INFINEON BOARD(TC22x) DOCUMENT

Features :

- 32 bit controller(1 core), RISC Architecture.
- Operating frequency : 133MHz, (Note : OSC--->20MHz)
- 16 Channel DMA
- ADC--->12Bit--->12+12(24) channels---->2 converter.
- Timer-->5 General Purpose Timer(GPT) and CCU6(Capture Compare Unit 6)
- General Timer Module(GTM)--->5 timer Modules(TIM,TOM,DTM,CMU/ICM,TBU).
- CAN--->1 Module--->3 Nodes---->128 Message Objects And 1 CANFD
- QSPI--->4 channel
- ASCLIN--->2 modules.
- Memory :

FLASH	: 1MB
DSPR	: 88 Kbyte
PSPRAM	: 8 Kbyte
BROM	: 32 Kbyte
DFLASH	: 96 Kbyte
PRGM CACHE	: 8 Kbyte

Three Operating Modes :

Prescaler Mode :

Fpll1=Fosc/k1

Fpll2 = Fosc/k2.

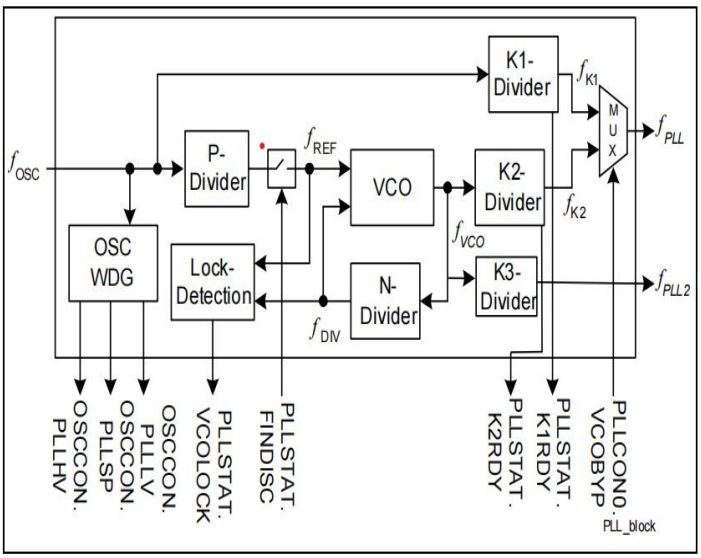
Normal Mode :

Fpll1=Fosc/k1

Free running Mode :

Fpll1 = (N/(p*k2))*Fosc

Fpll2 = (N/(p*k3))*Fosc



***Default Values : FOSC : 20MHz, CPU runs in Normal Mode(After reset) with a frequency of 100MHz

General Purpose Input and Output Port

Pins can be configured as Input and Output.

Input Pin : can be pull_Up or pull_Down or No_pull.
Output Pin : can be Push_Pull or Open_Drain.

Each port has 15 pins.And all pins are in input mode by default(i.e after reset).

***Note : Input /Output can be Selected by PCx[3:0] bits in IOCR register For alternate functionality ref data-sheet table 2-14

Table 14-5 PCx Coding

PCx[4:0]	I/O	Characteristics	Selected Pull-up / Pull-down / Selected Output Function
$0XX00_B$	Input[1]	–	No input pull device connected, tri-state mode
$0XX01_B$			Input pull-down device connected
$0XX10_B$			Input pull-up device connected
$0XX11_B$			No input pull device connected, tri-state mode
10000_B	Output	Push-pull	General-purpose output
10001_B			Alternate output function 1
10010_B			Alternate output function 2
10011_B			Alternate output function 3
10100_B			Alternate output function 4
10101_B			Alternate output function 5
10110_B			Alternate output function 6
10111_B			Alternate output function 7
11000_B		Open-drain	General-purpose output
11001_B			Alternate output function 1
11010_B			Alternate output function 2
11011_B			Alternate output function 3
11100_B			Alternate output function 4
11101_B			Alternate output function 5
11110_B			Alternate output function 6
11111_B			Alternate output function 7

1) Only input functions apply for P40 and P41.

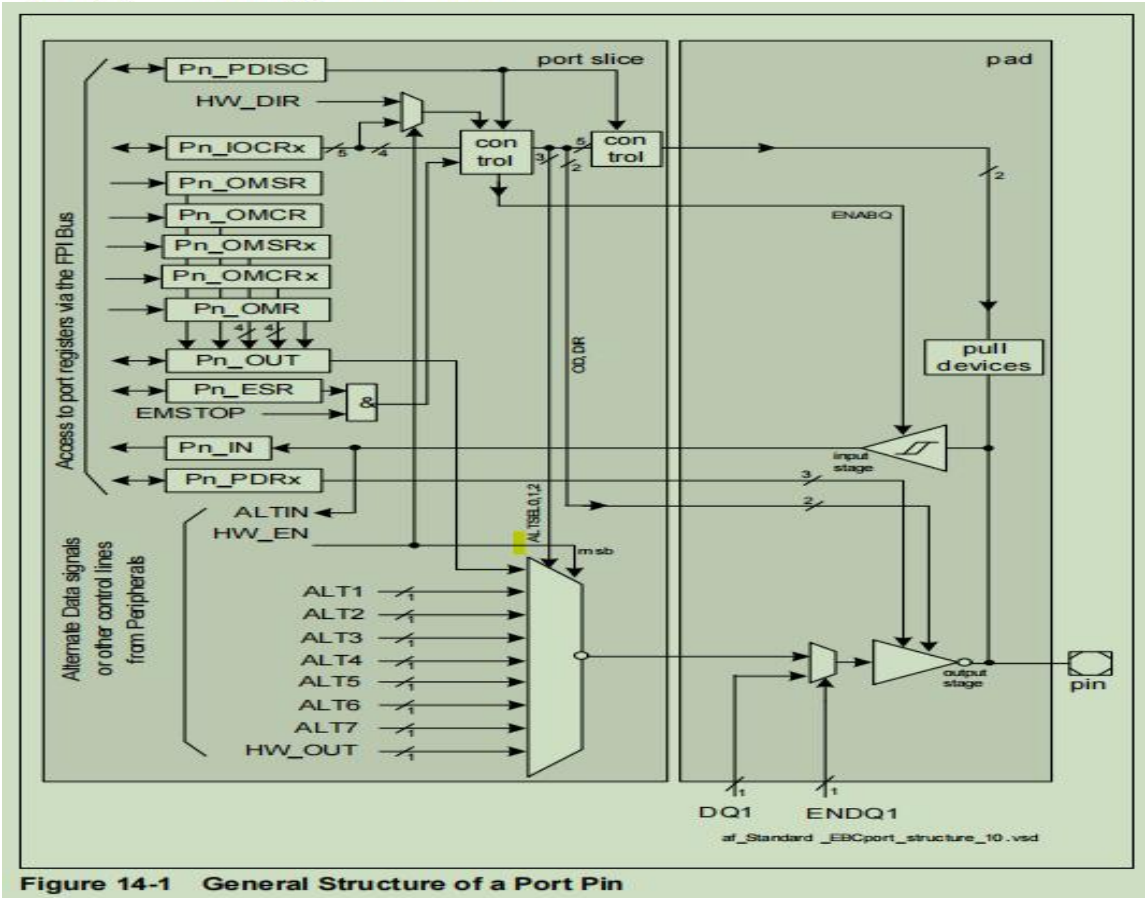


Figure 14-1 General Structure of a Port Pin

Output Mode : When any pins are configured as general-purpose output , you can change its states by changing the bit of Pn_OMSR(Output Modification Set Register) and Pn_OMCR(Output Modification Clear Register).

If any on-chip peripheral use the pin for output signals, the alternate output lines ALT1 to ALT7 can be switched via the multiplexer to the output driver. The data written into the output register Pn_OUT by software can be used as input data to an on-chip peripheral.

All digital GPIO lines has an emergency stop logic. This logic makes it possible to individually disconnect outputs and put them onto a well defined logic state in an emergency case. In an emergency case, the pin is switched to input function in tri-state mode. The Emergency Stop Register Pn_ESR determines whether an output is enabled or disabled in an emergency case.I this way you can Pin damage will never happens.

Port Address :

Table 14-3 Registers Address Space

Module	Base Address	End Address	Note
P00	$F003\ A000_H$	$F003\ A0FF_H$	13 pins; pins[12:0]
P02	$F003\ A200_H$	$F003\ A2FF_H$	9 pins; pins[8:0]
P10	$F003\ B000_H$	$F003\ B0FF_H$	5 pins; pins[6:5], [3:1]
P11	$F003\ B100_H$	$F003\ B1FF_H$	8 pins; pins [3:2], 6, [12:8]
P13	$F003\ B300_H$	$F003\ B3FF_H$	4 pins; pins[3:0]
P14	$F003\ B400_H$	$F003\ B4FF_H$	9 pins; pins[8:0]
P15	$F003\ B500_H$	$F003\ B5FF_H$	9 pins; pins[8:0]
P20	$F003\ C000_H$	$F003\ C0FF_H$	12 pins; pins [14:6], [3:2], 0
P21	$F003\ C100_H$	$F003\ C1FF_H$	6 pins; pins[7:2]
P22	$F003\ C200_H$	$F003\ C2FF_H$	5 pins; pins[4:0]
P23	$F003\ C300_H$	$F003\ C3FF_H$	1 pin; pin1
P33	$F003\ D300_H$	$F003\ D3FF_H$	13 pins; pins[12:0]
P34	$F003\ D400_H$	$F003\ D4FF_H$	4 pins; pins[3:0]
P40	$F003\ E000_H$	$F003\ E0FF_H$	12 pins; pins[11:0]
P41	$F003\ E100_H$	$F003\ E1FF_H$	12 pins; pins[11:0]

Important Registers :

1.**Port Input/Output Control Registers :** Helps in Configuring different functions(like GPIO, alternate Function).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PC3					0			PC2					0		
rw					r			rw					r		

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PC1					0			PC0					0		
rw					r			rw					r		

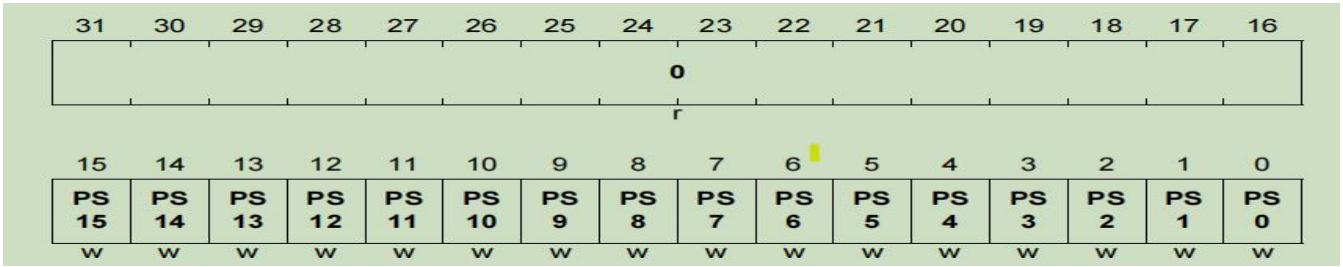
Field	Bits	Type	Description
PC0, PC1, PC2, PC3	[7:3], [15:11], [23:19], [31:27]	rw	**Port Control for Port n Pin 0 to 3** This bit field determines the Port n line x functionality (x = 0-3) according to the coding table (see Table 14-5).

2. ***Port Output Modification Register :*** Helps in driving output state to either LOW or HIGH.

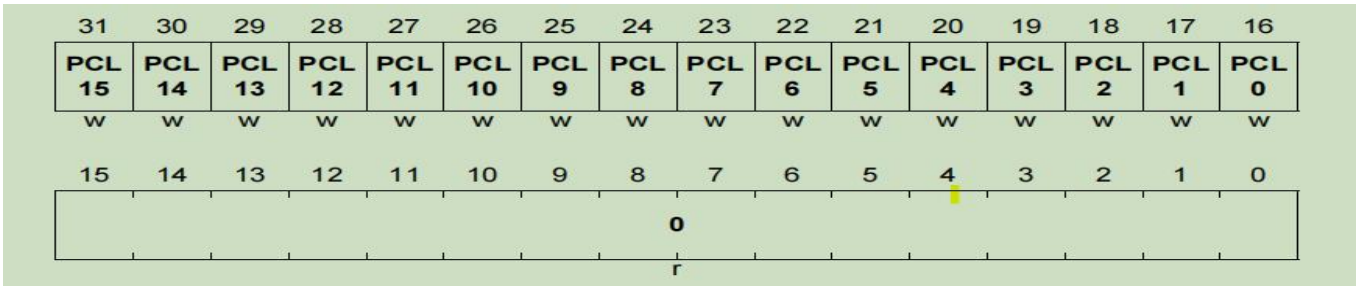
PS(Pin Set) : will Set the pin and PCL(Pin Clear) will Clear the Pin .

Note : If PS =1 and PCL = 1, then PIN will set to in Toggle mode.

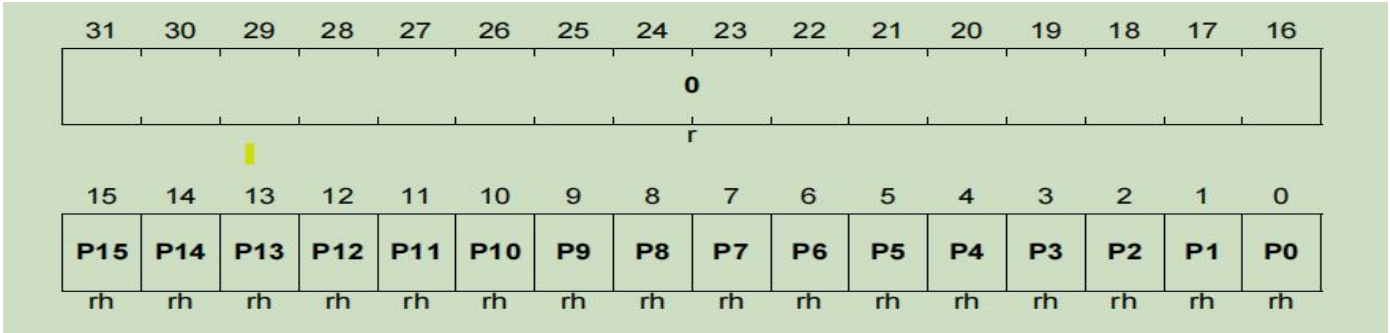
3. **Port Output Modification Set Register(PxOMSR) :** Alternate Register that will only Set the Pins state.



4. **Port Output Modification Clear Register(PxOMCR) :** Alternate Register that will only Clear the Pins state.



5.**Port Input Register(Px_IN) :** Helps to read PIN state. This readonly Register.



APIs : Related to Configure Pin to work in Different Modes : file to look for below APIs---->IfxPort.c

IfxPort_setPinMode(Ifx_P *port, uint8 pinIndex, IfxPort_Mode mode);

Affected Reg : Pn_IOCR(Pn_Input_output Control Register)
Port : will point to port Address
pinIndex : Pin_Number
Mode : Input/output/alternate_functions

IfxPort_setPinLow(Ifx_P *port, uint8 pinIndex) : Set Pin to Logic High[i.e in Push-pull Config]**

Affected Reg : Pn_OMSR(Pn_Output Modification Register Set)
Port : will point to port Address
pinIndex : Pin_Number

IfxPort_setPinHigh(Ifx_P *port, uint8 pinIndex) : Set Pin to Logic Low[i.e in Push-pull Config]**

Affected Reg : Pn_OMSC(Pn_Output Modification Register Clear)
Port : will point to port Address
pinIndex : Pin_Number

IfxPort_setPinState(Ifx_P *port, uint8 pinIndex, IfxPort_State action)) : Set Pin to Logic Low/High

Port : will point to port Address
pinIndex : Pin_Number
Action : High/Low

Example : To LED Blinking

Note : LED is Connected to PORT2.0,PORT2.1,PORT2.2,PORT2.3,PORT2.4,PORT2.5 [6LEDs]
PORT11.10, PORT11.11 [2Leds]

```
IfxPort_setPinMode(& PORT2,PIN0,IfxPort_OutputIdx_general);

while(1)
{
 IfxPort_setPinLow(& PORT2,PIN0);    //To Turn On LED
For(int I = 0; I < 9999; I++);                 //  Delay

IfxPort_setPinHigh(& PORT2,PIN0);    //To Turn Off LED
For(int I = 0; I < 9999; I++);       //  Delay
}
```

Other API's :

```
void IfxPort_togglePin(Ifx_P *port, uint8 pinIndex);

boolean IfxPort_getPinState(Ifx_P *port, uint8 pinIndex);

Ifx_P *IfxPort_getAddress(IfxPort_Index port);

IfxPort_Index IfxPort_getIndex(Ifx_P *port);
```

General Purpose Timer(GPT12) Module

---> Total Five 16 bit Timer.

----> Two Groups :

GPT1----> Three Timers of 16 bit. The maximum resolution is fGPT/4.
1. Timer 3[T3]----> Main Timer(Core Timer)
2. Timer 2[T2]----> Auxiliary Timer1
3. Timer 4[T4]----> Auxiliary Timer2

GPT2----> Two Timers of 16 bit.The maximum resolution is fGPT/2.
1.Timer 6[T6]----> Main Timer(Core Timer)
2. Timer 5[T5]----> Auxiliary Timer

GPT1 Operating Modes : Has 4 Modes

1. Timer Mode
2. Gated Timer Mode
3. Counter Mode
4. Incremental Interface Mode

Note : The interrupt requests of GPT1 are signaled on service request lines SR0(T2), SR1(T3), and SR2(T4).

GPT2 Operating Modes : Has 3 Modes

1. Timer Mode : Used to generate Delay
2. Gated Timer Mode : Can control start and stop using external pin
3. Counter Mode : Used to Count external events

Note : The interrupt requests of GPT2 are signaled on service request lines SR3(T5), SR4(T6), and SR5(CAPREL Service Request)

Each Timer can count up or Down and this configuration can be done through Software(By configuring the UD bit of Timer Control Register(TCON) or by altered by a signal at the External Up/Down control input TxEUD (alternate pin function).

An overflow/underflow of core timer [T3/T6] is indicated by the Output Toggle Latch T3OTL and T6OTL, whose state may be output on the associated pin T3OUT and T6OUT (alternate pin function).

Note: The auxiliary timers have no output toggle latch and no alternate output function.

The auxiliary timers T2 and T4 may additionally be concatenated with the core timer T3 (through T3OTL) or may be used as capture or reload registers for the core timer T3.

The Capture/Reload register CAPREL can be used to capture the contents of timer T5, or to reload timer T6.

**The Timers can be start and Stop using Timer Tx Run Bit in Timer Control Register

Ex :

T2R	6	rw	**Timer T2 Run Bit** 0_B Timer T2 stops 1_B Timer T2 runs *Note: This bit only controls timer T2 if bit T2RC = 0.*

Block Diagram of Two Timer Blocks (GPT1 and GPT2) :

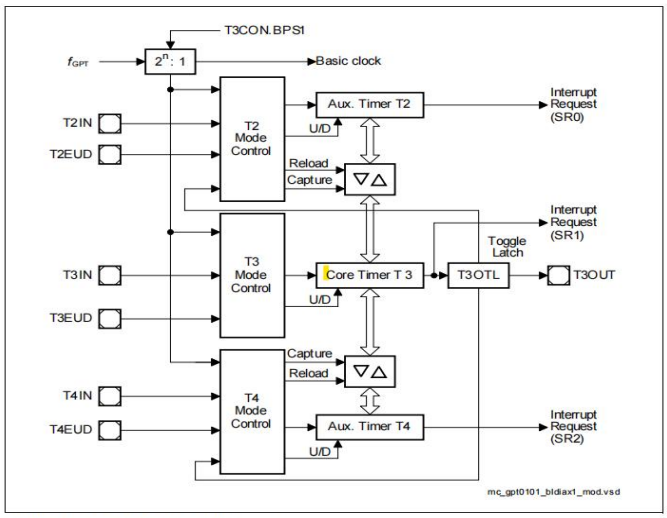


Figure 26-1 GPT1 Block Diagram

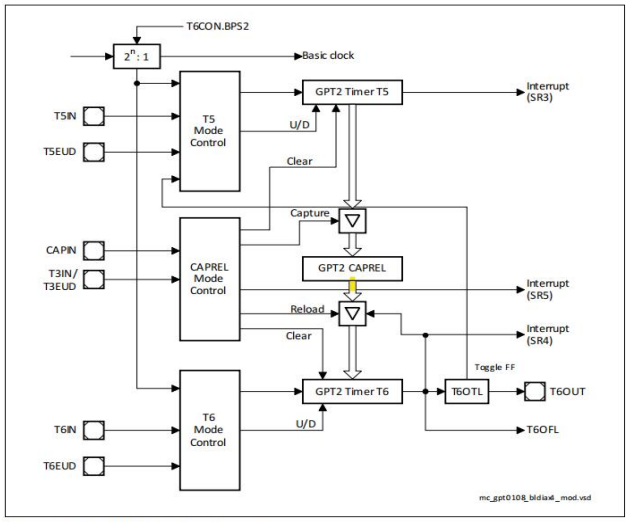


Figure 26-18 GPT2 Block Diagram

Timer clock Source is fgpt, which is connected to System Peripheral Bus(100MHz). Speed of Timer clock can be Controlled by two Registers, one by setting appropriate value in BSP bits and TxI bits in Timer Control Register(TxCON) .

(Timer Mode and Gated Timer Mode)

Individual Prescaler for Tx	Common Prescaler for Module Clock[1]			
	$BPS2 = 01_B$	$BPS2 = 00_B$	$BPS2 = 11_B$	$BPS2 = 10_B$
$TxI = 000_B$	2	4	8	16
$TxI = 001_B$	4	8	16	32
$TxI = 010_B$	8	16	32	64
$TxI = 011_B$	16	32	64	128
$TxI = 100_B$	32	64	128	256
$TxI = 101_B$	64	128	256	512
$TxI = 110_B$	128	256	512	1024
$TxI = 111_B$	256	512	1024	2048

Operating Modes :

Timer Mode : Timer Mode for the timer is selected by setting bitfield TxM in register TxCON. In Timer Mode, Tx is clocked with the module's input clock fGPT divided by two programmable prescalers controlled by bitfields BPSx and TxI in register TxCON.

Gated Timer Mode : Gated Timer Mode for the core timer T3 is selected by setting bitfield T3M in register T3CON to 010B or 011B. Bit T3M.0 (T3CON.3) selects the active level of the gate input. However, the input clock to the timer in this mode is gated by the external input pin TxIN (Timer Tx External Input).

To enable this operation, the associated pin T3IN must be configured as ouput mode.

If TxIN = 1, timer will start, TxIN = 0 , timer will stop. The timer will only run if TxR is 1 and the gate is active. It will stop if either TxR is 0 or the gate is inactive.

Timer in Counter Mode : Counter Mode for the core timer Tx(T3,T4,T5,T6,T2) is selected by setting bitfield TxM in register TxCON In Counter Mode, timer Tx is clocked by a transition at the external input pin TxIN. The event causing an increment or decrement of the timer can be a positive, a negative, or both a positive and a negative transition at this line. Bitfield TxI in control register TxCON selects the triggering transition.

**Note : For Counter Mode operation, pin T3IN must be configured as input.

Points to remember on Auxiliary Timers :

The start/stop function of the auxiliary timers can be remotely controlled by the T3 run control bit. Several timers may thus be controlled synchronously.

Note: The auxiliary timers have no output toggle latch and no alternate output function.
Each of the auxiliary timers T2 and T4 can be started or stopped by software in two different ways:

- Through the associated timer run bit (T2R or T4R). In this case it is required that the respective control bit TxRC = 0.
- Through the core timer's run bit (T3R). In this case the respective remote control bit must be set (TxRC = 1).

The selected run bit is relevant in all operating modes of T2/T4. Setting the bit will start the timer, clearing the bit stops the timer.

In Gated Timer Mode, the timer will only run if the selected run bit is set and the gate is active (high or low, as programmed).

In Reload Mode : The content of Auxiliary count register can loaded into core timer by two ways :
1. TxIN
2. TxOTL

****Note : When a T3OTL transition is selected for the trigger signal, service request SR1 will also become active, indicating T3's overflow or underflow. Modifications of T3OTL via software will NOT trigger the counter function of T2/T4.

Note : Timer_Module Base_Address is 0xF0002E00U

Procedure to Initialize and Configure the Timers :

1. Enable Timer Module :
   IfxGpt12_enableModule(Ifx_GPT12 *gpt12);
   Affected Reg : Clock Control register---> bit DISR (Module En/Dis) bit.
   gpt12 : Should pass base address of Timer Module

2. Set Prescaler Values :
   a.void IfxGpt12_setGpt2BlockPrescaler(Ifx_GPT12 *gpt12, IfxGpt12_Gpt2BlockPrescaler bps2);
   Or
   void IfxGpt12_setGpt1BlockPrescaler(Ifx_GPT12 *gpt12, IfxGpt12_Gpt1BlockPrescaler bps1);
   Affected Reg : Control register---->BPS1 bit (for GTP1) and BPS2 bit(for GPT2)
   gpt12 : Should pass base address of Timer Module
   bps2 : Value ans should be multiple of 2

   b. Tx---Can be T3,T2,T4,T5 and T6
   void IfxGpt12_Tx_setTimerPrescaler(Ifx_GPT12 *gpt12, IfxGpt12_TimerInputPrescaler val);
   Affected Reg : Control register of Tx timer(TxCON)---->TxI bit
   gpt12 : Should pass base address of Timer Module
   val : Value ans should be multiple of 2

3. Select mode of Operation : Tx--->Can be T3,T2,T4,T5 and T6

void IfxGpt12_Tx_setMode(Ifx_GPT12 *gpt12, IfxGpt12_Mode mode);

Affected Reg : Control register of Tx timer of(TxCON)---->TxM bit

gpt12 : Should pass base address of Timer Module

val : Pls Refer Manual(ex : page no:2631 for T3)

4. Set Direction of Counter (I.e up/Down counting) : Tx--->Can be T3,T2,T4,T5 and T6

void IfxGpt12_Tx_setTimerDirection(Ifx_GPT12 *gpt12, IfxGpt12_TimerDirection direction);

Affected Reg : Control register of Tx timer of(TxCON)---->TxUD bit

gpt12 : Should pass base address of Timer Module

val : 0-->counts Up, 1--->counts down

5. Configure interrupt if needed

6. Start the Timer : Tx--->Can be T3,T2,T4,T5 and T6

void IfxGpt12_Tx_run(Ifx_GPT12 *gpt12, IfxGpt12_TimerRun runTimer);

Affected Reg : Control register of Tx timer of(TxCON)---->TxR bit

gpt12 : Should pass base address of Timer Module

val : 0-->Stops, 1--->Starts

Example to Generate 1 sec of delay using Timer : Timer Module Address is 0xF0002E00

Note : frequency for Timer module is coming from SPB i.e 100MHz

STEP 1:

```
/**********Providing Clock to to Module**********/
IfxGpt12_enableModule(&TMR6);
```

STEP 2:

```
/**********Dividing Clock with 2 (i.e 100MHz/2---->50MHz)**********/
IfxGpt12_setGpt2BlockPrescaler(&TMR6, IfxGpt12_Gpt2BlockPrescaler_2);
```

STEP 3:

```
/**********Set Timer to operate in Timer Mode**********/
IfxGpt12_T6_setMode(&TMR6, IfxGpt12_Mode_timer);
```

STEP 4:

```
/**********Set Timer counter Direction to UP**********/
IfxGpt12_T6_setTimerDirection(&TMR6, IfxGpt12_TimerDirection_up);
```

STEP 5:

```
/**********Get the Interrupt src for Timer6 Module**********/
volatile Ifx_SRC_SRCR *src = IfxGpt12_T6_getSrc(&MODULE_GPT120);


/**********Assign Interrupt Request from Source to CPU to handle it**********/
IfxSrc_init(src, IfxSrc_Tos_cpu0, PRIORITY);

/**********Enable the Interrupt**********/
IfxSrc_enable(src);
```

STEP 6:

```
/**********Start the Timer**********/
IfxGpt12_T6_run(&MODULE_GPT120, IfxGpt12_TimerRun_start);


/********** Initialize the Led to toggle when we get the interrupt.**********/
LED_Init(LED_PORT2, LED8, IfxPort_OutputMode_pushPull,IfxPort_OutputIdx_general);
```

Interrupt Part :

Calculation part :

We've configured timer to run at 50Mhz , max delay that can be generated is 50Mhz*timer Resolution(2^16= 65535) = 20microsec

For every 20usec ISR is executed

```
//Declaring ISR
IFX_INTERRUPT(tmr6_isr, 0, 1);

void tmr6_isr(void)
{
   count++;

   if(count >= 769)          //20Microsec*769 = 1sec for evey one sec LED will toggle
   {
      IfxPort_togglePin(LED_PORT1, LED1);
      count = 0;
   }

}
```

Analog to Digital Converter

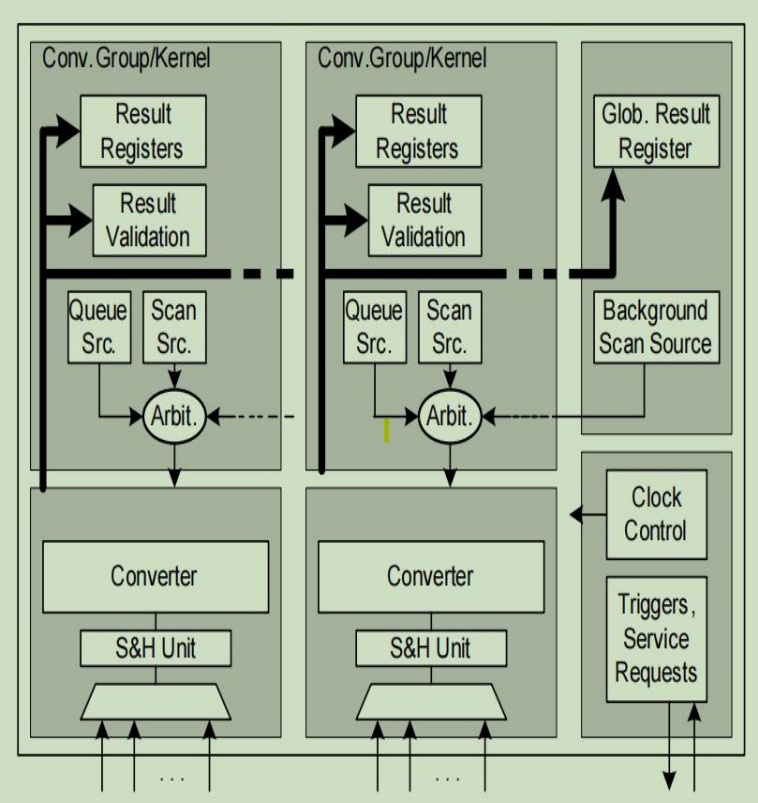
Introduction :

1. ADC(SAR Type)---->12 bit ADC, 12+12 Channels with 2 converter.
2. Nominal analog supply voltage 5.0 V, operation at 3.3 V (Can be Selected via SW)
3. Conversion time below 1 µs
4. Select-able result width of 8/10/12 bits

CMS	[10:8]	rw	**Conversion Mode for Standard Conversions** 000_B 12-bit conversion 001_B 10-bit conversion 010_B 8-bit conversion 011_B Reserved 100_B Reserved 101_B 10-bit fast compare mode 110_B Reserved 111_B Reserved
0	[15:11]	r	**Reserved, write 0, read as 0**

5. FIR/IIR filter with select-able coefficients

**Note : The basic module clock fADC is connected to the system clock signal fSPB(100MHz).



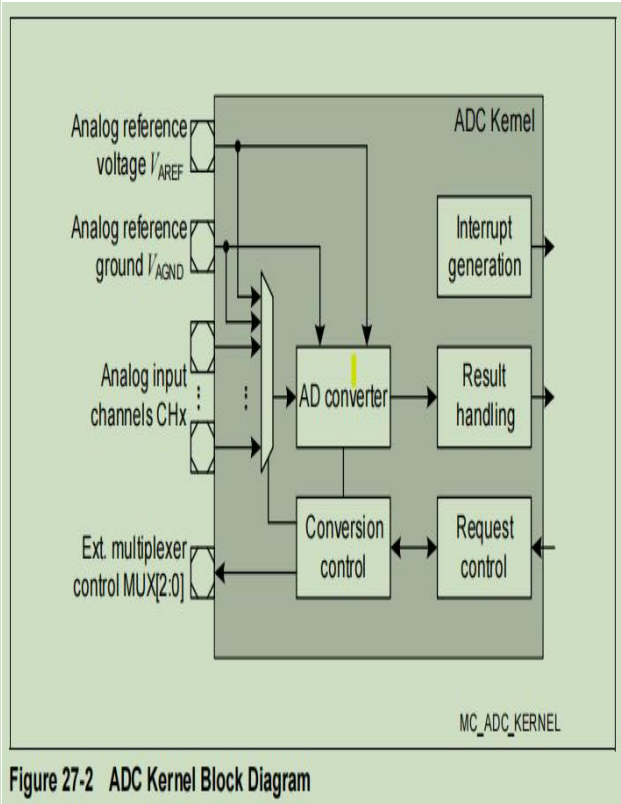


Figure 27-2 ADC Kernel Block Diagram

There are two converters, operates independently, and can be controlled by dedicated set of register and can be triggered by dedicated two group request source(Scan and Queued) and Background source. The request sources can be enabled concurrently with configurable priorities.

Conversion Modes :

Fixed Channel Conversion (single or continuous) : A specific channel source requests conversions of one selectable channel (once or repeatedly)

Auto Scan Conversion (single or continuous) : A channel scan source (request source 1 or 2) requests auto scan conversions of a configurable linear sequence of all available channels (once or repeatedly)

Channel Sequence Conversion (single or continuous) : A queued source (request source 0 or 3) requests a sequence of conversions of up to 8 arbitrarily selectable channels (once or repeatedly)

***Requests with higher priority can either cancel a running lower-priority conversion (cancel-inject-repeat mode) or be converted immediately after the currently running conversion (wait-for-start mode). If the target result register has not been read, a conversion can be deferred (wait-for-read mode).

Input Channel Selection :

The analog input multiplexer selects one of the available analog inputs (CH0 - CHx1)) to be converted. Three sources can select a linear sequence, an arbitrary sequence, or a specific channel. The priorities of these sources can be configured.

Note : CH0-CH12 belongs to Group0 and CH13-CH23 belongs to Group1. For details Ref Table 27-12

Note : The conversion results of each analog input channel can be directed to one of 16 group-specific result registers and one global result register to be stored . A result register Alternatively, an FIR or IIR filter can be enabled that prepossesses the conversion results before sending them to the result register.

Procedure to Initialize and Configure ADC :

1. Enable ADC Module and configure the module.
2. Configure the Group
3. Configure the channels
4. Add channels to Configured Group
5. Start Conversion.

Example code in Scan Mode :

Here single channel is configured and Scanned in scan Mode

/**********Initialisation Steps**********/

Global Variables initialisation :

```
IfxVadc_Adc                 vadc;                /* VADC configuration     */
IfxVadc_Adc_Config          adcConfig;

IfxVadc_Adc_Group           adcGroup;            /* Group configuration    */
IfxVadc_Adc_GroupConfig     adcGroupConfig;

IfxVadc_Adc_ChannelConfig   adcChannelConfig;    /* Channel configuration  */
IfxVadc_Adc_Channel         adcChannel;          /* Channel                */
```

STEP 1:

```
/**************ADC_Module_Configuration*************/
IfxVadc_Adc_initModuleConfig(&adcConfig, &MODULE_VADC);

IfxVadc_Adc_initModule(&vadc, &adcConfig); //Configuring with default settings.
```

STEP 2:

```
/************************ADC_Group_Configuration************************/
       /* Groupid can be Local or Global (  0,1,2,3)
         * Master id should be always equal to Group id
         * Scan requestSlot can be Groupscan, queuedscan and Backgroundscan .
        * TriggerConfig will triiger the converter to start the conversion and this triggering  signal is coming from CCU6 module(By default CCU6 is the triggering source).This has to       be enabled in order to start converstion.
         */

IfxVadc_Adc_initGroupConfig(&adcGroupConfig ,&vadc);//Get default Values
adcGroupConfig.groupId                                  = 0;
adcGroupConfig.master                                   = 0;
adcGroupConfig.scanRequest.autoscanEnabled              = TRUE;
adcGroupConfig.arbiter.requestSlotScanEnabled           = TRUE;
adcGroupConfig.scanRequest.triggerConfig.gatingMode     = IfxVadc_GatingMode_always;
IfxVadc_Adc_initGroup(&adcGroup, &adcGroupConfig);// set the group configuration
```

STEP 3:

```
/************************ADC_Channel_Configuration************************/
       /* Channel Id is channel number that you want to use
    * Chose the Result register where you want to store the converted data
        */

IfxVadc_Adc_initChannelConfig(&adcChannelConfig, &adcGroup);   //get Default Values
adcChannelConfig.channelId          = 5;
adcChannelConfig.resultRegister     = 5;
IfxVadc_Adc_initChannel(&adcChannel, &adcChannelConfig);// Initialise the channel
```

STEP 4:

```
/**************Assign channels to Configured Group*************/
IfxVadc_Adc_setScan(&adcGroup,(1<<adcChannelConfig.channelId),(1<<adcChannelConfig.channelId));
```

STEP 5:

```
/**************Start_Scaning of channels of particular group*************/
IfxVadc_Adc_startScan(&adcGroup);

/************************************End of Initialisation Steps*****************************/
```

User defined function to get converted values :

```
unsigned int get_Converted_Value( void )
{
   Ifx_VADC_RES conversionResult;

   /* Retrieve the conversion value until valid flag of the result register is true */
   do
   {
        conversionResult = IfxVadc_Adc_getResult(&adcChannel);
   }   while (!conversionResult.B.VF);

    return conversionResult.B.RESULT;

}
```

Controller Area Network (CAN)

Introduction :

One CAN ----> 3 Nodes, can run upto 1Mbps

One CANFD---> 5Mbps (ISO 11898-1)

128 Message objects--->shared by all nodes.

Interrupt requests can be routed individually to one of the 16 or 8 interrupt output lines.

Message post-processing notifications can be combined flexibly into a dedicated register field of 256 notification bits .

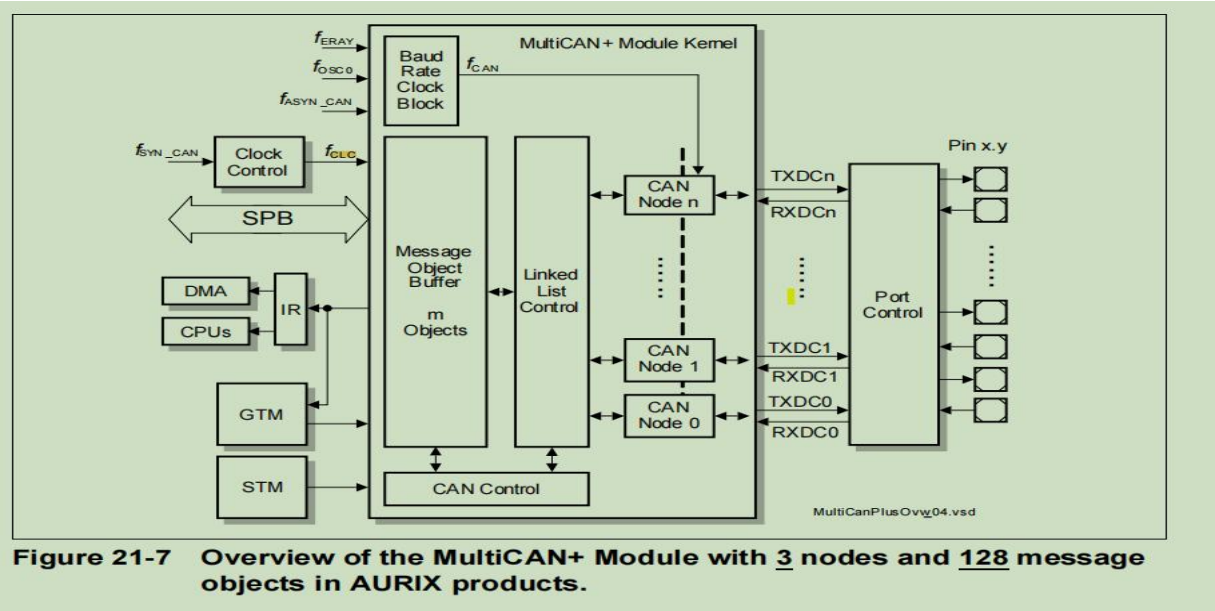


Figure 21-7 Overview of the MultiCAN+ Module with 3 nodes and 128 message objects in AURIX products.

The bit timings for the CAN nodes are derived from the module timer clock (fCAN) and are programmable up to a data rate of 1 Mbit/s in Classical CAN (ISO 11898-1:2003(E) mode or up to 5 MBaud in CAN FD mode.

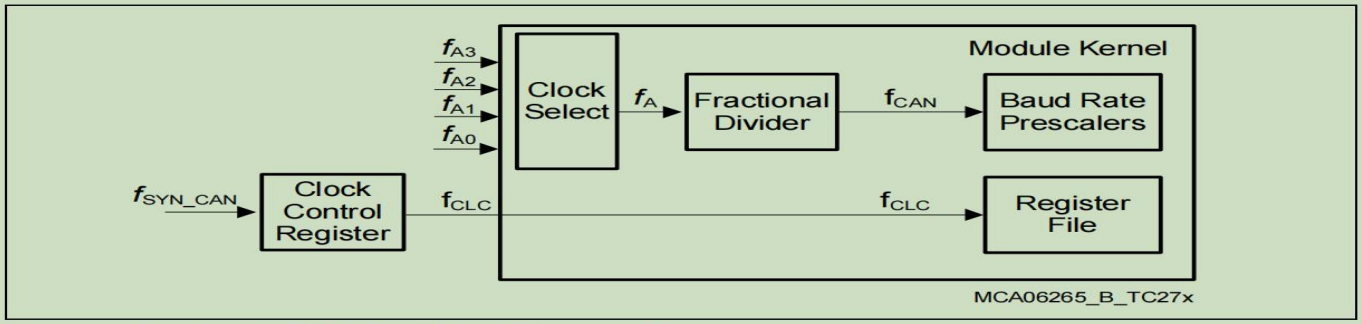


Figure 21-11 MultiCAN+ Clock Generation

The fSYN_CAN is identical to fSPB. fAi is the asynchronous clock input.Fa is responsible for generation of Baudrate.

$$\text{Baudrate} = [(8 * T_{CAN}) + (8 * T_{CLC}) + (4 * \text{No.of active CAN nodes} * T_{CLC})].$$

As an example, when $\mathbf{f_{CLC}}$ = 10MHz, $\mathbf{f_{CAN}}$ = 20MHz, No of active CAN nodes =2,
Baudrate_{max} = [(8x50ns) + (8x100ns) + (4x2x100ns)] = 2000ns = 500 KBaud.

Procedure to Initialize and Configure :

- Initialize the CAN nodes

IfxMultican_Status IfxMultican_Can_initModule(IfxMultican_Can *mcan, const IfxMultican_Can_Config *config);

mcan : CAN Module Base address

config : CAN module configuration structure[Like setting Module CLK, node priority etc].

- Allocate the message objects to the CAN nodes

IfxMultican_Status IfxMultican_Can_Node_init(IfxMultican_Can_Node *node, const IfxMultican_Can_NodeConfig *config)

node : CAN node handle data structure

config : CAN Node configuration[like giving node id, tx/rx pin config, baudrate etc..].

Note : In Aurix Board the NODE0 is connect to P33.8(TX) and P33.7(RX). NODE1 is connected to P14.0(TX) and P14.1(RX).Ref TriBoard Manual page number 40.

- Initialize the message objects

IfxMultican_Status IfxMultican_Can_MsgObj_init(IfxMultican_Can_MsgObj *msgObj, const IfxMultican_Can_MsgObjConfig *config)

msgObj : CAN message object handle data structure

Config : CAN message object configuration[like MSG id, type of frame etc..].

- Transmit Message

void IfxMultican_Message_init(IfxMultican_Message *msg, uint32 id, uint32 dataLow, uint32 dataHigh, IfxMultican_DataLengthCode lengthCode);

IfxMultican_Status IfxMultican_Can_MsgObj_sendMessage(IfxMultican_Can_MsgObj *msgObj, const IfxMultican_Message *msg);

- Receive Message

IfxMultican_Status IfxMultican_Can_MsgObj_readMessage(IfxMultican_Can_MsgObj *msgObj, IfxMultican_Message *msg).

Example : This example demonstrate Loop-back Mode (.i.e Node to Node communication).

Note : Interrupt is Generated on When you receive the data and this is indicated by turning on the Built in LED.

/*********Global structure for various configuration***********/**

```
typedef struct
{
    IfxMultican_Can  can;                       // CAN module handle to HW module SFR set
    IfxMultican_Can_Config canConfig;           // CAN module configuration structure
    IfxMultican_Can_Node canSrcNode;            // CAN source node handle data structure
    IfxMultican_Can_Node canDstNode;            // CAN destination node handle data structure
    IfxMultican_Can_NodeConfig canNodeConfig; // CAN node configuration structure
    IfxMultican_Can_MsgObj canSrcMsgObj;        // CAN source message object handle data
    IfxMultican_Can_MsgObj canDstMsgObj;        // CAN destination message object handle
     IfxMultican_Can_MsgObjConfig canMsgObjConfig;   // CAN message object configuration
    IfxMultican_Message  txMsg;                   // Transmitted CAN message structure
    IfxMultican_Message  rxMsg;                   // Received CAN message structure
} AppMulticanType;
```

AppMulticanType g_multican; //Declare a variable of type AppMulticanType

/*************Declare ISR**************/
// Parameters : ISR_Name, Priority No and Priority of ISR
IFX_INTERRUPT(canIsrRxHandler, 0, 1);

Note : Interrupt is enabled on Rx_Msg_Object(i.e Step 3)
/*****************************ISR*****************************/

```
void canIsrRxHandler(void)
{
    IfxMultican_Status readStatus;

    /* Read the received CAN message and store the status of the operation */
    readStatus = IfxMultican_Can_MsgObj_readMessage(&g_multican.canDstMsgObj,
&g_multican.rxMsg);

    /* If no new data has been received, report an error */
    if( !( readStatus & IfxMultican_Status_newData ) )
    {
        while(1);

    }

    /* If new data has been received but with one message lost, report an error */
    if( readStatus == IfxMultican_Status_newDataButOneLost )
    {
        while(1);

    }
    /* Finally, check if the received data matches with the transmitted one */
    if( readStatus == IfxMultican_Status_newData)
    {
        /* Turn on the RX_INDICATOR to indicate correctness of the received message */
        IfxPort_setPinLow(LED_PORT1, RX_INDICATOR);
        printf("%s\t",&g_multican.rxMsg.data[0]);
        printf("%s\n",&g_multican.rxMsg.data[1]);

    }
}
```

```
void main ( void  )
{
   //Led Initialisation
 IfxPort_setPinModeOutput(LED_PORT1,RX_INDICATOR, IfxPort_OutputMode_pushPull,
IfxPort_OutputIdx_general);
```

STEP 1:

```
/*********************CAN_MODULE_INIT_AND_CONFIG*********************/
IfxMultican_Can_initModuleConfig(&g_multican.canConfig, &MODULE_CAN);

g_multican.canConfig.nodePointer[TX_INTERRUPT_SRC_ID].priority = ISR_PRIORITY_CAN_TX;
g_multican.canConfig.nodePointer[RX_INTERRUPT_SRC_ID].priority = ISR_PRIORITY_CAN_RX;

IfxMultican_Can_initModule(&g_multican.can, &g_multican.canConfig);
```

STEP 12:

```
/*****************SOURCE_NODE(Node0)*********************/
IfxMultican_Can_Node_initConfig(&g_multican.canNodeConfig, &g_multican.can);
g_multican.canNodeConfig.loopBackMode    = TRUE;
g_multican.canNodeConfig.nodeId          = IfxMultican_NodeId_0;
IfxMultican_Can_Node_init(&g_multican.canSrcNode, &g_multican.canNodeConfig);

/*****************DSTINATION_NODE)Node1)*********************/
IfxMultican_Can_Node_initConfig(&g_multican.canNodeConfig, &g_multican.can);
g_multican.canNodeConfig.loopBackMode    = TRUE;
g_multican.canNodeConfig.nodeId          = IfxMultican_NodeId_1;

IfxMultican_Can_Node_init(&g_multican.canDstNode, &g_multican.canNodeConfig);
```

STEP 3:

```
/***************************RX_MSG_OBJ*******************************/
IfxMultican_Can_MsgObj_initConfig(&g_multican.canMsgObjConfig, &g_multican.canDstNode);
g_multican.canMsgObjConfig.msgObjId              = DST_MESSAGE_OBJECT_ID;
g_multican.canMsgObjConfig.messageId             = CAN_MESSAGE_ID;
g_multican.canMsgObjConfig.frame                 = IfxMultican_Frame_receive;
g_multican.canMsgObjConfig.rxInterrupt.enabled   = TRUE;
g_multican.canMsgObjConfig.rxInterrupt.srcId     = RX_INTERRUPT_SRC_ID;
IfxMultican_Can_MsgObj_init(&g_multican.canDstMsgObj, &g_multican.canMsgObjConfig);


/***************************TX_MSG_OBJ*******************************/
IfxMultican_Can_MsgObj_initConfig(&g_multican.canMsgObjConfig, &g_multican.canSrcNode);
g_multican.canMsgObjConfig.msgObjId              = SRC_MESSAGE_OBJECT_ID;
g_multican.canMsgObjConfig.messageId             = CAN_MESSAGE_ID;
g_multican.canMsgObjConfig.frame                 = IfxMultican_Frame_transmit;
g_multican.canMsgObjConfig.txInterrupt.enabled   = TRUE;
g_multican.canMsgObjConfig.txInterrupt.srcId     = TX_INTERRUPT_SRC_ID;
g_multican.canMsgObjConfig.control.messageLen    = IfxMultican_DataLengthCode_5;
```

STEP 4 :

```
IfxMultican_Can_MsgObj_init(&g_multican.canSrcMsgObj, &g_multican.canMsgObjConfig);

IfxMultican_Message_init(&g_multican.txMsg, g_multican.canMsgObjConfig.messageId,
     12345678, 87654321, g_multican.canMsgObjConfig.control.messageLen);

    while( IfxMultican_Status_notSentBusy ==
            IfxMultican_Can_MsgObj_sendMessage(&g_multican.canSrcMsgObj, &g_multican.txMsg) );

 While(1);

}
```