## 1 Introduction

The Texas Instruments® TivaWare™ supports two models for programming the Tiva™ family of ARM® Cortex™-M based microcontrollers, the *Direct Register Access Model* and the *Software Driver Model*.

In the **direct register access model**, the peripherals are programmed by the application by writing values directly into the peripheral's registers. A set of macros is provided that simplifies this process. These macros are stored in part-specific header files contained in the *inc* directory; the name of the header file matches the part number (for example, the header file for the TM4C129ENCPDT microcontroller is inc/tm4c129encpdt.h). By including the header file that matches the part being used, macros are available for accessing all registers on that part, as well as all bit fields within those registers. No macros are available for registers that do not exist on the part in question, making it difficult to access registers that do not exist.

In the **software driver model**, the API provided by the peripheral driver library is used by applications to control the peripherals. Because these drivers provide complete control of the peripherals in their normal mode of operation, it is possible to write an entire application without direct access to the hardware. This method provides for rapid development of the application without requiring detailed knowledge of how to program the peripherals.

In this lab, we will use the software driver model to program the timers/counters and the interrupt modules of the TM4C129ENCPDT microcontroller.

## 2 UART

The universal asynchronous receiver transmitter (UART) is one of the commonly employed peripherals in microcontroller chips for serial communications. The Universal Asynchronous Receiver/Transmitter (UART) API provides a set of functions for using the Tiva UART modules. Functions are provided to configure and control the UART modules, to send and receive data, and to manage interrupts for the UART modules. The Tiva UART performs the functions of parallel-to-serial and serial-to-parallel conversions.

## 3 Timers/Counters

### 2.1 Introduction

Timers can be used to count or time external events that drive the Timer input pins. The TM4C129ENCPDT General-Purpose Timer Module (GPTM) contains 16/32-bit GPTM blocks. Each 16/32-bit GPTM block provides two 16-bit timers/counters (referred to as Timer A and Timer B) that can be configured to operate independently as timers or event counters or concatenated to operate as one 32-bit timer or one 32-bit Real-Time Clock (RTC). Timers can also be used to trigger µDMA transfers **(read Chapter 16 of the *TM4C129ENCPDT Datasheet* to know more about the functional description of the module)**.

### 2.2 The Timer API

The timer API provides a set of functions for using the timer module. Functions are provided to configure and control the timer, modify timer/counter values, and manage timer interrupt handling. The timer module provides two half-width timers/counters that can be configured to operate independently as timers or event counters or to operate as a combined full-width timer or Real Time Clock (RTC). Some timers provide 16-bit half-width timers and a 32-bit full-width timer, while others provide 32-bit half-width timers and a 64-bit full-width timer.

When configured as either a full-width or half-width timer, a timer can be set up to run as a one-shot timer or a continuous timer. If configured in one-shot mode, the timer ceases counting when it reaches zero when counting down or the load value when counting up. If configured in continuous mode, the timer counts to zero (counting down) or the load value (counting up), then reloads and continues counting. When configured as a full-width timer, the timer can also be configured to operate as an RTC. In this mode, the timer expects to be driven by a 32.768-KHz external clock, which is divided down to produce 1 second clock ticks. When in half-width mode, the timer can also be configured for event capture or as a Pulse Width Modulation (PWM) generator. When configured for event capture, the timer acts as a counter. It can be configured to either count the time between events or the events themselves. The type of event being counted can be configured as a positive edge, a negative edge, or both edges. When a timer is configured as a PWM generator, the input signal used to capture events becomes an output signal, and the timer drives an edge-aligned pulse onto that signal.

The timer module also provides the ability to control other functional parameters, such as output inversion, output triggers, and timer behavior during stalls. Control is also provided over interrupt sources and events. Interrupts can be generated to indicate that an event has been captured, or that a certain number of events have been captured. Interrupts can also be generated when the timer has counted down to zero or when the timer matches a certain value.

On some parts, the counters from multiple timer modules can be synchronized. Synchronized counters are useful in PWM and edge time capture modes. In PWM mode, the PWM outputs from multiple timers can be in lock-step by having the same load value and synchronizing the counters (meaning that the counters always have the same value). Similarly, by using the same load value and synchronized counters in edge time capture mode, the absolute time between two input edges can be easily measured.

This driver is contained in *driverlib/timer.c*, with *driverlib/timer.h* containing the API declarations for use by applications (read Chapter 29 of the *TivaWare Peripheral Driver Library* for full description of the API functions).

## 2.3 A Programming Example

The following example shows how to use the timer API to configure the timer as a half-width one shot timer and a half-width edge capture counter.

```
// Enable the Timer0 peripheral
    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);

// Wait for the Timer0 module to be ready.
while(!SysCtlPeripheralReady(SYSCTL_PERIPH_TIMER0)){
}

// Configure TimerA as a half-width one-shot timer, and TimerB as a
// half-width edge capture counter.
    TimerConfigure(TIMER0_BASE, (TIMER_CFG_SPLIT_PAIR |
            TIMER_CFG_A_ONE_SHOT |
            TIMER_CFG_B_CAP_COUNT));
```

```
// Set the count time for the the one-shot timer (TimerA).
    TimerLoadSet(TIMER0_BASE, TIMER_A, 3000);

// Configure the counter (TimerB) to count both edges.
    TimerControlEvent(TIMER0_BASE, TIMER_B, TIMER_EVENT_BOTH_EDGES);

// Enable the timers.
    TimerEnable(TIMER0_BASE, TIMER_BOTH);
```

# 4 Interrupts

## 4.1 Introduction

Interrupts can be used to change the normal flow of software control in response to a hardware event that is asynchronous with the current software execution. In the system, peripherals use interrupts to communicate with the processor. The processor handles interrupts by invoking a special software called *Interrupt Service Routines (ISRs)* (read Chapter 2.5 of the TM4C129ENCPDT Datasheet to know more about the interrupts).

The TM4C129ENCPDT Nested Vectored Interrupt Controller (NVIC) is used to prioritize and handle all interrupts. The processor state is automatically stored to the stack on an interrupt and automatically restored from the stack at the end of the Interrupt Service Routine (ISR). The interrupt vector is fetched in parallel to the state saving, enabling efficient interrupt entry. The processor supports tail-chaining, meaning that back-to-back interrupts can be performed without the overhead of state saving and restoration. Software can set eight priority levels on 7 exceptions (system handlers) and 106 interrupts (read Chapter 3.4 of the TM4C129ENCPDT Datasheet to know more about the functional description of NVIC).

## 4.2 The Interrupt Controller API

The interrupt controller API provides a set of functions for dealing with the Nested Vectored Interrupt Controller (NVIC). Functions are provided to enable and disable interrupts, register interrupt handlers, and set the priority of interrupts. The NVIC provides global interrupt masking, prioritization, and handler dispatching. Devices within the Tiva family support up to 154 interrupt sources and eight priority levels. Individual interrupt sources can be masked, and the processor interrupt can be globally masked as well (without affecting the individual source masks).

The NVIC is tightly coupled with the Cortex-M microprocessor. When the processor responds to an interrupt, the NVIC supplies the address of the function to handle the interrupt directly to the processor. This action eliminates the need for a global interrupt handler that queries the interrupt controller to determine the cause of the interrupt and branch to the appropriate handler, reducing interrupt response time.

The interrupt prioritization in the NVIC allows higher priority interrupts to be handled before lower priority interrupts, as well as allowing preemption of lower priority interrupt handlers by higher priority interrupts. Again, this helps reduce interrupt response time (for example, a 1ms system control interrupt is not held off by the execution of a lower priority 1 second housekeeping interrupt handler).

Sub-prioritization is also possible; instead of having N bits of preemptable prioritization, the NVIC can be configured (via software) for N - M bits of preemptable prioritization and M bits of sub-priority.

In this scheme, two interrupts with the same preemptable prioritization but different sub-priorities do not cause a preemption; tail chaining is used instead to process the two interrupts back-to-back. If two interrupts with the same priority (and sub-priority if so configured) are asserted at the same time, the one with the lower interrupt number is processed first. The NVIC keeps track of the nesting of interrupt handlers, allowing the processor to return from interrupt context only once all nested and pending interrupts have been handled.

Interrupt handlers can be configured in one of two ways; statically at compile time or dynamically at run time. Static configuration of interrupt handlers is accomplished by editing the interrupt handler table in the application's startup code. When statically configured, the interrupts must be explicitly enabled in the NVIC via IntEnable() before the processor can respond to the interrupt (in addition to any interrupt enabling required within the peripheral itself). Statically configuring the interrupt table provides the fastest interrupt response time because the stacking operation (a write to SRAM) can be performed in parallel with the interrupt handler table fetch (a read from Flash), as well as the prefetch of the interrupt handler itself (assuming it is also in Flash).

Alternatively, interrupts can be configured at run-time using IntRegister() (or the analog in each individual driver). When using IntRegister(), the interrupt must also be enabled as before; when using the analogue in each individual driver, IntEnable() is called by the driver and does not need to be called by the application. Run-time configuration of interrupts adds a small latency to the interrupt response time because the stacking operation (a write to SRAM) and the interrupt handler table fetch (a read from SRAM) must be performed sequentially.

This driver is contained in *driverlib/interrupt.c*, with *driverlib/interrupt.h* containing the API declarations for use by applications transfers (read Chapter 17 of the *TivaWare Peripheral Driver Library* for full description of the API functions).

## 4.3 A Programming Example

The following example shows how to use the Interrupt Controller API to register an interrupt handler for UART 0 and enable the interrupt.

```
//
// The interrupt handler function.
//
    extern void IntHandler(void);
//
// Register the interrupt handler function for UART 0.
//
    IntRegister(INT_UART0, IntHandler);
//
// Enable the interrupt for UART 0.
//
    IntEnable(INT_UART0);
//
// Enable UART 0.
//
    IntMasterEnable();
```

## 5 Assignment (1.5 bonus points if completed before the deadline, which is specified in Canvas for each assignment)

**Stopwatch**. Create a stop watch. Use the serial terminal as a user interface. It should be possible to set initial time, start, stop, and reset the clock via the keyboard on the serial terminal. The updated time should be displayed on terminal screen. Use a hardware timer to update the time value periodically when the timer is running. You should divide the program into several source les and create a well-defined API for your main program. That is, all low-level code should be implemented as device drivers and its interface should be presented to the main program via descriptive function calls. The minimum requirement is that the clock should be able to handle hours, minutes and seconds.

**Hint**: It might be useful to use the cursor movement ANSI escape sequences to position the cursor at the screen. You are allowed to use the UART example available in C:\ti\TivaWare_C_Series-2.1.4.178\examples\boards\ek-tm4c129exl in this lab.

## 6 Optional Assignments (2 bonus points, if completed before the deadline, which is specified in Canvas for each assignment)

1. **IRQ serial driver**. Replace the *int32_t UARTCharGet (uint32_t ui32Base)* function in the Serial Driver API to use interrupts instead of polling.

**Note:** The bonus points collected after successfully completing the compulsory assignments and optional assignments before the corresponding deadlines will be added to your score in the final exam. These bonus points could help you in improving your final grade in the course.