Laboratory 3: Interrupt and Timer

Objectives

1. Understand the concept of Interrupt
2. Understand the concept of Timer

Interrupt

Device Services: Polling Versus Interrupts

The CPU in a microcontroller or microprocessor is a sequential machine. The control unit sequentially fetches, decodes, and executes instructions from the program memory according to the stored program. This implies that a single CPU can only be executing one instruction at any given time. When it comes to a CPU servicing its peripherals, the program execution needs to be taken to the specific set of instructions that perform the task associated to servicing each device. Take for example the case of serving a keypad.

A keypad can be visualized at its most basic level as an array of switches, one per key, where software gives meaning to each key. The switches are organized in a way that each key depressed yields a different code. For every keystroke, the CPU needs to retrieve the associated key code. The action of retrieving the code of each depressed key and passing them to a program for its interpretation, is what we call servicing the keypad. Like the keypad in this example, the CPU might serve many other peripherals, like a display by passing it the characters or data to be displayed, or a communication channel by receiving or sending characters that make up a message, etc.

source: http://racegrade.com/keypad.html

When it comes to the CPU serving a device, one of two different approaches can be followed: service by polling or service by interrupts. Let's look at each approach with some detail.

Service by Polling

In service by polling, the CPU continuously interrogates or polls the status of the device to determine if service is needed. When the service conditions are identified, the device is served. This action can be exemplified with a hypothetical case of real life where you will act like a polling CPU:

Assume you are given the task of answering a phone to take messages. You don't know when calls will arrive, and you are given a phone that has no ringer, no vibrator, or any other means of knowing that a call has arrived. Your only choice for not missing a single call is by periodically picking-up the phone, placing it to your ear and asking "Hello! Anybody there?" hoping

someone will be in the other side of the line. This would be quite an annoying job, particularly if you had other things to do. However, if you don't want to miss a single call you'll have to put everything else aside and devote yourself to continuously perform the polling sequence: pick-up the phone, bring it to your ear, and hope someone is on the line. Since the line must be available for calls to enter (sorry, no call-waiting service), you have to hang-up and repeat the sequence over and over to catch every incoming call and taking the messages. What a waste of time! Well, that is polling.

Service by Interrupts

When a peripheral is served by interrupts, it sends a signal to the CPU notifying of its need. This signal is called an interrupt request or IRQ. The CPU might be busy performing other tasks, or even in sleep mode if there were no other tasks to perform, and when the interrupt request arrives, if enabled, the CPU suspends the task it might be performing to execute the specific set of instructions needed to serve the device. This event is what we call an interrupt. The set of instructions executed to serve the device upon an IRQ form what is called the interrupt service routine or ISR.

We can bring this interrupting capability to our phone example above.

Let's assume that in this case your phone has a ringer. While expecting to receive incoming calls, now you can rely on the ringer to let you know that a call has arrived. In the mean time, while you wait for calls to arrive you are free to perform other tasks. You could even get a nap if there were nothing else to do. When a call arrives, the ringer sounds and you suspend whatever task you are doing to pick-up the phone, now with the certainty that a caller is in the other end of the line to take his or her message. The ring sound acts like an interrupt request to you. A much more efficient way to take the messages.

Interrupts can be used to serve different tasks. The following are just a few simple examples that illustrate the concept:

‣ A system that toggles an LED when a push-button is depressed. The push-button interface can be configured to trigger an interrupt request to the CPU when the push-button is depressed, having an associated ISR that executes the code that turns the LED on or off.

‣ A message arriving at a communication port can have the port interface configured to trigger an interrupt request to the CPU so that the ISR executes the program that receives, stores, and/or interprets the message.

‣ A voltage monitor in a battery operated system might be interfaced to trigger an interrupt when a low-voltage condition is detected. The ISR might be programmed to save the system state and shut it down or to warn the user about the situation.

Source: Introduction to Embedded Systems Using Microcontrollers and the MSP430 , Jiménez, Manuel, Palomera, Rogelio, Couvertier, Isidoro

Nested vectored interrupt controller (NVIC)

NVIC features

The nested vector interrupt controller NVIC includes the following features:

‣ 82 maskable interrupt channels for STM32F405xx/07xx and STM32F415xx/17xx, and up to 91 maskable interrupt channels for STM32F42xxx and STM32F43xxx (not including the 16 interrupt lines of Cortex ® -M4 with FPU)

‣ 16 programmable priority levels (4 bits of interrupt priority are used)

‣ low-latency exception and interrupt handling

‣ power management control

‣ implementation of system control registers

The NVIC and the processor core interface are closely coupled, which enables low latency interrupt processing and efficient processing of late arriving interrupts. All interrupts including the core exceptions are managed by the NVIC. For more information on exceptions and NVIC programming, refer to programming manual PM0214.

External interrupt/event controller (EXTI)

The external interrupt/event controller consists of up to 23 edge detectors for generating event/interrupt requests. Each input line can be independently configured to select the type (interrupt or event) and the corresponding trigger event (rising or falling or both). Each line can also masked independently. A pending register maintains the status line of the interrupt requests.

Timer

Base Timer Structure

In its most basic form, a timer is a counter driven by a periodic clock signal. Its operation can be summarized as follows: Starting from an arbitrary value, typically zero, the counter is incremented every time the clock signal makes a transition. Each clock transition is called a "clock event". The counter keeps track of the number of clock events. Notice that only a single type of transition, either low-to-high or high-to-low (not both), will create an event. If the clock were a periodic signal of known frequency f , then the number of events k in the counter would indicate the time kT elapsed between event 0 and the current event , being T = 1/ f the period of the clock signal. The name "timer" stems from this characteristic.

To enhance the capabilities of a timer, further blocks are typically added to the basic counter. The conglomerate of the counter plus the additional blocks enhancing the structure form the architecture of a particular timer.

Figure 7.11 illustrates the basic structure of a timer unit. Its fundamental component include the following:

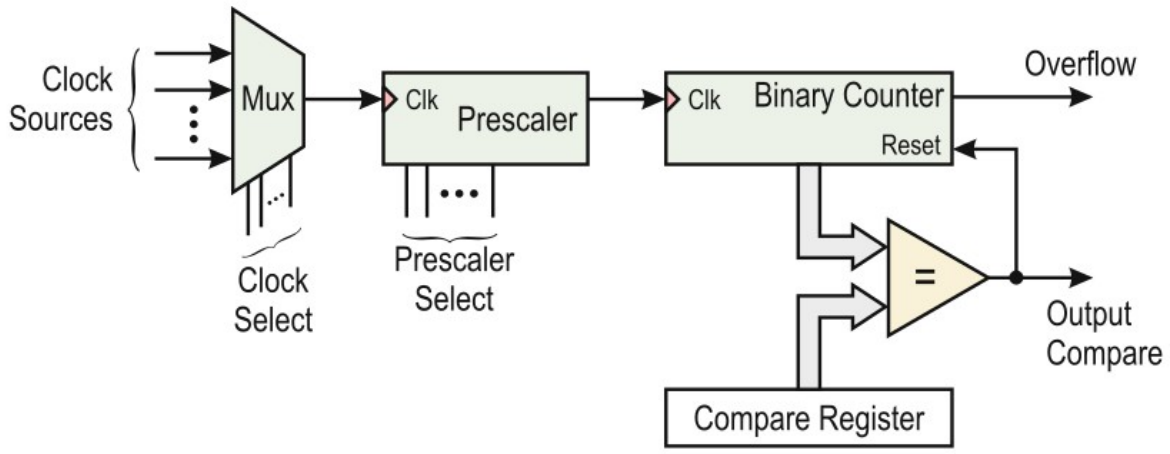

Fig. 7.11 Components of a base timer

- Some form of a clock selector (Mux) allowing for choosing one from multiple clock sources.
- A prescaler, that allows for pre-dividing the input clock frequency reaching the counter.
- An n-bit binary counter, providing the basic counting ability.
- An n-bit compare register, that allows for limiting the maximum value the counter can reach.
- A hardware comparator that allows for knowing when the binary counter reaches the value stored in the compare register. Note how a match of these values resets the binary counter.

The STM32 devices are built-in with various types of timers, with the following features for each:

▸ General-purpose timers are used in any application for output compare (timing and delay generation), one-pulse mode, input capture (for external signal frequency measurement), sensor interface (encoder, hall sensor)...

▸ Advanced timers : these timers have the most features. In addition to general purpose functions, they include several features related to motor control and digital power conversion applications: three complementary signals with deadtime insertion, emergency shut-down input.

▸ One or two channel timers : used as general-purpose timers with a limited number of channels.

▸ One or two channel timers with complementary output : same as previous type, but having a deadtime generator on one channel. This allows having complementary signals with a time base independent from the advanced timers.

▸ Basic timers have no input/outputs and are used either as timebase timers or for triggering the DAC peripheral.

▸ Low-power timers are simpler than general purpose timers and their advantage is the ability to continue working in low-power modes and generate a wake-up event.

▸ High-resolution timers are specialized timer peripherals designed to drive power conversion in lighting and power source applications. It is however also usable in other fields that require very fine timing resolution. AN4885 and AN4449 are practical examples of high-resolution timer use.

Lab Exercises

1. Create a new STM32 Project on STM32CubeIDE to blink (on/off) green LED with changeable period by pushing USER push-button from periods of 0.2 sec. → 1 sec. → 5 sec. and back to 0.2 sec. (using External interrupt/event controller (EXTI) is required)

2. Create a new STM32 Project on STM32CubeIDE to generate clock signal from any output pins with 100 microsecond period and use an oscilloscope to monitor the signal.

3. Create a new STM32 Project on STM32CubeIDE to blink (on/off) green LED (LD5) with 500 milliseconds period and green LED (LD4) with 490.5 milliseconds period and display the number of both red and green LED blinkings every 999.9 milliseconds on Serial terminal via UART. (using interrupt timer)

4. Create a new STM32 Project to echo back (transmit the receive data) the communication data from UART peripheral (USART2) interface similar to Lab 2 but use an interrupt for receiving the data instead.