In order to use the TTC interrupt you must first include a TTC module in the Zynq block in Vivado. As a reminder, you do this when configuring the Zynq in the block design. In particular, you need to enable the timer(s) in the MIO configuration step.



In my implementation, I enabled Timer 0. After completing the customization of the Zynq using the Re-customization IP wizard, you should notice that the Zynq block has additional input and output associated with TTCx (x is either 0 or 1).

When you are in Vitis, you can check that the TTC\_x (x is either 0 or 1) module was properly instantiate by looking in the xparameters.h file located at the folder shown below left – you will have to scroll down. Open the xparameters.h file and look for the definitions associated with TTC\_x. Since each TTC has 3 independent timers, the distinction between the timers inside a TTC\_x module and the TTC\_x module can be unclear. If you remember that there are only 2 TTC module (called 0 and 1) and there are 3 timers inside each module, any variable that has a 0,1, and 2 variation is referencing the timers inside a module. For example the define XPAR\_PS7\_TTC\_2\_BASEADDR in xparameters.h is referencing timer 2 inside the TTC0 module.



In order to use interrupts with a TTC module, you will need to understand the call hierarchy, functions and data structures used inside the two main functions.

**Hardware Instances**

Each hardware component that you work with will have an associated data structure. The name of this data structure is the same as the functions associated with the hardware component. For example, the TTC hardware component has:

 Data structure: XTtcPs Example Function: XTtcPs\_Start

When you are working with these functions, you will often need a pointer to “instance name”. This argument is just the name of the variable associated with the data structure preceeded with a “&” symbol, the address operator in C.

**Code hints in Vitis**

Loiter over a function name brings up its associated header. For example, when I left the cursor over the function SetupTimer the following pop-up appeared. Note this is the contents of the comment block preceding the function declaration. Nice use of comments.



There are times that you will want to look up the declaration of a function or constant. Do this by double clicking on the statement of interest, right clicking and then selecting Open Declaration. When the declaration occurs in more than one file, you will get the Open Declaration pop-up. I always just leave the default selected file and click OK.



In order to make the TTC subsystem generate interrupts to call an interrupt service routine (ISR), you will need to add the code modules and header information in your file. Let’s start with the data structures and definitions in your header given in the number code segment below

Lines 1-3 You will need to include these headers.

Lines 5-6 While not strictly necessary, these definitions give you a more human readable name for your devices so that you can recognize their function from their name. I was not very creative.

Line 9 In my code example, I only used one of the timers in TTC0

Lines 11-16 This data structure will hold the configuration for each timer in TTC0/TTC1

Lines 19-21 Each timer index (line 9) will have a row in this table. All you need to do is fill in the first entry, the frequency of the timer. In my example this is 1000Hz.

Lines 23-25 Function prototypes for the code. You may want to change the name of Ttc0IsrHander to reflect the function of your ISR.

Lines 27,28 This variable holds everything about the interrupt controller. Since there are multiple timers, we’ll store their instances in an array.

**#include** "xil\_exception.h"

**#include** "xttcps.h"

**#include** "xscugic.h"

**#define** TTC0\_0\_DEVICE\_ID XPAR\_XTTCPS\_0\_DEVICE\_ID

**#define** TTC0\_0\_INTR\_ID XPAR\_XTTCPS\_0\_INTR

**#define** INTC\_DEVICE\_ID XPAR\_SCUGIC\_SINGLE\_DEVICE\_ID

**#define** NUM\_TTC0\_INDEX 1

**typedef** **struct** {

 u32 OutputHz; /\* Output frequency \*/

 XInterval Interval; /\* Interval value \*/

 u8 Prescaler; /\* Prescaler value \*/

 u16 Options; /\* Option settings \*/

} TmrCntrSetup;

TmrCntrSetup SettingsTable[NUM\_TTC0\_INDEX] = {

 {1000, 0, 0, 0}

};

/\* Set up routines for timer counters \*/

**int** **SetupIntervalTimerWithInterrupt**(**void**);

**static** **int** **SetupInterruptSystem**(u16 IntcDeviceID, XScuGic \*IntcInstancePtr);

**static** **void** **Ttc0IsrHander**(**void** \*CallBackRef, u32 StatusEvent);

XScuGic InterruptController; /\* Interrupt controller instance \*/

XTtcPs TtcPsInst[NUM\_TTC0\_INDEX]; /\* Number of available timer counters \*/

The function SetupIntervalTimerWithInterrupt has a lot going on, so we’ll try to go through it slowly so that you

Lines 8-10 Pointers will allow you to change the values of the data structures inside functions

Lines 12 -13 Assign pointer to the instances that were defined in the global context

Lines 19-20 Setup TTC0 as an interval timer that does not manipulate a pin

Lines 22-38 Pulls the default configuration and then writes this to the TTC module. Sets a lot of small details that you probably do not want to mess with and some that you do.

Lines 46-53 No more timer math! You have a function that will do it for your. Takes the frequency you put for SettingTables and calculates the prescaler and interval count for the TTC module, neat.

Lines 59-77 Pretty much boiler plate stuff with the exceptions of line 66. You need to put the name of the ISR here.

Line 82 Easy peezy, lemon squeezy.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\* Removed comment block to fit on a page

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**int** **SetupIntervalTimerWithInterrupt**(**void**)

{

 **int** Status;

 TmrCntrSetup \*TimerSetup;

 XTtcPs\_Config \*Config;

 XTtcPs \*TtcTimerInstPtr;

 TtcTimerInstPtr = &(TtcPsInst[TTC0\_0\_DEVICE\_ID]);

 TimerSetup = &SettingsTable[TTC0\_0\_DEVICE\_ID];

 /\*

 \* Set up appropriate options for Ticker: interval mode without

 \* waveform output.

 \*/

 TimerSetup->Options |= (XTTCPS\_OPTION\_INTERVAL\_MODE |

 XTTCPS\_OPTION\_WAVE\_DISABLE);

 Config = XTtcPs\_LookupConfig(TTC0\_0\_DEVICE\_ID);

 **if** (NULL == Config) {

 **return** XST\_FAILURE;

 }

 /\*

 \* Initialize the device

 \*/

 Status = XTtcPs\_CfgInitialize(TtcTimerInstPtr, Config, Config->BaseAddress);

 **if** (Status != XST\_SUCCESS) {

 **return** XST\_FAILURE;

 }

 /\*

 \* Set the options

 \*/

 XTtcPs\_SetOptions(TtcTimerInstPtr, TimerSetup->Options);

 /\*

 \* Timer frequency is preset in the TimerSetup structure,

 \* however, the value is not reflected in its other fields, such as

 \* IntervalValue and PrescalerValue. The following call will map the

 \* frequency to the interval and prescaler values.

 \*/

 XTtcPs\_CalcIntervalFromFreq(TtcTimerInstPtr, TimerSetup->OutputHz,

 &(TimerSetup->Interval), &(TimerSetup->Prescaler));

 /\*

 \* Set the interval and prescaler

 \*/

 XTtcPs\_SetInterval(TtcTimerInstPtr, TimerSetup->Interval);

 XTtcPs\_SetPrescaler(TtcTimerInstPtr, TimerSetup->Prescaler);

 /\*

 \* Connect to the interrupt controller

 \*/

 Status = XScuGic\_Connect(&InterruptController, TTC0\_0\_INTR\_ID,

 (Xil\_ExceptionHandler)XTtcPs\_InterruptHandler, (**void** \*)TtcTimerInstPtr);

 **if** (Status != XST\_SUCCESS) {

 **return** XST\_FAILURE;

 }

 XTtcPs\_SetStatusHandler(&(TtcPsInst[TTC0\_0\_DEVICE\_ID]), &(TtcPsInst[TTC0\_0\_DEVICE\_ID]),

 (XTtcPs\_StatusHandler)Ttc0IsrHander);

 /\*

 \* Enable the interrupt for the Timer counter

 \*/

 XScuGic\_Enable(&InterruptController, TTC0\_0\_INTR\_ID);

 /\*

 \* Enable the interrupts for the tick timer/counter

 \* We only care about the interval timeout.

 \*/

 XTtcPs\_EnableInterrupts(TtcTimerInstPtr, XTTCPS\_IXR\_INTERVAL\_MASK);

 /\*

 \* Start the tick timer/counter

 \*/

 XTtcPs\_Start(TtcTimerInstPtr);

 **return** Status;

}