Freescale Semiconductor
Application Note

Document Number: AN3073
Rev. 0, 06/2006

MSC711*n* Debugging Techniques

Using the Event Port

by *Barbara Johnson*
DSP Applications
Freescale Semiconductor, Inc.
Austin, TX

The MSC711*n* system control unit controls the event port. The event port collects important core, device and external events and combines them to trigger a specific action, for example driving signals or generating interrupt requests. A high-level view of the event port in Figure 1 shows the various event inputs and the resulting actions that are used for system debugging.

Multiple input signals are combined in different ways for advanced system-level triggering. The event port combining unit combines the different input sources using OR, AND, XOR, set, set-reset, or toggle operations.

The event port consists of eight multiplexers, each of which can work independently or in a sequence. The sequencing unit allows each multiplexer to receive a trigger from the preceding multiplexer and to pass a trigger to the next multiplexer in the cascade.

When a multiplexer is triggered, it results in an action to drive external pins, drive the timer module, generate interrupt requests, wake up the core from sleep mode, or switch the crossbar to the alternate priority.

This application note provides methods for using the MSC711*n* event port to assist in debugging application

Contents

software. It presents examples of driving an external signal when a DMA event is detected, counting the number of cycles between DMA events, counting the number of core stalls due to instruction cache misses, and counting the number of TDM receive interrupt requests.

A basic knowledge of the event port is assumed. The event port is discussed in detail in the MSC711*n* reference manuals available at the Freescale website listed on the back cover of this application note.

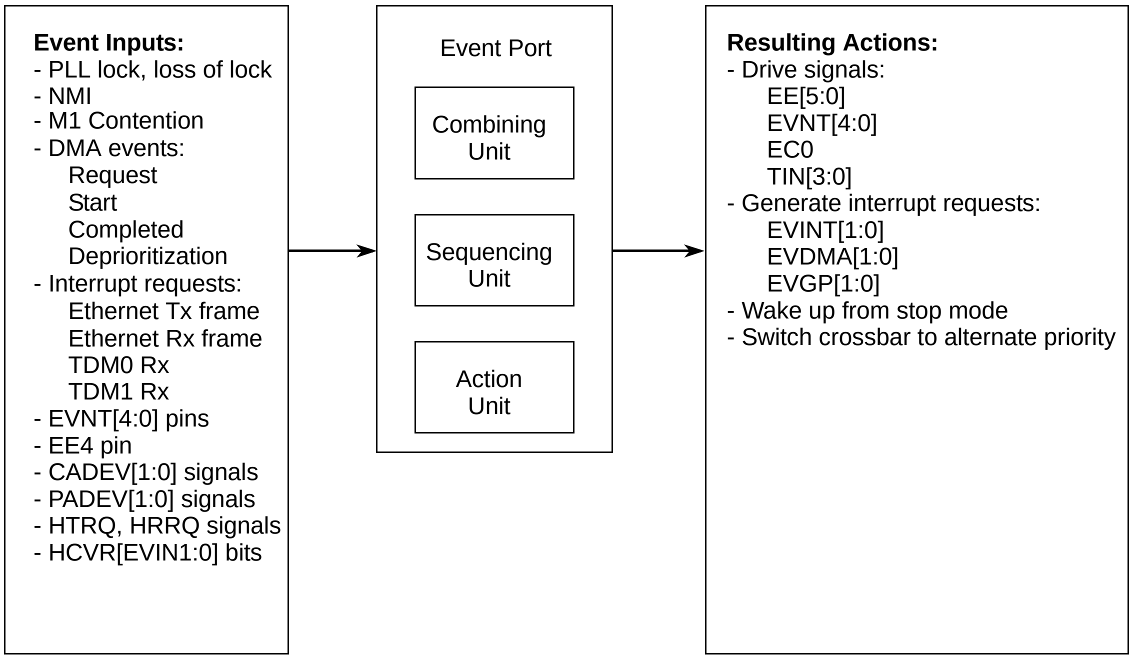

Figure 1. Event Port

1 Monitoring DMA Events

The event port is useful for monitoring DMA activity. An EVNT*n* pin is driven by the occurrence of a DMA event, such as channel request, start, completion, or deprioritization. The event port can also interact with the timer by driving the internal timer input (TIN*n*) signals when specified DMA events occur. When used with the timer, the event port provides a way to count the number of cycles between two DMA events.

1.1 Driving EVNT*x* Pins to Detect DMA Events

The event port multiplexers drive the EVNT*x* pins to indicate the occurrence of a DMA event. A DMA channel request, start, completion or deprioritization event may be selected as input to the event mux. When this event occurs, the selected EVNT*x* pin is asserted. Figure 2 shows an example of using the EVNT0 and EVNT1 pins to indicate the start and completion of the DMA channel 0 transfer. Driving external signals to indicate internal events is useful in debugging because it gives the user some visibility of when and how long system events occur.

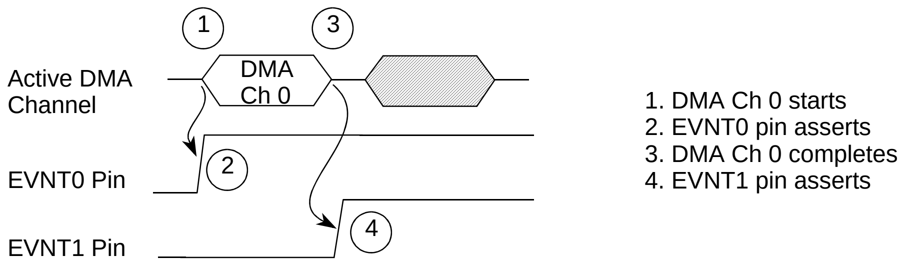

Figure 2. Driving EVNTx Pin to Detect DMA Events

The function `setup_genPulse_DMAEvent()` shown in Example 1 initializes the event port to generate a pulse on the specified EVNT*x* pin when a specific DMA event occurs. The function requires the parameters shown in Table 1 to be passed. The user must pass the DMA channel number, the type of DMA event, the event multiplexer number, and the EVNT*x* pin to drive. The parameters of type enum values are defined in Section 5.1, “File event.h.”

Table 1. `setup_genPulse_DMAEvent()` **Parameters**

Parameter	Valid Parameter Values	Description
int DMAChannel	0–31	DMA channels 0–31
enum EVDMATYPE DMAType	DMA_Request DMA_Start DMA_Completion DMA_Deprior	DMA channel request DMA channel start DMA channel completed DMA channel deprioritization
int EvMux	0–7	Event port multiplexers 0–7
enum EVDRIVEEVNTx EVNTpin	DRIVE_EVNT0 DRIVE_EVNT1 DRIVE_EVNT2 DRIVE_EVNT3 DRIVE_EVNT4	Drive EVNT0 pin Drive EVNT1 pin Drive EVNT2 pin Drive EVNT3 pin Drive EVNT4 pin

Before the EVNT*x* pin can be driven, it must be configured to operate as a peripheral pin instead of a GPIO pin. By default, IO pins are in software control mode out of reset, so these pins function as GPIO. The EVNT*x* pins must be set to hardware control mode in the Port A and Port C control registers (GPACTL and GPCCTL). Next, the event port control register (EVCTL) is cleared because none of the auxiliary inputs, such as PLL lock signal, loss of lock signal, or Ethernet frame interrupts, are required for this function. The event selective invert register (EVSELINV) is also cleared since the input signals will not be inverted. The event multiplexer input and event multiplexer output registers (EVINx and EVOUTx) are first cleared before configuring the source and required action. The EVIN*x* indicates which event multiplexer is used to detect the DMA event. For example, if the event mux 0 is selected to detect the start of DMA channel 0, then EVIN0 is programmed with this event input source. Similarly, if the event mux 1 is selected to detect the completion of DMA channel 0, EVIN1 is programmed with this event input source. The EVIN*x*[DMACH] and EVIN*x*[DMATYP] bits specify the DMA channel number and the type of DMA event to detect. The EVOUT*x*[COMB] bit configures the combining unit for the set operation so that when the enabled input source is asserted, it remains asserted until explicitly cleared. Because the event multiplexer operates independently in this case, the EVOUT*x*[ENABLE] bits are set to always enabled. The EE*x* pins should not be driven when the DMA event is detected. Leaving the EVOUT*x*[DEE] bits to the default setting asserts the EE0 pin upon detection of the DMA event, causing the core to enter debug mode. These bits must be explicitly set so they do not assert the EE*x* pins. The EVOUT*x*[DEVNT] bits are configured to drive the specified EVNT*x* pin when the DMA event occurs. Finally, the EVIN*x*[DMAEN] bit is set to enable detection of the DMA event.

Example 1. `setup_genPulse_DMAEvent()` Function

```
        void setup_genPulse_DMAEvent(                  int  DMAChannel,
                                                       enum EVDMATYPE DMAType,
                                                       int  EvMux,
                                                       enum EVDRIVEEVNTx EVNTpin )
        {
             setEvPort_EVNTpin(EVNTpin);                           //config EVNTx as periph pin
             clrEvPort_EVCTL();                                    //clr EVCTL
             clrEvPort_EVSELINV();                                 //clr EVSELINV
             clrEvPort_EVIN(EvMux);                                //clr EVIN
             clrEvPort_EVOUT(EvMux);                               //clr EVOUT
             setEvPort_DMAInput(EvMux, DMAChannel, DMAType);       //set mux, DMA ch and event
             setEvPort_Combine(EvMux, SET);                        //use set operation
             setEvPort_Enable(EvMux, Always_Enabled);              //mux always enabled
             setEvPort_Drive_EOnCE_EEpin(EvMux, DISABLE_EEx);      //do not drive EE pins
             setEvPort_Drive_EV_EVNTpin(EvMux, EVNTpin);           //drive EVNTx pin
             setEvPort_EnableDMAEvent(EvMux);                      //enable DMA event
        }
```

Example 2 shows how the function is called to drive the EVNT0 and EVNT1 pins to detect the start and completion of the DMA channel 0 transfer. Event multiplexer 0 detects the start of the transfer and drives EVNT0. Event multiplexer 1 detects the completion of the transfer and drives EVNT1. When these EVNT*x* pins are connected to an oscilloscope, the time elapsed between the two DMA events is measured to determine the length of the DMA transfer.

Example 2. Calling the `setup_genPulse_DMAEvent()` Function

```
             //DMA ch 0, DMA start, mux 0, drive EVNT0
             setup_genPulse_DMAEvent(0, DMA_Start,      0, DRIVE_EVNT0);

             //DMA ch 0, DMA complete, mux 1, drive EVNT1
             setup_genPulse_DMAEvent(0, DMA_Completion, 1, DRIVE_EVNT1);

             //*****************************
             //Place code to start DMA here
        //*****************************
```

1.2 Using the Timer to Measure Time Between DMA Events

In Example 2, the length of time between DMA events is measured by monitoring the EVNT0 and EVNT1 pins on an oscilloscope. Alternatively, the event port and the timer can be used to measure the time between DMA events. The timer counter increments for every rising edge of the timer input clock during from the first DMA event to the second DMA event. Figure 3 shows an example of how to use the timer to measure the length of time between DMA channel start and completion events as shown in .

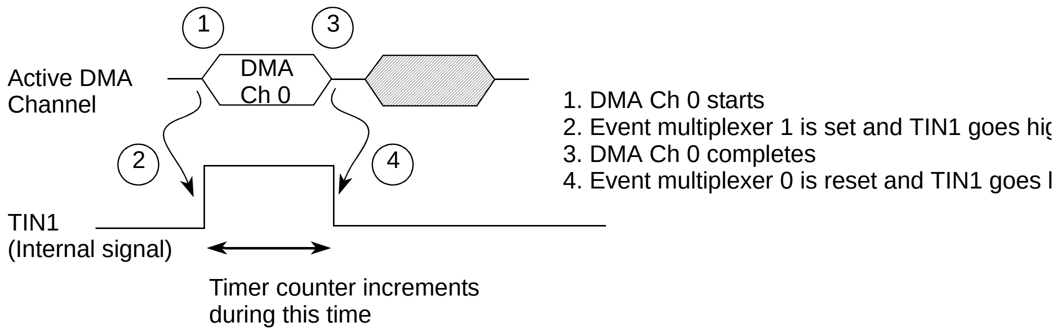
1
3
Active DMA Channel
DMA Ch 0
2
4
TIN1 (Internal signal)
Timer counter increments during this time
1. DMA Ch 0 starts
2. Event multiplexer 1 is set and TIN1 goes hig
3. DMA Ch 0 completes
4. Event multiplexer 0 is reset and TIN1 goes l

Figure 3. Using the Timer to Measure Time Between DMA Events

Example 3 shows the function `setup_countCyc_DMAStart2Done()` to initialize the event port to drive the internal timer signal TIN1 between the DMA start and completion events. Table 2 shows that the function requires the timer unit number and the DMA channel number to be passed. The timer unit number of type enum is defined in Section 5.2, “File timer.h.”

Table 2. `setup_countCyc_DMAStart2Done()` Parameters

Parameter	Valid Values	Description
enum TMRUNIT Unit	A0, A1, A2, A3, B0, B1, B2, B3	Timers A0, A1, A2, A3 Timers B0, B1, B2, B3
int DMAChannel	0–31	DMA channels 0–31

The function `setup_countCyc_DMAStart2Done()` consists of the timer initialization and the event port initialization. To access the timer registers, the clock control register CLKCTL[TMUX] bits must be set to select the APB clock as the source of the timer clock. If these bits are configured to any other value, writing to the timer registers will have no effect.

1. The timer control register TMR*x*CTL[PCS] selects the input clock divided by 1 as the primary clock source. This means that the APB clock divided by 1 is the source of the timer clock. The APB clock operates at half the core frequency. For example, when the core operates at 300 MHz, the timer clock operates at 150 MHz.
2. The TMR*x*CTL[SC] bits select internal timer input signal TIN1 as the secondary input.
3. The TMR*x*CTL[CM] bit is set up to count the rising edges of the primary input while the secondary input is high. The timer counter increments at a rate of 150 MHz while the TINx signal is high. When the TINx signal goes low, then the timer stops incrementing.
4. The TMR*x*CTL[ONCE] bit is cleared to allow the timer to count repeatedly for continuous counting.

5. The TMR*x*CTL[DIR] bit is cleared to select the normal counting up direction.
6. The Timer Status and Control Register TMR*x*SCTL[OEN] bit is set to enable the timer output.

The second part of the `setup_countCyc_DMAStart2Done()` function initializes the event port. In this function, event multiplexer 1 and 0 are configured to detect the DMA start and completion events, respectively. The combining unit uses the set-reset operation so that one set of conditions (DMA start event) performs a set operation and a second set of conditions (DMA completion event) performs a reset of the triggering signal (TIN*x*). The set-reset operation requires two adjacent event muxes. Event multiplexer *i+1* detects the set operation and event multiplexer *i* detects the reset operation. This example uses event multiplexer 1 and event multiplexer 0 for the set-reset operation. The EVOUT1[COMB] bits are configured for set operation and the EVOUT0[COMB] bits are configured for set-reset operation. All other event port register settings are the same as in previous examples except that the TIN1 is driven instead of an EVNT pin by configuring the EVOUT1[DTIN] and EVOUT0[DTIN] bits to drive the TIN1 timer input.

Example 3. `setup_countCyc_DMAStart2Done` Function

```
void setup_countCyc_DMAStart2Done(enum TMRUNIT Unit,int DMAChannel)
{
   //Timer Operation
   setTmr_InputClk();                              //set APB clk source of timer clk
   setTmr_PrimaryClk(Unit, InpClkDiv1);            //primary clk = timer input clk/1
   setTmr_CountMode(Unit, RisPrimHighSec);         //count rising prim while sec high
   setTmr_CntrReg(Unit, 0);                        //clr counter reg
   setTmr_CountOnce(Unit, Repeat);                 //repeat count
   setTmr_CountDir(Unit, Up);                      //count up

   clrEvPort_EVCTL();                              //clr EVCTL
   clrEvPort_EVSELINV();                           //clr EVSELINV

   //Mux 1 Set Operation
   clrEvPort_EVIN(1);                              //clr EVIN1
   clrEvPort_EVOUT(1);                             //clr EVOUT1
   setEvPort_DMAInput(1, DMAChannel, DMA_Start);//DMA ch , start on mux 1
   setEvPort_Combine(1, SET);                      //use set operation
   setEvPort_Enable(1, Always_Enabled);            //mux 1 always enabled
   setEvPort_Drive_EOnCE_EEpin(1, DISABLE_EEx);   //do not drive EEx pins
   setEvPort_Drive_Timer_TINpin(1, DRIVE_TIN1);   //drive timer input TIN1
   setEvPort_EnableDMAEvent(1);                    //enable DMA event
```

```
        //Mux 0 Reset Operation
        clrEvPort_EVIN(0);                              //clr EVIN0
        clrEvPort_EVOUT(0);                             //clr EVOUT0
        setEvPort_DMAInput(0, DMAChannel, DMA_Completion);//DMA ch, complete on mux 0
        setEvPort_Combine(0, SETRESET);                 //use set reset operation
        setEvPort_Enable(0, Always_Enabled);            //mux 0 always enabled
        setEvPort_Drive_EOnCE_EEpin(0, DISABLE_EEx);   //do not drive EEx pins
        setEvPort_Drive_Timer_TINpin(0, DRIVE_TIN1);   //drive timer input TIN1
        setEvPort_EnableDMAEvent(0);                    //enable DMA event
    }
```

Example 4 shows how the function is called to use the Timer A1 counter to count the number of cycles between the start and completion events of the DMA channel 0 transfer, as shown in Figure 3. When the DMA transfer is complete, the final value of the timer counter can be read by calling the `getTmr_Counter()` function.

Example 4. Calling the `setup_countCyc_DMAStart2Done()` Function

```
        //count cycles from DMA ch 0 start to complete
        setup_countCyc_DMAStart2Done(A1, 0);

        //*****************************
        //Place code to start DMA here
        //*****************************

        //get final counter value
        counter = getTmr_Counter(A1);
```

2 Counting Stall Cycles from I-Cache Misses

When a cache miss occurs, the I-Cache fetches instructions from the M2 or external DDR memory to the cache array. The SC1400 core stalls for the number of clock cycles required to fetch instruction into the cache array. The cache miss penalty degrades performance.

The event port is useful for analyzing the performance degradation that results from instruction cache misses. It supports I-Cache misses as an input to the event port muxes. When an I-Cache miss occurs, this information is sent to the event port and this signal asserts for one AHB clock for every two core clock cycles that the core is stalled by the I-Cache miss. For example, if the core stalls for eight core clock cycles, the I-Cache miss signal asserts for four AHB clocks. For a stall of seven core clock cycles, the I-Cache miss signal asserts for three or four AHB clock cycles, depending on the previous state. This allows the timers connected to the event port to count the number of stall cycles due to cache misses more accurately.

When used with the timer, the event port can count the number of stall cycles attributed to all I-Cache misses or the I-Cache misses to external memory only. Software reads the final timer counter value to determine the number of stall cycles. Table 3 shows that the `setup_countCyc_ICacheMiss()` and `setup_countCyc_ICacheMissToDDR()` functions require only the timer unit number as the parameter to be passed.

Table 3. `setup_countCyc_ICacheMiss()`and `setup_countCyc_ICacheMissToDDR()`Parameters

Parameter	Valid Values	Description
enum TMRUNIT Unit	A0, A1, A2, A3, B0, B1, B2, B3	Timers A0, A1, A2, A3 Timers B0, B1, B2, B3

Example 5 shows the steps in setting up the timer to count stalls due to I-Cache misses. Most of the settings are the same as in the previous example that uses the timer. Only the timer count mode programmed in the TMR*x*CTL[PCS] is different. In this case, the count mode is set to count the rising edges of the primary source, which is the APB clock. All other timer settings remain the same.

This function uses event multiplexer 1, but other multiplexers can be used instead. The event multiplexer input source is selected by setting the EVOUT*x*[IMISS] or EVOUT*x*[IMISSE] bit, depending on whether all cases of I-Cache misses or just the I-Cache misses to external memory are detected. In both cases, the timer input signal TIN1 is driven by the event multiplexer when an I-Cache miss event occurs, which then increments the timer counter. Other timer input signals can also be used.

Example 5. `setup_countCyc_ICacheMiss()` and `setup_countCyc_ICacheMissToDDR()` Functions

```
void setup_countCyc_ICacheMiss(enum TMRUNIT Unit)
{
   setTmr_InputClk();                         //set APB clk source of timer clk
   setTmr_PrimaryClk(Unit, InpCtr1);          //primary clk = timer input clk/1
   setTmr_CountMode(Unit, RisPrim);           //count rising prim high
   setTmr_CntrReg(Unit, 0);                   //clr counter reg
   setTmr_CountLen(Unit, Rollover);           //count rollover
   setTmr_Cmp1Reg(Unit, 0);                   //clr compare reg

   clrEvPort_EVIN(1);                         //clr EVIN1
   clrEvPort_EVOUT(1);                        //clr EVOUT1
   clrEvPort_EVCTL();                         //clr EVCTL
   clrEvPort_EVSELINV();                      //clr EVSELINV

   setEvPort_Drive_Timer_TINpin(1, DRIVE_TIN1);//drive timer input TIN1
   setEvPort_EventInput(1, IC_Miss_All);      //set all ICache misses mux 1 input
   setEvPort_Combine(1, OR);                  //use or operation
   setEvPort_Enable(1, Always_Enabled);       //mux 1 always enabled
```

```
    setEvPort_Drive_EOnCE_EEpin(1, DISABLE_EEx);//do not drive EEx pins
}

    // *****************************************************************************
void setup_countCyc_ICacheMissToDDR(enum TMRUNIT Unit)
{
    setTmr_InputClk();                           //set APB clk source of timer clk
    setTmr_PrimaryClk(Unit, InpCtr1);            //primary clk = timer input clk/1
    setTmr_CountMode(Unit, RisPrim);             //count rising prim high
    setTmr_CntrReg(Unit, 0);                     //clr counter reg
    setTmr_CountLen(Unit, Rollover);             //count rollover
    setTmr_Cmp1Reg(Unit, 0);                     //clr compare reg

    clrEvPort_EVIN(1);                           //clr EVIN1
    clrEvPort_EVOUT(1);                          //clr EVOUT1
    clrEvPort_EVCTL();                           //clr EVCTL
    clrEvPort_EVSELINV();                        //clr EVSELINV

    setEvPort_Drive_Timer_TINpin(1, DRIVE_TIN1);//drive timer input TIN1
    setEvPort_EventInput(1, IC_Miss_DDR);        //set ICache misses DDR mux 1 input
    setEvPort_Combine(1, OR);                    //use or operation
    setEvPort_Enable(1, Always_Enabled);         //mux 1 always enabled
    setEvPort_Drive_EOnCE_EEpin(1, DISABLE_EEx);//do not drive EEx pins
}
```

Example 6 shows how the function is called. In this example, timer unit A0 is used to count stalls caused by I-Cache misses to both M2 and external memory. After the code being profiled completes, the final value of the timer counter can be read to determine the number of core stalls due to I-Cache misses.

Example 6. Calling the `setup_countCyc_ICacheMiss()` Function

```
//count stall cycles due to icache misses using timer A0
setup_countCyc_ICacheMiss(A0);

//******************************
//Place code to profile here
//******************************
```

```
        //get final counter value
        icachemiss = getTmr_Counter(A0);
```

3 Counting TDM Receive Interrupts

Different events can trigger a TDM receive interrupt. A receive interrupt is generated when new data is loaded into the receive data register or the receive FIFO. It can also be generated when the receive FIFO reaches the programmed FIFO watermark. Software tracking of the number of TDM receive interrupts requires processing overhead. Instead, this task can be handled by the event port and the timer.

The function `setup_countCyc_TDM0RxIntReq()` and the `setup_countCyc_TDM1RxIntReq()` set up the event port and the timer to count the number of TDM0 and TDM1 receive interrupts. Table 4 shows that only the timer unit parameter is required when these functions are called.

Table 4. `setup_countCyc_TDM0RxIntReq()`and `setup_countCyc_TDM1RxIntReq()`Parameters

Parameter	Valid Values	Description
enum TMRUNIT Unit	A0, A1, A2, A3, B0, B1, B2, B3	Timers A0, A1, A2, A3 Timers B0, B1, B2, B3

Example 7 shows the function to count the TDM receive interrupt requests. The input event is the only setting that differs from the Example 6. The EVOUT*x*[TDM0] or EVOUT*x*[TDM1] bit is set to enable detection of a TDM0 or TDM1 receive interrupt request.

Example 7. `setup_countCyc_TDM0RxIntReq()` and `setup_countCyc_TDM1RxIntReq()` Functions

```
        void setup_countCyc_TDM0RxIntReq(enum TMRUNIT Unit)
        {
             setTmr_InputClk();                          //set APB clk source of timer clk
             setTmr_PrimaryClk(Unit, InpCtr1);           //primary clk = timer input clk/1
             setTmr_CountMode(Unit, RisPrim);            //count rising prim high
             setTmr_CntrReg(Unit, 0);                    //clr counter reg
             setTmr_CountLen(Unit, Rollover);            //count rollover
             setTmr_Cmp1Reg(Unit, 0);                    //clr compare reg

             clrEvPort_EVIN(1);                          //clr EVIN1
             clrEvPort_EVOUT(1);                         //clr EVOUT1
             clrEvPort_EVCTL();                          //clr EVCTL
             clrEvPort_EVSELINV();                       //clr EVSELINV

             setEvPort_Drive_Timer_TINpin(1, DRIVE_TIN1);//drive timer input TIN1
             setEvPort_EventInput(1, TDM0_Rx_IntReq);    //set TDM0 rx interr mux 1 input
             setEvPort_Combine(1, OR);                   //use or operation
             setEvPort_Enable(1, Always_Enabled);        //mux 1 always enabled
             setEvPort_Drive_EOnCE_EEpin(1, DISABLE_EEx);//do not drive EEx pins
        }

        // ******************************************************************************
        void setup_countCyc_TDM1RxIntReq(enum TMRUNIT Unit)
        {
             setTmr_InputClk();                          //set APB clk source of timer clk
             setTmr_PrimaryClk(Unit, InpCtr1);           //primary clk = timer input clk/1
             setTmr_CountMode(Unit, RisPrim);            //count rising prim high
             setTmr_CntrReg(Unit, 0);                     //clr counter reg
             setTmr_CountLen(Unit, Rollover);             //count rollover
             setTmr_Cmp1Reg(Unit, 0);                     //clr compare reg

             clrEvPort_EVIN(1);                           //clr EVIN1
             clrEvPort_EVOUT(1);                          //clr EVOUT1
             clrEvPort_EVCTL();                           //clr EVCTL
             clrEvPort_EVSELINV();                        //clr EVSELINV

             setEvPort_Drive_Timer_TINpin(1, DRIVE_TIN1);//drive timer input TIN1
             setEvPort_EventInput(1, TDM1_Rx_IntReq);     //set TDM1 rx interr mux 1 input
             setEvPort_Combine(1, OR);                    //use or operation
             setEvPort_Enable(1, Always_Enabled);         //mux 1 always enabled
             setEvPort_Drive_EOnCE_EEpin(1, DISABLE_EEx);//do not drive EEx pins
        }
```

Example 8 shows how the function is called. Timer unit A0 is used to count TDM receive interrupt requests. After the TDM code executes, the final value of the timer counter can be read to determine the number of receive interrupts.

Example 8. Calling the `setup_countCyc_TDM0RxIntReq()` Function

```
    //count TDM0 rx interrupts using timer A1

    setup_countCyc_TDM0RxIntReq(A1);


    //*****************************

    //Place code to start TDM here

    //*****************************


    //get final counter value

    counter = getTmr_Counter(A1);
```

4 Summary

Table 5 summarizes the event port and timer settings required to perform the various functions discussed in this application note.

Table 5. Summary of Event Port and Timer Settings

Function	Event Port Settings	Timer Settings
Drive EVNTx pins to detect DMA event	EVIN[DMACH] = Ch 0–31	
	EVIN[DMATYP] = Request, start, completed, or deprioritization	
	EVIN[DMAEN] = Enabled	
	EVOUT[ENABLE] = Always enabled	
	EVOUT[COMB] = Set	
	EVOUT[DEVNT] = EVNT[4:0]	
	EVOUT[DEE] = Disabled	
Use timer to count cycles between DMA start and complete	EVIN$_i$[DMACH] = Ch 0–31 EVIN$_{i+1}$[DMACH] = Ch 0–31	CLKCTL[TMUX] = APB clock
	EVIN$_i$[DMATYP] = Completed EVIN$_{i+1}$[DMATYP] = Start	TMRxCTL[CM] = Count rising edges of primary source while secondary input is high
	EVIN$_i$[DMAEN] = Enabled EVIN$_{i+1}$[DMAEN] = Enabled	TMRxCTL[PCS] = Input clock/1
	EVOUT$_i$[ENABLE] = Always enabled EVOUT$_{i+1}$[ENABLE] = Always enabled	TMRxCTL[SC] = TIN[3:0]
	EVOUT$_i$[COMB] = Set-reset EVOUT$_{i+1}$[COMB] = Set	TMRxCTL[ONCE] = Count repeatedly
	EVOUT$_i$[DTIN] = TIN[3:0] EVOUT$_{i+1}$[DTIN] = TIN[3:0]	TMRxCTL[LEN] = Roll over
	EVOUT$_i$[DEE] = Disabled EVOUT$_{i+1}$[DEE] = Disabled	TMRxCTL[DIR] = Up
Use timer to count I-Cache misses	EVOUT[ENABLE] = Always enabled	CLKCTL[TMUX] = APB clock
	EVOUT[COMB] = Or	TMRxCTL[CM] = Count rising edges of primary source
	EVOUT[IMISS or IMISSE] = Enabled (I-Cache miss all or DDR only)	TMRxCTL[PCS] = Input clock/1
		TMRxCTL[SC] = TIN[3:0]
	EVOUT[DTIN] = TIN[3:0]	TMRxCTL[ONCE] = Count repeatedly
	EVOUT[DEE] = Disabled	TMRxCTL[LEN] = Roll over

Table 5. Summary of Event Port and Timer Settings

Function	Event Port Settings	Timer Settings
Use timer to count TDM receive interrupts	EVOUT[ENABLE] = Always enabled	CLKCTL[TMUX] = APB clock
	EVOUT[COMB] = Or	TMRxCTL[CM] = Count rising edges of primary source
	EVOUT[TDM0 or TDM1] = Enabled	TMRxCTL[PCS] = Input clock/1
		TMRxCTL[SC] = TIN[3:0]
	EVOUT[DTIN] = TIN[3:0]	TMRxCTL[ONCE] = Count repeatedly
	EVOUT[DEE] = Disabled	TMRxCTL[LEN] = Roll over

5 Code Listing

5.1 File event.h

```
enum EVEVENTINPUT
{
    DMA_Priority_Elev= 0x00400000,
    IC_Miss_All      = 0x00200000,
    IC_Miss_DDR      = 0x00100000,
    TDM0_Rx_IntReq   = 0x00080000,
    TDM1_Rx_IntReq   = 0x00040000,
    NMI_IntReq       = 0x00020000,
    M1_Contention    = 0x00010000,
};

enum EVDMATYPE
{
    DMA_Request   = 0x00000000,
    DMA_Start     = 0x00800000,
    DMA_Completion= 0x01000000,
    DMA_Deprior   = 0x01800000,
};

enum EVMUXENABLE
{
    EE0_pin          = 0x00000000,
    EE1_pin          = 0x08000000,
    EE2_pin          = 0x10000000,
    EE3_pin          = 0x18000000,
    EE4_pin          = 0x20000000,
    EE5_pin          = 0x28000000,
    EED_and_EE0_pins = 0x40000000,
    EED_and_EE1_pins = 0x48000000,
    EED_and_EE2_pins = 0x50000000,
    EED_and_EE3_pins = 0x58000000,
    EED_and_EE4_pins = 0x60000000,
    EED_and_EE5_pins = 0x68000000,
    Mux_i_plus_1     = 0x70000000,
    Always_Enabled   = 0x78000000,
};

enum EVDRIVETINx
{
    DISABLE_TINx  = 0x00000000,
    DRIVE_TIN0    = 0x00004000,
    DRIVE_TIN1    = 0x00005000,
    DRIVE_TIN2    = 0x00006000,
    DRIVE_TIN3    = 0x00007000,
};

enum EVDRIVEEVx
{
    DRIVE_EE0     = 0x00000000,
    DRIVE_EE1     = 0x00000008,
    DRIVE_EE2     = 0x00000010,
    DRIVE_EE3     = 0x00000018,
    DRIVE_EE4     = 0x00000020,
    DRIVE_EE5     = 0x00000028,
    DISABLE_EEx   = 0x00000038,
};

enum EVDRIVEEEVNTx
{
    DRIVE_EVNT0     = 0x00000000,
    DRIVE_EVNT1     = 0x00000200,
    DRIVE_EVNT2     = 0x00000400,
    DRIVE_EVNT3     = 0x00000600,
    DRIVE_EVNT4     = 0x00000800,
};

enum EVCOMBINE
{
    OR            = 0x00000000,
    AND           = 0x00800000,
    SET           = 0x01000000,
    TOGGLE        = 0x01800000,
    XOR           = 0x02000000,
    SETRESET      = 0x03800000,
};
```

5.2 File timer.h

```
enum TMRUNIT
{
        A0 = 0,
        A1 = 2,
        A2 = 4,
        A3 = 6,
        B0 = 1,
        B1 = 3,
        B2 = 5,
        B3 = 6,
};

enum PRIMARYCLK
{
        InpCtr0       = 0x0000,
        InpCtr1       = 0x0200,
        InpCtr2       = 0x0400,
        InpCtr3       = 0x0600,
        OutCtr0       = 0x0800,
        OutCtr1       = 0x0A00,
        OutCtr2       = 0x0C00,
        OutCtr3       = 0x0E00,
        InpClkDiv1    = 0x1000,
        InpClkDiv2    = 0x1200,
        InpClkDiv4    = 0x1400,
        InpClkDiv8    = 0x1600,
        InpClkDiv16   = 0x1800,
        InpClkDiv32   = 0x1A00,
        InpClkDiv64   = 0x1C00,
        InpClkDiv128  = 0x1E00,
};

enum SECONDARYCLK
{
        TIN0           = 0x0000,
        TIN1           = 0x0080,
        TIN2           = 0x0100,
        TIN3           = 0x0180,
};

enum COUNTMODE
{
        Disabled      = 0x0000,
        RisPrim       = 0x2000,
        RisFallPrim   = 0x4000,
        RisPrimHighSec  = 0x6000,
        Quadrature      = 0x8000,
        RisPrimDirSec   = 0xA000,
        InpSecTrigPrim  = 0xC000,
        Cascaded        = 0xE000,
};

enum COUNTONCE
{
        Repeat        = 0x0000,
        CompareStop   = 0x0040,
};

enum COUNTLEN
{
        Rollover      = 0x0000,
        CompareReinst = 0x0020,
};

enum COUNTDIR
{
        Up            = 0x0000,
        Down          = 0x0001
};

enum OFLAGMODE
{
        Assert                = 0x0000,
        ClrOFLAG              = 0x0001,
        SetOFLAG              = 0x0002,
        TglOFLAGCmp           = 0x0003,
        TglOFLAGAlt           = 0x0004,
        SetCmpClrSec          = 0x0005,
        SetCmpClrRollover     = 0x0006,
        EnGatedClk            = 0x0007
};
```

```
enum TMRINTERR
{
        CmpFlagEn            = 0x4000,
        OverflowFlagEn       = 0x1000,
        InpEdgeFlagEn        = 0x0400,
        CaptureDis           = 0x0000,
        LoadCapRisEdgeSec    = 0x0040,
        LoadCapFallEdgeSec   = 0x0080,
        LoadCapAnyEdgeSec    = 0x00C0,
        Mstr                 = 0x0020,
        ExtOFLAGEn           = 0x0010,
        ForcedOflagVal       = 0x0008,
        ForceOflagOut        = 0x0004,
        OutPlInv             = 0x0002,
        OutEn                = 0x0001
};
```

5.3 File event.c

```
// ****************************************************************************
void setEvPort_GPIOEVNTPin(int EVNTpin)
{
        GPIO   *pstGPIO   = (GPIO  *) (GPIO_BASE);

        switch(EVNTpin)
        {
                case 0:
                        //EVNT0 GPCCTL[13] = 1
                        pstGPIO->astPort[2].vuliPortControl |= 0x00002000;
                        break;
                case 1:
                        //EVNT1 GPACTL[17] = 1
                        pstGPIO->astPort[0].vuliPortControl |= 0x00020000;
                        break;
                case 2:
                        //EVNT2 GPCCTL[14] = 1
                        pstGPIO->astPort[2].vuliPortControl |= 0x00004000;
                        break;
                case 3:
                        //EVNT3 GPCCTL[15] = 1
                        pstGPIO->astPort[2].vuliPortControl |= 0x00008000;
                        break;
                case 4:
                        //EVNT4 GPACTL[16] = 1
                        pstGPIO->astPort[0].vuliPortControl |= 0x00010000;
                        break;
        }
}

// ****************************************************************************
void setEvPort_EVNTpin(enum EVDRIVEEVNTx EVNTpin)
{
        switch(EVNTpin)
        {
                case DRIVE_EVNT0:
                        setEvPort_GPIOEVNTPin(0);
                        break;

                case DRIVE_EVNT1:
                        setEvPort_GPIOEVNTPin(1);
                        break;

                case DRIVE_EVNT2:
                        setEvPort_GPIOEVNTPin(2);
                        break;

                case DRIVE_EVNT3:
                        setEvPort_GPIOEVNTPin(3);
                        break;

                case DRIVE_EVNT4:
                        setEvPort_GPIOEVNTPin(4);
                        break;
        }
}

// ******************************************************************************
void clrEvPort_EVCTL(void)
{
        pstEV->vuliEVCTRL = 0;
}

// ******************************************************************************
void clrEvPort_EVSELINV(void)
{
        pstEV->vuliEVSELINV = 0;
}

// ******************************************************************************
void clrEvPort_EVIN(int EvMux)
{
        pstEV->astEVIN[EvMux].vuliIN = 0;
}

// ******************************************************************************
void clrEvPort_EVOUT(int EvMux)
{
        pstEV->astEVOUT[EvMux].vuliOUT = 0;
}

// ******************************************************************************
void setEvPort_DMAInput(int EvMux, int DMAChannel, enum EVDMATYPE DMAType)
{
        pstEV->astEVIN[EvMux].vuliIN |= (DMAChannel << 26) | DMAType;
}

// ******************************************************************************
void setEvPort_Combine(int EvMux, enum EVCOMBINE Combine)
```

```
{
        pstEV->astEVOUT[EvMux].vuliOUT |= Combine;
}

// ****************************************************************************
void setEvPort_Enable(int EvMux, enum EVMUXENABLE Enable)
{
        pstEV->astEVOUT[EvMux].vuliOUT |= Enable;
}

// ****************************************************************************
void setEvPort_Drive_EOnCE_EEpin(int EvMux, enum EVDRIVEEEVx EExPin)
{
        pstEV->astEVOUT[EvMux].vuliOUT |= EExPin;
}

// ****************************************************************************
void setEvPort_Drive_EV_EVNTpin(int EvMux, enum EVDRIVEEVNTx EVNTxPin)
{
        pstEV->astEVOUT[EvMux].vuliOUT |= EVNTxPin;
}

// ****************************************************************************
void setEvPort_EnableDMAEvent(int EvMux)
{
        pstEV->astEVIN[EvMux].vuliIN |= 0x00200000;
}

// ****************************************************************************
void setup_genPulse_DMAEvent(int  DMAChannel,
                             enum EVDMATYPE DMAType,
                             int  EvMux,
                             enum EVDRIVEEVNTx EVNTpin)
{
        setEvPort_EVNTpin(EVNTpin);
        clrEvPort_EVCTL();
        clrEvPort_EVSELINV();
        clrEvPort_EVIN(EvMux);
        clrEvPort_EVOUT(EvMux);
        setEvPort_DMAInput(EvMux, DMAChannel, DMAType);
        setEvPort_Combine(EvMux, SET);
        setEvPort_Enable(EvMux, Always_Enabled);
        setEvPort_Drive_EOnCE_EEpin(EvMux, DISABLE_EEx);
        setEvPort_Drive_EV_EVNTpin(EvMux, EVNTpin);
        setEvPort_EnableDMAEvent(EvMux);
}

// ****************************************************************************
void setup_countCyc_DMAStart2Done(enum TMRUNIT Unit, int DMAChannel)
{
        setTmr_InputClk();
        setTmr_PrimaryClk(Unit, InpClkDiv1);
        setTmr_SecondaryClk(Unit,TIN1 );
        setTmr_CountMode(Unit, RisPrimHighSec);
        setTmr_CntrReg(Unit, 0);
        setTmr_CountLen(Unit, Rollover);
        setTmr_Cmp1Reg(Unit, 0);

        clrEvPort_EVCTL();
        clrEvPort_EVSELINV();

        //Mux 1 Set Operation
        clrEvPort_EVIN(1);
        clrEvPort_EVOUT(1);
        setEvPort_DMAInput(1, DMAChannel, DMA_Start);
        setEvPort_Combine(1, SET);
        setEvPort_Enable(1, Always_Enabled);
        setEvPort_Drive_EOnCE_EEpin(1, DISABLE_EEx);
        setEvPort_Drive_Timer_TINpin(1, DRIVE_TIN1);
        setEvPort_EnableDMAEvent(1);

        //Mux 0 Reset Operation
        clrEvPort_EVIN(0);
        clrEvPort_EVOUT(0);
        setEvPort_DMAInput(0, DMAChannel, DMA_Completion);
        setEvPort_Combine(0, SETRESET);
        setEvPort_Enable(0, Always_Enabled);
        setEvPort_Drive_EOnCE_EEpin(0, DISABLE_EEx);
        setEvPort_Drive_Timer_TINpin(0, DRIVE_TIN1);
        setEvPort_EnableDMAEvent(0);
}

// ******************************************************************************
void setup_countCyc_ICacheMiss(enum TMRUNIT Unit)
{
        setTmr_InputClk();
        setTmr_PrimaryClk(Unit, InpCtr1);
        setTmr_CountMode(Unit, RisPrim);
        setTmr_CntrReg(Unit, 0);
        setTmr_CountLen(Unit, Rollover);
        setTmr_Cmp1Reg(Unit, 0);

        clrEvPort_EVIN(1);
        clrEvPort_EVOUT(1);
        clrEvPort_EVCTL();
        clrEvPort_EVSELINV();

        setEvPort_Drive_Timer_TINpin(1, DRIVE_TIN1);
        setEvPort_EventInput(1, IC_Miss_All);
        setEvPort_Combine(1, OR);
        setEvPort_Enable(1, Always_Enabled);
        setEvPort_Drive_EOnCE_EEpin(1, DISABLE_EEx);
}

// ******************************************************************************
void setup_countCyc_ICacheMissToDDR(enum TMRUNIT Unit)
{
        setTmr_InputClk();
        setTmr_PrimaryClk(Unit, InpCtr1);
        setTmr_CountMode(Unit, RisPrim);
        setTmr_CntrReg(Unit, 0);
        setTmr_CountLen(Unit, Rollover);
        setTmr_Cmp1Reg(Unit, 0);

        clrEvPort_EVIN(1);
        clrEvPort_EVOUT(1);
        clrEvPort_EVCTL();
        clrEvPort_EVSELINV();

        setEvPort_Drive_Timer_TINpin(1, DRIVE_TIN1);
        setEvPort_EventInput(1, IC_Miss_DDR);
        setEvPort_Combine(1, OR);
        setEvPort_Enable(1, Always_Enabled);
        setEvPort_Drive_EOnCE_EEpin(1, DISABLE_EEx);
}

// ****************************************************************************
void setup_countCyc_M1Contention(enum TMRUNIT Unit)
{
        setTmr_InputClk();
        setTmr_PrimaryClk(Unit, InpCtr1);
        setTmr_CountMode(Unit, RisPrim);
        setTmr_CntrReg(Unit, 0);
        setTmr_CountLen(Unit, Rollover);
        setTmr_Cmp1Reg(Unit, 0);

        clrEvPort_EVIN(1);
        clrEvPort_EVOUT(1);
        clrEvPort_EVCTL();
        clrEvPort_EVSELINV();

        setEvPort_Drive_Timer_TINpin(1, DRIVE_TIN1);
        setEvPort_EventInput(1, M1_Contention);
        setEvPort_Combine(1, OR);
        setEvPort_Enable(1, Always_Enabled);
        setEvPort_Drive_EOnCE_EEpin(1, DISABLE_EEx);
}

// ****************************************************************************
void setup_countCyc_TDM0RxIntReq(enum TMRUNIT Unit)
{
        setTmr_InputClk();
        setTmr_PrimaryClk(Unit, InpCtr1);
        setTmr_CountMode(Unit, RisPrim);
        setTmr_CntrReg(Unit, 0);
        setTmr_CountLen(Unit, Rollover);
        setTmr_Cmp1Reg(Unit, 0);

        clrEvPort_EVIN(1);
        clrEvPort_EVOUT(1);
        clrEvPort_EVCTL();
        clrEvPort_EVSELINV();

        setEvPort_Drive_Timer_TINpin(1, DRIVE_TIN1);
        setEvPort_EventInput(1, TDM0_Rx_IntReq);
        setEvPort_Combine(1, OR);
        setEvPort_Enable(1, Always_Enabled);
        setEvPort_Drive_EOnCE_EEpin(1, DISABLE_EEx);
}

// ******************************************************************************
void setup_countCyc_TDM1RxIntReq(enum TMRUNIT Unit)
{
        setTmr_InputClk();
        setTmr_PrimaryClk(Unit, InpCtr1);
        setTmr_CountMode(Unit, RisPrim);
        setTmr_CntrReg(Unit, 0);
        setTmr_CountLen(Unit, Rollover);
        setTmr_Cmp1Reg(Unit, 0);

        clrEvPort_EVIN(1);
        clrEvPort_EVOUT(1);
        clrEvPort_EVCTL();
        clrEvPort_EVSELINV();

        setEvPort_GPIOEVNTPin(1);
        setEvPort_Drive_Timer_TINpin(1, DRIVE_TIN1);
        setEvPort_EventInput(1, TDM1_Rx_IntReq);
        setEvPort_Combine(1, OR);
        setEvPort_Enable(1, Always_Enabled);
        setEvPort_Drive_EOnCE_EEpin(1, DISABLE_EEx);
}
```

5.4 File timer.c

```
// ******************************************************************************
long getTmr_ModuleUnit(enum TMRUNIT Unit)
{
        if(Unit == A0 || Unit == A1 || Unit == A2 || Unit == A3 )
                return TMRA_BASE;
        else
                return TMRB_BASE;
}

// ******************************************************************************
short getTmr_Counter(enum TMRUNIT Unit)
{
        int i = Unit / 2;

        pstTMR = (TMRXn*)(getTmr_ModuleUnit(Unit));

        return (pstTMR->astTIMER[i].vusiTMR_CNTR);
}

// ******************************************************************************
void setTmr_InputClk(void)
{
        // Source of Timer clock is APB clock
        pstCLK->vuliCLKCTRL = pstCLK->vuliCLKCTRL | 0xC0000000;
}
```

```
// ****************************************************************************
void setTmr_PrimaryClk(enum TMRUNIT Unit, enum PRIMARYCLK PrimaryClk)
{
       int i = Unit / 2;
       pstTMR = (TMRXn*)(getTmr_ModuleUnit(Unit));
       pstTMR->astTIMER[i].vusiTMR_CTRL |= PrimaryClk;
}

// ****************************************************************************
void setTmr_SecondaryClk(enum TMRUNIT Unit, enum SECONDARYCLK SecondaryClk)
{
       int i = Unit / 2;
       pstTMR = (TMRXn*)(getTmr_ModuleUnit(Unit));
       pstTMR->astTIMER[i].vusiTMR_CTRL |= SecondaryClk;
}

// ****************************************************************************
void setTmr_CountOnce(enum TMRUNIT Unit, enum COUNTONCE CountOnce)
{
       int i = Unit / 2;
       pstTMR = (TMRXn*)(getTmr_ModuleUnit(Unit));
       pstTMR->astTIMER[i].vusiTMR_CTRL |= CountOnce;
}

// ****************************************************************************
void setTmr_LoadReg(enum TMRUNIT Unit, unsigned short int LoadValue)
{
       int i = Unit / 2;
       pstTMR = (TMRXn*)(getTmr_ModuleUnit(Unit));
       pstTMR->astTIMER[i].vusiTMR_LOAD = LoadValue;
}

// ****************************************************************************
void setTmr_Cmp1Reg(enum TMRUNIT Unit, unsigned short int Cmp1Value)
{
       int i = Unit / 2;
       pstTMR = (TMRXn*)(getTmr_ModuleUnit(Unit));
       pstTMR->astTIMER[i].vusiTMR_CMP1 = Cmp1Value;
}

// ****************************************************************************
void setTmr_Cmp2Reg(enum TMRUNIT Unit, unsigned short int Cmp2Value)
{
       int i = Unit / 2;
       pstTMR = (TMRXn*)(getTmr_ModuleUnit(Unit));
       pstTMR->astTIMER[i].vusiTMR_CMP2 = Cmp2Value;
}

// ****************************************************************************
void setTmr_CntrReg(enum TMRUNIT Unit, unsigned short int CntrValue)
{
       int i = Unit / 2;
       pstTMR = (TMRXn*)(getTmr_ModuleUnit(Unit));
       pstTMR->astTIMER[i].vusiTMR_CNTR = CntrValue;
}

// ****************************************************************************
void setTmr_CountMode(enum TMRUNIT Unit, enum COUNTMODE CountMode)
{
        int i = Unit / 2;
        pstTMR = (TMRXn*)(getTmr_ModuleUnit(Unit));
        pstTMR->astTIMER[i].vusiTMR_CTRL |= CountMode;
}

// ****************************************************************************
void setTmr_CountDir(enum TMRUNIT Unit, enum COUNTDIR CountDir)
{
        int i = Unit / 2;
        pstTMR = (TMRXn*)(getTmr_ModuleUnit(Unit));
        pstTMR->astTIMER[i].vusiTMR_CTRL |= CountDir;
}


// ****************************************************************************
void setTmr_CountLen(enum TMRUNIT Unit, enum COUNTLEN CountLen)
{
        int i = Unit / 2;
        pstTMR = (TMRXn*)(getTmr_ModuleUnit(Unit));
        pstTMR->astTIMER[i].vusiTMR_CTRL |= CountLen;
}

// ****************************************************************************
void setTmr_OFLAGMode(enum TMRUNIT Unit, enum OFLAGMODE OflagMode)
{
        int i = Unit / 2;
        pstTMR = (TMRXn*)(getTmr_ModuleUnit(Unit));
        pstTMR->astTIMER[i].vusiTMR_CTRL |= OflagMode;
}

// ****************************************************************************
void setTmr_Interr(enum TMRUNIT Unit, enum TMRINTERR TmrInterr)
{
        int i = Unit / 2;
        pstTMR = (TMRXn*)(getTmr_ModuleUnit(Unit));
        pstTMR->astTIMER[i].vusiTMR_SCR |= TmrInterr;
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

Document Number: AN3073
Rev. 0
06/2006