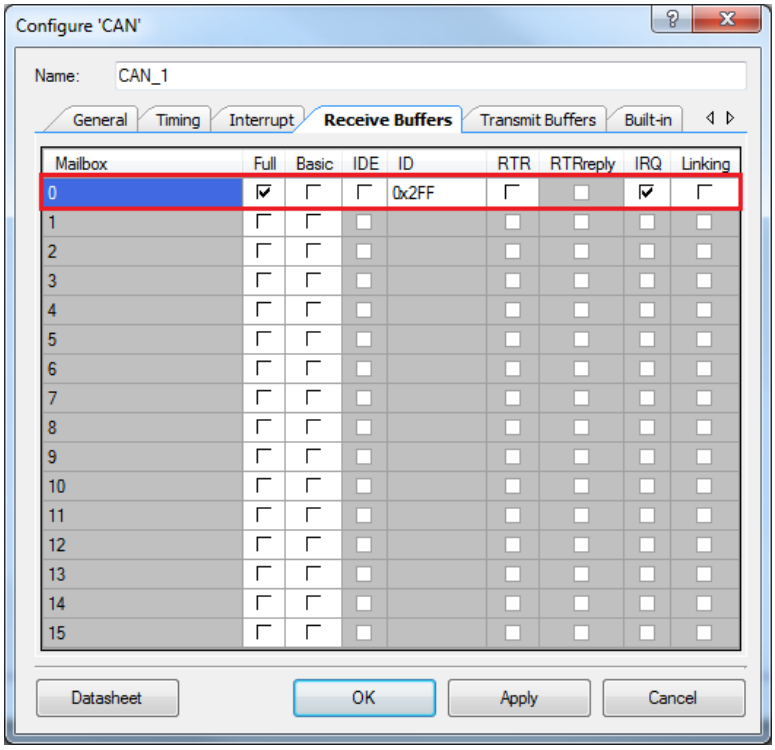
Make sure that the check boxes 'Enable Interrupts', 'Message Received', and 'Bus Off State' are checked, to enable these interrupts. All other boxes should be unchecked.

3.3.3 CAN Receive Buffer

The CAN Component has 16 input buffers, or mailboxes, for receiving messages. Thus, the CAN Component can receive at most 16 different CAN message types.

The mailboxes are numbered from 0 to 15 by default and can be replaced with any name of your choice. To do this, select the "Full" mode, as Figure 13 shows. Selecting Full mode enables other configurable options in the row.

Figure 13. Receive Buffer Settings for CAN Component



1. Select checkboxes, as Figure 13 shows, to enable a Full mailbox with ID 0x2FF. The ID field is used to define the identifier for the message to be received in the corresponding mailbox, and can be set to any 11-bit value.
2. Check the IRQ box to trigger an interrupt when a message is received in that mailbox.

   This completes configuration of the CAN Component on the receiver side. The transmit buffers need not be configured in this case, since this node does not need to transmit a message.

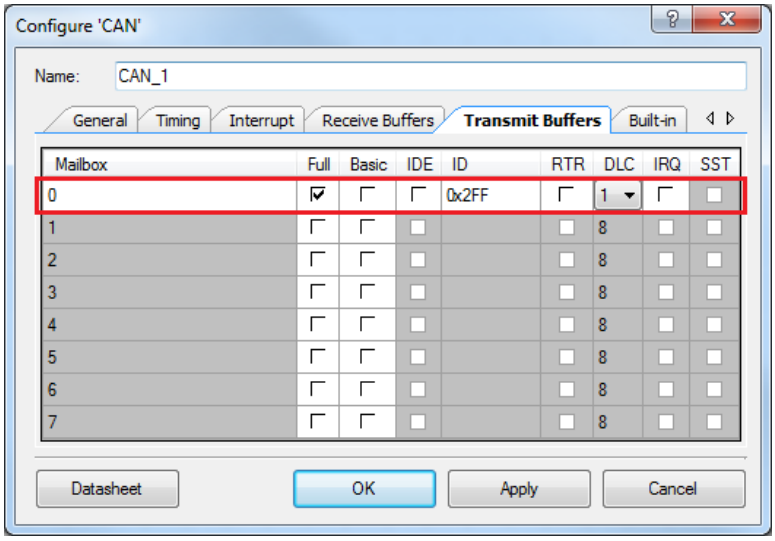
3.3.4 CAN Transmit Buffer

Now, let us configure the transmitter side. Since we do this for the other PSoC, we need to create another project for that PSoC.

1. Add another project to the Workspace: Right-click on the Workspace name inside the Workspace Explorer window and select **Add** > **New Project**. Name this project 'Transmitter'.
2. Drag and drop the CAN Controller Macro into the TopDesign of this project. Configure the CAN Component as described in Sections 3.3.1 to 3.3.3 similar to the receiver side.

The Transmit Buffers tab is used to configure the transmit mailboxes, as Figure 14 shows. A transmit mailbox can be either Full or Basic. Selecting Full mode enables other configurable options in the row.

Figure 14. Transmit Buffer Settings for CAN Component



3. Select the checkboxes as Figure 14 shows, to enable a Full mailbox with ID 0x2FF, same as the receiver side.

   The Data Length Count (DLC) field is 1 because we only need to transmit one byte.

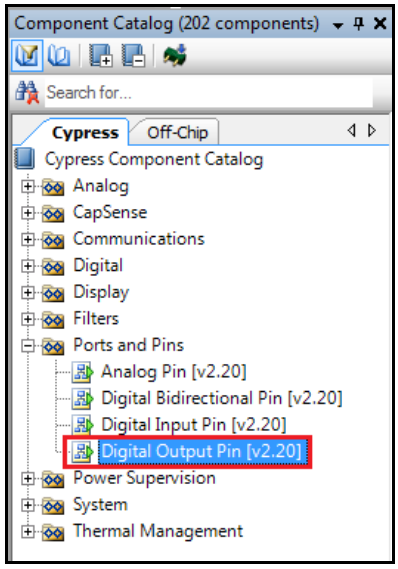
3.3.5 Pin Configurations

The steps in this section apply to both the Receiver and Transmitter projects.

Let us now add and configure the pins in the projects. The Tx_1 and Rx_1 pins were included as part of the CAN macro in step 2. We need to add a third pin, as Figure 10 shows.

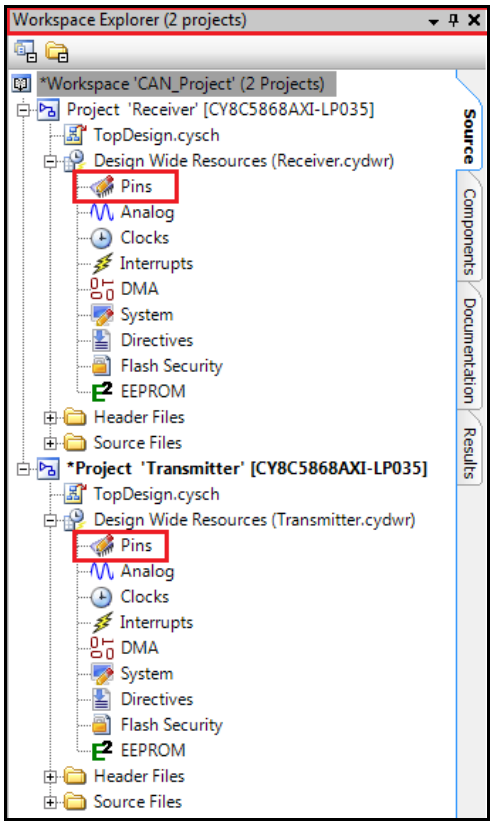
1. Drag and drop a Digital Output Pin on to the TopDesign schematics of both the projects, as Figure 10 and Figure 15 show. Connect this pin to the tx_en output of the CAN Component. This pin is used to enable the external transceiver.

Figure 15. Locating the Digital Output Pin



3. Open each project's Pins tab in the design wide resources (*.cydwr*) file in the Workspace Explorer, as Figure 16 shows.

Figure 16. Locating the .cydwr File



4. Assign the ports for each of the pins as Table 1 shows.

Table 1. Pin Assignments

Pin	CY8CKIT-001	CY8CKIT-030/ CY8CKIT-050
Rx_1	P3[4]	P3[4]
Tx_1	P3[3]	P3[3]
Pin_1	P3[2]	P3[2]
LCD_Char_1	P2[6:0]	P2[6:0]

3.3.6 Clock Configurations

The steps in this section apply to both the Receiver and Transmitter projects.

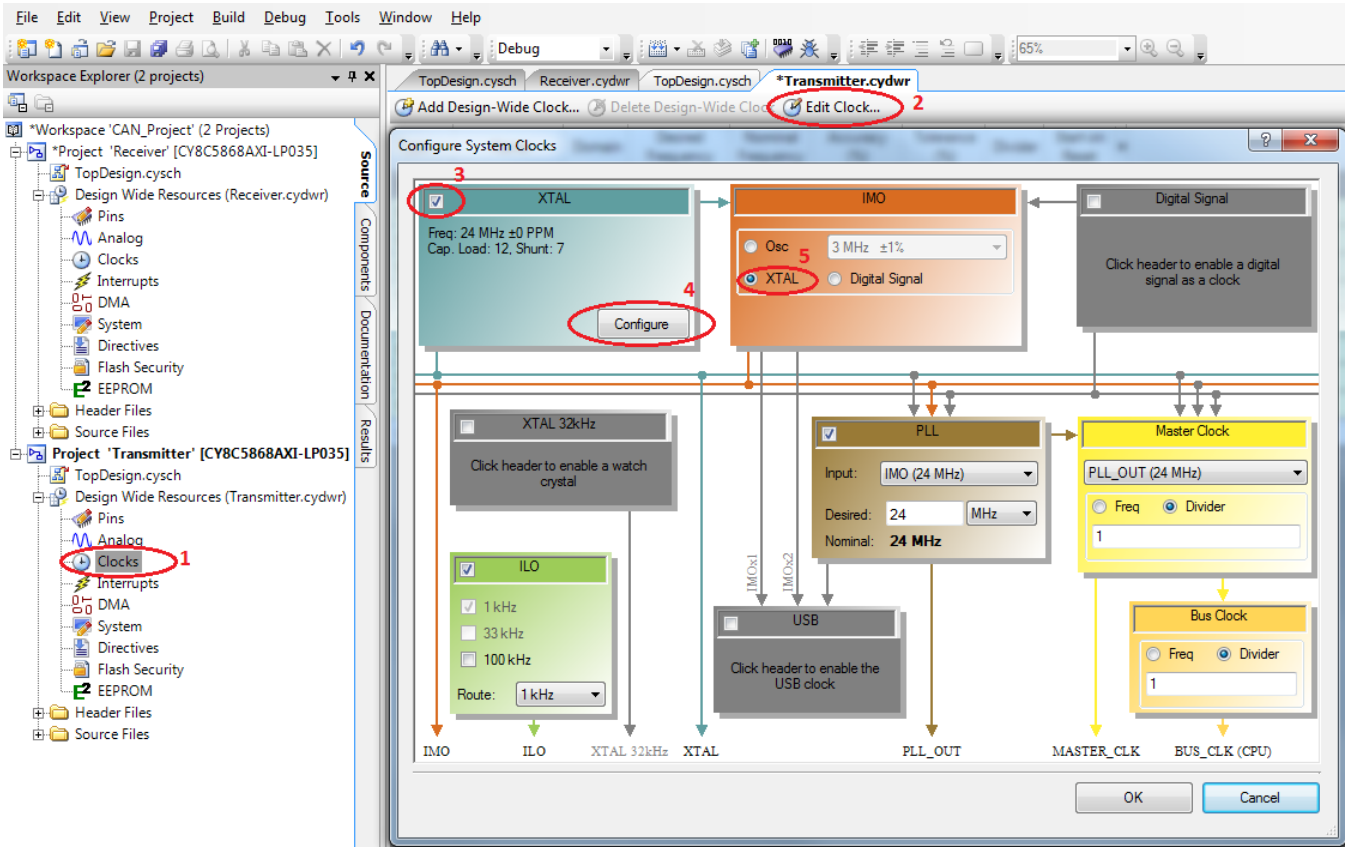
The CAN protocol does not transmit a clock to synchronize the bits. Synchronization between nodes is done for every bit transmitted, during the Sync Segment, as Figure 5 on page 4 shows. This requires the use of a highly accurate oscillator for baud rates greater than 125 Kbps.

The CAN protocol specifies that the clock accuracy must be less than or equal to 0.5%. An error less than 0.1% can be achieved in PSoC by using an external crystal. The clock tree dialog in the design wide resources file (that is, the file with extension. cydwr) is used to configure this.

1. Open the design wide resources file, as Figure 17 shows. From the Clocks tab, click **Edit Clocks**. The clock tree dialog opens.

2. Enable the crystal - click the check box XTAL on the top left of the clock tree. Click the **Configure** button and enter 24 MHz for the crystal frequency.
3. Select XTAL as the IMO source. Do not change other settings.
4. If your kit does not already have it, connect a 24-MHz crystal and two capacitors to pins P15[0] and P15[1]. See a PSoC 3 or PSoC 5LP datasheet for details.

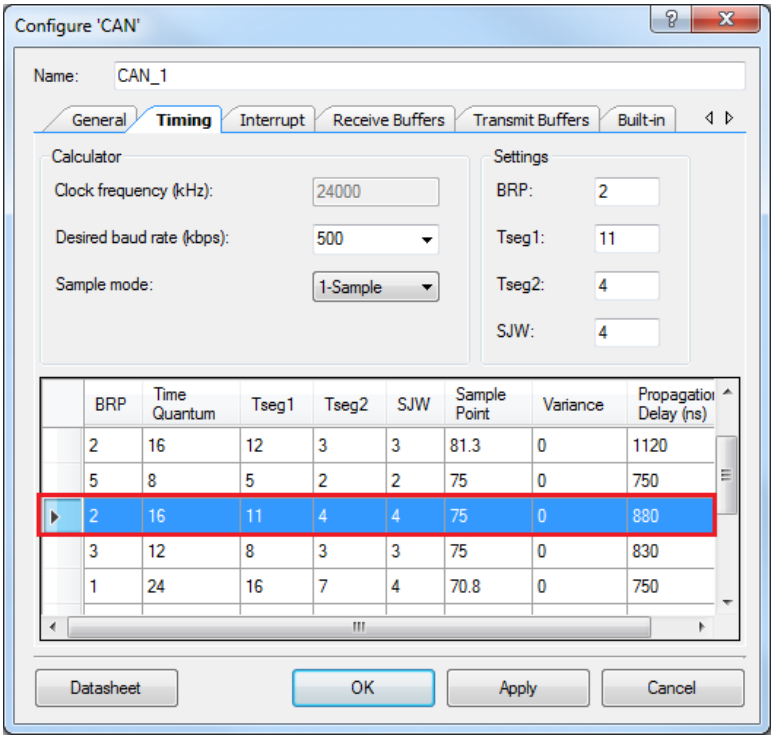
Figure 17. Clock Tree Configuration



3.3.7 CAN Timing Configuration

Figure 18 shows the timing configuration tab for the CAN Component.

Figure 18. Timing Settings Tab for CAN Component



1. Set the baud-rate to be 500 kbps. This automatically modifies the values as shown in Figure 18.

   The baud rate determines the speed of communication between the devices. It can be as high as 1 Mbps. All CAN nodes on a bus must operate at the same baud rate.

   Note Selecting the baud rate only provides a list of possible timing parameters in the table (see Figure 18). You must double-click on a row in the table to update the parameters in the respective fields.

2. Double-click on the row with BRP = 2, Time Quantum = 16, Sample Point = 75 and, SWJ = 4 (see Figure 18) to add these values to the "Settings".

   For best performance, select a row with a Sample Point value between 60 and 80 and a Variance of 0.

   Synchronization Jump Width (SJW) is the number of time quanta that a sample point is allowed to vary from its mean position. This value must be less than or equal to both TSEG1 and TSEG2 in Figure 5 on page 4.

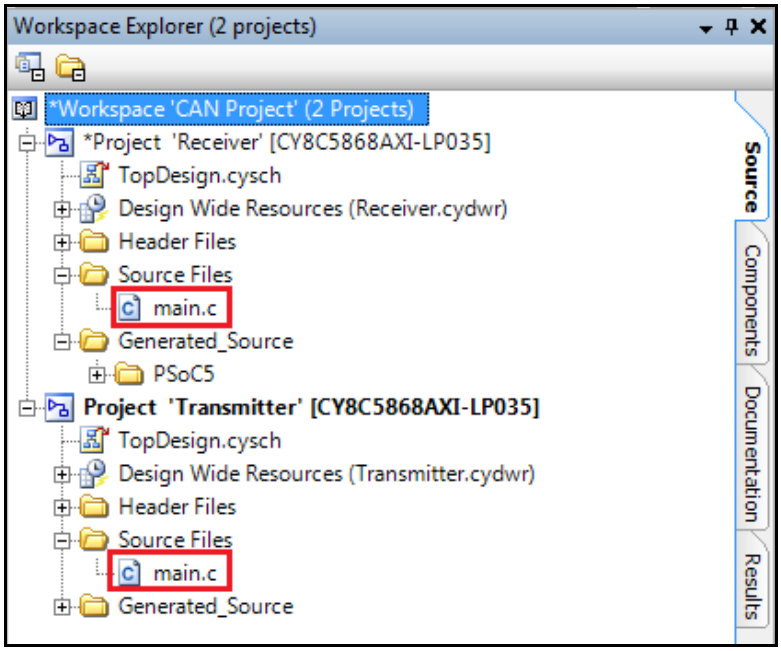
   The sample mode is the number of samples that are taken to determine the state of the bus. A single sample mode or 3-sample mode may be chosen here.

3.4 Firmware

The steps in this section apply to both the Receiver and Transmitter projects. A small amount of firmware must be added to each project.

1. Build each project by selecting the menu ***Build*** > ***Build All Projects***. The CAN Component and other API files are automatically generated.
2. Open the *main.c* file of the transmitter project by double-clicking the file name inside the Workspace Explorer (see Figure 19). Add the code from Code 1 to the *main.c* file.

Figure 19. Locating the *main.c* File



Code 1. Transmitter *main.c* Code

```
#include <device.h>

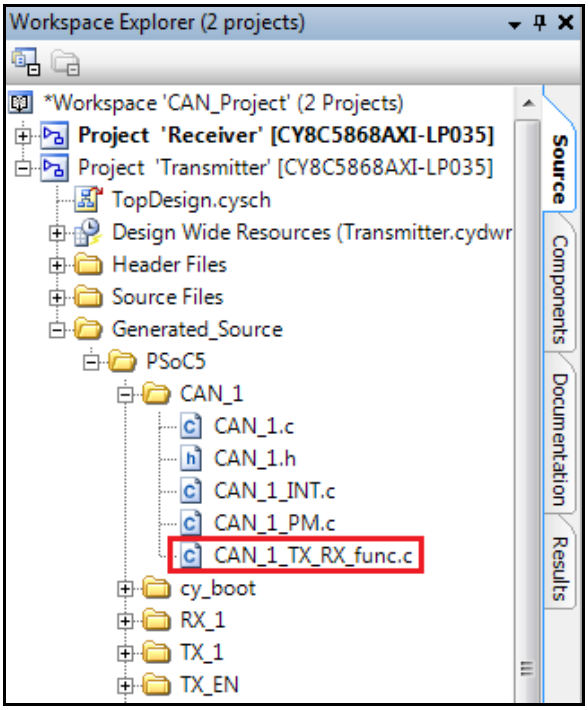
uint8 Tx_Data = 0;

int main()
{
  CAN_1_Start();
  LCD_Char_1_Start();
  CyGlobalIntEnable;

  for(;;) /* do forever */
  {
    LCD_Char_1_ClearDisplay();
    LCD_Char_1_Position(0,0);
    LCD_Char_1_PrintNumber(Tx_Data);
    CAN_1_SendMsg0();
    Tx_Data++;
    CyDelay(500);
  }
}
```

3. Open the *CAN_1_TX_RX_func.c* file in the Transmitter project by double-clicking the file name inside the Workspace Explorer, as Figure 20 shows. This file is generated after the project is built.

Figure 20. Locating *CAN_1_TX_RX_func.c*



4. Navigate to the lines:

```
/* `#START TX_RX_FUNCTION` */

/* `#END` */
```

5. Type the following line of code between these lines.

```
extern uint8 Tx_Data;
```

6. Locate the function CAN_1_SendMsg0() in the same file. Navigate to the code:

```
/* `#START MESSAGE_0_TRASMITTED` */

/* `#END` */
```

7. Type the following line of code between these lines.

```
CAN_1_TX_DATA_BYTE1(0) = Tx_Data;
```

8. Open the *main.c* file of the Receiver project by double-clicking on the filename inside the Workspace Explorer. Add the code from Code 2 to the *main.c* file.

Code 2. Receiver main.c Code

```
#include <device.h>

uint8 Rx_Data;

int main()
{
  CAN_1_Start();
  LCD_Char_1_Start();
  CyGlobalIntEnable;

  for(;;) /* do forever */
  {
    LCD_Char_1_ClearDisplay();
    LCD_Char_1_Position(0,0);
    LCD_Char_1_PrintNumber(Rx_Data);
  }
}
```

9. Open the *CAN_1_TX_RX_func.c* file for the Receiver project by double-clicking the file name inside the Workspace Explorer. This file is generated after the project is built.

10. Navigate to the lines:

```
/* `#START TX_RX_FUNCTION` */

/* `#END` */
```

11. Type the following line of code between these lines.

```
extern uint8 Rx_Data;
```

12. Locate the function CAN_1_ReceiveMsg0() in the same file. Navigate to the code:

```
/* `#START MESSAGE_0_RECEIVED` */

/* `#END` */
```

13. Type the following line of code between these lines.

```
Rx_Data = CAN_1_RX_DATA_BYTE1(0);
```

14. Build both projects and program each of them into the PSoC devices in the two kits. The program option is found in the menu Debug in PSoC Creator: **Debug** > **Program**.

3.4.1 Other Firmware Considerations

Consider the following points when writing firmware for a CAN Component:

- The CAN Component needs to be initialized and started in the *main.c* file before using it.
- Global interrupts should be enabled to service the interrupt requests from the CAN Component.
- To transmit a Full CAN message from a particular transmit mailbox, call the `CAN_TX_SendMsgx()` function, where x is replaced by the mailbox number and CAN_TX is the name given to the CAN Component in the PSoC Creator TopDesign schematic.
- The necessary functions for transmitting and receiving Full CAN messages are available in the *CAN_TX_TX_RX_func.c* file. This file is generated when the project is built.

Refer to the example projects provided with this application note to understand how the firmware is written to configure CAN in PSoC.

4 Implementing the Hardware

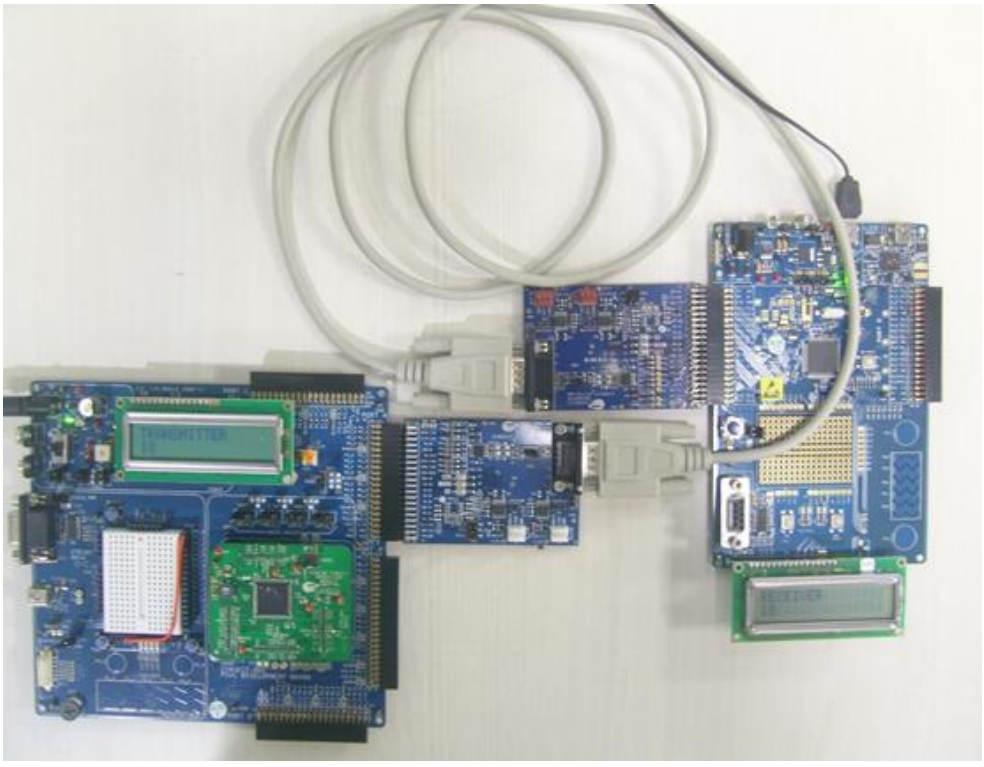
The PSoC outputs from the CAN Component are TX and RX. You must level-translate these outputs to obtain the CANH and CANL signals. The CY8CKIT-017 Expansion Board Kit is used as an external transceiver for level translation for the examples given in this application note. The CAN transceiver IC used in this kit is the TJA1050. The transceiver may be implemented with minimal connections as Figure 25 on page 20 shows, where the CY8CKIT-017 may be replaced by a TJA1050. A detailed figure is shown in Appendix B.

CAN requires only two wires for a CAN bus. In the CY8CKIT-017, a standard male-to-male DB9 connector may be used to connect between two CAN nodes, as Figure 21 shows. The cable length between any two CAN nodes in a CAN network should be restricted to the values shown in Table 2. These values are not mentioned in the CAN specification, but are typical values used in a design.

Table 2. Typical Cable Lengths for Different Baud Rates

Baud Rate (in bits per second)	Typical Cable Length (in meters)
1 Mbps	40
500 kbps	100
250 kbps	200
125 kbps	500
10 kbps	6000

Figure 21. CAN Physical Connections

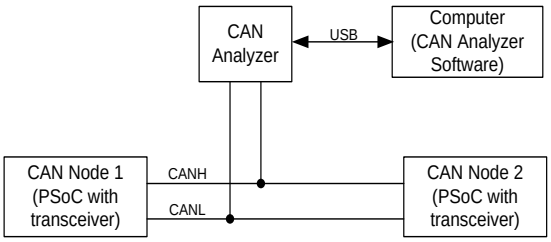


4.1 Working with a CAN Analyzer

A CAN analyzer is a device used to monitor the data traffic on a CAN bus, for debugging purposes. These devices are available from manufacturers such as Microchip and Peak-system. Figure 22 shows a block diagram illustrating the connection of a USB CAN analyzer to a CAN bus.

A CAN analyzer also requires software such as PCAN View to be installed on the computer system to which the CAN analyzer is connected. This software interprets the data received by the CAN analyzer and shows it in the application's interface.

Figure 22. USB CAN Analyzer Connected to a CAN Bus



5 Example Projects

There are four example projects included with this application note.

5.1 Examples 1 and 2: Simplex Communication

Examples 1 and 2, shown in Figure 25 on page 20, demonstrate a simplex communication between two CAN nodes.

Example 1 acts as a transmitter. It monitors key presses on a switch and communicates the number of key presses through CAN. Example 2 acts as the receiver. It receives the data and displays it on the character LCD display. The flowcharts shown in Figure 23 and Figure 24 on page 20 illustrate the program flow for these examples.

Follow these steps to set up example projects 1 and 2:

1. Build the system as Figure 25 on page 20 shows. Two CY8CKIT-001 DVKs and two CY8CKIT-017 EBKs may be used. A CY8CKIT-030 or CY8CKIT-050 DVK can also be used instead of a CY8CKIT-001 DVK. You can also use a PSoC 5LP FreeSoC2 Development Board along with CY8CKIT-026 for testing these projects.

   Note: The FreeSoC2 development kit does not have a crystal by default. You must populate a 24-MHz crystal on the FreeSoC2 board if you wish to test this project on the FreeSoC2 development kit.

2. Open the project files and make the correct pin assignments in the ***.cydwr*** file. Pin assignments are as shown in Table 3 and Table 4.

Table 3. Pin Usage for CAN Simplex Transmitter

Pin	CY8CKIT-001	CY8CKIT-030 / CY8CKIT-050	FreeSoC2 Kit
LCD_Tx	P2[6:0]	P2[6:0]	N/A
Data_In	P0[0]	P6[1]	P1[2]
RX	P3[4]	P3[4]	P3[4]
TX	P3[3]	P3[3]	P3[3]
Tx_En	P3[2]	P3[2]	P3[2]
UART_Tx	N/A	N/A	P4[0]

Table 4. Pin Usage for CAN Simplex Receiver

Pin	CY8CKIT-001	CY8CKIT-030 / CY8CKIT-050	FreeSoC2
LCD_Rx	P2[6:0]	P2[6:0]	N/A
RX	P3[4]	P3[4]	P3[4]
TX	P3[3]	P3[3]	P3[3]
Tx_En	P3[2]	P3[2]	P3[2]
UART_Tx	N/A	N/A	P4[0]

3. Build both projects. Program CAN_SimplexCommunication_Tx into the first PSoC device and CAN_SimplexCommunication_Rx into the second PSoC device.

   The CAN_SimplexCommunication_Tx project displays 'TRANSMITTER' on the first row of the LCD display. The second row shows the number of key presses registered on the Data_in pin.

   The CAN_SimplexCommunication_Rx project displays 'RECEIVER' on the first row of the LCD display. The second row displays the data sent by the first PSoC through CAN. This is the same as the number of key presses shown on the second row of the CAN_SimplexCommunication_Tx project.

Figure 23. Flowchart for Example 1 (Transmitter)

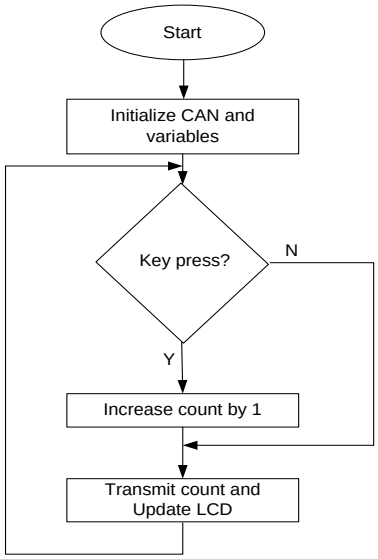


Figure 24. Flowchart for Example 2 (Receiver)

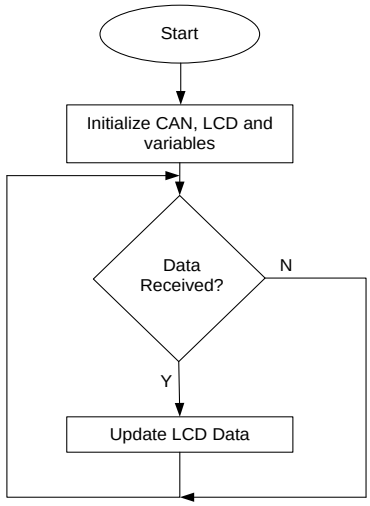
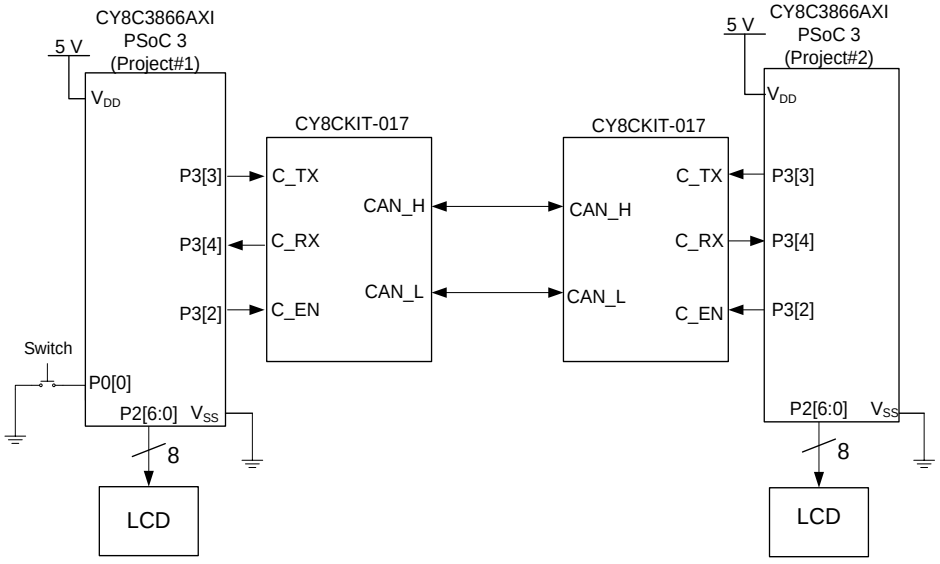


Figure 25. Physical Configuration for Example 1 and 2 (CY8CKIT-030)



5.2 Examples 3 and 4: RTR feature in CAN

Examples 3 and 4, shown in Figure 28 on page 22, demonstrate the RTR feature of CAN.

Example 3 is named Node1 and Example 4 is named Node 2. Node 1 monitors the ADC data input and displays the ADC value on the LCD display. In the event of an RTR request from Node 2, the current value of the ADC is transmitted to Node 2. Node 2 continuously checks for a key press on a switch and issues an RTR request to Node 1 in event of a key press. The ADC data value sent by Node 1 is received by Node 2 and is displayed on its LCD display. Flowcharts shown in Figure 26 and Figure 27 on page 22 illustrate the program flow for these examples.

Follow these steps to set up example projects 3 and 4:

1. Build the system as Figure 28 on 22 shows. Two CY8CKIT-001 DVKs and two CY8CKIT-017 EBKs may be used for this. A CY8CKIT-030 or CY8CKIT-050 DVK can also be used instead of a CY8CKIT-001 DVK,
2. Open the project files and make the correct pin assignments in the *.cydwr* file. Pin assignments are as shown in Table 5 and Table 6:

Table 5. Pin Usage for CAN_RTR_Node1

Pin	CY8CKIT-001	CY8CKIT-030 / CY8CKIT-050
LCD	P2[6:0]	P2[6:0]
ADC_In	P0[0]	P6[5]
RX	P3[4]	P3[4]
TX	P3[3]	P3[3]
Tx_En	P3[2]	P3[2]

Table 6. Pin Usage for CAN_RTR_Node2

Pin	CY8CKIT-001	CY8CKIT–030 / CY8CKIT-050
LCD	P2[6:0]	P2[6:0]
RTR_In	P0[0]	P6[1]
RX	P3[4]	P3[4]
TX	P3[3]	P3[3]
Tx_En	P3[2]	P3[2]

3. Build both projects. Program CAN_RTR_Node1 into the first PSoC and CAN_RTR_Node2 into the second PSoC.

   The CAN_RTR_Node1 project displays ‘Node1’ on the first row and the present value of the ADC output on the second row of the LCD display.

   The CAN_RTR_Node2 project displays ‘Node2’ on the first row of the LCD display. When the RTR_In key is pressed, the first-row shows ‘RTR Sent’ and the second row shows the value of the ADC output received from Node1 in response to the RTR.

Figure 26. Flowchart for Example 3 (Node 1)

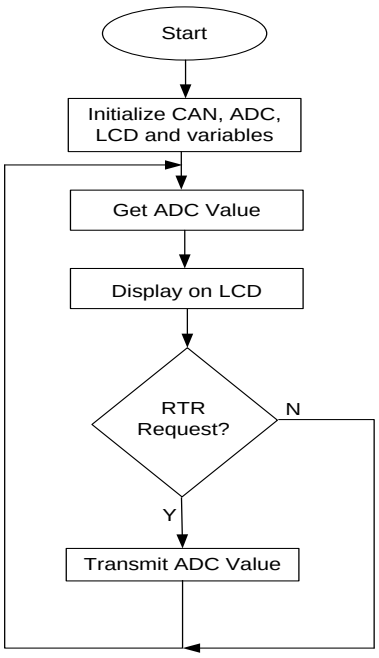


Figure 27. Flowchart for Example 4 (Node 2)

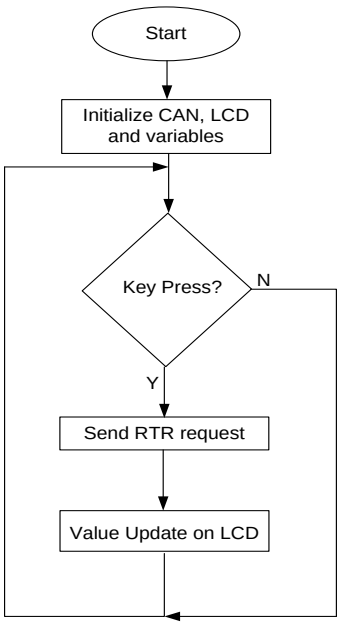
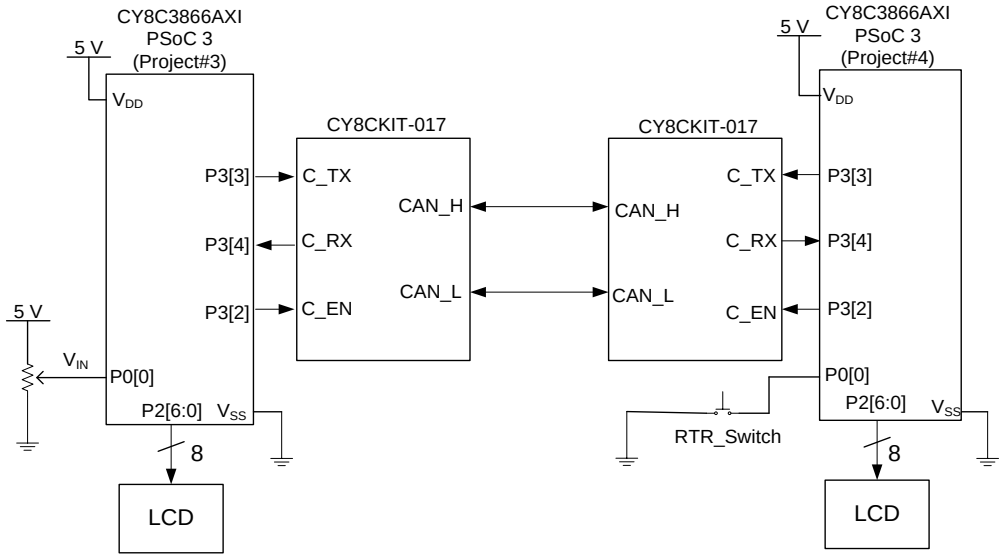


Figure 28. Physical Configurations for Examples 3 and 4



For proper operation of these projects, ensure that jumper JP2 of the CY8CKIT-017 is populated. In addition, jumper JP6 of the CY8CKIT-017 should be connected between V5_0 and V_{DD}. A standard male-to-male DB9 connector can be used to connect between two CAN nodes.

Two more example projects are available in PSoC Creator Example Projects.

6 Summary

CAN is a reliable serial communication protocol used mainly in automotive applications. The protocol allows bi-directional communication between devices and offers a flexible network.

Cypress PSoC 3 and PSoC 5LP, along with PSoC Creator, support the CAN 2.0a and CAN 2.0b specifications and offer a user-friendly interface and collection of APIs to set up the CAN network with ease. This application note guides you in implementing CAN with PSoC smoothly.

7 Related Resources

- CE95282 - CAN as Control Node with PSoC 3/5LP

  Demonstrates CAN Configuration, transmitting and receiving Full messages, and handling Tx and Rx errors.

- CE95283 - CAN as Remove Node with PSoC 3/5LP

  Demonstrates CAN Configuration, transmitting and receiving Full messages, and handling Tx and Rx errors.

- CE97311 - PSoC® 4 M: CAN Simplex Communication with CapSense®

  Demonstrates how to send and receive data over the CAN bus. he status of the CapSense® button is sent over the CAN to control LEDs at the CAN receiver.

- CE211027 - CAN Communication Between FM4-S6E2Gx Series and PSoC® 4 M-Series Using CY8CKIT- 026 CAN and LIN Shield Kit

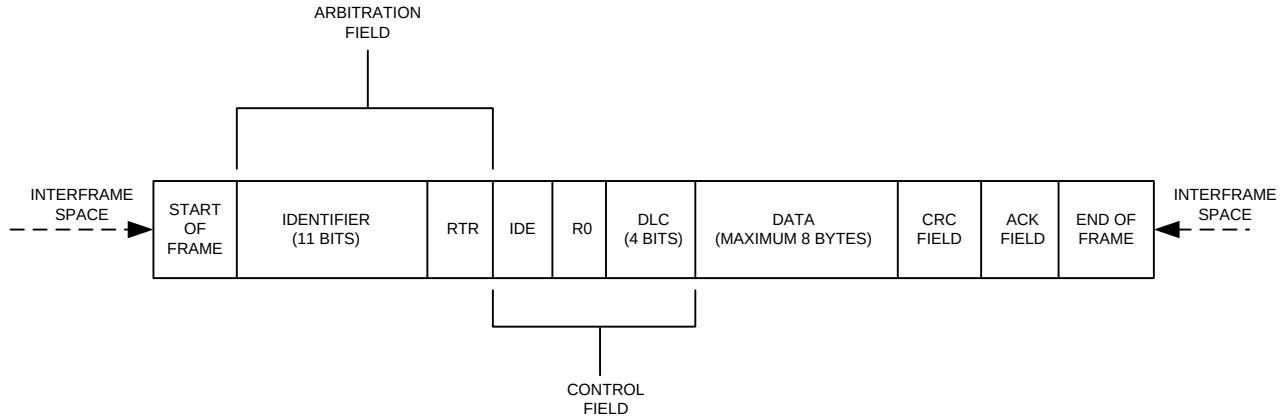
  Demonstrates CAN communication between FM4-S6E2Gx Series & PSoC® 4 M-Series using the CY8CKIT-026 CAN and LIN Shield Kit.

About the Author

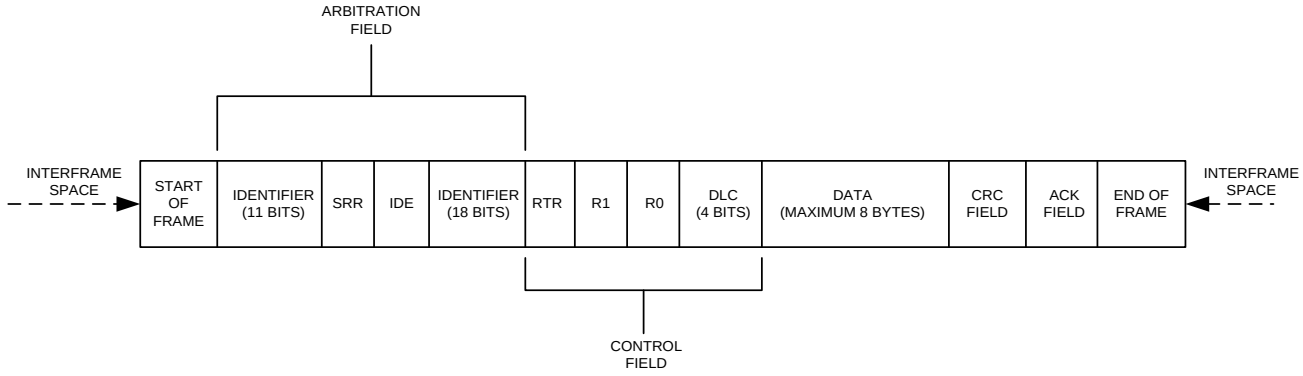
Name:	Ranjith M
Title:	Applications Engineer
Background:	Ranjith graduated from Government Engineering College, Thrissur with a Bachelor's Degree in Electronics and Communications Engineering.

Appendix A. Data Frames

STANDARD DATA FRAME

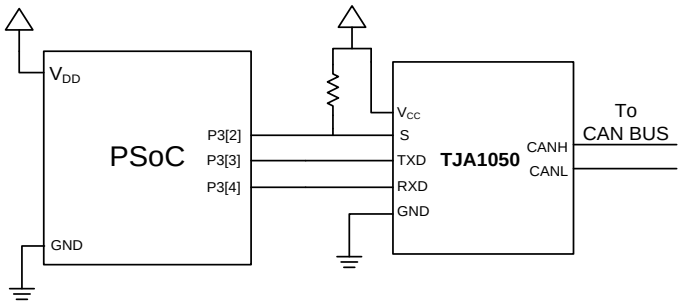


EXTENDED DATA FRAME



Appendix B. Interfacing PSoC with TJA1050 CAN Transceiver

Interfacing with TJA1050



Document History

Document Title: AN52701 - PSoC 3 and PSoC 5LP – Getting Started with Controller Area Network (CAN)

Document Number: 001-52701

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	2710279	ANUP	05/22/2009	New application note
*A	2763879	ANUP	09/15/2009	Updated Figure 2: Basic CAN Frame Schematic
*B	2947435	ANUP	06/08/2010	Changed document title. Updated to PSoC Creator Beta 4.1 and made the projects PSoC 5 compatible
*C	3032350	ANUP	09/17/2010	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.
*G	3756544	RNJT	11/12/2012	Changed title. Complete rewrite of application note.
*H	3857178	RNJT	01/03/2013	Major changes in the section CAN in PSoC.
*I	4031275	RNJT	06/17/2013	Added note in the CAN Timing Configuration section.
*J	5371117	RNJT	08/24/2016	Updated the example projects to PSoC Creator 3.3 CP3 Updated Introduction section to add FM4 and PSoC 4 references Updated example projects section to add FreeSoC2 board support Updated template
*K	5698515	AESATP12	06/13/2017	Updated logo and copyright
*L	5850993	AJYA	08/18/2017	Project updated to PSoC Creator 4.1 Updated Section PSoC Creator Projects Added references to the related code example

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

ARM® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6

Cypress Developer Community

Forums | WICED IOT Forums | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

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

© Cypress Semiconductor Corporation, 2009-2017. This document is the property of Cypress Semiconductor Corporation and its subsidiaries, including Spansion LLC ("Cypress"). This document, including any software or firmware included or referenced in this document ("Software"), is owned by Cypress under the intellectual property laws and treaties of the United States and other countries worldwide. Cypress reserves all rights under such laws and treaties and does not, except as specifically stated in this paragraph, grant any license under its patents, copyrights, trademarks, or other intellectual property rights. If the Software is not accompanied by a license agreement and you do not otherwise have a written agreement with Cypress governing the use of the Software, then Cypress hereby grants you a personal, non-exclusive, nontransferable license (without the right to sublicense) (1) under its copyright rights in the Software (a) for Software provided in source code form, to modify and reproduce the Software solely for use with Cypress hardware products, only internally within your organization, and (b) to distribute the Software in binary code form externally to end users (either directly or indirectly through resellers and distributors), solely for use on Cypress hardware product units, and (2) under those claims of Cypress's patents that are infringed by the Software (as provided by Cypress, unmodified) to make, use, distribute, and import the Software solely for use with Cypress hardware products. Any other use, reproduction, modification, translation, or compilation of the Software is prohibited.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS DOCUMENT OR ANY SOFTWARE OR ACCOMPANYING HARDWARE, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. To the extent permitted by applicable law, Cypress reserves the right to make changes to this document without further notice. Cypress does not assume any liability arising out of the application or use of any product or circuit described in this document. Any information provided in this document, including any sample design information or programming code, is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Cypress products are not designed, intended, or authorized for use as critical components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or system could cause personal injury, death, or property damage ("Unintended Uses"). A critical component is any component of a device or system whose failure to perform can be reasonably expected to cause the failure of the device or system, or to affect its safety or effectiveness. Cypress is not liable, in whole or in part, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from or related to all Unintended Uses of Cypress products. You shall indemnify and hold Cypress harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of Cypress products.

Cypress, the Cypress logo, Spansion, the Spansion logo, and combinations thereof, WICED, PSoC, CapSense, EZ-USB, F-RAM, and Traveo are trademarks or registered trademarks of Cypress in the United States and other countries. For a more complete list of Cypress trademarks, visit cypress.com. Other names and brands may be claimed as property of their respective owners.