AN213891

How to Use CAN FD for Traveo™ Family

Author: Shinichi Takko

Associated Part Family: Traveo Family

Related Documents: For a complete list, see Related Documents.

This application note explains how to use CAN with Flexible Data rate (CAN FD) for Traveo family microcontrollers.

Contents

1 Introduction

This application note is intended for users of the Cypress Traveo family microcontrollers. It describes how to use Controller Area Network with Flexible Data rate (CAN FD) for Cypress Traveo family devices.

The CAN, the predecessor standard, is a safe and economical communication standard for automotive, and it is an automotive industry standard for data communication for many years.

CAN FD is an extension of CAN, which is nowadays called ‘Classical CAN’. CAN FD can transmit data frames of up to 64 bytes at bit rates exceeding the 1 Mbps limit of Classical CAN. The maximum achievable bus speed in the data segment is limited only by external components such as transceivers and the particular network topology of an application.

The CAN FD Controller conforms to ISO 11898-1:2015 and has been certified according ISO16845:2016.

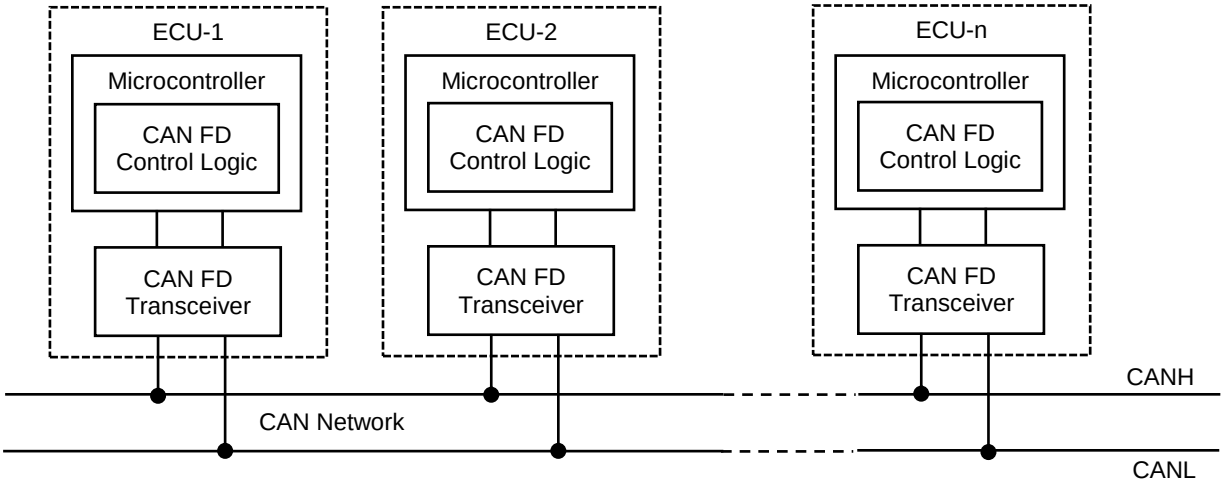
2 CAN FD Network

This section describes the operation of CAN FD communication. Figure 1 shows an example of the CAN FD network.

Two communication lines (CANH, CANL) are used in the CAN FD network to make it resilient against noise. Multiple ECUs can be connected to the CAN FD network; data is exchanged between the ECUs.

A receiver node converts the differential bus voltage to a digital signal by the CAN FD Transceiver, and received data is handled by the CAN FD Control Logic of the microcontroller. In transmission, data is transmitted from the CAN FD Control Logic to the CAN FD Transceiver that drives a differential signal onto the CANH and CANL lines of the CAN FD network.

Figure 1. CAN FD Network



ECU - Electronic Control Unit

CANH - CAN network line High

CANL - CAN network line Low

3 CAN FD Messages

CAN FD communication enlarges the bandwidth compared to Classical CAN. Figure 2 shows the CAN FD frame format. CAN and CAN FD message formats are equal in the arbitration segment and after the CRC field. Differences occur in the data segment which has more bytes, and which can be transmitted at higher speeds than the arbitration baud rate.

Data length in Classical CAN is 8 bytes but it is expanded to 64 bytes in CAN FD. Data communication speed can exceed the 1-Mbps limit set by Classical CAN. There are CAN transceivers supporting 5 Mbps, but there are announcements for 8 Mbps too.

3.1 CAN FD Fields

The fields of the CAN FD frame format include an Arbitration field, a Control field, a Data field, a CRC field, and an ACK field.

The Arbitration field contains the message ID number, and determines the priority of the message among other messages from other nodes trying to start a transmission at the same time.

Three bits of FD Format Indicator (FDF), Bit Rate Switch (BRS), and Error State Indicator (ESI) are newly adopted in the control field.

The FDF bit identifies the frame type as CAN or CAN FD. It is recessive for CAN FD frames and dominant for CAN frames. If the BRS bit is recessive, the bit rate of the data field is switched to another, typically higher speed, and if it is dominant, the bit rate of the data field keeps the arbitration bus speed. The ESI bit is used for the identification of the error state of the CAN FD node.

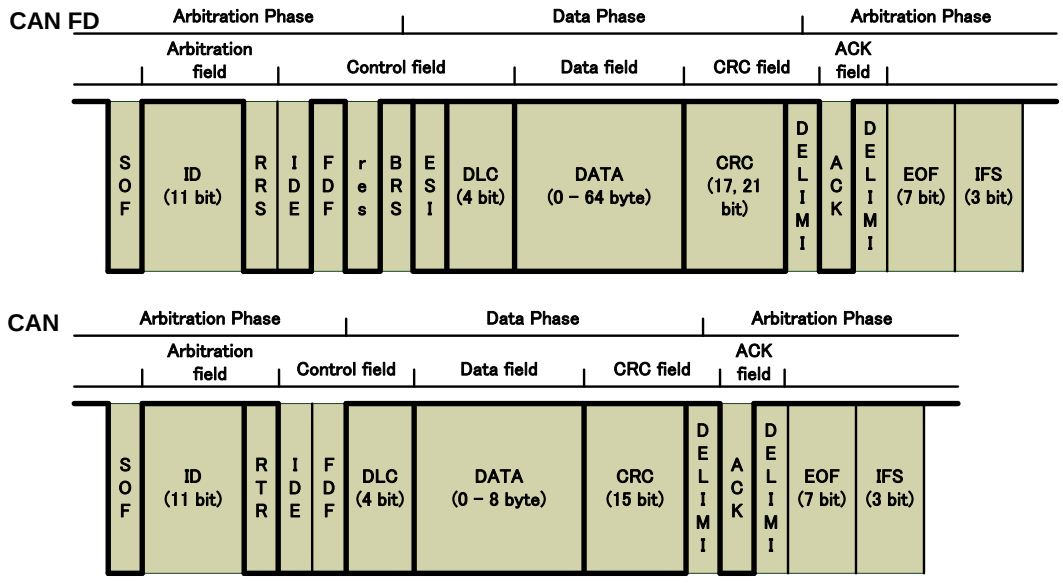
Furthermore, the Data Length Code (DLC) indicates how many bytes of data are transmitted. This settable range has been expanded from 8 bytes of CAN to 64 bytes in CAN FD.

The Data field carries the message data, and is sized by the data length set in DLC.

The CRC field consists of a CRC sequence and a CRC delimiter. The CRC sequence is 17 bits when the data length is 0 - 16 bytes, 21 bits in the case of 17 - 64 bytes. Any receiver can analyze the received data stream of a message and compare it to the transmitted CRC, and thus identify a valid or incorrectly received message.

The ACK field consists of an ACK slot and an ACK delimiter. If the message reception is successful, the dominant is transmitted from other nodes.

Figure 2. CAN FD Frame

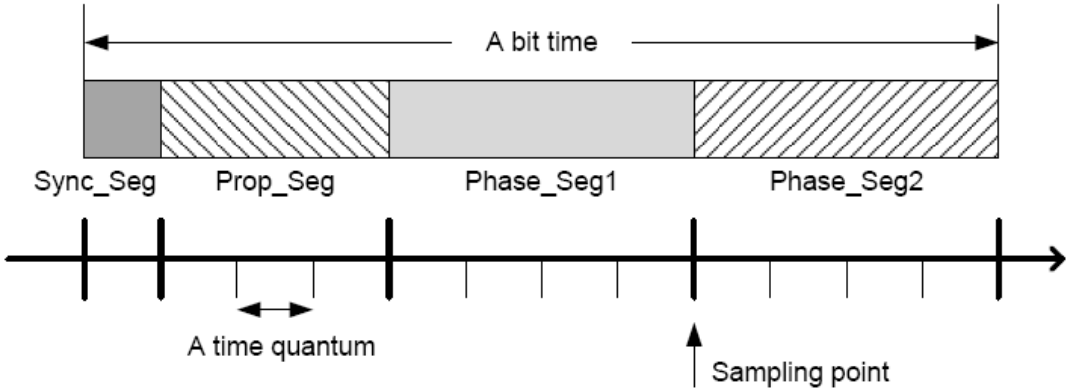


3.2 Bit Timing

The CAN FD operation defines two bit times - nominal bit time and data bit time. The nominal bit time is for the arbitration phase. The data bit time has an equal or shorter length and can be used to accelerate the data phase.

The basic construction of a bit time is shared with both the nominal and data bit times. The bit time can be divided into four segments according to the CAN specifications (see Figure 3): the synchronization segment (Sync_Seg), the propagation time segment (Prop_Seg), the phase buffer segment 1 (Phase_Seg1), and the phase buffer segment 2 (Phase_Seg2). The sample point, the point of time at which the bus level is read and interpreted as the value of that respective bit, is located at the end of Phase_Seg1.

Figure 3. Bit Time Construction



Each segment consists of a programmable number of time quanta, which is a multiple of the time quantum that is defined by the CAN clock and a prescaler. The values and prescalers used to define these parameters differ for the nominal and data bit times, and are configured by Nominal Bit Timing & Prescaler Register (NBTP) and Data Bit Timing & Prescaler Register (DBTP) as shown in below.

Parameter	Description
Time quantum tq (nominal) and tqd (data)	Time quantum. Derived by multiplying the basic unit time quanta (i.e. the CAN clock period) with the respective prescaler. The time quantum is configured by the CAN FD Controller as nominal : tq = (NBTP.NBRP[8:0] +1) × CAN clock period data : tqd = (DBTP.DBRP[4:0] + 1) × CAN clock period
Sync_Seg	Sync_Seg is fixed to 1 time quantum as defined by the CAN specifications and is therefore not configurable (inherently built into the CAN FD Controller). nominal : 1 tq data : 1 tqd
Prop_Seg	Prop_Seg is the part of the bit time that is used to compensate for the physical delay times within the network. The CAN FD Controller configures the sum of Prop_Seg and Phase_Seg1 with a single parameter, i.e., nominal : Prop_Seg + Phase_Seg1 = NBTP.NTSEG1[7:0] + 1 data : Prop_Seg + Phase_Seg1 = DBTP.DTSEG1[4:0] + 1
Phase_Seg1	Phase_Seg1 is used to compensate for edge phase errors before the sampling point. Can be lengthened by the resynchronization jump width. The sum of Prop_Seg and Phase_Seg1 is configured by the CAN FD Controller as nominal : NBTP.NTSEG1[7:0] + 1 data : DBTP.DTSEG1[4:0] + 1
Phase_Seg2	Phase_Seg2 is used to compensate for edge phase errors after the sampling point. Can be shortened by the resynchronization jump width. Phase_Seg2 is configured by the CAN FD Controller as nominal : NBTP.NTSEG2[6:0] + 1 data : DBTP.DTSEG2[3:0] + 1
SJW	Resynchronization Jump Width. Used to automatically compensate timing fluctuation between nodes and adjust the length of Phase_Seg1 and Phase_Seg2. SJW will not be longer than either Phase_Seg1 or Phase_Seg2. SJW is configured by the CAN FD Controller as nominal : NBTP.NSJW[6:0] + 1 data : DBTP.DSJW[3:0] + 1

These relations result in the following equations for the nominal and data bit times:

Nominal Bit time

$$= [Sync\_Seg + Prop\_Seg + Phase\_Seg1 + Phase\_Seg2] \times tq$$

$$= [1 + (NBTP.NTSEG1[7:0] + 1) + (NBTP.NTSEG2[6:0] + 1)] \times [(NBTP.NBRP[8:0] + 1) \times CAN\ clock\ period]$$

Example (500 kbps)

$$= [1 + (13 + 1) + (4 + 1)] \times [(3 + 1) \times (1/40000000)] = 0.000002\ \ (500\ kbps)$$

Data bit time

$$= [1 + (DBTP.DTSEG1[4:0] + 1) + (DBTP.DTSEG2[3:0] + 1)] \times [(DBTP.DBRP[4:0] + 1) \times CAN\ clock\ period]$$

Example (5 Mbps)

$$= [1 + (3 + 1) + (2 + 1)] \times [(0 + 1) \times (1/40000000)] = 0.0000002\ \ (5\ Mbps)$$

Example (2 Mbps)

$$= [1 + (10 + 1) + (7 + 1)] \times [(0 + 1) \times (1/40000000)] = 0.0000005\ \ (2\ Mbps)$$

4 CAN FD Settings

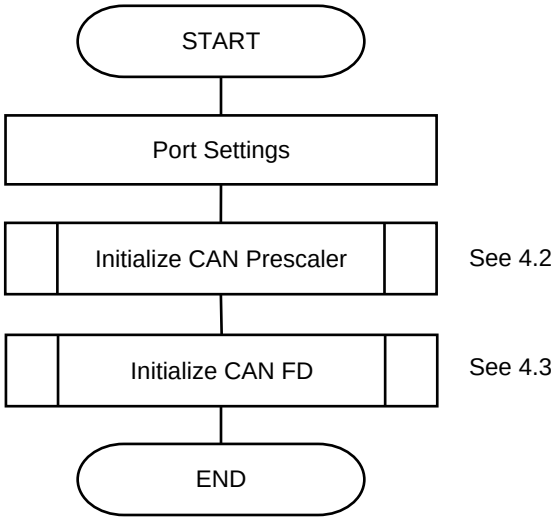
This section explains setting CAN FD for Traveo family. For details of registers, refer to the CAN FD chapter in the Hardware manual of target products.

4.1 CAN FD Setup

Figure 4 is an example of the CAN FD setup flow. First, enable the I/O port used for CAN FD communication. Then configure the clock supplied to the CAN FD controller unit and initialize CAN FD controller.

Section 4.1.1 shows a code example of the setup flow illustrated in Figure 4.

Figure 4. Example of the CAN FD Setup Flow



4.1.1 Example Program of CAN FD Setup

Code 1 demonstrates the example program of CAN FD Setup

Code 1. CAN FD Setup

```
int main(void)
{
    uint32_t    u32count = 0;

    /* Finalize initialization to default settings. */
    /* (this will do IRQ and NMI initialization and global IRQ/NMI enable) */
    Start_Init();

    /* CAN FD transceiver device Initialize */
    CanFD_Device_Init();

    /* PORT - port pin configuration */
    /* TX0(P304) POF=2 */
    PPC_KEYCDR=0x100000C8;
    PPC_KEYCDR=0x500000C8;
    PPC_KEYCDR=0x900000C8;
    PPC_KEYCDR=0xD00000C8;
    PPC_PCFGR304=0x0002;
    /* Output direction */
    GPIO_KEYCDR=0x20000038;
```

Port settings

Keycode register setting

PPC_PCFGR register setting for TX0 port

```
Port settings
    GPIO_KEYCDR=0x60000038;
    GPIO_KEYCDR=0xA0000038;
    GPIO_KEYCDR=0xE0000038;
    GPIO_DDSR3=0x00000010;

    /* RX0(P303) POF=0 */
    PPC_KEYCDR=0x100000C6;
    PPC_KEYCDR=0x500000C6;
    PPC_KEYCDR=0x900000C6;
    PPC_KEYCDR=0xD00000C6;
    PPC_PCFGR303=0x1000;
    /* Input direction */
    GPIO_KEYCDR=0x2000003C;
    GPIO_KEYCDR=0x6000003C;
    GPIO_KEYCDR=0xA000003C;
    GPIO_KEYCDR=0xE000003C;
    GPIO_DDCR3=0x00000008;

    /* PORT enable */
    GPIO_KEYCDR=0x20000400;
    GPIO_KEYCDR=0x60000400;
    GPIO_KEYCDR=0xA0000400;
    GPIO_KEYCDR=0xE0000400;
    GPIO_PORTEN=0x00000001;

Initialize CAN prescaler
    /* Initialize CAN Prescaler */
    Can_PrescalerInit();

Initialize CAN FD
    /* Initialize CAN FD 0ch */
    CanFD_Init();

    /* Endless main loop */
    for (;;)
    {
        u32count++;
        if ( u32count > 100000 )
        {
            u32count = 0;

Message transmission
            /* Transmit CAN FD Data */
            /* Tx Buffer 0 */
            CanFD_UpdateAndTransmitMsgBuffer();
        }
        /* Clear hardware watchdog */
        ClearWatchdog();
    }
}
```

GPIO_DDSR register setting for TX0 port

PPC_PCFGR register setting for RX0 port

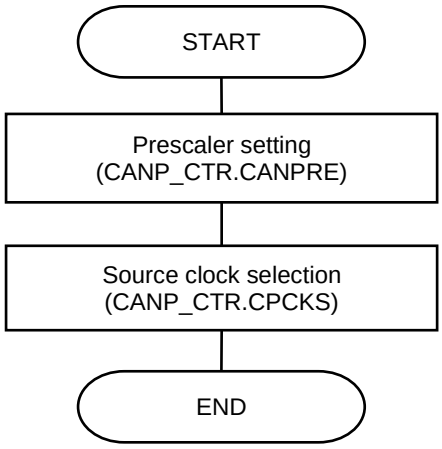
GPIO_DDCR register setting for RX0 port

4.2 Initialize CAN Prescaler

Figure 5 is an example of CAN Prescaler initialization flow. In the prescaler setting, set the division ratio for the clock supplied to the CAN FD controller (CANP_CTR.CANPRE). PLL clock and main clock are selectable for a source clock of the CAN Prescaler (CANP_CTR.CPCKS).

4.2.1 shows a code example of the following flow.

Figure 5. Example of CAN Prescaler Initialization Flow



4.2.1 Example Program of CAN Prescaler Initialization

Code 2 demonstrates the example program of CAN Prescaler settings.

Code 2. CAN Prescaler Settings

```
void Can_PrescalerInit(void)
{
    /* CAN prescaler division setting : 11 (Divided by 12) */
    CANP_CTR_CANPRE = 11;  /* CAN clock = 40MHz (PLL clock = 480MHz) */
    while( CANP_STR_BUSY == 1 )
    {
    }
    /* Select the source clock of CAN Prescaler */
    CANP_CTR_CPCKS = 0;    /* Select the PLL clock */
}
```

Prescaler setting

Source clock selection

4.3 Initialize CAN FD

Figure 6 is an example of CAN FD initialization flow; the general flow of initialization is described in this section.

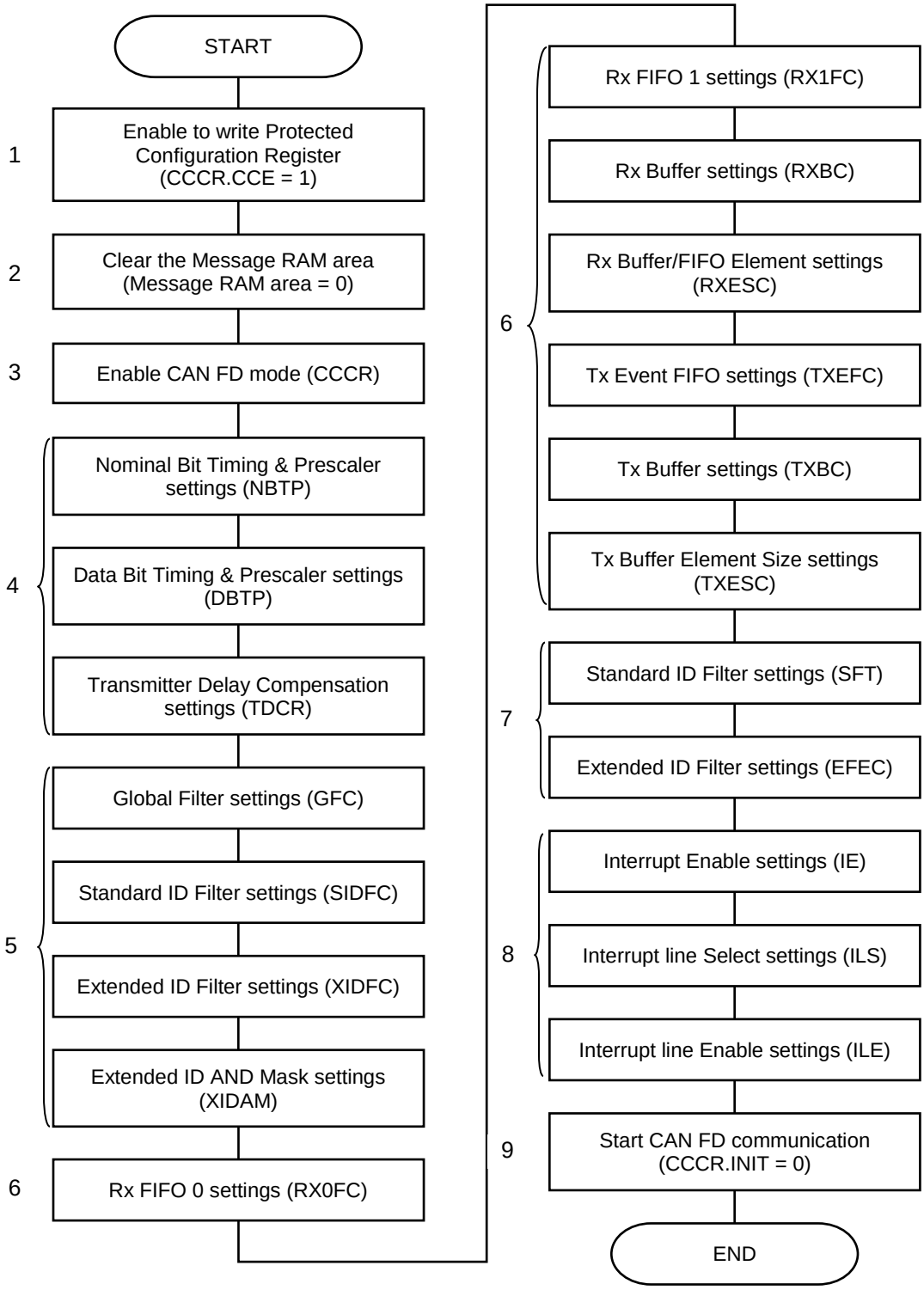
1. Enable the Configuration Change Enable register (CCCR.CCE) to write in the write-protected CAN FD configuration register.
2. Clear the message RAM area that has Rx/Tx message area and filter configurations area.
3. Set the Bit Rate Switch (CCCR.BRSE) and CAN FD mode (CCCR.FDOE) in the CC Control Register (CCCR).
4. Bit timing is set with the Nominal Bit Timing & Prescaler Register (NBTP) used in the arbitration phase and the Data Bit Timing & Prescaler Register (DBTP) used in the data phase when the bit rate switch is enabled.
5. For message filters, determine the handling of received frames with message IDs that do not match any filters as set in the Global Filter Configuration Register (GFC). The number of elements of the message filter and the start address offset in the message RAM are configured with the Standard ID Filter Configuration register (SIDFC) and the Extended ID Filter Configuration register (XIDFC).
6. For Rx/Tx messages, configure the element size of the Rx FIFO and start address offset in message RAM with Rx FIFO 0 Configuration (RXF0C) and Rx FIFO 1 Configuration (RXF1C).

   The Rx buffer is configured with Rx Buffer Configuration (RXBC). The data field of the size set by Rx Buffer / FIFO Element Size Configuration (RXESC) is stored in the Rx FIFO or Rx buffer.

   The number of Tx buffer and start address offset in message RAM are configured with Tx Buffer Configuration (TXBC). Set the size of the data field of the Tx buffer with Tx Buffer Element Size Configuration (TXESC).
7. The message filter is set to the address obtained by adding the start address offset set by SIDFC/XIDFC to the start address of the message RAM. Range Filter, Dual Filter, or Classic Bit Mask Filter can be set.
8. Each interrupt is configured with Interrupt Enable (IE). The CAN FD controller has Dual Interrupt Lines, and the Interrupt Line Select (ILS) determines which line the interrupt is assigned to. Enable the interrupt line with Interrupt Line Enable (ILE).
9. Set the Initialization register (CCCR.INIT) to 0 to start the operation of CANFD.

Section 4.3.1 shows a code example of the following flow.

Figure 6. Example of CAN FD Initialization Flow



4.3.1 Example Program of CAN FD Initialization

Code 3 demonstrates the example program of CAN FD initialization settings

Code 3. Initialization Settings

```
void CanFD_Init(void)
{
    stc_id_filter_t*         pstcIdFilter;
    stc_extid_filter_t*      pstcExtIdFilter;
    uint32_t*                pulAdrs;
    uint16_t                 u16count;

    /* Shadow data to avoid RMW and speed up HW access */
    un_cpg_canfdn_sidfc_t unSIDFC = { 0 };
    un_cpg_canfdn_xidfc_t unXIDFC = { 0 };
    un_cpg_canfdn_xidam_t unXIDAM = { 0 };
    un_cpg_canfdn_rxf0c_t unRXF0C = { 0 };
    un_cpg_canfdn_rxf1c_t unRXF1C = { 0 };
    un_cpg_canfdn_rxbc_t  unRXBC  = { 0 };
    un_cpg_canfdn_txefc_t unTXEFC = { 0 };
    un_cpg_canfdn_txbc_t  unTXBC  = { 0 };
    un_cpg_canfdn_cccr_t  unCCCR  = { 0 };
    un_cpg_canfdn_nbtp_t  unNBTP  = { 0 };
    un_cpg_canfdn_dbtp_t  unDBTP  = { 0 };
    un_cpg_canfdn_tdcr_t  unTDCR  = { 0 };
    un_cpg_canfdn_gfc_t   unGFC   = { 0 };
    un_cpg_canfdn_rxesc_t unRXESC = { 0 };
    un_cpg_canfdn_txesc_t unTXESC = { 0 };
    un_cpg_canfdn_ie_t    unIE    = { 0 };
    un_cpg_canfdn_ils_t   unILS   = { 0 };
    un_cpg_canfdn_ile_t   unILE   = { 0 };

    /* Set CCCR.INIT to 1 and wait until it will be updated. */
    unCCCR.stcField.u1INIT = 1;
    CPG_CANFD0_CCCR = unCCCR.u32Register;

    while ( CPG_CANFD0_CCCR_INIT != 1 )
    {
    }

    /* Cancel protection */
    unCCCR.stcField.u1CCE = 1;
    CPG_CANFD0_CCCR = unCCCR.u32Register;

    /* Clear the message RAM area */
    pulAdrs = (uint32_t *)((uint32_t)&CPG_CANFD0 + (uint32_t)0x00008000);
    for (u16count = 0; u16count < 4096; u16count++)
    {
        *pulAdrs++ = 0;
    }

    /* Configuration of CAN bus */

    /* CCCR register */
    unCCCR.stcField.u1TXP  = 0; /* Transmit pause disabled. */
    unCCCR.stcField.u1BRSE = 1; /* Bit rate switching for transmissions enabled. */
    unCCCR.stcField.u1FDOE = 1; /* FD operation enabled. */
    unCCCR.stcField.u1TEST = 0; /* Normal operation */
    unCCCR.stcField.u1DAR  = 0; /* Automatic retransmission enabled. */
```

Stop CAN FD communication

Enable to write Protected Configuration Register

Clear the Message RAM area

```
unCCCR.stcField.u1MON  = 0; /* Bus Monitoring Mode is disabled. */
unCCCR.stcField.u1CSR  = 0; /* No clock stop is requested. */
unCCCR.stcField.u1ASM  = 0; /* Normal CAN operation. */
CPG_CANFD0_CCCR = unCCCR.u32Register;

/* Normal Bit Timing & Prescaler Register */
unNBTP.stcField.u9NBRP   = 3;
unNBTP.stcField.u8NTSEG1 = 13;
unNBTP.stcField.u7NTSEG2 = 4;
unNBTP.stcField.u7NSJW   = 4;
CPG_CANFD0_NBTP = unNBTP.u32Register;

/* Data Bit Timing & Prescaler */
unDBTP.stcField.u1TDC    = 1; /* Transceiver Delay Compensation enabled. */
unDBTP.stcField.u5DBRP   = 0;
unDBTP.stcField.u5DTSEG1 = 3;
unDBTP.stcField.u4DTSEG2 = 2;
unDBTP.stcField.u4DSJW   = 2;
CPG_CANFD0_DBTP = unDBTP.u32Register;

/* Transmitter Delay Compensation */
unTDCR.stcField.u7TDCO   = 4; /* Transmitter Delay Compensation Offset */
unTDCR.stcField.u7TDCF   = 0; /* Transmitter Delay Compensation Filter Window Length */
CPG_CANFD0_TDCR = unTDCR.u32Register;

/* Configuration of Message RAM */

/* Configuration of ID Filter List */
/* Configuration of Global Filter */
unGFC.stcField.u2ANFS = 2; /* Reject when unmatch id */
unGFC.stcField.u2ANFE = 2; /* Reject when unmatch id */
unGFC.stcField.u1RRFS = 1; /* Reject all remote frame */
unGFC.stcField.u1RRFE = 1; /* Reject all remote frame */
CPG_CANFD0_GFC = unGFC.u32Register;

/* Standard ID filter */
unSIDFC.stcField.u8LSS = 1; /* Number of standard Message ID filter elements = 1 */
unSIDFC.stcField.u14FLSSA = 0x00000000; /* offset(word) */
CPG_CANFD0_SIDFC = unSIDFC.u32Register;

/* Extended ID filter */
unXIDFC.stcField.u7LSE = 1; /* Number of extended Message ID filter elements = 1 */
unXIDFC.stcField.u14FLESA = 0x00000004; /* offset(word) */
CPG_CANFD0_XIDFC = unXIDFC.u32Register;

unXIDAM.stcField.u29EIDM = 0x1fffffff;   /* not filtering(initial value) */
CPG_CANFD0_XIDAM = unXIDAM.u32Register;

/* Configuration of Rx Buffer and Rx FIFO */
/* Rx FIFO 0 (not use) */
unRXF0C.stcField.u1F0OM = 0;    /* Rx FIFO 0 blocking mode */
unRXF0C.stcField.u7F0WM = 0;    /* Watermark interrupt disabled */
unRXF0C.stcField.u7F0S = 0;     /* FIFO Element Number = 0 */
unRXF0C.stcField.u14F0SA = 0x00000010; /* offset(word) */
CPG_CANFD0_RXF0C = unRXF0C.u32Register;

/* Rx FIFO 1 (not use) */
unRXF1C.stcField.u1F1OM = 0;    /* Rx FIFO 1 blocking mode */
unRXF1C.stcField.u7F1WM = 0;    /* Watermark interrupt disabled */
unRXF1C.stcField.u7F1S = 0;     /* FIFO Element Number = 0 */
unRXF1C.stcField.u14F1SA = 0x00000020; /* offset(word) */
```

Enable CAN FD mode

Nominal Bit Timing & Prescaler settings (500kbps)

Data Bit Timing & Prescaler settings (5Mbps)

Transmitter Delay Compensation settings

Global Filter settings

Standard ID Filter settings

Extended ID Filter settings

Extended ID AND Mask settings

Rx FIFO 0 settings

```
CPG_CANFD0_RXF1C = unRXF1C.u32Register;

/* Rx buffer */
unRXBC.stcField.u14RBSA = 0x00000030; /* offset(word) */
CPG_CANFD0_RXBC = unRXBC.u32Register;

/* Configuration of Rx Buffer and Rx FIFO */
unRXESC.stcField.u3RBDS = 7; /* 64 byte data field. */
unRXESC.stcField.u3F1DS = 7; /* FIFO1 64 byte data field. */
unRXESC.stcField.u3F0DS = 7; /* FIFO0 64 byte data field. */
CPG_CANFD0_RXESC = unRXESC.u32Register;

/* Configuration of Tx Buffer and Tx FIFO/Queue */
/* Tx Event FIFO (not use) */
unTXEFC.stcField.u6EFWM = 0; /* Watermark interrupt disabled. */
unTXEFC.stcField.u6EFS = 0;  /* Tx Event FIFO disabled. */
unTXEFC.stcField.u14EFSA = 0x00000100; /* offset(word) */
CPG_CANFD0_TXEFC = unTXEFC.u32Register;

/* Tx buffer */
unTXBC.stcField.u1TFQM = 0;  /* Tx FIFO operation */
unTXBC.stcField.u6TFQS = 0;  /* No Tx FIFO/Queue */
unTXBC.stcField.u6NDTB = 1;  /* Number of Dedicated Tx Buffers = 1 */
unTXBC.stcField.u14TBSA = 0x00000200; /* offset(word) */
CPG_CANFD0_TXBC = unTXBC.u32Register;

/* Configuration of Tx Buffer Element Size */
unTXESC.stcField.u3TBDS = 7; /* 64 byte data field. */
CPG_CANFD0_TXESC = unTXESC.u32Register;

/* Configuration of ID Filter */
/* Standard Message ID Filter */
pstcIdFilter = (stc_id_filter_t*)((uint32_t*)((uint32_t)&CPG_CANFD0 +
               (uint32_t)0x00008000) + CPG_CANFD0_SIDFC_FLSSA);
pstcIdFilter->SFT   = 2;        /* Standard Filter Type : Classic filter */
pstcIdFilter->SFEC  = 7;        /* Store into dedicated Rx Buffer,    */
                                /* configuration of SFT[1:0] ignored. */
pstcIdFilter->SFID1 = 0x010;    /* Filter ID : 0x10(16) */
pstcIdFilter->SFID2 = (0 << 9)  /* Store message into a dedicated Rx Buffer */
                    | 0;        /* Buffer index : 0 */

/* Extended Message ID Filter */
pstcExtIdFilter = (stc_extid_filter_t*)((uint32_t*)((uint32_t)&CPG_CANFD0 +
                  (uint32_t)0x00008000) + CPG_CANFD0_XIDFC_FLESA);
pstcExtIdFilter->F1_f.EFT   = 2;        /* Extended Filter Type : Classic filter */
pstcExtIdFilter->F0_f.EFEC  = 7;        /* Store into dedicated Rx Buffer,    */
                                        /* configuration of EFT[1:0] ignored. */
pstcExtIdFilter->F0_f.EFID1 = 0x10001;  /* Filter ID : 0x10001(65537) */
pstcExtIdFilter->F1_f.EFID2 = (0 << 9)  /* Store message into a dedicated Rx Buffer */
                            | 1;        /* Buffer index : 1 */

/* Configuration of Interrupt */

/* Interrupt Enable */
unIE.stcField.u1DRXE  = 1; /* Message stored to Dedicated Rx Buffer */
CPG_CANFD0_IE = unIE.u32Register;

/* Interrupt Line Select */
unILS.u32Register = 0;  /* Interrupt line canfd_int0 */
CPG_CANFD0_ILS = unILS.u32Register;
```

Rx FIFO 1 settings

Rx Buffer settings

Rx Buffer/FIFO Element settings

Tx Event FIFO settings

Tx Buffer settings

Tx Buffer Element Size settings

Standard ID Filter settings

Extended ID Filter settings

Interrupt Enable settings

Interrupt line Select settings

```
    /* Interrupt Line Enable */
    unILE.stcField.u1EINT0 = 1; /* Enable Interrupt Line 0 */
    unILE.stcField.u1EINT1 = 0; /* Disable Interrupt Line 1 */
    CPG_CANFD0_ILE = unILE.u32Register;

    /* CAN FD operation start */
    /* Set CCCR.INIT to 0 and wait until it will be updated. */
    unCCCR.stcField.u1INIT = 0;
    CPG_CANFD0_CCCR = unCCCR.u32Register;

    while ( CPG_CANFD0_CCCR_INIT != 0 )
    {
    }

}
```

Interrupt line Enable settings

Start CAN FD communication

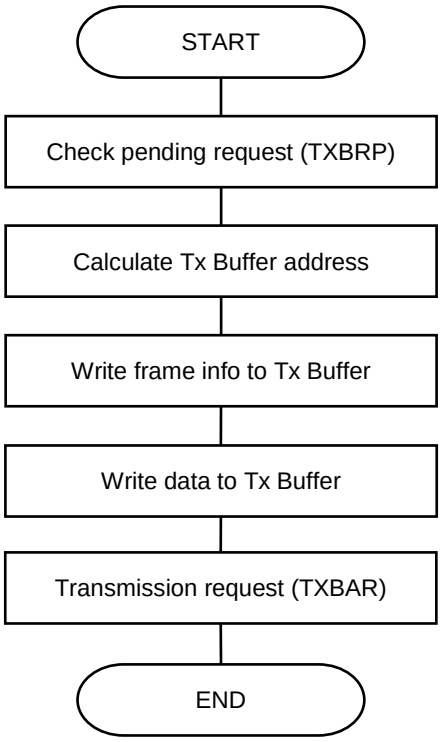
4.4 Message Transmission

Figure 7 is an example of Message Transmission flow. This example does not use the Tx interrupt.

The Message is sent via the Tx Buffer in the message RAM area. Ensure that there are no pending requests (TXBRP). If there is no pending request, calculate the Tx Buffer address of the message RAM and write the control information and data of the frame transmitted by the CAN FD controller. A message transmission is started by writing to Tx Buffer Add Request (TXBAR).

Section 4.4.1 shows a code example of the following flow.

Figure 7. Example of Message Transmission Flow



4.4.1 Example Program of Message Transmission

Code 4 demonstrates the example program of CAN FD Message transmission. This program do not use the Tx interrupt.

Code 4. Message Transmission

```
void CanFD_UpdateAndTransmitMsgBuffer(void)
{
    uint8_t u8DataLengthWord;
    stc_canfd_tx_buffer_t* pstcCanFDTxBuffer;
    uint16_t u16count;
    uint16_t u16dlcTmp;
    uint16_t u16MessageBufferNumber ;
    uint8_t  u8DataLengthCode;
    uint32_t au32Data[16] = { 0 };

    /* Message Buffer 0 */
    u16MessageBufferNumber = 0;

    /* Check whether Tx buffer is empty or not */
    while ( 0 != (CPG_CANFD0_TXBRP & 0x00000001 )) /* Check the TRP0 */
    {
    }

    /* Get Tx Buffer address */
    pstcCanFDTxBuffer = (stc_canfd_tx_buffer_t *)((uint32_t *)((uint32_t)&CPG_CANFD0 +
                        (uint32_t)0x00008000) + CPG_CANFD0_TXBC_TBSA +
                        (2 + iDlcInWord[CPG_CANFD0_TXESC_TBDS]) * u16MessageBufferNumber) ;

    /* Set data to Tx buffer */
    pstcCanFDTxBuffer->T0_f.RTR = 0; /* Transmit data frame. */
    pstcCanFDTxBuffer->T0_f.XTD = 0; /* 11-bit ID */
    pstcCanFDTxBuffer->T0_f.ID  = 0x200 << 18; /* Identifier */
    pstcCanFDTxBuffer->T1_f.EFC = 0; /* Not store Tx event FIFO */
    pstcCanFDTxBuffer->T1_f.MM  = 0; /* Not used */
    pstcCanFDTxBuffer->T1_f.DLC = 15; /* DataLengthCode */
    pstcCanFDTxBuffer->T1_f.FDF = 1; /* CAN FD format */
    pstcCanFDTxBuffer->T1_f.BRS = 1; /* Transmitted with bit rate switching */

    /* Convert the DLC to word data type */
    u8DataLengthCode = pstcCanFDTxBuffer->T1_f.DLC;
    u16dlcTmp = u8DataLengthCode - 8;
    u8DataLengthWord = iDlcInWord[u16dlcTmp];

    /* Data set */
    au32Data[0]              = 0x12345678;    /* Data of CAN FD message */
    au32Data[1]              = 0x9abcdef9;    /* Data of CAN FD message */
    au32Data[2]              = 0x12345677;    /* Data of CAN FD message */
    au32Data[15]             = 0x87654321;    /* Data of CAN FD message */

    for ( u16count = 0; u16count < u8DataLengthWord; u16count++ )
    {
        pstcCanFDTxBuffer->DATA_AREA_f[u16count] = au32Data[u16count];
    }

    /* Transmission request */
    CPG_CANFD0_TXBAR = 0x00000001; /* request for buffer 0 */
}
```

Tx Buffer 0

Check pending request

Calculate Tx Buffer 0 address

Write frame info to Tx Buffer 0

Convert DLC to word data type

Write data to TxBuffer 0

Transmission request

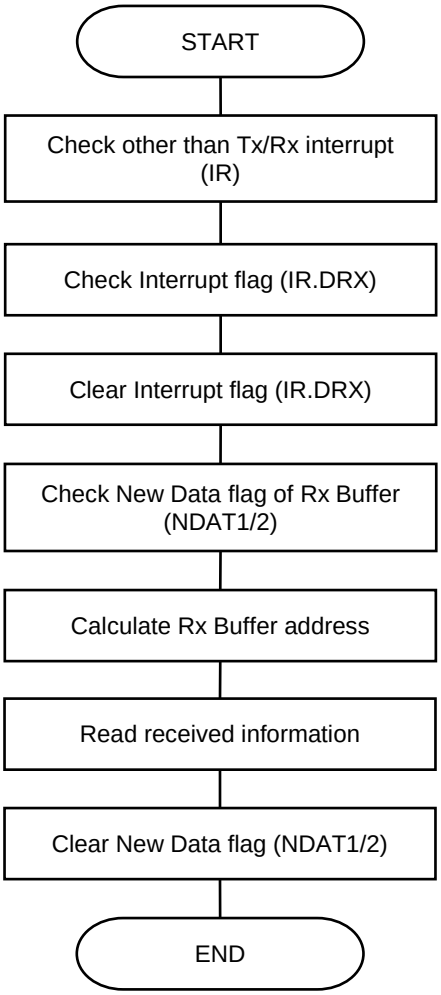
4.5 Message Reception

Figure 8 is an example of Message Reception flow using the Rx buffer and the Rx interrupt.

The received message is filtered by the message filter and stored in the Rx buffer of the message RAM area; an interrupt occurs at this event. When a message is stored in the dedicated Rx Buffer, the corresponding bits of the Interrupt Register (IR.DRX) and New Data 1/2 (NDAT 1/2) are set. Calculate the start address of the Rx Buffer and read the received message information from the calculated address.

Section 4.5.1 shows a code example of the following flow.

Figure 8. Example of Message Reception Flow



4.5.1 Example Program of Message Reception

Code 5 demonstrates the example program of CAN FD Message reception. This program use the Rx buffer and the Rx interrupt.

Code 5. Message Reception

```
/* CAN FD Reception interrupt routine */
FN_IRQ_DEFINE_BEGIN(CanFD_Isr_CanFD0, INTERRUPTS_IRQ_NUMBER_56)
{
    uint32_t*    pulAdrs = 0;
    uint16_t     u16count = 0;
    uint16_t     u16dlcTmp = 0;
    uint16_t     u16MessageBufferNumber ;
    stc_canfd_msg_t stcCanFDmsg;

    /* Other than Tx/Rx interrupt occured */
    if ( CPG_CANFD0_IR & 0x3ff7E0EE )
    {
        /* User handling */
    }

    /* Received a data frame */
    if ( CPG_CANFD0_IR_DRX == 1 ) /* At least one received message stored */
                                  /* into an Rx Buffer */
    {
        /* Clear the Message stored to Dedicated Rx Buffer flag */
        CPG_CANFD0_IR_DRX = 1;

        /* New data is exist */
        if (CPG_CANFD0_NDAT1 & 0x00000001)      /* Check the message buffer 0 */
        {
            /* Message Buffer 0 */
            u16MessageBufferNumber = 0;
            /* Rx Buffer address */
            pulAdrs = (uint32_t *)((uint32_t)&CPG_CANFD0 + (uint32_t)0x00008000) +
                      CPG_CANFD0_RXBC_RBSA + (2 + iDlcInWord[CPG_CANFD0_RXESC_RBDS]) *
                      u16MessageBufferNumber;

            /* Clear NDAT1 register */
            CPG_CANFD0_NDAT1 = 0x00000001;
        }
        else if (CPG_CANFD0_NDAT1 & 0x00000002)   /* Check the message
        {
            /* Message Buffer 1 */
            u16MessageBufferNumber = 1;
            /* Rx Buffer address */
            pulAdrs = (uint32_t *)((uint32_t)&CPG_CANFD0 + (uint32_t)0x00008000) +
                      CPG_CANFD0_RXBC_RBSA + (2 + iDlcInWord[CPG_CANFD0_RXESC_RBDS]) *
                      u16MessageBufferNumber;

            /* Clear NDAT1 register */
            CPG_CANFD0_NDAT1 = 0x00000002;
        }

        if (pulAdrs)
        {
            /* Save received data */
            /* XTD : Extended Identifier */
            stcCanFDmsg.stcIdentifier.bExtended = ((stc_canfd_rx_buffer_t *) pulAdrs)
                                                  ->R0_f.XTD;
```

Check other than Tx/Rx interrupt

Check IR.DRX flag

Clear Interrupt flag

Check New Data flag of Rx Buffer 0

Calculate Rx Buffer 0 address

Clear New Data flag

Check New Data flag of Rx Buffer 1

Calculate Rx Buffer 1 address

Clear New Data flag

Read Extended Identifier

```
            /* ID : RxID */
            if ( stcCanFDmsg.stcIdentifier.bExtended == FALSE )
            {

                stcCanFDmsg.stcIdentifier.u32Identifier =
                                    ((stc_canfd_rx_buffer_t *) pulAdrs)->R0_f.ID >> 18;
            }
            else
            {
                stcCanFDmsg.stcIdentifier.u32Identifier =
                                    ((stc_canfd_rx_buffer_t *) pulAdrs)->R0_f.ID;
            }

            /* FDF : Extended Data Length */
            stcCanFDmsg.bCanFDFormat = ((stc_canfd_rx_buffer_t *) pulAdrs)->R1_f.FDF;


            /* DLC : Data Length Code */
            stcCanFDmsg.stcData.u8DataLengthCode =
                                    ((stc_canfd_rx_buffer_t *) pulAdrs)->R1_f.DLC;

            /* Copy 0-64 byte of data area */
            if ( stcCanFDmsg.stcData.u8DataLengthCode < 8 )
            {
                u16dlcTmp = 0;
            }
            else
            {
                u16dlcTmp = stcCanFDmsg.stcData.u8DataLengthCode - 8;
            }

            for ( u16count = 0; u16count < iDlcInWord[u16dlcTmp]; u16count++ )
            {
                stcCanFDmsg.stcData.au32Data[u16count] =
                            ((stc_canfd_rx_buffer_t *) pulAdrs)->DATA_AREA_f[u16count];
            }
        }
    }
    else if ( CPG_CANFD0_IR_TC == 1 ) /* Transmission completed */
    {
    }
}
FN_IRQ_DEFINE_END()
```

Read Identifier

Read Extended Data Length

Read Data Length Code

Read Data

5 Related Documents

- S6J311E/D/C/B Series 32-bit Microcontroller Traveo Family Datasheet
- S6J311A/9/8 Series 32-bit Microcontroller Traveo Family Datasheet
- S6J3110 Series 32-bit Microcontroller Traveo Family Hardware Manual
- S6J3120 Series 32-bit Microcontroller Traveo Family Datasheet
- S6J3120 Series 32-bit Microcontroller Traveo Family Hardware Manual
- S6J3200 Series 32-bit Microcontroller Traveo Family Datasheet
- S6J3200 Series 32-bit Microcontroller Traveo Family Hardware Manual
- S6J32E/F/G Series 32-bit Microcontroller Traveo Family Datasheet
- S6J32E/F/G Series 32-bit Microcontroller Traveo Family Hardware Manual
- Traveo Family Hardware Manual Platform Part for S6J3200 Series
- S6J3300 Series 32-bit Microcontroller Traveo Family Datasheet
- S6J3350 Series 32-bit Microcontroller Traveo Family Datasheet
- S6J3300/S6J3350 Series 32-bit Microcontroller Traveo Family Hardware Manual
- S6J3400 Series 32-bit Microcontroller Traveo Family Datasheet
- S6J3400 Series 32-bit Microcontroller Traveo Family Hardware Manual
- Traveo Family Hardware Manual Platform Part for S6J3300/S6J3350/S6J3400 Series
- S6J3360/70 Series Datasheet (Doc.No.002-03359)
- S6J3360/70 Series Hardware Manual (Doc.No.002-18302)
- Traveo Family HardwareManual Platform Part for S6J3360/3370 Series (Doc.No.002-07884)
- S6J3510 Series Datasheet (Doc.No.002-18647)
- S6J3510 Series Hardware Manual (Doc.No.002-18642)
- Traveo Family Hardware Manual Platform Part for S6J3510 Series (Doc.No.002-07884)

6 Summary

This application note described how to use CAN FD used mainly in automotive applications.

Document History

Document Title: AN213891 - How to Use CAN FD for Traveo™ Family

Document Number: 002-13891

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	5380108	SITO	02/21/2017	New application note
*A	5747377	SITO	06/21/2017	Added S6J3360/S6J3370/S6J3510 series to target products.

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

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709