

CAN Node on AWR1642

Raghunandan Kamath

ABSTRACT

AWR1642 has two CAN interfaces: one DCAN controller supporting bit rates of up to 1 Mbit/s, and compliant to the controller area network (CAN) 2.0B protocol specification, and one MCAN controller supporting bit rates of up to 10 Mbit/s. compliant to the controller area network (CAN) 2.0 part A, B protocol specification and ISO 11898-1, and the CAN FD V1.0 specification with up to 64 data bytes support.

This application report describes the procedure to configure a CAN node on AWR1642 and perform CAN communication over the network.

Contents

1	Introduction	2
2	Initializing the CAN Peripheral	5
3	Configuring Message Objects	11
Appendix A	ECO's on AWR1642BOOST	18

List of Figures

1	CAN Transceiver Interface	3
2	DCAN Module Block Diagram	4
3	MCAN Module Block Diagram	5
4	MCAN Rx Buffer/Rx FIFO Element	10
5	ECO[1] on AWR1642BOOST	20
6	ECO[2] on AWR1642BOOST	21
7	ECO on MMWAVE-DEVPACK	21
8	Nominal Bit-Timing 500KBit/s	22
9	Nominal Bit-Timing for 1MBit/s	23
10	Dataphase Bit-Timing for 5Mbit/s	24

List of Tables

1	Structure of Message Object	9
---	-----------------------------------	---

Trademarks

All trademarks are the property of their respective owners.

1 Introduction

1.1 Controller Area Network (DCAN)

The Controller Area Network is a high-integrity, serial, multi-master communication protocol for distributed real-time applications. This CAN module is implemented according to ISO 11898-1 and is suitable for industrial, automotive and general embedded communications.

1.1.1 Features

- Protocol
 - Supports CAN protocol version 2.0 part A, B
- Speed
 - Bit rates up to 1 MBit/s
- MailBox
 - Configurable Message objects
 - Individual identifier masks for each message object
 - Programmable FIFO mode for message objects
- High Speed MailBox Access
 - DMA access to Message RAM
- Debug
 - Suspend mode for debug support
 - Programmable loop-back modes for self-test operation
 - Direct access to Message RAM in test mode
 - Supports two interrupt lines - Level 0 and Level 1
- Others
 - Automatic message RAM initialization
 - Automatic bus on after bus-off state by a programmable 32-bit timer
 - CAN Rx/Tx pins configurable as general purpose IO pins
 - Software module reset

1.2 Modular Controller Area Network (MCAN)

The MCAN module supports both classic CAN with Flexible Data-Rate (CAN and CAN FD) specifications. CAN FD feature allow high throughput and increased payload per data frame. The classic CAN and CANFD devices can coexist on the same network without any conflict.

1.2.1 Features

- Conforms with CAN protocol 2.0 A, B and ISO 11898-1
- Full CAN FD support (up to 64 data bytes)
- AUTOSAR and SAE J1939 support
- Up to 32 dedicated transmit buffers
- Configurable transmit FIFO, up to 32 elements
- Configurable transmit queue, up to 32 elements
- Configurable transmit event FIFO, up to 32 elements
- Up to 64 dedicated receive buffers
- Two configurable receive FIFOs, up to 64 elements each
- Up to 128 filter elements
- Internal loopback mode for self-test
- Maskable interrupts, two interrupt lines

- Two clock domains: CAN clock and Host clock
- Parity/ECC support - message RAM single error correction and double error detection (SECEDED)
- Mechanism
- Local power-down and wakeup support
- Timestamp counter

1.3 CAN Transceiver Interface Diagram

Figure 1 shows a typical CAN transceiver interface.

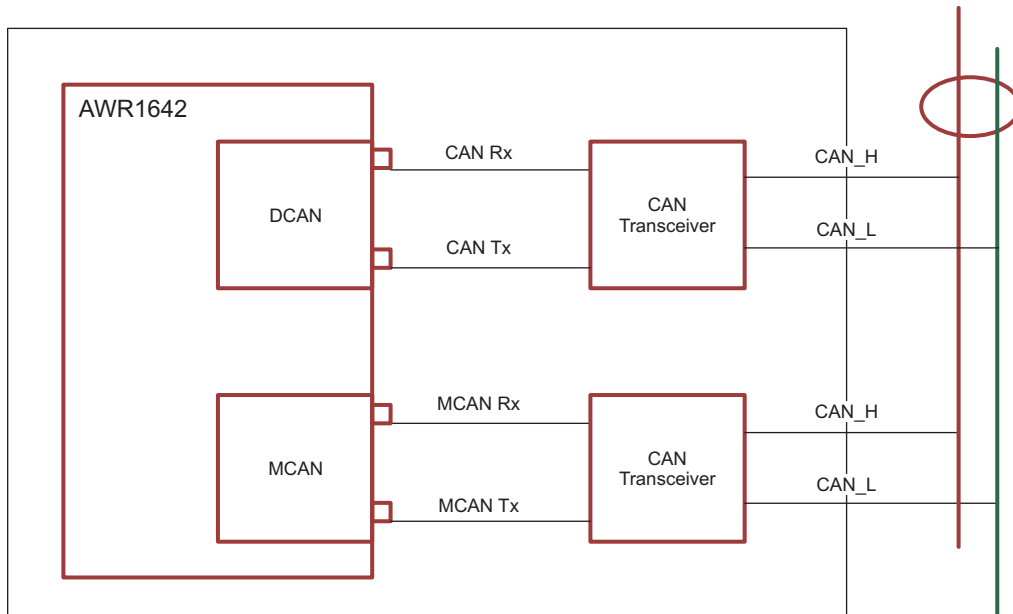


Figure 1. CAN Transceiver Interface

1.4 Block Diagram

1.4.1 DCAN Block Diagram

Figure 2 shows a typical DCAN module block diagram

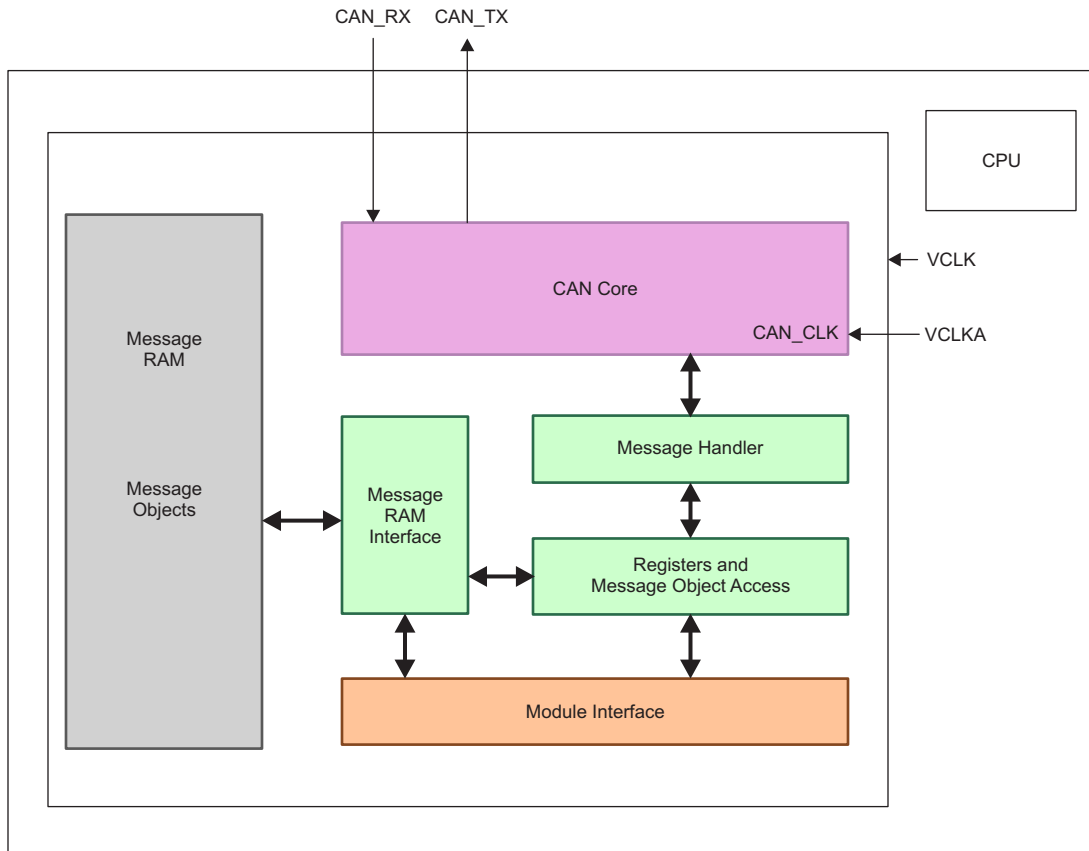


Figure 2. DCAN Module Block Diagram

1.4.2 MCAN Block Diagram

Figure 3 shows a typical MCAN module block diagram

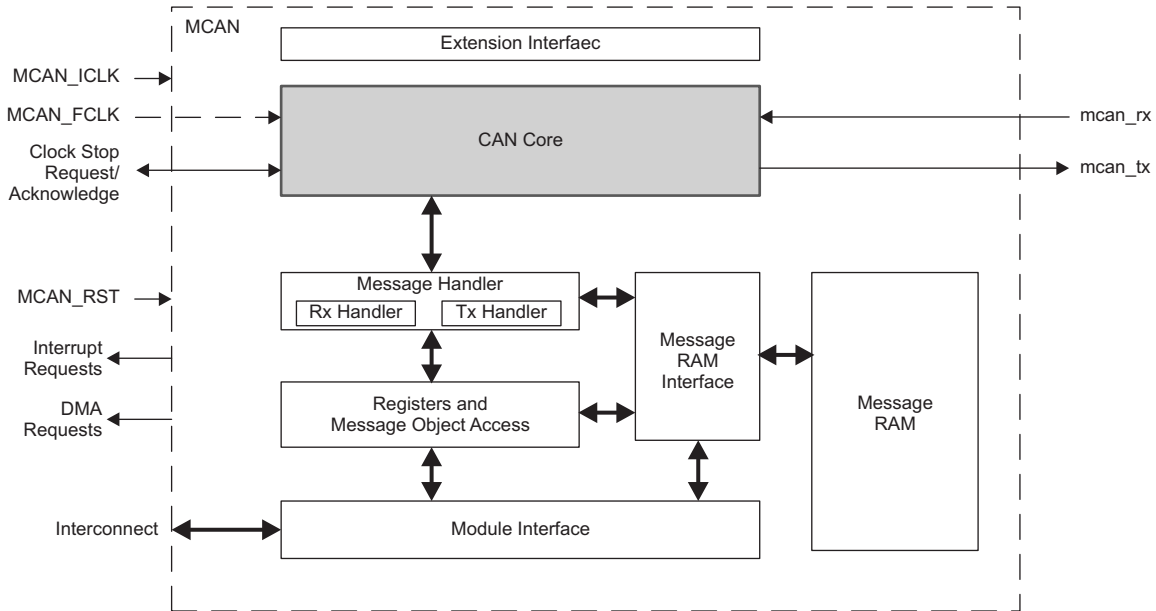


Figure 3. MCAN Module Block Diagram

2 Initializing the CAN Peripheral

Initialization of the CAN module involves the following common steps:

- CAN Clock source configuration
- CAN RAM initialization
- CAN register Configuration
- CAN Bit Timing Configuration

2.1 CAN Clock Source Configuration

The CAN module requires the configuration of the clock source and the clock divider values to be configured for its operation.

2.1.1 DCAN Clock Configuration

Below is the clock initialization sequence:

1. Gate the Clock by programming "0x1" to MSS_RCM: CLKGATE: DCANCLKGATE.
2. Set the divider value by programming divider value to MSS_RCM: CLKDIVCTL0: DCANCLKDIV.
3. Set the clock source by programming clock source to MSS_RCM: CLKSRCSEL0: DCANCLKSRCSEL.
4. Ungate the clock by programming "0x0" to MSS_RCM: CLKGATE: DCANCLKGATE.

```

SDK Code
/* Configure the divide value for DCAN source clock */
SOC_setPeripheralClock(socHandle, SOC_MODULE_DCAN, SOC_CLKSOURCE_VCLK, 9U, &errCode);
    
```

2.1.2 MCAN Clock Configuration

Below is the clock initialization sequence:

1. Gate the Clock by programming “0x1” to MSS_RCM: CLKGATE: FDCANCLKGATE.
2. Set the divider value by programming divider value to MSS_RCM: CLKDIVCTL0: FDCANCLKDIV.
3. Set the clock source by programming clock source to MSS_RCM: CLKSRCSEL0: FDCANCLKSRCSEL.
4. Ungate the clock by programming “0x0” to MSS_RCM: CLKGATE: FDCANCLKGATE.

```

SDK Code
/* Configure the divide value for MCAN source clock */
SOC_setPeripheralClock(socHandle, SOC_MODULE_MCAN, SOC_CLKSOURCE_VCLK, 4U, &errCode);

```

2.2 CAN RAM Initialization

The CAN RAM holds the CAN message objects (also called mail box). To begin, the RAM space should be initialized to zeros through the hardware by configuring the system registers.

2.2.1 DCAN RAM Initialization

Below is the sequence for RAM initialization:

1. Switch to memory initialization mode by programming the key “0xAD” to MSS_RCM: MEMINITSTART: MEMINITKEY.
2. Start the memory initialization for the DCAN memories by programming ‘0x1’ to MSS_RCM: MEMINITSTART: DCANMEM,
3. Wait on the confirmation of the memory initialization complete. Wait until MSS_RCM: MEMINITDONE: DCANMEM to become “0x1”.

```

SDK Code
/* Initialize peripheral memory */
SOC_initPeripheralRam(socHandle, SOC_MODULE_DCAN, &errCode);

```

2.3 CAN Register Configuration

2.3.1 DCAN Register Configuration

The following steps are involved in the DCAN configuration:

1. Enable the “Normal” operation by programming ‘0x1’ to DCAN_CTL: INIT.
2. Enable the “Configuration Change Enable” by programming ‘0x1’ to DCAN_CTL: CCE.
3. Enable the “Interrupt line 0” by programming ‘0x1’ to DCAN_CTL:IE0.
4. Enable the “Status Change Interrupt Enable” by programming ‘0x1’ to DCAN_CTL:SIE.
5. Enable the “Error Interrupt Enable” by programming ‘0x1’ to DCAN_CTL:EIE.
6. Enable the “Disable automatic retransmission” by programming ‘0x1’ to DCAN_CTL:DAR.
7. Enable the “Test mode enable” by programming ‘0x1’ to DCAN_CTL:TEST.
8. Enable the “Interruption debug support enable” by programming ‘0x1’ to DCAN_CTL:IDS.
9. Enable the “Auto-Bus-On enable” by programming ‘0x1’ to DCAN_CTL:ABO.
10. Enable the “Parity On/Off bit” by programming ‘0xF’ to DCAN_CTL:PMD.
11. Enable the “Internal init state while debug access” by programming ‘0xF’ to DCAN_CTL: INITDBG.
12. Enable the “Interrupt line 1” by programming ‘0x1’ to DCAN_CTL:IE1.

13. Enable the “Enable DMA request line for IF1” by programming ‘0x1’ to DCAN_CTL:DE1.
14. Enable the “Enable DMA request line for IF2” by programming ‘0x1’ to DCAN_CTL:DE2.
15. Enable the “Enable DMA request line for IF3” by programming ‘0x1’ to DCAN_CTL:DE3.
16. Program the “Auto BUS on Timer register ” with the timer value in DCAN_ABOTR:ABO_TIME .

```

SDK Code
/* Initialize the DCAN parameters that need to be specified by the application */
DCANAppInitParams(&appDcanCfgParams,
                  &appDcanTxCfgParams,
                  &appDcanRxCfgParams,
                  &appDcanTxData);

/* Initialize the CAN driver */
canHandle = CAN_init(&appDcanCfgParams, &errCode);

```

2.3.2 MCAN Register Configuration

The following steps are involved in the MCAN configuration:

1. Check whether or not the memory initialization is complete. Check if the value of MCANSS_STAT: MMI_DONE is set to 0x1.
2. Initiate a soft reset by programming value “0x1” to MCANSS_CTRL: RESET.
3. Put the MCAN in “Software Initialization Mode” by programming ‘0x1’ to MCAN_CCCR: INIT.
4. Configure MCAN wakeup and clock stop controls.
 - a. Enable “Wakeup Request Enable” programming value “0x1” to MCANSS_CTRL: WAKEUPREGEN.
 - b. Enable “Automatic Wakeup Enable” programming value “0x1” to MCANSS_CTRL: AUTOWAKEUP.
 - c. Disable “Emulation Enable” programming value “0x0” to MCANSS_CTRL: EMUEN.
 - d. Disable “Emulation Fast Ack” programming value “0x0” to MCANSS_CTRL: EMUFACK.
 - e. Disable “Clock Fast Ack” programming value “0x0” to MCANSS_CTRL: CLKFACK.
5. Configure MCAN mode (FD vs Classic CAN operation) and controls.
 - a. Enable the “FD Operation Enable” by programming value ‘0x1’ to MCAN_CCCR: FDOE.
 - b. Enable the “Bit Rate Switch Enable” by programming value ‘0x1’ to MCAN_CCCR: BRSE.
 - c. Disable the “Transmit Pause” by programming value ‘0x0’ to MCAN_CCCR: TXP.
 - d. Disable the “Edge Filtering during Bus Integration” by programming value ‘0x0’ to MCAN_CCCR: EFBI.
 - e. Disable the “Protocol Exception Handling Disable” by programming value ‘0x0’ to MCAN_CCCR: PXHD.
 - f. Enable the “Disable Automatic Retransmission” by programming value ‘0x1’ to MCAN_CCCR: DAR.
6. Configure transceiver delay compensation.
 - a. Configure the “Transmitter Delay Compensation Filter Window Length” by programming the length to MCAN_TDCR: TDCF.
 - b. Configure the “Transmitter Delay Compensation Offset” by programming the length to MCAN_TDCR: TDCO.
7. Configure MSG RAM watchdog counter preload value by programming the preload value to MCAN_RWD: WDC.

8. Enable the “Transceiver Delay Compensation” by programming the MCAN_DBTP: TDC.

```

SDK Code
MCANAppInitParams (&mcanCfgParams);

/* Initialize the CANFD driver */
canHandle = CANFD_init(&mcanCfgParams, &errCode);

```

2.4 CAN Bit Timing Configuration

2.4.1 DCAN Bit Timing Configuration

The following steps are used for the DCAN bit timing configuration:

1. Enable the “Init” operation by programming ‘0x1’ to DCAN_CTL: INIT.
2. Enable the “Configuration Change Enable” by programming ‘0x1’ to DCAN_CTL: CCE.
3. Program the DCAN bit time value by programming the BRP Extension Register DCAN_BTR with BRP value.
4. Disable the “Configuration Change Enable” by programming ‘0x0’ to DCAN_CTL: CCE.
5. Enable the “Normal” operation by programming ‘0x0’ to DCAN_CTL: INIT.

```

SDK Code
/* Set the desired bit rate based on input clock */
retVal = DCANAppCalcBitTimeParams(DCAN_APP_INPUT_CLK / 1000000,
                                   DCAN_APP_BIT_RATE / 1000,
                                   DCAN_APP_SAMP_PT,
                                   DCAN_APP_PROP_DELAY,
                                   &appDcanBitTimeParams);

```

2.4.2 MCAN Bit Timing Configuration

The following steps are used for the MCAN bit timing configuration:

1. Enable the “Configuration Change Enable” by programming ‘0x1’ to MCAN_CCCR: CCE.
2. Configure the nominal bitrate values.
 - a. Program the “Nominal Resynchronization Jump Width” with value to MCAN_NBTP: NSJW.
 - b. Program the “Nominal Baud Rate Prescaler” with value to MCAN_NBTP: NBRP.
 - c. Program the “Nominal Time segment before sample point” with value to MCAN_NBTP: NTSEG1.
 - d. Program the “Nominal Time segment after sample point” with value to MCAN_NBTP: NTSEG2.
3. Configure the dataphase bitrate values.
 - a. Program the “Data Resynchronization Jump Width” with value to MCAN_ MCAN_DBTP: DSJW.
 - b. Program the “Data time segment after sample point” with value to MCAN_ MCAN_DBTP: DTSEG2.
 - c. Program the “Data time segment before sample point” with value to MCAN_ MCAN_DBTP: DTSEG1.
 - d. Program the “Data Baud Rate Prescaler” with value to MCAN_ MCAN_DBTP: DBRP.
 - e. Program the “Data Baud Rate Prescaler” with value to MCAN_ MCAN_DBTP: DBRP.

4. Disable the “Configuration Change Enable” by programming ‘0x0’ to MCAN_CCCR: CCE

```

SDK Code
mcanBitTimingParams.nomBrp      = 0x4U;
mcanBitTimingParams.nomPropSeg  = 0x8U;
mcanBitTimingParams.nomPseg1    = 0x6U;
mcanBitTimingParams.nomPseg2    = 0x5U;
mcanBitTimingParams.nomSjw      = 0x1U;

mcanBitTimingParams.dataBrp     = 0x1U;
mcanBitTimingParams.dataPropSeg = 0x2U;
mcanBitTimingParams.dataPseg1   = 0x2U;
mcanBitTimingParams.dataPseg2   = 0x3U;
mcanBitTimingParams.dataSjw     = 0x1U;

/* Configure the CAN driver */
retVal = CANFD_configBitTime (canHandle, &mcanBitTimingParams, &errCode);
if (retVal < 0)
{
    System_printf ("Error: CANFD Module configure bit time failed [Error code %d]\n",
errCode);
    return -1;
}

```

2.5 CAN Messaging

2.5.1 DCAN Message Objects

The structure of the CAN message object (also known as the CAN Mailbox) is shown in [Table 1](#).

Table 1. Structure of Message Object

Message Object												
UMask	Msk[28:0]	MXtd	MDir	EoB	Unused	NewDat	MsgLst	RxIE	TxIE	IntPnd	RmtEn	TxRqst
MsgVal	ID[28:0]	Xtd	Dir	DLC[3:0]	Data 0	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7

It has three main components:

- ID: Message ID (with Extended Message ID, Xtd)
- DLC: Message Length
- Datax : Message Data (up to 8 Bytes)

Below are the control bits:

- Message Valid (MsgVal): Indicates that the message is valid.
- Direction (Dir): Indicates whether the mailbox is to transmit or receive
- Identifier Mask (Msk): Indicates the bits that are to be masked in ID
- Mask Extended Identifier (MXtd): Indicates the Mask bits for extended ID Xtd
- Mask Message Direction (MDir): Indicates whether or not the Dir is supposed to be masked
- Use Acceptance Mask (UMask): Indicates whether or not Mask bits are to be used
- End of Block (EoB): Indicates the last message of the FIFO buffer
- Transmit Interrupt Enable (TxIE): Provides an interrupt after transmission of the data.
- Receive Interrupt Enable (RxIE): Provides an interrupt after reception of the data.
- Interrupt Pending (IntPnd) : Indicates that interrupt is pending for this message object.
- New Data (NewDat): Indicates that new data is available in the message object.

- Transmit Request (TxRqst): Requests the transmission of the data
- Remote Enable (RmtEn): Enables the message object to accept remote frame

2.5.2 MCAN FIFO and Buffer Elements

2.5.2.1 RX Buffer /FIFO

- MCAN has upto 64 Rx buffers and 2 Tx FIFO elements that can be configured in the RAM.
- Each Rx FIFO section can be configured to store up to 64 received messages.

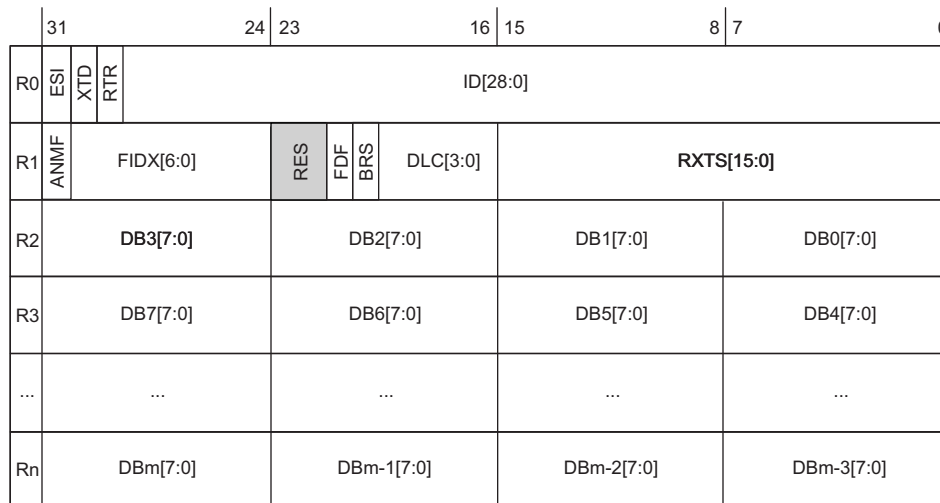


Figure 4. MCAN Rx Buffer/Rx FIFO Element

It has three main components:

- ID: Message ID
- DLC: Message Length
- DBx : Message Data (up to 64 Bytes)

The following are Control bits:

- ESI: Indicates the error state
- XTD: Indicates the extended identifier
- RTR: Indicates remote transmission request
- ID: Indicates the message identifier
- ANMF: Indicates the accepted non-matching frame
- FIDX: Indicates the filter index
- FDL: Indicates if the format is flexible datarate
- BRS: Indicates the bitrate switch

3 Configuring Message Objects

3.1 DCAN Messaging

This section describes the possible message object configurations for DCAN communication.

3.1.1 Configuring DCAN Message Object to Transmit

The following steps are used in configuring the DCAN message object for Tx:

1. Configure the identifier mask, Mask message Direction and Mask extended identifier in the IFx Mask register.
2. Configure the message identifier, message direction(Tx = 0x1) and Message Valid bit (ID = 1) in the IFx Arbitration register.
3. Configure the Use acceptance mask, transmit interrupt enable, set the EoB to define the FIFO buffer or single message and remote enable bit in the IFx Message control register.
4. Configure command Access for TX in the IFx Command register.
5. Now transfer from Interface register to RAM.

3.1.2 Configuring DCAN Message Object to Receive

The following steps are used in configuring the DCAN message object for Rx:

1. Configure the identifier mask, Mask message Direction and Mask extended identifier in the IFx Mask register.
2. Configure the message identifier, message direction(Tx = 0x0) and Message Valid bit (ID = 1) in the IFx Arbitration register.
3. Configure the Use acceptance mask, receive interrupt enable, set the EoB to define the FIFO buffer or single message and remote enable bit in the IFx Message control register.
4. Configure command Access for RX in the IFx Command register.
5. Now transfer from Interface register to RAM

```

SDK Code
/* Setup the transmit message object */
txMsgObjHandle = CAN_createMsgObject (canHandle, DCAN_MSG_OBJ_1, &appDcanTxCfgParams,
&errCode);
if (txMsgObjHandle == NULL)
{
    System_printf ("Error: CAN create Tx message object failed [Error code %d]\n", errCode);
    return -1;
}

/* Setup the receive message object */
rxMsgObjHandle = CAN_createMsgObject (canHandle, DCAN_MSG_OBJ_2, &appDcanRxCfgParams,
&errCode);
if (rxMsgObjHandle == NULL)
{
    System_printf ("Error: CAN create Rx message object failed [Error code %d]\n", errCode);
    return -1;
}

```

3.2 MCAN Messaging

This section describes the possible message object configurations for MCAN communication.

3.2.1 Configuring MCAN Message Object to Transmit

The following steps are used for configuring the message objects in MCAN:

1. Allocate memory for the CAN Message Object.
2. Store the message object handle for book keeping.
3. Enable the Transmission interrupt enable by setting MCAN:TXBTIE.
4. Enable the Cancellation Finished Interrupt Enable by setting MCAN:TXBCIE

3.2.2 Configuring MCAN Message Object to Receive

1. Allocate memory for the CAN Message Object.
2. Store the message object handle for book keeping.
3. Add the standard/Extended message ID filter to message RAM.

3.3 CAN Transmit/Receive Operation

3.3.1 DCAN Transmit operation

```

int32_t CAN_transmitData(CAN_MsgObjHandle handle, const CAN_DCANData* data, int32_t* errCode)
{
    CAN_MessageObject* ptrCanMsgObj;
    CAN_DriverMCB* ptrCanMCB;
    int32_t retVal = 0;
    uint32_t baseAddr;
    uint32_t ifRegNum;

    /* Get the message object pointer */
    ptrCanMsgObj = (CAN_MessageObject*)handle;
    if ((ptrCanMsgObj == NULL) || (data == NULL))
    {
        *errCode = CAN_EINVAL;
        retVal = MINUS_ONE;
    }
    else
    {
        /* Get the pointer to the CAN Driver Block */
        ptrCanMCB = (CAN_DriverMCB*)ptrCanMsgObj->ptrDriverMCB;

        if (ptrCanMCB == NULL)
        {
            *errCode = CAN_EINVAL;
            retVal = MINUS_ONE;
        }
        else
        {
            baseAddr = ptrCanMCB->hwCfg.regBaseAddress;

            ifRegNum = CAN_DCANIfRegNum_1;

            *errCode = DCANTransmitData(baseAddr, ptrCanMsgObj->
            >msgObjectNum, ifRegNum, data, 100U);
            if (*errCode != CAN_EOK)
            {
                retVal = MINUS_ONE;
            }
            else
            {

```

```

        /* Increment the stats */
        ptrCanMsgObj->messageProcessed++;
    }
}
return retVal;
}

```

3.3.2 DCAN Receive Operation

If the receive interrupts are enabled and a callback function has been registered when creating the receive message object, the driver notifies the application when the data has arrived. The application needs to call the CAN_getData function to read the received data.

```

int32_t CAN_getData(CAN_MsgObjHandle handle, CAN_DCANData* data, int32_t* errCode)
{
    CAN_MessageObject* ptrCanMsgObj;
    CAN_DriverMCB* ptrCanMCB;
    int32_t retVal = 0;
    uint32_t baseAddr;
    uint32_t ifRegNum;

    /* Get the message object pointer */
    ptrCanMsgObj = (CAN_MessageObject*)handle;
    if ((ptrCanMsgObj == NULL) || (data == NULL))
    {
        *errCode = CAN_EINVAL;
        retVal = MINUS_ONE;
    }
    else
    {
        /* Get the pointer to the CAN Driver Block */
        ptrCanMCB = (CAN_DriverMCB*)ptrCanMsgObj->ptrDriverMCB;

        if (ptrCanMCB == NULL)
        {
            *errCode = CAN_EINVAL;
            retVal = MINUS_ONE;
        }
        else
        {
            baseAddr = ptrCanMCB->hwCfg.regBaseAddress;

            ifRegNum = CAN_DCANIfRegNum_2;

            /* Read the pending data */
            *errCode = DCANGetData(baseAddr, ptrCanMsgObj->msgObjectNum, ifRegNum, data, 100U);
            if (*errCode == CAN_EOK)
            {
                /* Increment the stats */
                ptrCanMsgObj->messageProcessed++;
                do
                {
                    /* Processing loop? */
                    if (DCANIsIfRegBusy(baseAddr, ifRegNum) == 0U)
                    {
                        break;
                    }
                }
                while (1);

                /* Clear the interrupts for message object */
                DCANIntrClearStatus(baseAddr, ptrCanMsgObj->msgObjectNum, ifRegNum);
            }
        }
    }
}

```

```

        /* Wait for the operation to complete */
        do
        {
            /* Processing loop? */
            if (DCANIsIfRegBusy(baseAddr, ifRegNum) == 0U)
            {
                break;
            }
        }
        while (1);
    }
}
return retVal;
}

```

3.4 MCAN Transmit Operation

```

int32_t CANFD_transmitData(CANFD_MsgObjHandle handle, uint32_t id, CANFD_MCANFrameType
frameType, uint32_t dataLength, const uint8_t* data, int32_t* errCode)
{
    CANFD_MessageObject*   ptrCanMsgObj;
    CANFD_DriverMCB*       ptrCanFdmcb;
    int32_t                 retVal = 0;
    uint32_t                baseAddr;
    MCAN_TxBufElement      txBuffElem;
    uint32_t                index;

    /* Get the message object pointer */
    ptrCanMsgObj = (CANFD_MessageObject*)handle;
    if ((ptrCanMsgObj == NULL) || (data == NULL) || (dataLength < 1U) || (dataLength > 64U))
    {
        *errCode = CANFD_EINVAL;
        retVal = MINUS_ONE;
    }
    else
    {
        /* Get the pointer to the CAN Driver Block */
        ptrCanFdmcb = (CANFD_DriverMCB*)ptrCanMsgObj->ptrDriverMCB;

        if (ptrCanFdmcb == NULL)
        {
            *errCode = CANFD_EINVAL;
            retVal = MINUS_ONE;
        }
        else
        {
            baseAddr = ptrCanFdmcb->hwCfg.regBaseAddress;
            /* Check for pending messages */
            index = (uint32_t)1U << ptrCanMsgObj->txElement;
            if (index == (MCAN_getTxBufReqPend(baseAddr) & index))
            {
                *errCode = CANFD_EINUSE;
                retVal = MINUS_ONE;
            }
            else
            {
                /* populate the Tx buffer message element */
                txBuffElem.rtr = 0;
                txBuffElem.esi = 0;
                txBuffElem.efc = 0;
                txBuffElem.mm = 0;
            }
        }
    }
}

```

```

        if(frameType == CANFD_MCANFrameType_CLASSIC)
        {
            txBuffElem.brs = 0;
            txBuffElem.fdf = 0;
        }
        else
        {
            txBuffElem.brs = 1U;
            txBuffElem.fdf = 1U;
        }
        /* Populate the Id */
        if (ptrCanMsgObj->msgIdType == CANFD_MCANXidType_11_BIT)
        {
            txBuffElem.xtd = CANFD_MCANXidType_11_BIT;
            txBuffElem.id = (id & STD_MSGID_MASK) << STD_MSGID_SHIFT;
        }
        else
        {
            txBuffElem.xtd = CANFD_MCANXidType_29_BIT;
            txBuffElem.id = id & XTD_MSGID_MASK;
        }

        /* Copy the data */
        memcpy ((void*)&txBuffElem.data, data, dataLength);
        memcpy ((void*)&txBuffElem.data, data, dataLength);

        /* Compute the DLC value */
        for(index = 0U ; index < 16U ; index++)
        {
            if(dataLength <= ptrCanFdMCB->mcanDataSize[index])
            {
                txBuffElem.dlc = index;
                break;
            }
        }
        txBuffElem.dlc = index;
        if (index == 16)
        {
            *errCode = CANFD_EINVAL;
            retVal = MINUS_ONE;
        }
        else
        {
            /* Pad the unused data in payload */
            for(index = dataLength; index < ptrCanFdMCB-
>mcanDataSize[txBuffElem.dlc]; index++)
            {
                txBuffElem.data[index] = (uint8_t)0xCCU;
            }

            MCAN_writeMsgRam(baseAddr, MCAN_MemType_BUF, ptrCanMsgObj-
>txElement, &txBuffElem);
            MCAN_txBufAddReq(baseAddr, ptrCanMsgObj->txElement);

            /* Increment the stats */
            ptrCanMsgObj->messageProcessed++;
        }
    }
}
return retVal;
}

```

3.5 MCAN Receive Operation

If the receive interrupts are enabled using and a callback function has been registered when initializing the CANFD driver, the driver notifies the application when the data has arrived. The application needs to call the CANFD_getData function to read the received data.

```

int32_t CANFD_getData(CANFD_MsgObjHandle handle, uint32_t* id, CANFD_MCANFrameType*
ptrFrameType, CANFD_MCANIdType* idType, uint32_t* ptrDataLength, uint8_t* data, int32_t*
errCode)
{
    CANFD_MessageObject*   ptrCanMsgObj;
    CANFD_DriverMCB*       ptrCanFdMCB;
    int32_t                 retVal = 0;
    uint32_t               baseAddr;
    MCAN_RxBufElement      rxBuffElem;

    /* Get the message object pointer */
    ptrCanMsgObj = (CANFD_MessageObject*)handle;
    if ((ptrCanMsgObj == NULL) || (id == NULL) || (ptrDataLength == NULL) || (data == NULL))
    {
        *errCode = CANFD_EINVAL;
        retVal = MINUS_ONE;
    }
    else
    {
        /* Get the pointer to the CAN Driver Block */
        ptrCanFdMCB = (CANFD_DriverMCB*)ptrCanMsgObj->ptrDriverMCB;

        if (ptrCanFdMCB == NULL)
        {
            *errCode = CANFD_EINVAL;
            retVal = MINUS_ONE;
        }
        else
        {
            baseAddr = ptrCanFdMCB->hwCfg.regBaseAddress;

            /* Read the pending data */
            MCAN_readMsgRam(baseAddr, MCAN_MemType_BUF, ptrCanMsgObj-
>rxElement, 0, &rxBuffElem);

            /* Get the data length from DLC */
            *ptrDataLength = ptrCanFdMCB->mcanDataSize[rxBuffElem.dlc];

            /* Get the message Identifier */
            if (rxBuffElem.xtd == 1U)
            {
                /* Received frame with Extended ID */
                *id = (uint32_t)(rxBuffElem.id);
                *idType = CANFD_MCANIdType_29_BIT;
            }
            else
            {
                /* Received frame with Standard ID */
                *id = (uint32_t)((rxBuffElem.id >> 18U) & 0x7FFU);
                *idType = CANFD_MCANIdType_11_BIT;
            }

            /* Get the frame type */
            if (rxBuffElem.fdf == 1U)
            {
                /* FD frame Received */
                *ptrFrameType = CANFD_MCANFrameType_FD;
            }
            else
        }
    }
}
    
```



```
    {
        /* Classic frame Received */
        *ptrFrameType = CANFD_MCANFrameType_CLASSIC;
    }

    /* Copy the data */
    memcpy ((void *)data, rxBuffElem.data, *ptrDataLength);

    /* Increment the stats */
    ptrCanMsgObj->messageProcessed++;
}
return retVal;
}
```

ECO's on AWR1642BOOST

A.1 Driver Configuration

The driver runtime configurations are set during the initialization. The sample configuration below can be included in the driver.

```

static void MCANAppInitParams(CANFD_MCANInitParams* mcanCfgParams)
{
    /*Intialize MCAN Config Params*/
    memset (mcanCfgParams, sizeof (CANFD_MCANInitParams), 0);

    mcanCfgParams->fdMode           = 0x1U;
    mcanCfgParams->brsEnable        = 0x1U;
    mcanCfgParams->txpEnable        = 0x0U;
    mcanCfgParams->efbi             = 0x0U;
    mcanCfgParams->pxhddisable      = 0x0U;
    mcanCfgParams->darEnable        = 0x1U;
    mcanCfgParams->wkupReqEnable     = 0x1U;
    mcanCfgParams->autoWkupEnable   = 0x1U;
    mcanCfgParams->emulationEnable  = 0x0U;
    mcanCfgParams->emulationFAck    = 0x0U;
    mcanCfgParams->clkStopFAck     = 0x0U;
    mcanCfgParams->wdcPreload       = 0x0U;
    mcanCfgParams->tdcEnable        = 0x1U;
    mcanCfgParams->tdcConfig.tdcf   = 0U;
    mcanCfgParams->tdcConfig.tdco  = 8U;
    mcanCfgParams->monEnable        = 0x0U;
    mcanCfgParams->asmEnable        = 0x0U;
    mcanCfgParams->tsPrescalar      = 0x0U;
    mcanCfgParams->tsSelect         = 0x0U;
    mcanCfgParams->timeoutSelect    = CANFD_MCANTimeOutSelect_CONT;
    mcanCfgParams->timeoutPreload  = 0x0U;
    mcanCfgParams->timeoutCntEnable= 0x0U;
    mcanCfgParams->filterConfig.rrfe      = 0x1U;
    mcanCfgParams->filterConfig.rrfs     = 0x1U;
    mcanCfgParams->filterConfig.anfe     = 0x1U;
    mcanCfgParams->filterConfig.anfs     = 0x1U;
    mcanCfgParams->msgRAMConfig.lss      = 127U;
    mcanCfgParams->msgRAMConfig.lse     = 64U;
    mcanCfgParams->msgRAMConfig.txBufNum = 32U;
    mcanCfgParams->msgRAMConfig.txFIFOSize = 0U;
    mcanCfgParams->msgRAMConfig.txBufMode = 0U;
    mcanCfgParams->msgRAMConfig.txEventFIFOSize      = 0U;
    mcanCfgParams->msgRAMConfig.txEventFIFOWaterMark = 0U;
    mcanCfgParams->msgRAMConfig.rxFIFO0size         = 0U;
    mcanCfgParams->msgRAMConfig.rxFIFO0OpMode       = 0U;
    mcanCfgParams->msgRAMConfig.rxFIFO0waterMark    = 0U;
    mcanCfgParams->msgRAMConfig.rxFIFO1size         = 64U;
    mcanCfgParams->msgRAMConfig.rxFIFO1waterMark    = 64U;
    mcanCfgParams->msgRAMConfig.rxFIFO1OpMode       = 64U;
    mcanCfgParams->eccConfig.enable                 = 1;
    mcanCfgParams->eccConfig.enableChk              = 1;
    mcanCfgParams->eccConfig.enableRdModWr         = 1;
    mcanCfgParams->errInterruptEnable              = 1U;

```

```

mcanCfgParams->dataInterruptEnable = 1U;
mcanCfgParams->appErrCallBack      = MCANAppErrStatusCallback;
mcanCfgParams->appDataCallBack     = MCANAppCallback;
}

```

A.2 Error and Status Interrupts

The error and status need to be registered during the initialization of the driver. The sample below can be used for callback definition.

The application can monitor the ECC error, Bus off error and Protocol Errors by enabling the error interrupts. The driver calls the registered callback function to indicate which error fields caused the interrupt. It is up to the application to take appropriate action.

```

static void MCANAppErrStatusCallback(CANFD_Handle handle, CANFD_Reason reason,
CANFD_ErrStatusResp* errStatusResp)
{
    /*Record the error count */
    ...
}

```

A.3 CAN-FD SET/GET Functions

The runtime query and configuration of various statistics, ecc, error counter, diagnostic are supported via get/set function.

```

optionTLV.type = CANFD_Option_MCAN_MSG_OBJECT_STATS;
optionTLV.length = sizeof(CANFD_MCANMsgObjectStats);
optionTLV.value = (void*) &msgObjStats;

retVal = CANFD_getOptions(canHandle, &optionTLV, &errCode);
if (retVal < 0)
{
    System_printf ("Error: CANFD get stats failed [Error code %d]\n", errCode);
    return -1;
}

```

```

optionTLV.type = CANFD_Option_MCAN_MODE;
optionTLV.length = sizeof(uint8_t);
value = 1U;
optionTLV.value = (void*) &value;

retVal = CANFD_setOptions(canHandle, &optionTLV, &errCode);
if (retVal < 0)
{
    System_printf ("Error: CANFD set option Mode -
SW INIT failed [Error code %d]\n", errCode);
    return -1;
}

```

A.4 ECO's on AWR1642BOOST

There are certain ECO's required on the [AWR1642BOOST](#) for the external MCAN communication to be functional. These modifications are required to be performed on EVM.

Runtime query and configuration of various statistics, error counters, ECC diagnostics, power down, and so forth, have been provided via the get/set_ Options.

1. Mount 0 Ω on R11 and R12.

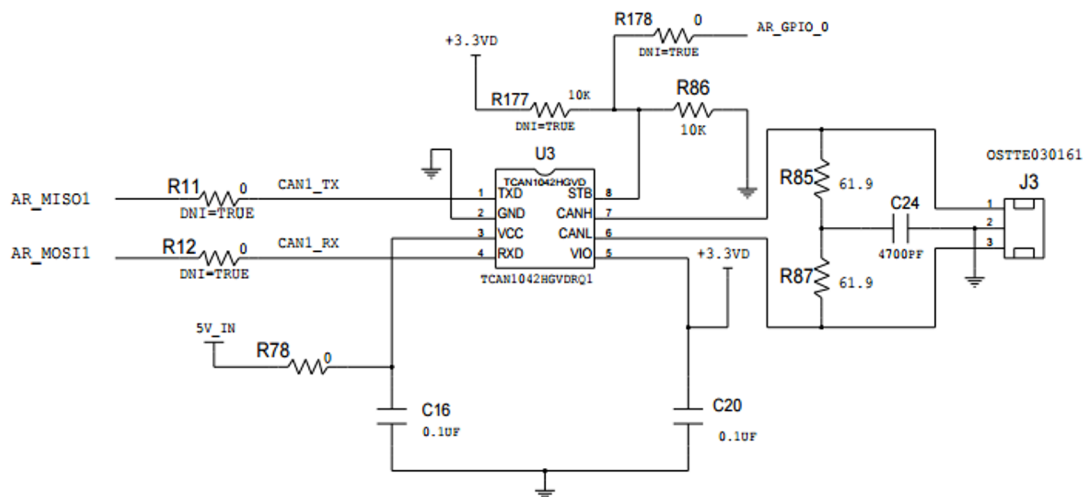


Figure 5. ECO[1] on AWR1642BOOST

- Remove R6 and R4.

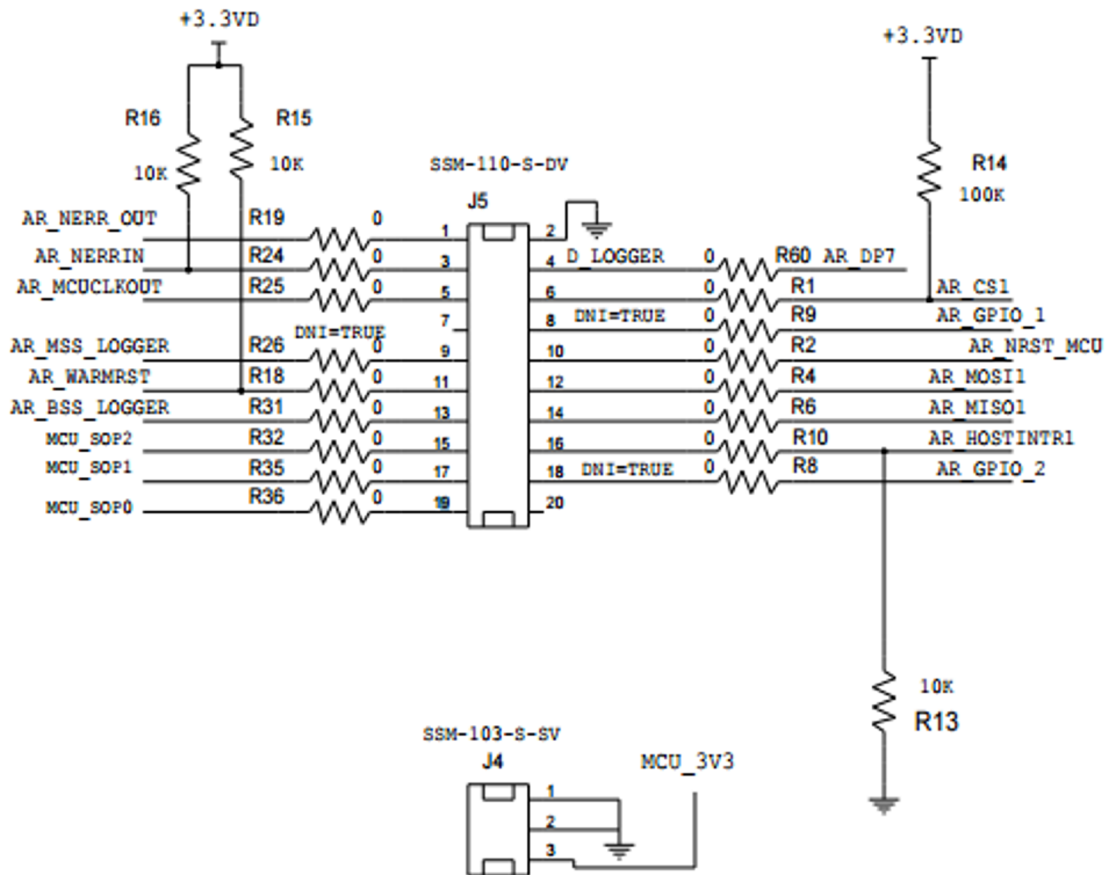


Figure 6. ECO[2] on AWR1642BOOST

A.5 ECO's on MMWAVE-DEVPACK

There are certain ECO's required on the [MMWAVE-DEVPACK](#) for the external DCAN communication to be functional. These modifications are required to be performed on EVM.

- Remove R16 & R43/
- Mount 0 Ω on R17 and R39.

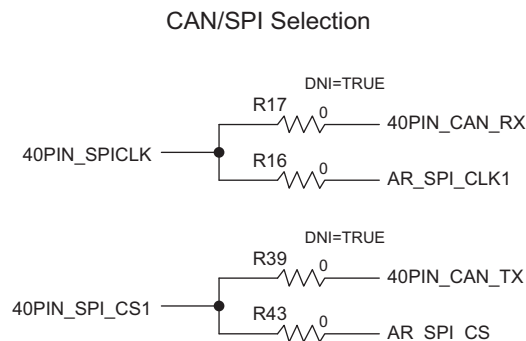
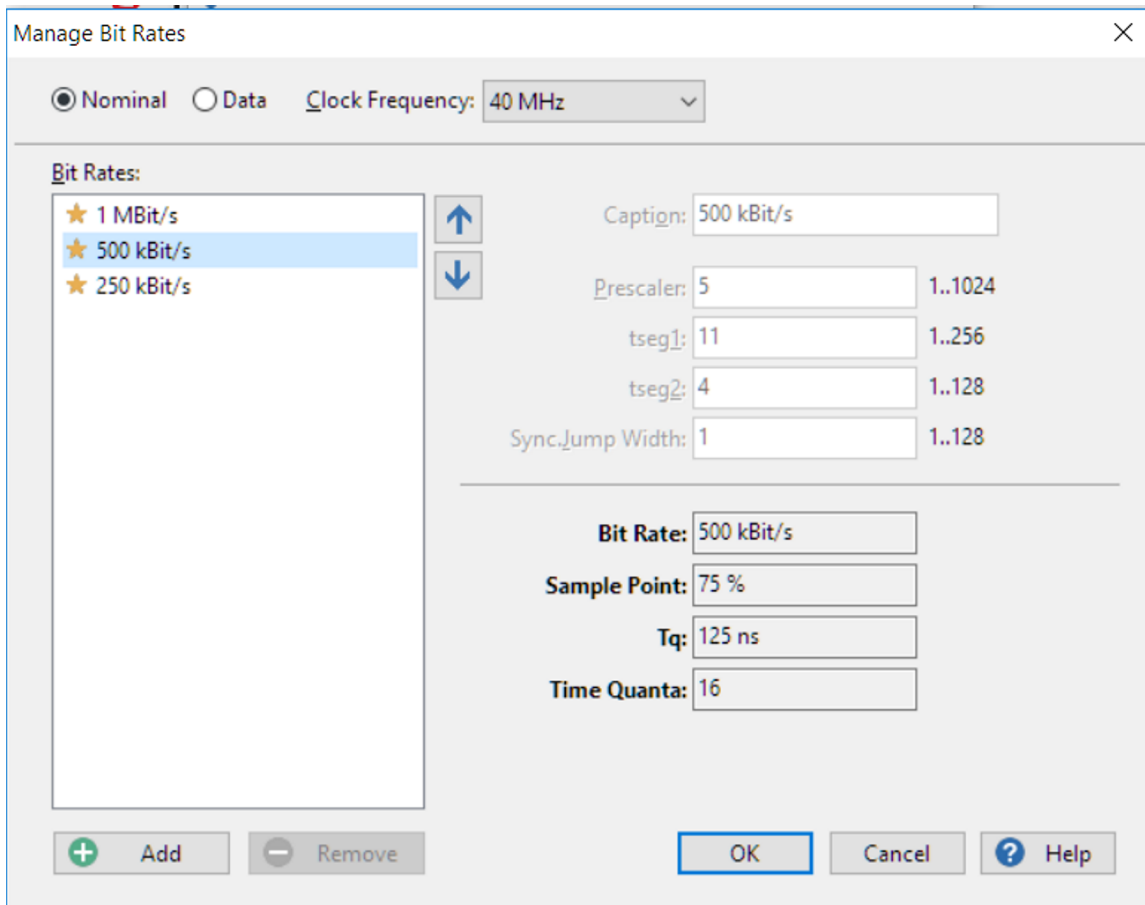


Figure 7. ECO on MMWAVE-DEVPACK

A.6 Bit Timing Calculation

- Sample Bit Timing calculation for Nominal bitrate 500Kbit/s
Nominal bitrate of 500Kbit/s valid in case of both DCAN and MCAN.



Manage Bit Rates

Nominal Data Clock Frequency: 40 MHz

Bit Rates:

- ★ 1 MBit/s
- ★ 500 kBit/s
- ★ 250 kBit/s

Caption: 500 kBit/s
 Prescaler: 5 1..1024
 tseg1: 11 1..256
 tseg2: 4 1..128
 Sync_Jump Width: 1 1..128

Bit Rate: 500 kBit/s
 Sample Point: 75 %
 Tq: 125 ns
 Time Quanta: 16

Figure 8. Nominal Bit-Timing 500KBit/s

- Sample Bit Timing calculation for Nominal bitrate 1Mbit/s
Nominal Bitrate of 1Mbit/s valid in case of both DCAN and MCAN.

The screenshot shows the 'Manage Bit Rates' dialog box with the following configuration:

- Mode:** Nominal, Data
- Clock Frequency:** 40 MHz
- Bit Rates List:**
 - ★ 1 MBit/s (Selected)
 - ★ 500 kBit/s
 - ★ 250 kBit/s
- Parameters:**
 - Caption: 1 MBit/s
 - prescaler: 5 (Range: 1..1024)
 - tseg1: 5 (Range: 1..256)
 - tseg2: 2 (Range: 1..128)
 - Sync_Jump Width: 1 (Range: 1..128)
- Calculated Values:**
 - Bit Rate: 1 MBit/s
 - Sample Point: 75 %
 - Tq: 125 ns
 - Time Quanta: 8
- Buttons:** Add, Remove, OK, Cancel, Help

Figure 9. Nominal Bit-Timing for 1MBit/s

- Sample Bit Timing calculation for Data bitrate 5Mbit/s
Data Bitrate setting is valid only in case of MCAN. This is valid in case of the FD mode of operation where the BRS is enabled.

Manage Bit Rates
✕

Nominal
 Data
 Clock Frequency: 40 MHz

Bit Rates:

- ★ 2 MBit/s
- ★ 4 MBit/s
- ★ 8 MBit/s
- ★ 10 MBit/s
- ★ 5MBit/s

Caption:

Prescaler: 1..1024

tseg1: 1..32

tseg2: 1..16

Sync_Jump Width: 1..16

Bit Rate:

Sample Point:

Tq:

Time Quanta:

+ Add
- Remove

OK
Cancel
? Help

Figure 10. Dataphase Bit-Timing for 5Mbit/s

IMPORTANT NOTICE FOR TI DESIGN INFORMATION AND RESOURCES

Texas Instruments Incorporated ("TI") technical, application or other design advice, services or information, including, but not limited to, reference designs and materials relating to evaluation modules, (collectively, "TI Resources") are intended to assist designers who are developing applications that incorporate TI products; by downloading, accessing or using any particular TI Resource in any way, you (individually or, if you are acting on behalf of a company, your company) agree to use it solely for this purpose and subject to the terms of this Notice.

TI's provision of TI Resources does not expand or otherwise alter TI's applicable published warranties or warranty disclaimers for TI products, and no additional obligations or liabilities arise from TI providing such TI Resources. TI reserves the right to make corrections, enhancements, improvements and other changes to its TI Resources.

You understand and agree that you remain responsible for using your independent analysis, evaluation and judgment in designing your applications and that you have full and exclusive responsibility to assure the safety of your applications and compliance of your applications (and of all TI products used in or for your applications) with all applicable regulations, laws and other applicable requirements. You represent that, with respect to your applications, you have all the necessary expertise to create and implement safeguards that (1) anticipate dangerous consequences of failures, (2) monitor failures and their consequences, and (3) lessen the likelihood of failures that might cause harm and take appropriate actions. You agree that prior to using or distributing any applications that include TI products, you will thoroughly test such applications and the functionality of such TI products as used in such applications. TI has not conducted any testing other than that specifically described in the published documentation for a particular TI Resource.

You are authorized to use, copy and modify any individual TI Resource only in connection with the development of applications that include the TI product(s) identified in such TI Resource. NO OTHER LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE TO ANY OTHER TI INTELLECTUAL PROPERTY RIGHT, AND NO LICENSE TO ANY TECHNOLOGY OR INTELLECTUAL PROPERTY RIGHT OF TI OR ANY THIRD PARTY IS GRANTED HEREIN, including but not limited to any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information regarding or referencing third-party products or services does not constitute a license to use such products or services, or a warranty or endorsement thereof. Use of TI Resources may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

TI RESOURCES ARE PROVIDED "AS IS" AND WITH ALL FAULTS. TI DISCLAIMS ALL OTHER WARRANTIES OR REPRESENTATIONS, EXPRESS OR IMPLIED, REGARDING TI RESOURCES OR USE THEREOF, INCLUDING BUT NOT LIMITED TO ACCURACY OR COMPLETENESS, TITLE, ANY EPIDEMIC FAILURE WARRANTY AND ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT OF ANY THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

TI SHALL NOT BE LIABLE FOR AND SHALL NOT DEFEND OR INDEMNIFY YOU AGAINST ANY CLAIM, INCLUDING BUT NOT LIMITED TO ANY INFRINGEMENT CLAIM THAT RELATES TO OR IS BASED ON ANY COMBINATION OF PRODUCTS EVEN IF DESCRIBED IN TI RESOURCES OR OTHERWISE. IN NO EVENT SHALL TI BE LIABLE FOR ANY ACTUAL, DIRECT, SPECIAL, COLLATERAL, INDIRECT, PUNITIVE, INCIDENTAL, CONSEQUENTIAL OR EXEMPLARY DAMAGES IN CONNECTION WITH OR ARISING OUT OF TI RESOURCES OR USE THEREOF, AND REGARDLESS OF WHETHER TI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You agree to fully indemnify TI and its representatives against any damages, costs, losses, and/or liabilities arising out of your non-compliance with the terms and provisions of this Notice.

This Notice applies to TI Resources. Additional terms apply to the use and purchase of certain types of materials, TI products and services. These include; without limitation, TI's standard terms for semiconductor products (<http://www.ti.com/sc/docs/stdterms.htm>), [evaluation modules](#), and [samples](http://www.ti.com/sc/docs/sampterm.htm) (<http://www.ti.com/sc/docs/sampterm.htm>).

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2018, Texas Instruments Incorporated