Revision: 30SEP2015 (JFF)
Richard W. Wall, University of Idaho, rwall@uidaho.edu

DIGILENT®
www.digilentinc.com
1300 NE Henley Court, Suite 3
Pullman, WA 99163
(509) 334 6306 Voice | (509) 334 6300 Fax

Project 5: Process Speed Control Using Interrupts

Richard W. Wall, University of Idaho, rwall@uidaho.edu

1300 NE Henley Court, Suite 3
Pullman, WA 99163
(509) 334 6306 Voice | (509) 334 6300 Fax

Purpose

The purpose of this project is to explore detecting events using interrupts or preemption that implements a nested interrupt management scheme. Concepts concerning processor context are introduced as well as interrupt prioritization. The functional requirements for Project 5 are essentially identical to those of Project 4. The biggest difference is that time and button action events are detected by interrupts rather than software that polls processor flags and levels on input pins.

Minimum Knowledge and Programming Skills

1. Knowledge of C or C++ programming
2. Working knowledge of MPLAB IDE
3. Understanding of Finite State Machines
4. IO pin control
5. Use of logic analyzer or oscilloscope

Equipment List

1. Cerebot 32MX7cK processor board with USB cable
2. Microchip MPLAB ® X IDE
3. PmodSTEP
4. Stepper Motor (5V – 12V, 25Ω, unipolar or bi-polar)
5. Logic analyzer, or oscilloscope (Suggestion - Digilent Analog Discovery)

Software Resources

1. XC32 C/C++ Compiler Users Guide
2. PIC32 Peripheral Libraries for MBLAB C32 Compiler
3. Cerebot MX7cK Board Reference Manual
4. PIC32 Family Reference Manual Section 8: Interrupts
5. PIC32 Family Reference Manual Section 12: IO Ports
6. PIC32 Family Reference Manual Section 14: Timers
7. MPLAB ® X Integrated Development Environment (IDE)

8. C Programming Reference

References

Micri C – Interrupt Tutorial

Extreme Electronics – Introduction to PIC Interrupts and their Handling in C.

Interrupt Programming Concepts

"*In computing, preemption is the act of temporarily interrupting a task being carried out by a computer system, without requiring its cooperation, and with the intention of resuming the task at a later time. It is normally carried out by a privileged task or part of the system that has the power to interrupt, and later resume, other tasks in the system.*"[1] In software, a task is a basic unit of programming that has a constrained defined purpose. For example, reading the state of the input pins to determine if buttons are pressed or not can be considered a task. Multitasking implies that more than one task is running at the same time. Single core processors appear to be multitasking by time-sharing the processor resources. No time sharing is required in Projects 3 and 4 where each task in the program ran to its completion before the next task could begin. When multitasking with a single core processor, some tasks are suspended so that higher priority tasks can be serviced (executed).

Mechanics of Interrupts

Preemption by an interrupt causes the processor to stop the execution of the current code flow and begin execution of a special function called an interrupt handler or interrupt service routine (ISR). In previous projects, the order in which tasks were started and completed was strictly controlled by the order in which code was written in the program. In this project, tasks are started and completed based on a demand prioritization.

An interrupt is a signal triggered by an event. The computer suspends what would normally be the next instruction of a task in one part of a program and begins executing code to complete an entirely different task. The occurrence of events can be periodic (deterministic) or aperiodic (sporadic). An example of a deterministic event is a timer flag being set. Sporadic interrupts are the result of an unpredictable event such as a low power detection, or serial data received flag. These signals cause the microprocessor to stop executing the progression of current instructions and start the process of "handling" or "servicing" the interrupt by executing code that is specific to the source of the interrupt.

[1] http://en.wikipedia.org/wiki/Preemption_(computing)

In this project the interrupts are generated by hardware peripherals within the PIC32. However, it is also possible to trigger interrupts via software by setting the appropriate interrupt flag. In addition to interrupts, program execution can be altered by the occurrence of an exception. Exceptions are caused by unexpected (and undesireable) events, indicating some sort of software error or hardware malfunction. Examples include divide by zero, misaligned memory access, or invalid instruction. Exceptions are typically serviced by special system code.

The context defines the state or operating conditions of the processor. In order to properly resume executing the task that was interrupted, the ISR is responsible for saving and restoring the context of the code that is interrupted. At a minimum, the context consists of the processor register values, the flag register and the program address of the next instruction to be executed when the ISR has completed. The C compiler is directed to generate code segments called a prolog and an epilog. The prolog code saves the processor's context prior to executing the code that you have generated to service the interrupt. The epilog code restores the context at the end of the ISR. For the PIC32, the C compiler generates 33 assembly language instructions to save the processor context and an equal number of instructions to restore the context, thus requiring approximately 825 ns before and after executing the "useful" ISR code.

Interrupt latency is defined as the time between the instant when an event occurs that generates the interrupt signal and the time when the first useful instruction in the ISR is executed. This latency is made up of two major components: the time required to save the processor context and the amount of time that an interrupt is disabled or deferred by higher priority interrupts.

Managing Interrupts

Preemptive programs usually consist of two types of tasks: foreground and background. Foreground tasks are those preemptive tasks that the processor executes as soon as the need arises. Interrupts are assigned priority levels that dictate the order that interrupts will be serviced. In the case of multiple foreground tasks both needing service at the same time, the higher the interrupt priority, the sooner the microprocessor executes the code to service that interrupt. Code executed immediately following a power up reset runs at priority level zero and has no preemptive capability. This initial background task is responsible for setting up the resources for tasks that will eventually run as foreground tasks.

Foreground tasks have preemptive capability by having higher priorities than background tasks. The foreground task ensures adequate response times, while the background task manages deferred processing of the foreground data. Background tasks are those that the processor executes whenever it has time available and is waiting for something more important to do. Background tasks generally run at the interrupt priority level zero within a "while(1) { … }" loop where event detection by polling is common. Interrupt only systems that only have foreground processes respond quickly to both periodic and sporadic events but ignore potential work that can be allocated to a background process. The use of interrupts fall under the broad category of real time task scheduling and a thorough investigation of this topic is beyond the scope of this project.

There are two types of preemptive operating schemes: nested and non-nested. Nested interrupt schemes allow higher priority level interrupts to preempt code that is servicing a lower priority interrupt. Non-nested interrupt schemes complete the execution of the code servicing the current interrupt before it begins to service the code for any interrupt awaiting service.

Non-nested preemptive schemes are usually easier to manage but can result in priority inversion where a low priority task blocks a higher priority task from running. This scheme is easier to manage because the developer only has to focus on one ISR and the operation that determines which interrupts to service resembles the process of polling that we did in Project 4. For non-nested interrupt schemes, flags that are set by the interrupting event must be individually checked to determine which event to service. All interrupts are assigned to priority level 1 and are vectored (sent) to the same ISR where the particular interrupt flag must be cleared. For multiple simultaneous interrupts, tasks are serviced in the order that the interrupt flags are polled. A non-nested priority scheme results whenever interrupts are enabled using the instruction "*INTEnableSystemSingleVectoredINT();*" provided in the peripheral library. (Note: there is actually a little more to this. "Single Vector" really means that there is a single ISR, not necessarily non-nesting – it's just that is the norm. Having said that, there is no reason to use a single vector scheme in today's world.)

Nested priority schemes are generally more responsive by taking advantage of the fact that higher priority tasks can preempt (interrupt) lower priority tasks. The highest priority interrupt is guaranteed to have the lowest latency. However, nested priority schemes require more computer resources such as time and memory to accommodate context saving and restoring. Some low-end microcontrollers simply do not have sufficient memory and/or speed to support fully nested priority schemes. In reality, many embedded systems manage tasks using both nested and non-nested preemptive schemes. Either interrupt management scheme can be used in combination with polling to detect events as will be the case for Project 5.

The PIC32MX family of processors can use polling, nested priority, and non-nested priority schemes simultaneously. The PIC32MX processors utilize two major software level priorities called the "Group Priority" and the "Subgroup Priority". There are seven levels that can be assigned for the group priority (1-7), with one being the lowest priority and seven being the highest. An interrupt with a higher group priority will preempt an interrupt of a lower priority. There are four subgroup priority levels that can be assigned at each of the seven levels of group priority. Interrupts assigned to different group priority levels operate as nested interrupts.

Multiple events can be assigned interrupt priorities at the same group level but different subgroup levels. The sub-priority will not cause preemption of an interrupt in the same priority; rather, if two interrupts with the same priority are pending, the interrupt with the highest sub-priority will be handled first. The natural (hardware) priority scheme is asserted whenever multiple interrupts are generated simultaneously for events that are set for the same group and subgroup priority levels. (The notion of "simultaneous events" must be expanded to mean "if two interrupts are detected as waiting for service" whether or not they occurred at the same instant

of time.) See Section 8 of the PIC32MX Family Reference Manual for additional details of interrupt operations on the PIC32MX family of processors.

In all cases, for the PIC32MX family of processors, the interrupt flag bit must be cleared in software prior to completing the service of the interrupt otherwise the same interrupt will be serviced again without an event to initiate its service.

PIC32 Interrupts

1. General Interrupt Requirements:

There are three essential code elements required for a program to process interrupts using C: the declaration of interrupt functions, the code to initialize the resources that generate interrupts, and the ISR code that will be executed in response to an interrupt. Functions that have been declared as an ISR cannot be called by any other C function. There are two ways that the ISR code can be invoked: either in response to the event that causes the interrupt flag to be set, or by software that sets the corresponding bit in the interrupt flag register. A function that is declared as an ISR cannot have any variables passed to it (no argument list) and must return a void data type.

The processor must be initialized to generate the interrupt when the interrupt flag for a peripheral is set. This is normally done only once in the program for an application. In this project, two different interrupts will be generated: Timer 1 interrupt and an IO pin change notice interrupt. The programming requirements for each of these two interrupts are discussed below. The two statements in Listing 1 apply to all interrupts and should be executed only once after all interrupts have been initialized. In this instance, the program is using multi-vectored interrupts. Selected segments of the code used in an application can be protected from disruption by any and all interrupts by bracketing the code segment with the instructions *INTEnableInterrupts();* and *INTDisableInterrupts();*.

Listing 1

```
// Enable multi vectored interrupts
      INTEnableSystemMultiVectoredInt(); //done only once
      INTEnableInterrupts();        //use as needed
```

2. Timer Interrupts:

In project 4, you configured Timer 1 to set the interrupt flag once each millisecond. However, no interrupt was ever generated. In order to set up a timer interrupt, the timer must first be initialized to run as was done in Project 4. This is accomplished using the *OpenTimer1* instruction shown in Listing 2. The next three macro functions enable Timer 1 interrupts. The first macro instruction sets the Timer 1 interrupt priority level to 2, the second sets the Timer 1 sub priority level to 0 and the third enables Timer 1 interrupts.

Listing 2

```
#define T1_INTR_RATE 10000    // For 1 ms interrupt rate with T1 clock=10E6

void timer1_interrupt_initialize(void)
{

//configure Timer 1 with internal clock, 1:1 prescale, PR1 for 1 ms period
      OpenTimer1(T1_ON | T1_SOURCE_INT | T1_PS_1_1, T1_INTR_RATE-1);

// set up the timer interrupt with a priority of 2, sub priority 0
      mT1SetIntPriority(2);          // Group priority range: 1 to 7
      mT1SetIntSubPriority(0);       // Subgroup priority range: 0 to 3
      mT1IntEnable(1);               // Enable T1 interrupts

// Global interrupts must enabled to complete the initialization.
}
```

Timer 1 interrupts can be disabled at any point in the application software by using the instruction *mT1IntEnable(0);*.Other methods for initializing Timer 1 interrupts are shown in Appendix A.

Listing 3 code shows how to both declare a function to be an ISR and the general format of an ISR function. This method of declaring an ISR eliminates the requirement of a function prototype. Additional methods are provided in Appendix A because they may be used by other sources that the reader may encounter. *Note the two underscore characters preceding "ISR"; all others are a single underscore.*

Listing 3

```
   void __ISR(_TIMER_1_VECTOR, IPL2) Timer1Handler(void)
   {

      /* User generated code to service the interrupt is inserted here */

      mT1ClearIntFlag();      // Macro function to clear the interrupt flag

   }
```

3. Change Notice Interrupts:

Change notice (CN) interrupts are generated on selected enabled digital IO pins whenever the voltage of the pin changes causing the processor to read a logic value that is different from the previous reading of the PORT register. Pins designated as CN interrupt pins are shown in Table I. Only BTN1 and BTN2 on the Cerebot MX7cK processor board have the capability to generate CN interrupts. BTN3 cannot generate a CN interrupt.

Table I. IO pins with CN interrupt capability

CN x	Port	CN x	Port
CN0	RC13	CN10	RG8

CN1	RC14	CN11	RG9
CN2	RB0	CN12	RB15
CN3	RB1	CN13	RD4
CN4	RB2	CN14	RD5
CN5	RB3	CN15	RD6
CN6	RB4	CN16	RD7
CN7	RB5	CN17	RF4
CN8	**RG6 – BTN1**	CN18	RF5
CN9	**RG7 – BTN2**	CN19-CN21	NA

CN interrupts can be disabled to protect segments of code using *mCNIntEnable(0);* to disable the interrupts and *mCNIntEnable(1);* to enable them again . Listing 4 shows how to setup change notice interrupts using peripheral library code macros. In this code, CN interrupts are turned on for CN8 and CN9 that correspond to BTN1 (RG6) and BTN2 (RG7). The last parameter with value zero directs that internal pull-up resistors be disabled because these pins have external pull-up resistors on the Cerebot MX7cK processor board. The code in Listing 4 enable CN interrupts at priority level 1 and sub priority level zero.

Listing 4

```
/* Initialization of CN peripheral for interrupt level 1 */
   void cn_interrupt_initialize(void) // Code that is executed only once
   {

      unsigned int dummy; // used to hold PORT read value

      // BTN1 and BTN2 pins set for input by Cerebot header file
      // PORTSetPinsDigitalIn(IOPORT_G, BIT_6 | BIT7); //

      // Enable CN for BTN1 and BTN2
      mCNOpen(CN_ON,(CN8_ENABLE | CN9_ENABLE), 0);

      // Set CN interrupts priority level 1 sub priority level 0
      mCNSetIntPriority(1);                    // Group priority (1 to 7)
      mCNSetIntSubPriority(0);                 // Subgroup priority (0 to 3)
      // read port to clear difference
      dummy = PORTReadBits(IOPORT_G, BTN1 | BTN2);
      mCNClearIntFlag();                       // Clear CN interrupt flag
      mCNIntEnable(1);                         // Enable CN interrupts

      // Global interrupts must enabled to complete the initialization.
}
```

A single interrupt vector is used for all CN interrupts. The interrupt does not tell you if the pin went high or low; only that the condition on one of the selected pins has changed. The ISR code must return a type *void* and have no parameters passed to it. The CN interrupt flag must be cleared prior to exiting the ISR using the instructions *mCNClearIntFlag();*.

Listing 5

```
void __ISR(_CHANGE_NOTICE_VECTOR, IPL1) CNIntHandler(void)
{
      /* User ISR code inserted here */

      /* Required to clear the interrupt flag in the ISR */
      mCNClearIntFlag();            // Macro function
}
```

Project Tasks

This project implements a system that runs entirely using foreground processes. The Timer 1 interrupt flag is used to generate an interrupt once each millisecond. The PIC32 Change Notice Interrupt generates an interrupt when a button is pressed or released. The project consists of writing a program that meets the specifications listed below:

1. Functionality for “main” function
    a. Calls “system_init” function (see step 2)
    b. Continuously executes the statement “while(1);”
2. Create a “system_init” function that implements a fully nested interrupt scheme for Timer 1 and change notice interrupts using the following steps.
    a. Initialize the processor board using the function *Cerebot_mx7cK_setup();.* This function initializes IO port G to read the button inputs for BTN1 and BTN2.
    b. Initialize IO port B for LEDA, LEDB, LEDC, and stepper motor set as outputs.
    c. Initialize Timer 1 to generate an interrupt once each ms . Set the group priority for level 2 and the subgroup level for 0.
    d. Initialize the PIC32MX7 system for CHANGE NOTICE detection. Set change notice interrupts to detect activity on BTN1 and BTN2 only, the group priority level 1 and the subgroup level 0.
    e. Set the system for multiple vectored interrupts.
3. Functionality for “change_notice_ISR” (replaces “read_buttons” function in Project 4)
    a. Set LEDC on at the beginning of the ISR and off at the end of the ISR
    b. Delay for 20 ms to allow the buttons to settle, i.e., debounce period
    c. Determine the button status
    d. Decode the button status
    e. Clear the Change Notice interrupt flag (CNIF)
4. Functionality for “Timer1_ISR”
    a. Set for a 1 ms interval (Refer to Project 4)

b. Toggle LEDA
c. Decrement the “step_delay” variable.
d. When step delay is zero
    i. Call “stepper_state_machine” and “output_to_stepper_motor” functions.
    ii. Reset step_delay variable to the value stored in the step_period global variable
e. Clear the Timer 1 interrupt flag (T1IF)

5. Functionality for “decode_buttons” (Refer to Project 4)
    a. Determine values for global variables “step_dir”, “step_mode”, and “step_period”
6. Functionality for “stepper_state_machine” (Refer to Project 4)
    a. Toggle LEDB
    b. Determine stepper motor output signals for “step_code” (local)
    c. Call “output_to_stepper_motor” function with value of step_code
7. Functionality for “output_to_stepper_motor” (Refer to Project4)
    a. Output step_code to Port B using read-modify-write sequence.
8. Functionality for the button debounce period
    a. Implement a hardware assisted software delay using the core timer that can be preempted

Project Testing

There are significant timing requirements for this project: the Timer 1 interrupt rate, the stepper motor step rate, and the button debounce delay. Table 1 is provided to assist you in determining where to place the instructions to control the instrumentation LEDs. Appendix B shows where to connect an oscilloscope or logic analyzer to the test points for LEDA through LEDC on the PmodSTEP to instrument this project. An example of a logic analyzer screen capture is shown in **Figure 1**.

Table 1. Instrumentation LED operations

LED	Operation	When
LEDA	Toggle	Each millisecond
LEDB	Toggle	Each step
LEDC	SET (on)	Start of CN ISR
LEDC	Cleared (off)	End of CN ISR

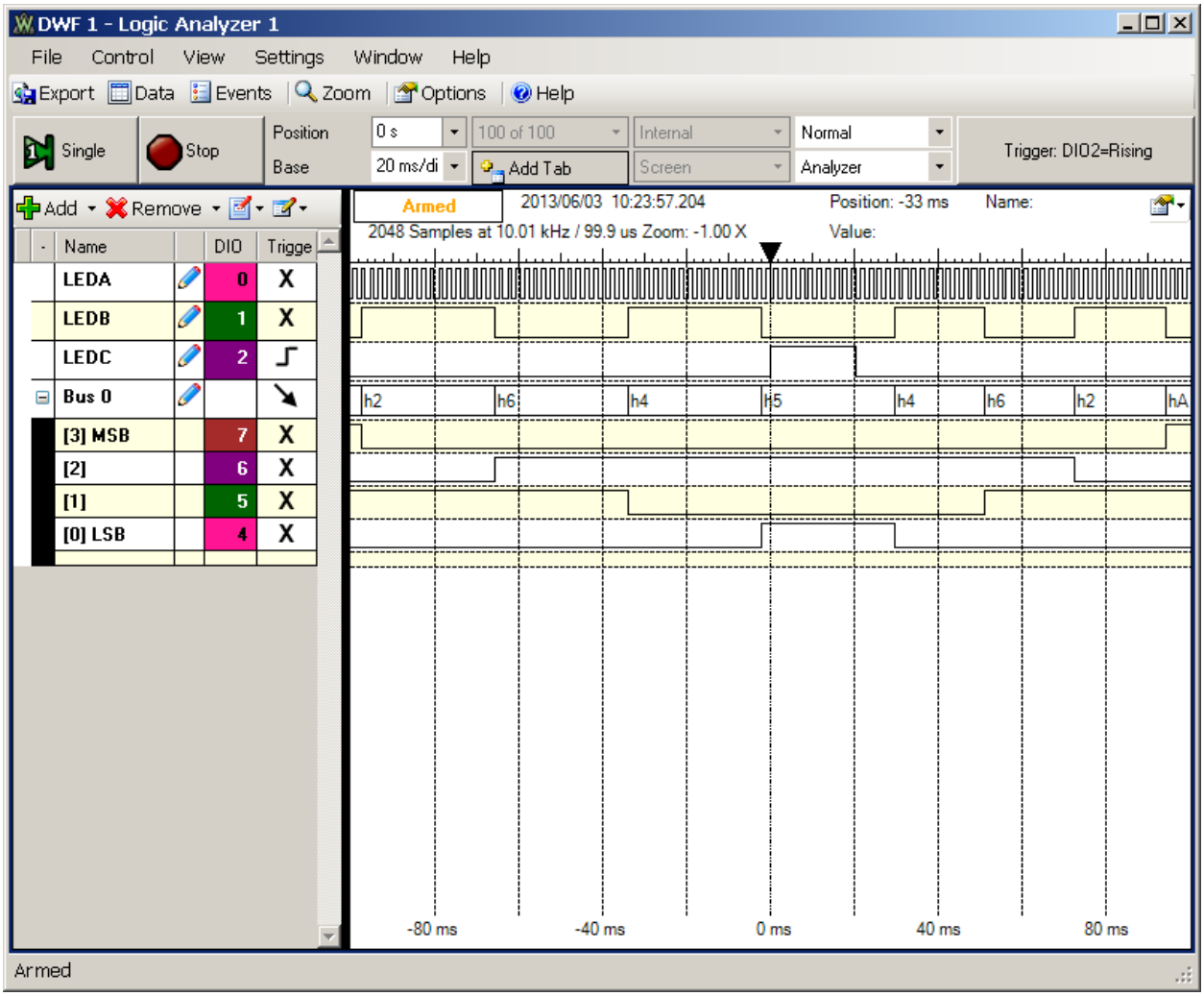

Figure 1. Screen capture for the instrumentation of Project 5.

Appendix A: PIC32 Interrupts

1. Alternate Methods for Declaring Interrupt Service Routines

The following three examples of declaring a function to service a Timer 1 interrupt as an ISR are equivalent. Although method "a" is probably the simplest to write, methods "b" and c" are provided because they may be used by other sources that the reader may encounter. **Note: the use of <u>two</u> underscores below.**

a. This method eliminates the requirement of a function prototype. __ISR is a macro that is expanded by the compiler to method b below. Saves some typing. (Section 13.3.3 of XC32 Compiler UG.)

```
void __ISR(_TIMER_1_VECTOR, IPL2) Timer1Handler(void)
{
      /* ISR code inserted here */
}
```

b. Using the attribute declaration (Section 13.3.1 XC32 Compiler UG.)

```
void __attribute__( (interrupt(IPL2),
vector(_TIMER_1_VECTOR))) Timer1Handler ( void );

void Timer1Handler(void)
{
      /* ISR code inserted here */
}
```

c. Using a pragma declaration (no longer recommended!) Requires the vector number be entered numerically.

```
#pragma interrupt Timer1Handler IPL2 vector 4

void Timer1Handler(void)
{
      /* ISR code inserted here */
}
```

2. Alternate Methods for configuring Timer Interrupts

There are (at least!) three methods for enabling Timer 1 interrupts. The first method is to use the following three C program statements that employ macro functions from the peripheral library. See Section 7.4 of the Peripheral Library Guide. The specific peripheral is identified by replacing "(xx)" with identifiers from Table 8-2. (Don't ask why Table 8-2 is in Section 7!) ☹

```
mT1SetIntPriority(p);          // Group priority: 1 <= p <= 7
mT1SetIntSubPriority(sp);      // Subgroup priority: 0 <= sp <= 3
mT1IntEnable();                // Enable T1 interrupts
```

The second method uses system three system functions from the peripheral library, Section 7. The specific peripheral is identified using enumerations from Table 8-1.

```
INTSetPriority(INT_T1, INT_PRIORITY_LEVEL_p); // 1 <= p <= 7

INTSetSubPriority(INT_T1, INT_SUB_PRIORITY_LEVEL_sp);
                                                    // 0 <= sp <= 3

INTEnable(INT_T1, EN)    // EN = 1 for enable, 0 for disable
```

A third method uses a single statement that requires multiple bits be logically ORed in a configuration parameter as show below. (see Section 11.2 of the Peripheral Library Guide)

```
void ConfigIntTimerx(unsigned int config);// x = 0,1,2,3,4,5, 23, 45
```

config definitions (logically OR selections)

Timer interrupt enable/disable (These bit fields are mutually exclusive):
Tx_INT_ON / Tx_INT_OFF

Timer interrupt priorities (These bit fields are mutually exclusive):
Tx_INT_PRIOR_7, Tx_INT_PRIOR_6, Tx_INT_PRIOR_5,
Tx_INT_PRIOR_4, Tx_INT_PRIOR_3, Tx_INT_PRIOR_2,
Tx_INT_PRIOR_1, Tx_INT_PRIOR_0

Timer interrupt sub- priorities (These bit fields are mutually exclusive):
Tx_INT_SUB_PRIOR_3, Tx_INT_SUB_PRIOR_2,
Tx_INT_SUB_PRIOR_1, Tx_INT_SUB_PRIOR_0

Note: the “23”, and “45” are used when combining two timers to create a 32-bit timer.