AN204777

16FX Family, C_CAN-Controller

This application note describes the features and the programming of the CAN Controller implemented on the 16FX Family 16-bit MCUs

Contents

1 Introduction....1
1.1 Feature List of C_CAN Controller....1
2 Prerequisite for using the C_CAN Controller....2
2.1 Clock Generation and Supply for C_CAN Controller....2
2.2 CAN functionality of GPIO Ports....2
2.3 Interface to the CAN Bus....2
3 C_CAN Channel Base Addresses....4
4 Configuration of C_CAN Channel....4
4.1 C_CAN Channel Baud Rate....4
4.2 Special Modes of C_CAN Channel....5
5 Configuration of Message Objects (Message Buffers)....7
5.1 Arbitration (Message ID)....8
5.2 ID Mask....9
6 Operation of C_CAN Channel....12
6.1 Basic Mode....12
6.2 Normal Mode....13
6.3 Interrupts....23
7 Additional Information....28
Document History....29
Worldwide Sales and Design Support....30
Products....30
PSoC® Solutions....30
Cypress Developer Community....30
Technical Support....30

1 Introduction

This application note describes the features and the programming of the CAN Controller implemented on the 16FX Family 16-bit MCUs. This CAN controller is the so-called C_CAN that uses the Bosch IP.

1.1 Feature List of C_CAN Controller

Following are the features of the CAN Controller:

- Supports CAN protocol version 2.0 part A and B
- Bit rates up to 1 MBits/s
- Upto 128 Message Objects
- Each Message Object has its own identifier mask
- Programmable FIFO mode (concatenation of Message Objects)
- Maskable interrupt
- Disabled Automatic Re-transmission Mode for Time Triggered CAN applications
- Programmable Loop Back Mode for self-test operation

For a comparison of the features of the CAN Controller implemented for example on the 16LX Family MCUs called F-CAN to the features of the new CAN Controller (called C_CAN) please refer to Table 1.

Table 1. Comparison of Features Supported by F-CAN Controller (MB90340 Series) and C_CAN Controller (16FX Family MCUs)

	F-CAN	C_CAN
Support CAN protocol version 2.0 part A and B	Yes	Yes
Supports Remote Frame	Yes	Yes
Maximum Baud Rate	1 MBits/s (when CAN clock is 16 MHz)	1 Mbit/s
Number of Message Objects	16	32 (up to 128)
Multi-Level Buffer	Yes	Yes
Mask Filtering		
Full Bit Comparison	Yes	Yes
Full Bit Mask	Yes	Yes
Partial Bit Mask	Yes	Yes
Masks	2 Masks for all Message Objects	1 Mask for each Message Object
Disabled Automatic Re-transmission Mode	No	Yes
Loop Back Mode for self-test	No	Yes

2 Prerequisite for using the C_CAN Controller

The 16FX Family MCUs can be used for a wide range of applications, including CAN applications. To use a CAN controller of the MCU, some configuration needs to be taken care of, depending on the requirements of the application. This configuration is discussed herein.

2.1 Clock Generation and Supply for C_CAN Controller

For details on the clock generation scheme and on the C_CAN Controller clock supply please refer to the Hardware Manual of the 16FX Family MCUs Chapter “Clocks”.

16FX Family MCUs support a clock generation scheme which supplies the Peripheral Clock 2 CLKP2 to the C_CAN Controller.

2.2 CAN functionality of GPIO Ports

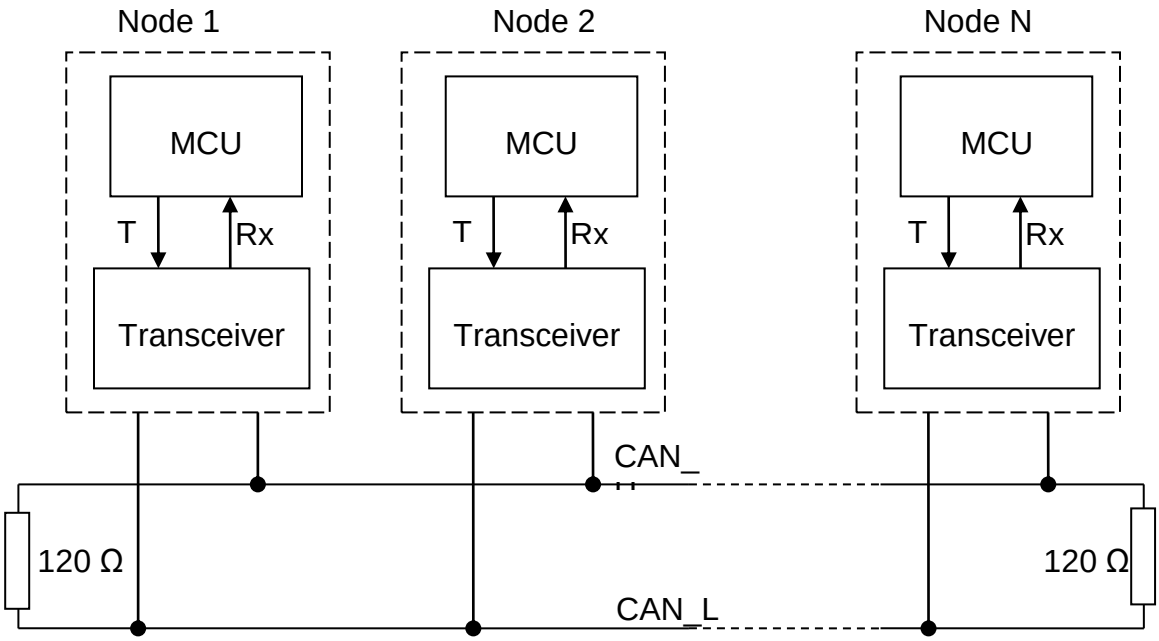
Since most of the pins of the MCU are shared to several IO functionalities one has to enable the CAN IO function for the corresponding port pins.

The Receive Pin RX is enabled by setting the corresponding PIER register bit. The output is enabled by setting the COER register of the C_CAN Controller to 0x01.

2.3 Interface to the CAN Bus

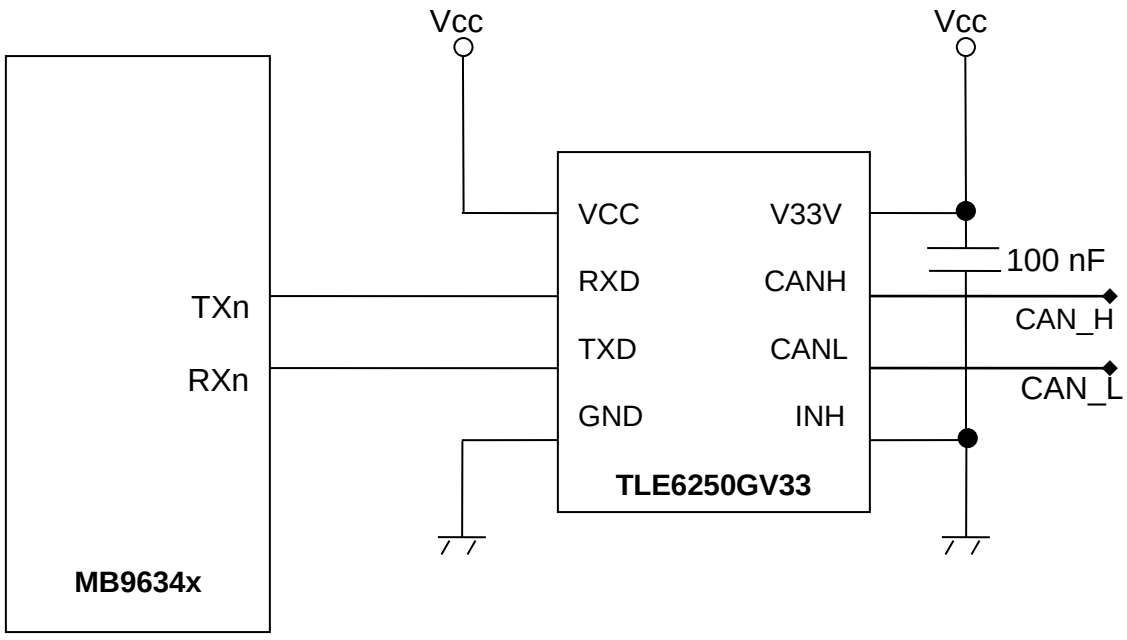
The Figure 1 shows the example of the Differential High-Speed CAN bus. Such arrangement may provide the speed up to 1 MBits/s. The termination resistor of 120 Ω is located at each end of the bus to minimize reflections and ringing on the waveforms. The logic states of the bits are determined by the differential voltage between the CAN_H and CAN_L signals. In most systems, the recessive state represents logic ‘1’, while logic‘0’ is provided by the dominant state.

Figure 1. High-Speed CAN Bus



C_CAN controller can be interfaced to such CAN bus via a transceiver. Transceiver provides the ability to receive and transmit the messages over the bus. Figure 2 shows interfacing of MB9634x microcontroller to Transceiver TLE6250GV33. The CAN_H and CAN_L outputs of the transceiver would be connected to corresponding signals of the CAN bus.

Figure 2. MB9634x Interfacing to Transceiver TLE6250GV33



3 C_CAN Channel Base Addresses

The base addresses of the C_CAN Channels on 16FX Family MCUs are:

CAN0: 0x0700

CAN1: 0x0800

CAN2: 0x0900

CAN3: 0x0A00

CAN4: 0x0B00

For details on the available CAN channels on a given 16FX Family MCU, please refer the datasheet of the MCU.

4 Configuration of C_CAN Channel

This Chapter describes how to configure a C_CAN channel. This includes setting the CAN baud rate corresponding to your CAN network and enabling special function modes (if needed).

4.1 C_CAN Channel Baud Rate

Configuration of baud rate using the Bit Timing Register `BTRn` (also BRP Extension Register `BRPERn` in some cases) is an integral part CAN initialization.

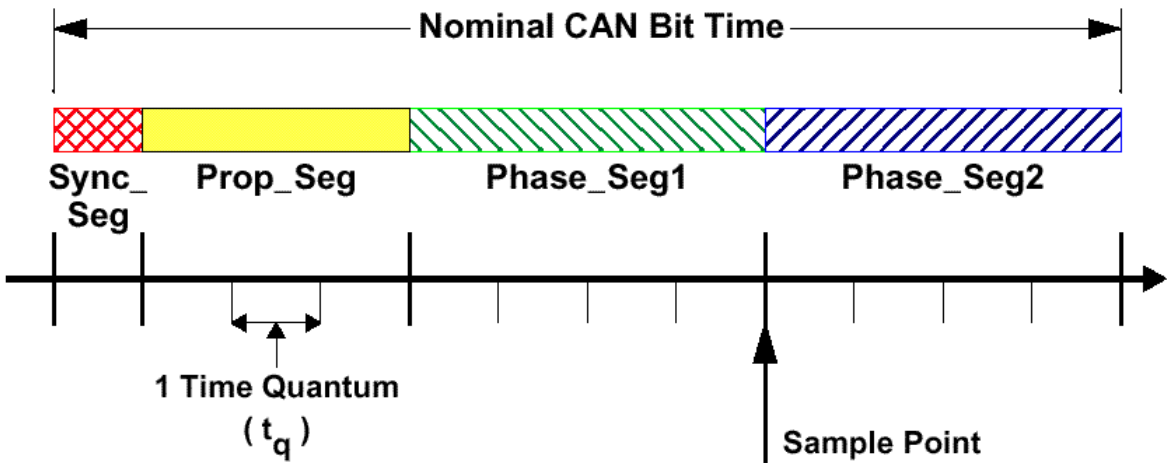
C_CAN supports a wide range of bit rate from 1kBPS to 1MBPS. Each CAN node available on a 16-FX series of microcontroller has an independent baud rate generator which can be tuned to get the desired bit or baud rate from CAN clock (i.e. CLKP2).

According to the CAN specification, the bit time is divided into the following four segments:

- Synchronization Segment (Sync_Seg)
- Propagation Time Segment (Prop_Seg)
- Phase Buffer Segment 1 (Phase_Seg1)
- Phase Buffer Segment 2 (Phase_Seg2)

The following figure shows CAN Bit Time and the segments:

Figure 3. CAN Bit Timing



Each segment consists of a specific, programmable number of time quanta. The length of the time quantum (t_q), which is the basic time unit of the bit time, is defined by the CAN controller's system clock (i.e. CLKP2) and the Baud Rate Prescaler (BRP):

$$t_q = BRP / CLKP2$$

Baud Rate Prescaler Extension (BRPE) may also need to be considered if the lower baud rate is required at higher CLKP2 clock values.

The following table discusses various parameter related to one CAN Bit and bits of `BTRn` register related to them:

Figure 4. Parameters of CAN Bit Time

Parameter	Range	Remarks
Sync_Seg	1 t_q	This segment is always fixed.
Prop_Seg	[1..8] t_q	These two segments together can be configured using TSEG1 bits (bits 8-11) of `BTRn` register as below: `Prop_Seg + Phase_Seg1 = TSEG1 Value + 1`
Phase_Seg1	[1..8] t_q	
Phase_Seg2	[1..8] t_q	This segment can be configured using TSEG2 bits (bits 12-14) of `BTRn` register as below: `Phase_Seg2 = TSEG2 Value + 1`

Example 1: To have a Bit rate of 500Kbps with a frequency of CLKP2 = 24 MHz it is possible to set: BRP = 5, TSEG1 = 3, TSEG2 = 2, then the `BTRn` value would be 0x2305.

4.2 Special Modes of C_CAN Channel

4.2.1 Disable Automatic Re-transmission

If there is any error while the transmission or in case of lost arbitration, the CAN has a provision of automatic re-transmission of such frames. By default this automatic re-transmission of frames is enabled. It can be disabled to enable the CAN to work within a Time Triggered CAN environment. The Disabled Automatic Retransmission mode is enabled by programming bit DAR in the CAN Control Register `CTRLRn` to "1". In this mode the behavior of `TXRQST` and `NEWDAT` bits are different than the normal one.

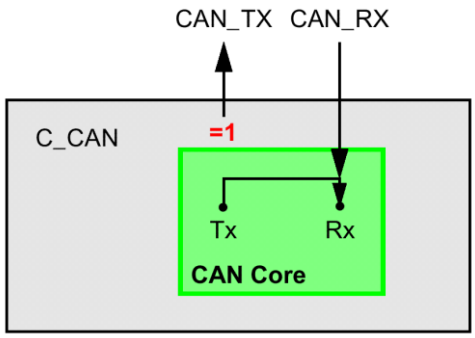
4.2.2 Test Modes

Silent Mode

In the silent mode, CAN node is not able to transmit any message frames however it can receive message frames such as Data Frame and Remote Frames. In this mode it cannot send any dominant bits on the bus (such as ACK bit).

Hence if there are two nodes on the CAN bus: one is in normal mode (Node 1) and the other being in Silent mode (Node 2) and when the Node 1 sends a message, Node 2 would receive the message but will not be able to acknowledge it. In such situation, the transmit error counter at the Node 1 would get increased to reflect this error condition. Node 2 would not encounter any error, since the ACK bit is rerouted internally so that it monitors this dominant bit, although the CAN bus may remain in recessive state.

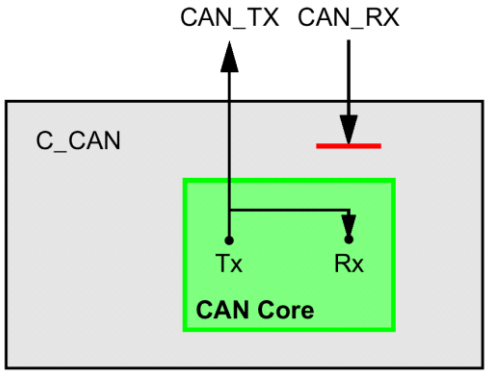
Figure 5. CAN Core in Silent Mode



Loop Back Mode

In Loop Back Mode, the CAN Core treats its own transmitted messages as received messages and stores them (if they pass acceptance filtering) into a Receive Buffer. In this mode, the message that CAN node transmits, can be monitored on CAN_TX pin and hence would be received by the nodes available on the bus. The same message would be received by the same node since CAN Core performs an internal feedback from its Tx output to its Rx input. The CAN Core ignores acknowledge errors in this mode.

Figure 6. CAN Core in Loop Back Mode



Basic Mode

In the Basic Mode, the C_CAN node runs without using the Message Objects in the RAM. It uses the IF1 registers as Transmit Buffer and uses the IF2 registers as Receive Buffer. Using this mode, one can avoid initializing all the message objects (32 up to 128) in the Message RAM.

The BUSY bit of IF1CREQn registers needs to be used in this mode only and has no meaning in the normal mode (i.e. while using Message RAM).

Transmission: After setting all the IF1 registers according to the transmit message requirements, the transmission (of contents of IF1 registers) is requested by setting the BUSY bit in IF1CREQn to 1. After setting the bit to 1, transmission data (contents of IF1 registers) is locked. Once the message gets transmitted (depending upon the condition on the bus) the CAN core clears the BUSY bit to indicate that the transmission is successfully completed. The pending transmission (BUSY = 1) can also be aborted by the CPU by clearing the BUSY bit to 0.

Reception: In the Basic Mode, the evaluation of all Message Object related control and status bits and also the evaluation of the control bits of the IFx Command Mask Registers IF1CMSKn is turned off. Hence, after the reception of a message the contents of the shift register is stored into the IF2 Registers, without any acceptance filtering.

5 Configuration of Message Objects (Message Buffers)

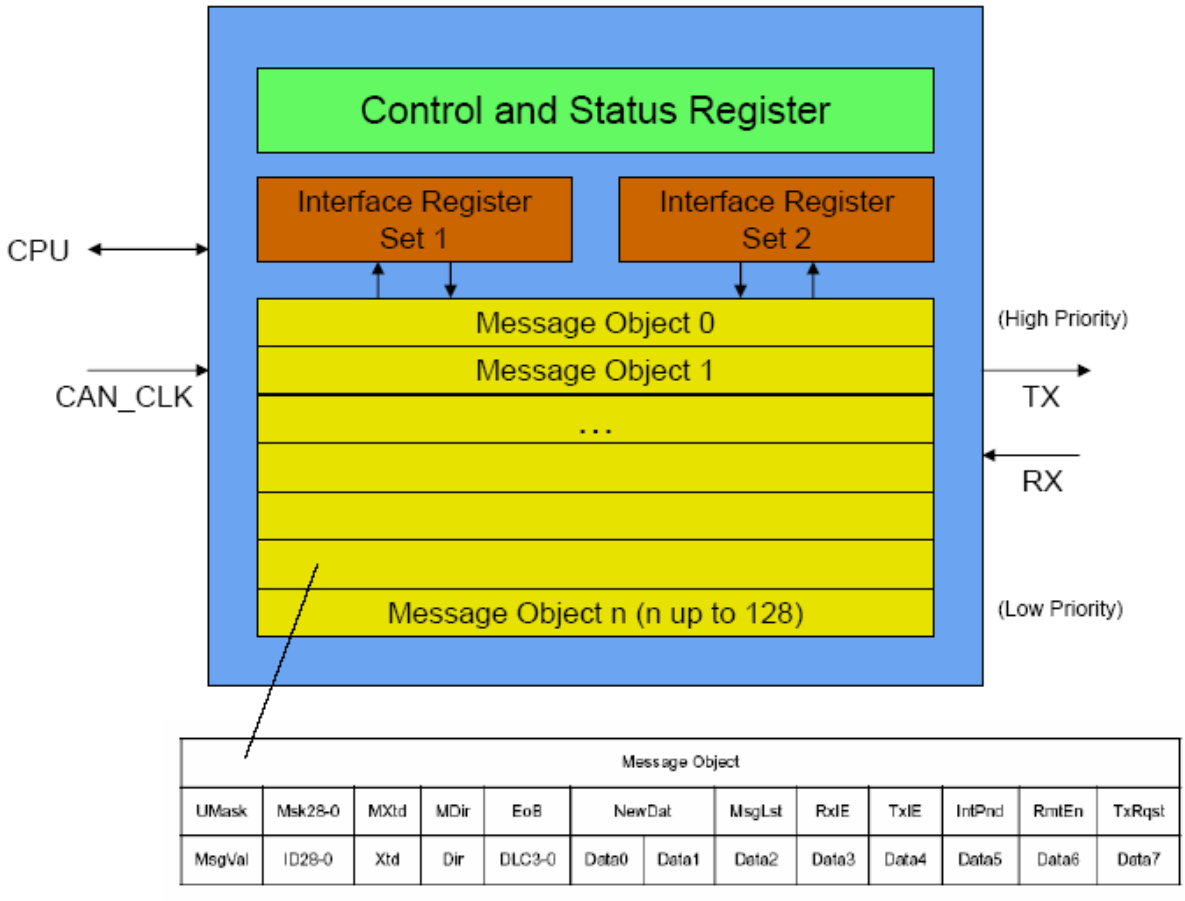
For efficient handling of CAN messages sent or received by C_CAN channel the message objects (message buffers) provided by C_CAN channel need to be prepared before starting the C_CAN channel. Nonetheless, it is also possible to change the configuration of the message objects while the C_CAN channel is participating in the CAN communication.

Handling of the message objects from the CPU is significantly different from the way this is done on the former F-CAN controller. In C_CAN controller, to ensure data consistency at all times the CPU has no direct access to the message buffers.

Each C_CAN channel contains two sets of interface registers. These interface registers can be modified/read directly by the CPU. The information which has to be read from / written to one of the message objects is transferred by these interface registers.

In Figure 7 the register structure of a C_CAN channel containing the two sets of interface registers is shown:

Figure 7. Register Structure of C_CAN Channel



Both sets of interface registers behave exactly in the same way and can be used interchangeably. However, when using interrupts, one must be careful that the register set used in the interrupt handler does not corrupt a partially configured register set. A corruption can be avoided if the interrupt handler uses one register set (in following IF2) while the main application loop uses the other register set (IF1 in the following). A different method would be to make sure the interrupt handler restores the interface register set to its previous state.

A set of interface registers consists of the following parts described next.

5.1 Arbitration (Message ID)

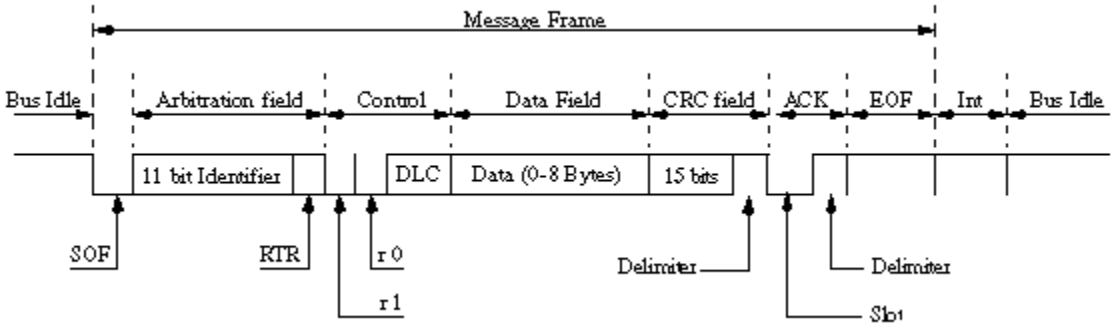
The Figure 8 shows the arbitration bits those are parts of IFxARB1n and IFxARB2n registers. In the case of transmission, these bits indicates ID of the message to transmitted and in the case of reception, these bits indicates the acceptance filter.

Figure 8. Arbitration Bits of Message Object

Message Object											
UMASK	MSK28-0	MXTD	MDIR	EOB	NEWDAT		MSGLST	RXIE	TXIE	INTPND	RMTEN	TXRQST
MSGVAL	ID28-0	XTD	DIR	DLC3-0	Data0	Data1	Data2	Data3	Data4	Data5	Data6	Data7

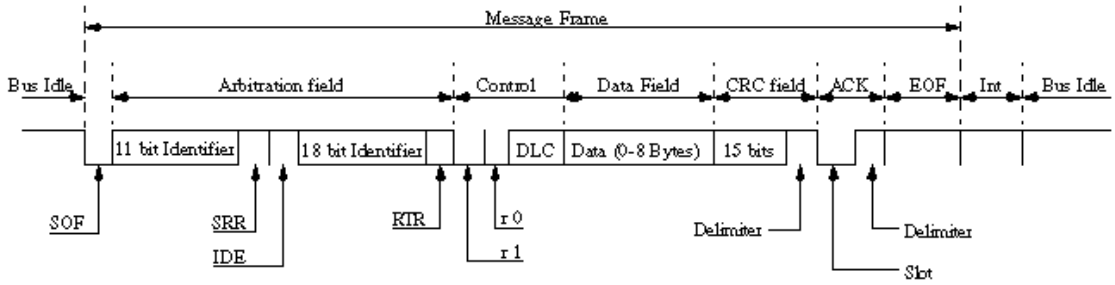
If the Standard Identifier (11-bit) is used, then it is programmed to bits ID28 to ID18. The Standard Identifier is also transmitted in the same order (i.e. from ID28 to ID18 – 11 bit identifier as shown in the Figure 9). The 7 most significant bits ID28 to ID22 must not be all ‘recessive’.The bits ID17 to ID0 are all ignored.

Figure 9. CAN Data Frame with Standard Identifier



If the Extended Identifier (29-bit) is used, then it is programmed to bits ID28 to ID0. In the Figure 10, the 11 bit identifier corresponds to bit ID28 to ID18 and the 18 bit identifier corresponds to ID17 to ID0.

Figure 10. CAN Data Frame with Extended Identifier



The XTD bit is set to 1, if Extended Identifier is used. For Standard Identifier it is cleared to 0. If the message object is required to be set for transmission, then the DIR bit needs to be set, else for reception it is required to be cleared. For the unused message object MSGVAL bit needs to be cleared and the same can be set if a particular message object is configured completely and should be considered by the message handler.

The following code snippet shows how to configure a given message object to transmit a message with Extended Identifier `0x1237FFFF`.

```
IF1ARB10 = 0xFFFF;
IF1ARB20 = 0xF237;          // =>
// MSGVAL = 1;              // msg buffer valid
// XTD = 1;                 // id = extended (29-bit)
// DIR = 1;                 // direction = transmit
// ID = 0x1237FFFF;         // ID = 0x1237FFFF
```

The following code snippet shows how to configure a given message object to transmit a message with Standard Identifier `0x60F`.

```
IF1ARB20 = 0xB83C;          // =>
// MSGVAL = 1;              // msg buffer valid
// XTD = 0;                 // id = standard (11-bit)
// DIR = 1;                 // direction = transmit
// ID = 0x60F;// ID = 0x60F
```

The following code snippet shows how to configure a given message object to receive a message with Extended Identifier `0x1237FFFF`.

```
IF1ARB10 = 0xFFFF;
IF1ARB20 = 0xD237;          // =>
// MSGVAL = 1;              // msg buffer valid
// XTD = 1;                 // id = extended (29-bit)
// DIR = 0;                 // direction = receive
// ID = 0x1237FFFF;         // ID = 0x1237FFFF
```

The following code snippet shows how to configure a given message object to receive a message with Standard Identifier `0x60F`.

```
IF1ARB20 = 0x983C;          // =>
// MSGVAL = 1;              // msg buffer valid
// XTD = 0;                 // id = standard (11-bit)
// DIR = 0;                 // direction = receive
// ID = 0x60F;// ID = 0x60F
```

5.2 ID Mask

The mask bits are only relevant to the received messages (data and remote frames).

The figure shows the Mask bits those are parts of IFxMSK1n and IFxMSK2n registers. These bits specifies which parts arbitration registers are considered and evaluated. If any of the highlighted bits if the mask registers is 1 then only the correspoding bit of arbitraration register is evaluated for the message acceptance filtering, else it would ignored.

Example: While reception of message/frame with Extended Identifier, if the bit MSK19 of mask register is set to 1 and if the bit ID19 bit of arbitration register is 0, the ID19 of the incoming message should be 0 to pass the acceptance filtering.

Figure 11. Mask Bits of Message Object

Message Object												
UMASK	MSK28-0	MXTD	MDIR	EOB	NEWDAT		MSGLST	RXIE	TXIE	INTPND	RMTEN	TXRQST
MSGVAL	ID28-0	XTD	DIR	DLC3-0	Data0	Data1	Data2	Data3	Data4	Data5	Data6	Data7

The following code snippet shows how to configure a given message object to receive a message with Extended Identifier `0x1237FFFF`.

```
IF1MSK10 = 0xFFFF;
IF1MSK20 = 0xFFFF;          // =>
// MXTD = 1;                // use XTD for acceptance filtering
// MDIR = 1;                // use DIR for acceptance filtering
                            // use ID28-ID0 for acceptance filtering
IF1ARB10 = 0xFFFF;
IF1ARB20 = 0xD237;          // =>
// MSGVAL = 1;              // msg buffer valid
// XTD = 1;                 // id = extended (29-bit)
// DIR = 0;                 // direction = receive
// ID = 0x1237FFFF;         // ID = 0x1237FFFF
```

The same can also be done by simply clearing the `UMSK` bit of `IF1MCTR0` register to 0. This way there is no need to configure mask registers at all.

```
IF1ARB10 = 0xFFFF;
IF1ARB20 = 0xD237;          // =>
// MSGVAL = 1;              // msg buffer valid
// XTD = 1;                 // id = extended (29-bit)
// DIR = 0;                 // direction = receive
// ID = 0x1237FFFF;         // ID = 0x1237FFFF

IF1MCTR0_UMSK = 0;          // ignore mask
```

The following code snippet shows how to configure a given message object to receive a messages with Extended Identifier ranging from `0x1237FFF0` to `0x1237FFFF`.

```
IF1MSK10 = 0xFFF0;
IF1MSK20 = 0xFFFF;          // =>
// MXTD = 1;                // use XTD for acceptance filtering
// MDIR = 1;                // use DIR for acceptance filtering
                            // use ID28-ID4 for acceptance filtering
IF1ARB10 = 0xFFFF;
IF1ARB20 = 0xD237;          // =>
// MSGVAL = 1;              // msg buffer valid
// XTD = 1;                 // id = extended (29-bit)
// DIR = 0;                 // direction = receive
// ID = 0x1237FFFF;         // ID = 0x1237FFFF
```

The following code snippet shows how to configure a given message object to receive a messages with Extended Identifier `0x1237FFF7` and `0x1237FFFF`.

```
IF1MSK10 = 0xFFF7;
IF1MSK20 = 0xFFFF;          // =>
// MXTD = 1;                // use XTD for acceptance filtering
// MDIR = 1;                // use DIR for acceptance filtering
                            // use ID28-ID0 (except ID3) for acceptance
                            //filtering
IF1ARB10 = 0xFFFF;
IF1ARB20 = 0xD237;          // =>
// MSGVAL = 1;              // msg buffer valid
// XTD = 1;                 // id = extended (29-bit)
// DIR = 0;                 // direction = receive
// ID = 0x1237FFFF;         // ID = 0x1237FFFF
```

The following code snippet shows how to configure a given message object to receive a message with Standard Identifier `0x60F`.

```
IF1MSK20 = 0xFFFC;          // =>
// MXTD = 1;                // use XTD for acceptance filtering
// MDIR = 1;                // use DIR for acceptance filtering
                            // use ID28-ID18 for acceptance filtering
IF1ARB20 = 0x983C;          // =>
// MSGVAL = 1;              // msg buffer valid
// XTD = 0;                 // id = standard (11-bit)
// DIR = 0;                 // direction = receive
// ID = 0x60F;              // ID = 0x60F
```

The same can also be done by simply clearing the `UMSK` bit of `IF1MCTR0` register to 0. This way there is no need to configure mask registers at all.

```
IF1ARB20 = 0x983C;          // =>
// MSGVAL = 1;              // msg buffer valid
// XTD = 0;                 // id = standard (11-bit)
// DIR = 0;                 // direction = receive
// ID = 0x60F;              // ID = 0x60F

IF1MCTR0 UMSK = 0;          // ignore mask
```

The following code snippet shows how to configure a given message object to receive a messages with Standard Identifier ranging from `0x600` to `0x60F`.

```
IF1MSK20 = 0xFFC0;          // =>
// MXTD = 1;                // use XTD for acceptance filtering
// MDIR = 1;                // use DIR for acceptance filtering
                            // use ID28-ID12 for acceptance filtering
IF1ARB20 = 0x983C;          // =>
// MSGVAL = 1;              // msg buffer valid
// XTD = 0;                 // id = standard (11-bit)
// DIR = 0;                 // direction = receive
// ID = 0x60F;              // ID = 0x60F
```

6 Operation of C_CAN Channel

6.1 Basic Mode

As already discussed, in the Basic Mode the C_CAN node runs without using the Message Objects in the RAM. It uses the IF1 registers as Transmit Buffer and uses the IF2 registers as Receive Buffer. The following example demonstrated the entry to the basic mode.

```
  COER0_OE = 1;                // enable CAN0 output
  PIER10_IE0 = 1;              // enable CAN0 input
  CTRLR0_INIT = 1;             // stop CAN Operation

CTRLR0_CCE = 1;           // enable BTR Configuration Change
  BTR0 = value;     // set bit timing to desired value
  CTRLR0_CCE = 0;          // disable BTR Configuration Change
  CTRLR0_TEST = 1;// enable test mode
TESTR0_BASIC = 1;           // enable basic mode
CTRLR0_INIT = 0;          // start CAN Operation
```

6.1.1 Transmission of CAN Data Frames

Setting Transmission Requests

After the configuration of mask, arbitration, control and data registers, the transmission is requested by writing the Busy bit of command request register to 1. Once the CAN bus is free, the IF1 register contents are loaded into the shift register of the CAN core and the transmission gets started. Once the transmission is over, the busy bit is cleared by the CAN core.

```
  IF1CMSK0 = 0x00B2;        // =>
    // WRRD = 1;            // write
    // ARB = 1;             // access arbitration
    // CONTROL = 1;         // access control
  // DATAA = 1            // access data bytes 0-3

  IF1ARB20 = 0xA00C;        // =>
    // MSGVAL = 1;          // msg buffer valid
    // DIR = 1;             // direction = transmit
    // ID = 3               // ID = 3

  IF1MCTR0 = 0x0182;       // =>
    // TXRQST = 1;         // set transmit request
    // EOB = 1;            // end of buffer = 1
    // DLC = 2;            // data length = 2 bytes
  IF1DTA10 = 0xFFAA;       //set data
  IF1CREQ0 = 0x8000;       // transmit contents ofIF Register 1
```

6.1.2 Reception of CAN Data Frames

In basic mode the evaluation of all message object related control and status bits and evaluation of the control bits of the command mask registers is turned off. The message number (Identifier) of the command request register is not evaluated either. Hence no configuration is required as it would have been required in the normal opeartion mode. The NEWDAT bit indicates whether a new message frame is arrived. The message related information such as DLC, data bytes and identifier can then be copied to the software buffer. At the end, the NEWDAT bit should be cleared.

```
  char length;
  long int data[2];
  long int id;

if (1 == IF2MCTR0_NEWDAT)  // if new message arrived?
  {
       length = IF2MCTR0_DLC;     // get received data length
       data[0] = IF2DTA0;           // get the data bytes 0 - 3
       data[1] = IF2DTB0;           // get the data bytes 4 - 7

       if(1 == IF2ARB0_XTD)
       {
         id = (IF2ARB0 & 0x1FFFFFFF);                              // Extended ID
       }
       else
       {
         id = (((IF2ARB0 & 0x1FFFFFFFL) >> 18) & 0x000007FFL));   // Standard ID
       }
       IF2MCTR0_NEWDAT = 0;       // clear NEWDAT flag
  }
```

6.2 Normal Mode

```
  COER0_OE = 1;               // enable CAN0 output
  PIER10_IE0 = 1;             // enable CAN0 input
  CTRLR0_INIT = 1;            // stop CAN Operation

CTRLR0_CCE = 1;           // enable BTR Configuration Change
  BTR0 = value;     // set bit timing to desired value
  CTRLR0_CCE = 0;           // disable BTR Configuration Change
CTRLR0_INIT = 0;           // start CAN Operation
```

The following example demonstrated the entry to the normal mode.

6.2.1 Transmission of CAN Data Frames

Before transmitting a CAN message, the message is constructed in one of the two Interface Register Sets. Then, the message must be transferred to a message object.

Setting Transmission Requests

There are two different methods to request the transmission of a message. First, the Interface Register Set for the message can set the transmit request in the Message Control Register as bit IFxMCTRn:TXRQST. If this bit is set and the message is transferred from the Interface Register Set to the Message Buffer, the message will be transferred.

```
IF1CMSK0 = 0x00B2;          // =>
  // WRRD = 1;              // write
  // ARB = 1;               // access arbitration
  // CONTROL = 1;           // access control
  // DATAA = 1              // access data bytes 0-3

IF1ARB20 = 0xA00C;         // =>
  // MSGVAL = 1;           // msg buffer valid
  // DIR = 1;              // direction = transmit
  // ID = 3                // ID = 3

IF1MCTR0 = 0x0182;    // =>
  // TXRQST = 1;         // set transmit request
  // EOB = 1;            // end of buffer = 1
  // DLC = 2;            // data length = 2 bytes
IF1DTA10 = 0xFFAA;     //set data
IF1CREQ0 = 1;           // transfer IF Register 1 to message object 1
```

Second, a message can be configured without the TXRQST bit set. The command to transfer the message to the Message Buffer is given using the IFxCMSKn register. In this register, the TXREQ bit can be set. If the TXREQ bit is set, a transmission is requested for that message, even if the IFxMCTRn:TXRQST bit is not set.

```
IF1CMSK0 = 0x00B6;       // =>
  // WRRD = 1;           // write
  // ARB = 1;            // access arbitration
  // CONTROL = 1;        // access control
  // DATAA = 1           // access data bytes 0-3
  // TXREQ = 1           // set TXRQST

IF1ARB20 = 0xA00C;        // =>
  // MSGVAL = 1;          // msg buffer valid
  // DIR = 1;             // direction = transmit
  // ID = 3               // ID = 3

IF1MCTR0 = 0x0082;        // =>
  // EOB = 1;           // end of buffer = 1
  // DLC = 2;            // data length = 2 bytes
IF1DTA10 = 0xFFAA;        //set data
IF1CREQ0 = 1;             // transfer IF Register 1 to message object 1
```

Setting Transmission Requests with TXRQST and NEWDAT

When a message needs to be transmitted with just the updated data bytes (and everything else including the arbitration bits and control bits are same), then along with the new data bytes, the TXRQST and NEWDAT flag needs to be set together. This prevents the reset of TXRQST at the end of a transmission that may already be in progress while the data is updated. When NEWDAT is set together with the TXRQST, NEWDAT will be reset as soon as new transmission has started.

```
 IF1CMSK0 = 0x00B2;       // =>
  // WRRD = 1;             // write
  // ARB = 1;              // access arbitration
  // CONTROL = 1;          // access control
  // DATAA = 1             // access data bytes 0-3

IF1ARB20 = 0xA00C;       // =>
  // MSGVAL = 1;          // msg buffer valid
  // DIR = 1;             // direction = transmit
  // ID = 3               // ID = 3

IF1MCTR0 = 0x0182;      // =>
  // TXRQST = 1;         // set transmit request
  // EOB = 1;            // end of buffer = 1
  // DLC = 2;            // data length = 2 bytes
IF1DTA10 = 0xFFAA;    //set data
IF1CREQ0 = 1;           // transfer IF Register 1 to message buffer 1

            . . .

IF1CMSK0 = 0x0093;        // set no transmit request, wr
IF1MCTR0 = 0x8188;        // =>
  // NEWDAT = 1;          // set new data
// TXRQST = 1;            // set transmit request
  // DLC = 8;             // data length = 8 byte
IF1DTA0 = 0xFFFF0000;     // set data
IF1DTB0 = 0xAAAA5555;     // set data
IF1CREQ0 = 1;             // transfer IF Register 1 to message object 1
```

Clearing Transmission Requests

The transmission request of a message, that is stored in a message buffer but is not yet transmitted, can be cleared. As with all operations on a message, the message or at least the corresponding bits must be loaded into an Interface Register Set. Then the existing request can be cleared by clearing the `IFxMCTRn:TXRQST` and storing the message back into the message buffer.

```
IF1CMSK0 = 0x0010;                    // read control bits
IF1CREQ0 = 0x01;                      // message object 1
IF1MCTR0_TXRQST = 0;                  // ... then cancel Transmit
IF1CMSK0 = 0x0090;                    // write control
IF1CREQ0 = 0x01;                      // to message object 1
```

6.2.2 Reception of CAN Data Frames

Configuring Message Object to Receive CAN Data Frames

Before receiving a message, the Message Object has to be configured appropriately with the command mask, ID mask and arbitration fields.

The command mask register should be configured to write the mask, arbitration and control bits. The mask register configuration decides whether a single message would be received by the Message Object or it would allow a group of messages thru its acceptance filter. Here the acceptance filtering is done in such a way that it would allow just a single message with. The arbitration register is configured to receive message with Standard Identifier (11-bit) 0x004.

```
IF1CMSK0 = 0x00F0;// =>
// WRRD = 1;                // write
// MSK = 1;                 // access mask
// ARB = 1;                 // access arbitration
// CONTROL = 1;             // access control

IF1MSK20 = 0xFFFC;          // =>
// MXTD = 1;              // use XTD for acceptance filtering
// MDIR = 1;                // use DIR for acceptance filtering
// use ID28-ID18 for acceptance filtering

IF1ARB20 = 0x8010;          // =>
// MSGVAL = 1;              // msg buffer valid
// XTD = 0;                 // id = standard (11-bit)
// DIR = 0;                 // direction = receive
// ID = 4;// ID = 0x004

IF1MCTR0 = 0x1080;         // =>
// UMASK = 1;  // use mask for acceptance filtering
// EOB = 1; // end of buffer = 1

IF1CREQ0 = 2;             // transfer IF Register 1 to message object 2
```

The same can also be done by simply clearing the `UMSK` bit of `IF1MCTR0` register to 0. This way there is no need to configure mask registers at all.

```
IF1CMSK0 = 0x00B0;// =>
// WRRD = 1;                // write
// MSK = 0;                 // don't access mask
// ARB = 1;                 // access arbitration
// CONTROL = 1;             // access control

IF1ARB20 = 0x8010;          // =>
// MSGVAL = 1;              // msg buffer valid
// XTD = 0;                 // id = standard (11-bit)
// DIR = 0;                 // direction = receive
// ID = 4;// ID = 0x004

IF1MCTR0 = 0x0080;        // =>
// UMASK = 0;  // ignore mask
// EOB = 1; // end of buffer = 1

IF1CREQ0 = 2;            // transfer IF Register 1 to message object 2
```

Receiving CAN Data Frames

When a CAN message is received, this is signaled by the NEWDAT register. This register also shows, in which message buffer the message was received. This message must then be transferred into an Interface Register Set. From there, the data can be read. Also, the flag in the NEWDAT and INTPND register must be cleared. This can be done by writing a value equal to 0x003F to command mask register and then the number of the message object to the command request register. Once the all the information is transferred from the message object to the IF2 registers, then it can be then copied to the software buffer.

```
  char length;
  long int data[2];
  long int id;

  if (NEWDT10 & 2)
  {
// read message-buffer 2
    IF2CMSK0 = 0x003F;              // =>
       // WRRD = 0                 // read
       // MASK = 0                  // don't transfer mask
       // ARB = 1                   // transfer arbitration
       // CONTROL = 1               // transfer control
       // CIP = 1                   // clear INTPND bit (if RXIE is set)
       // TXREQ = 1                 // clear NEWDAT bit
       // DATAA = 1                 // transfer data bytes 0-3
       // DATAB = 1                 // transfer data bytes 4-7
    IF2CREQ0 = 2;                 // transfer Message Register 2 to IF 2

  length = IF2MCTR0_DLC;   // get received data length
data[0] = IF2DTA0;           // get the data bytes 0 - 3
data[1] = IF2DTB0;           // get the data bytes 4 - 7

  id = (((IF2ARB0 & 0x1FFFFFFFL) >> 18) & 0x000007FFL));   //Standard ID
  }
```

6.2.3 Reception of CAN Data Frames using FIFO Buffer

C_CAN provides the possibilty to concatenate two or more message objects to form a FIFO buffer. The identifiers and the masks of these objects have to be programmed to the matching values and the EOB bit of all such message objects (to be concatenated) except the last messsage object of a FIFO buffer is required to be set to 0. The EOB bit of the last message object of a FIFO buffer is set to 1.

Configuring Message Objects to Receive CAN Data Frames

The following snippet of code demonstrates how a FIFO buffer described above is realized. Here 4 message objects from 2 to 5 are concatenated to form a FIFO buffer with the same message id 0x004.

```
int cnt;

IF1CMSK0 = 0x00F0;// =>
// WRRD = 1;          // write
// MSK = 1;           // access mask
// ARB = 1;           // access arbitration
// CONTROL = 1;       // access control

IF1MSK20 = 0xFFFC;    // =>
// MXTD = 1;         // use XTD for acceptance filtering
// MDIR = 1;          // use DIR for acceptance filtering
                      // use ID28-ID18 for acceptance filtering

IF1ARB20 = 0x8010;    // =>
// MSGVAL = 1;        // msg buffer valid
// XTD = 0;           // id = standard (11-bit)
// DIR = 0;           // direction = receive
// ID = 4;// ID = 0x004

for (cnt = 2; cnt <= 5; cnt++)
{
       if (cnt < 5)
       {
              IF1MCTR0 = 0x1000;  // =>
              // UMASK = 1;       // use mask for acceptance filtering
              // EOB = 0;         // end of buffer = 0
       }
       else
       {
              IF1MCTR0 = 0x1080;  // =>
              // UMASK = 1;       // use mask for acceptance filtering
              // EOB = 1;         // end of buffer = 1
       }

       IF1CREQ0 = cnt;    // transfer IF Register 1 to message object 2to 5
}
```

Receiving CAN Data Frames

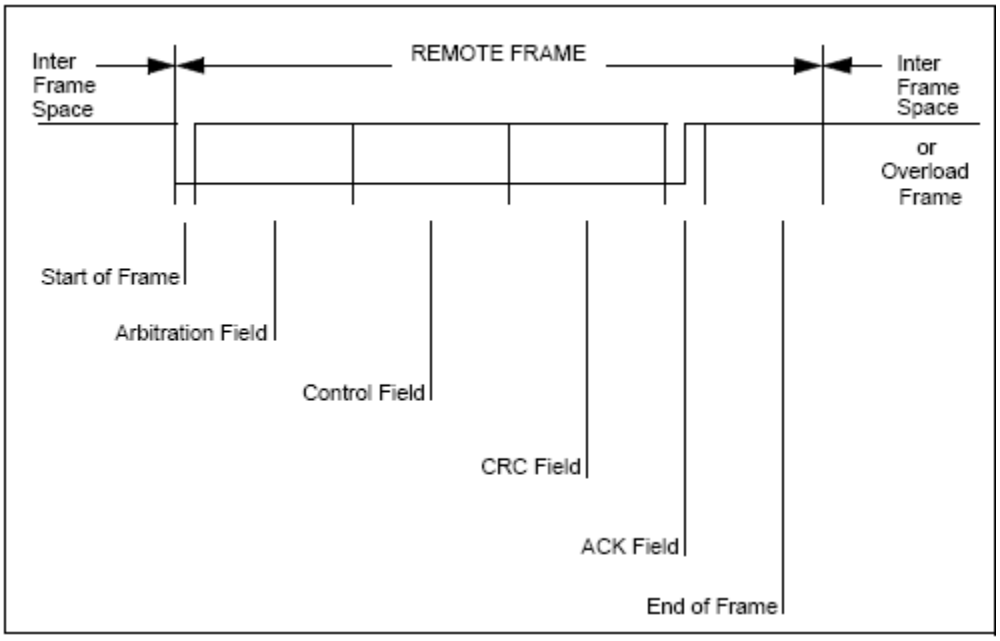
The following snippet of code demonstrates how a FIFO buffer data can be read and transferred. Here it is considered that message objects 2 to 5 are concatenated.

```
  typedef struct
  {
    char length;
    long int data[2];
    long int id;
  } received_data;

int cnt;
  received_data RX_data[4];

  for (cnt = 2; cnt <= 5; cnt++)
  {
  if ((NEWDT10 & (0x01 << (cnt-1))))     // new data arrived?
    {
  // read message-buffer
      IF2CMSK0 = 0x003F;          // =>
       // WRRD = 1                  // read
       // MASK = 0                  // don't transfer mask
       // ARB = 1                   // transfer arbitration
       // CONTROL = 1               // transfer control
       // CIP = 1                   // clear INTPND bit (if RXIE is set)
       // TXREQ = 1                 // clear NEWDAT bit
       // DATAA = 1                 // transfer data bytes 0-3
       // DATAB = 1                 // transfer data bytes 4-7
      IF2CREQ0 = cnt;             // transfer Message Register 2 to IF 2

      RX_data[cnt-2].length = IF2MCTR0_DLC;    // get received data length
  RX_data[cnt-2].data[0] = IF2DTA0;              // get the data bytes 0 - 3
  RX_data[cnt-2].data[1] = IF2DTB0;              // get the data bytes 4 - 7

      RX_data[cnt-2].id = (((IF2ARB0 & 0x1FFFFFFFL) >> 18) & 0x000007FFL));
                                                 // Standard ID
    }

    if (1 == IF2MCTR0_EOB)        // end of buffer reached?
    {
       break;
    }
  }
```

6.2.4 Transmission of CAN Remote Frames

CAN Remote frame is different than the data frame from the fact that it does not have data bytes and the RTR bit is set. The following figure shows a typical remote frame (remote transmission request).

Figure 12. CAN Remote Frame



To transmit the remote frame with a given identifier the message object needs to be configured as **Receive Object** and the `TXRQST` bit of the message control register needs to be set to 1 in order to tramsmit the same. Once the bus is found free the CAN controller will transmit the remote transmission request on the bus. If the matching data frame is received by **Receive Object** before the remote frame could be transmitted, the `TXRQST` bit is automatically reset.

```
IF1CMSK0 = 0x00B0;          // =>
// WRRD = 1;                // write
// MSK = 0;                 // don't access mask
// ARB = 1;                 // access arbitration
// CONTROL = 1;             // access control

IF1ARB20 = 0x8010;          // =>
// MSGVAL = 1;              // msg buffer valid
// XTD = 0;                 // id = standard (11-bit)
// DIR = 0;                 // direction = receive
// ID = 4;                  // ID = 0x004

IF1MCTR0 = 0x0182;          // =>
// TXRQST = 1;              // set transmit request for remote frame
// UMASK = 0;               // ignore mask
// EOB = 1;                 // end of buffer = 1
// DLC = 2;                 // requested data length = 2 bytes

IF1CREQ0 = 2;               // transfer IF Register 1 to message object 2
```

6.2.5 Reception of CAN Remote Frames

To receive the remote transmision request with a given identifier the message object needs to be configured as **TransmitObject** and the RMTEN bit of the message control register needs to be set to 1.

Once the remote transmission request is transmitted by the **Receive Object** (described in the above section), it would be received by **TransmitObject** since both has the matching identifiers. Since the RMTEN bit of message control register is set, the TXRQST of this message object is set automatically. Thats how the remote frame would be autonomously be answered by the data frame. That means the data frame with the identifier 0x004 with 2 data bytes 0xAA and 0xFF is transmitted on the bus by **TransmitObject**.

```
IF1CMSK0 = 0x00B2; // =>
   // WRRD = 1;            // write
   // ARB = 1;             // access arbitration
   // CONTROL = 1;         // access control
   // DATAA = 1            // access data bytes 0-3

 IF1ARB20 = 0xA010;       // =>
   // MSGVAL = 1;          // msg buffer valid
   // DIR = 1;             // direction = transmit
   // ID = 4               // ID = 0x004

 IF1MCTR0 = 0x0282;       // =>
   // RMTEN = 1;           // remote enable
   // EOB = 1;             // end of buffer = 1
   // DLC = 2;             // data length = 2 bytes
 IF1DTA10 = 0xFFAA;       //set data
 IF1CREQ0 = 1;            // transfer IF Register 1 to message object 1
```

6.3 Interrupts

If several interrupts are pending, the CAN Interrupt Register will point to the pending interrupt with the highest priority, disregarding their chronological order. An interrupt remains pending until the CPU has cleared it.

6.3.1 Transmission Interrupts

Setting Transmission Requests

For the transmission hadling using interrupts, the IE bit of CAN control register needs to be set and also the TXIE bit of message control register needs to be set. The rest of the transmit object initialization remains same.

```
CTRLR0_IE = 1;            // module interrupt enable

IF1CMSK0 = 0x00B2;        // =>
  // WRRD = 1;            // write
  // ARB = 1;             // access arbitration
  // CONTROL = 1;         // access control
  // DATAA = 1            // access data bytes 0-3

IF1ARB20 = 0xA00C;        // =>
  // MSGVAL = 1;          // msg buffer valid
  // DIR = 1;             // direction = transmit
  // ID = 3               // ID = 3

IF1MCTR0 = 0x0982;        // =>
  // TXIE = 1;            // transmit interrupt enable
  // TXRQST = 1;          // set transmit request
  // EOB = 1;             // end of buffer = 1
  // DLC = 2;             // data length = 2 bytes
IF1DTA10 = 0xFFAA;        //set data
IF1CREQ0 = 1;             // transfer IF Register 1 to message object 1
```

Transmit Interrupt Handler

Transmssion handler part of the C_CAN ISR need to be determine the cause of the interrupt first. This is done with the help of interrrupt, interrupt pending, message valid, new data and transmission request register as shown below. It should also clear the interrupt pending bit using the command mask register.

```
unsigned int IntPointer = 0x0000;
unsigned long IntBuffer;

IntPointer = INTR0;

IntPointer = IntPointer & 0x00FF; // use only the lower six bits

if( (IntPointer >= 1) && (IntPointer <= 0x20) )       // valid buffer number
{
       IntBuffer = 0x01 << (IntPointer-1);
       // Check whether the interrupt source is a valid buffer
       if(((MSGVAL0 & IntBuffer) != 0) && ((INTPND0 & IntBuffer) != 0)))
       // message buffer is valid
       {
              // Check whether the interrupt cause is receive or transmit
              if( (NEWDT0 & IntBuffer) != 0 )         // is a receive interrupt
              {
                     // call the receive handler (must clear NEWDAT & INTPND)
              }
              else if( (TREQR0 & IntBuffer) == 0 )    // is transmission done
              {
                     IF2CMSK0 = 0x0018;          // =>
                     // WRRD = 0                 // read
                     // CONTROL = 1              // transfer control
                     // CIP = 1                  // clear INTPND bit
                     IF2CREQ0 = IntPointer;      // start transfer
              }
       }
}
```

6.3.2 Receive Interrupts

Configuring Receive Object

For the reception handling using interrupts, the IE bit of CAN control register needs to be set and also the RXIE bit of message control register needs to be set. The rest of the receive object initialization remains same.

```
CTRLR0_IE = 1;              // module interrupt enable

IF1CMSK0 = 0x00B0;          // =>
// WRRD = 1;                // write
// MSK = 0;                 // don't access mask
// ARB = 1;                 // access arbitration
// CONTROL = 1;             // access control

IF1ARB20 = 0x8010;          // =>
// MSGVAL = 1;              // msg buffer valid
// XTD = 0;                 // id = standard (11-bit)
// DIR = 0;                 // direction = receive
// ID = 4;                  // ID = 0x004

IF1MCTR0 = 0x0480;          // =>
// RXIE = 1;                // receive interrupt enable
// UMASK = 0;               // ignore mask
// EOB = 1;                 // end of buffer = 1

IF1CREQ0 = 2;               // transfer IF Register 1 to message object 2
```

Receive Interrupt Handler

Reception handler part of the C_CAN ISR needs to determine the cause of the interrupt first. This is done with the help of interrupt, interrupt pending, message valid and new data register as shown below. It should also clear the interrupt pending bit and new data bit using the command mask register.

```
unsigned int IntPointer = 0x0000;
unsigned long IntBuffer;

IntPointer = INTR0;

IntPointer = IntPointer & 0x00FF; // use only the lower six bits

if( (IntPointer >= 1) && (IntPointer <= 0x20) )       // valid buffer number
{
       IntBuffer = 0x01 << (IntPointer-1);
       // Check whether the interrupt source is a valid buffer
       if(((MSGVAL0 & IntBuffer) != 0) && ((INTPND0 & IntBuffer) != 0)))
       // message buffer is valid
       {
              // Check whether the interrupt cause is receive or transmit
              if( (NEWDT0 & IntBuffer) != 0 )         // is a receive interrupt
              {
                     IF2CMSK0 = 0x003F;  // =>
                     // WRRD = 0         // read
                     // MASK = 0         // don't transfer mask
                     // ARB = 1          // transfer arbitration
                     // CONTROL = 1      // transfer control
                     // CIP = 1          // clear INTPND bit
                     // TXREQ = 1        // clear NEWDAT bit
                     // DATAA = 1        // transfer data bytes 0-3
                     // DATAB = 1        // transfer data bytes 4-7
                     IF2CREQ0 = IntPointer;     // start transfer
              }
       }
}
```

6.3.3 Status Change Interrupts

Configuration

For the status change handling using interrupts, the IE, SIE and EIE bit of CAN control register needs to be set.

```
CTRLR0_IE = 1;           // module interrupt enable
CTRLR0_SIE = 1;           // status change interrupt enable
CTRLR0_EIE = 1;           // error interrupt enable
```

Status Interrupt Handler

A Status Interrupt is generated by bits BOFF and EWARN (Error Interrupt) or by RXOK, TXOK, and LEC(Status Change Interrupt) assuming that the corresponding enable bits in the CAN Control Register are set. In case of Status Interrupt the Interrupt register INTRn reflects a value of 0x8000. Reading the Status Register STATRn will clear the Status Interrupt value (0x8000) in the Interrupt Register. Then the value of the status register determines the cause of status interrupt and then the corresponding action can be taken. The following is an example of Status handler part of the C_CAN ISR.

```
unsigned int IntPointer = 0x0000;
unsigned short canstatus;

IntPointer = INTR0;

if((IntPointer &0x8000)==0x8000)// Status Interrupt??
{
// Read the Status Register
// This will also clear pending Status/Error Interrupt
    canstatus = STATR0;

if((canstatus &0x80)==0x80)// BusOff??
{
// Busoff handling
...
}
if((canstatus &0x40)==0x40)// Error Warning??
{
// Error Warning handling
...
}
if((canstatus &0x20)==0x20)// Error Passive??
{
// Error Passive handling
...
}
if((canstatus &0x10)==0x10)// RXOK??
{
// Clear RXOK Flag in Status register
STATR0 =~(0xF0|0x10);
...
}
if((canstatus &0x08)==0x08)// TXOK??
{
// Clear TXOK Flag in Status register
STATR0 =~(0xF0|0x08);
...
}
if((canstatus &0x07)!=0x00)// LEC??
{
// Clear LEC in Status register
STATR0 =~(0xF0|0x07);
...
}
}
```

7 Additional Information

Information about Cypress Microcontrollers can be found on the following Internet page:

http://www.cypress.com/cypress-microcontrollers

The software example related to this application note is:

96340_can

96340_can_uart_terminal

It can be found on the following Internet page:

http://www.cypress.com/products/16FX

Document History

Document Title: AN204777 - 16FX Family, C_CAN-Controller

Document Number: 002-04777

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	-	MKEA	09/29/2006	Initial Release, based on application note mcu-an-300035
			07/05/2007	Reviewed the document and updated with review findings
			11/12/2007	Fix typos
			04/11//2008	Several corrections
			05/19/2008	Fix extended IDs
			12/15/2008	Use different IF-register sets for writing into and reading from message buffers – this should avoid corruption when using interrupts
			11/30/2012	Source code in chapter “Status Interrupt Handler” changed
*A	5060908	MKEA	12/22/2015	Migrated Spansion Application Note from MCU-AN-300228-E-V16 to Cypress format
*B	5834999	AESATP12	07/27/2017	Updated logo and copyright.
*C	6036665	NOFL	01/20/2018	Updated logo. Updated links. Updated Sales page and Copyright year.

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 MCU

Cypress Developer Community

Community | 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, 2006-2018. 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. No computing device can be absolutely secure. Therefore, despite security measures implemented in Cypress hardware or software products, Cypress does not assume any liability arising out of any security breach, such as unauthorized access to or use of a Cypress product. In addition, the products described in these materials may contain design defects or errors known as errata which may cause the product to deviate from published specifications. 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.