4.10 Timers

Using timers for accurate timing

EE302 – Real time and embedded systems

1

Overview

- Aims
  - To understand and use timer peripherals to aid real time applications
- Learning outcomes – you should be able to…
  - Explain the operation of a timer device (based on the PIC16 family integrated timers)
  - Configure a timer device (prescaler, postscaler, etc) for a particular timeout or period
  - Calculate the maximum, minimum timeouts possible with a particular timeout
  - Write pseudocode that uses timer based timing

2

Programmable timer peripheral

3

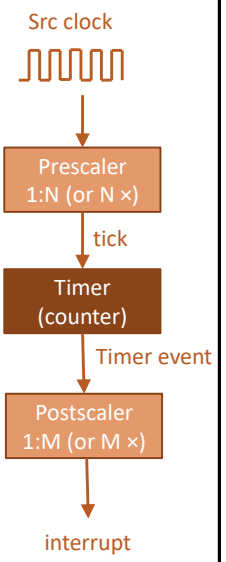
- A timer is a hardware device which can be used to accurately mark time to schedule events in the future or time the duration of events. It is essentially a counter (often just an 8-bit or 16-bit counter)
- Available as standalone devices or commonly integrated with microcontrollers
- A timer marks time by incrementing or decrementing a counter register once every tick
- When the timer reaches a specific value it can generate an output signal (and perhaps an interrupt)
- Code can use timer output signals to generate regular/periodic occurrences such as sampling an ADC at perfectly regular intervals or to measure a specific precise delay

3

Key timer terms

4

- (source) Clock
  - A timer can usually be configured to use one of several different clock sources for its underlying timing input, E.g. PIC allows instruction clock, external clock, etc.
- Tick
  - The timer/counter value increments by one on each tick.
- Pre-scaler
  - Sits between the source clock signal and the timer.
  - A prescaler value of N means a timer tick takes place every N clock cycles instead of every clock cycle
- Post-scaler
  - A post-scaler sits between the timer reset and the timer event signalling functionality
  - A post-scaler value of M, means a timer event will be signalled only after the timer resets M times instead of each time it resets



4

Prescaler and postscaler in more detail

5

- Prescaler
  - A timer marks time by incrementing or decrementing a counter register once every tick
  - If there is no prescaler (or the prescaling factor is 1:1 or 1x), then there is one tick per clock period of the source clock signal
  - If a prescaler is used with a factor bigger than 1x, then the tick does not occur on each source clock period but on specified multiple of the clock period. E.g. 1:4 or 4x would mean one tick every 4 source clock periods
  - A prescaler allows a timer (which usually has a limited counter) to measure longer durations than would otherwise be possible

5

Prescaler and postscaler in more detail

6

- Postscaler
  - Normally when the timer reaches a specific value a timer event occurs
  - When there is no postscaler or the postscaling factor is 1x, the timer event directly results in a timeout signal or interrupt
  - When the postscaling factor is more than 1x, then the specified number of timer events must occur before a timeout or interrupt is signalled, e.g. 8x would mean 8 timer events must occur before a timeout is signalled
  - Like a prescaler, a postscaler allows a timer (which usually has a limited counter) to measure longer durations than would otherwise be possible

6

Timer events, timeouts, and interrupts

- A **timer event** occurs when the **timer counter reaches a configured value** or the **timer counter register rolls over** (e.g. an 8-bit timer register rollover is from 0xFF to 0x00)
  - Whether it is a configured value or rollover that we use usually depends on the specific timer
- When a timer event occurs
  - it may be configured to automatically reset the timer which allows the timer to generate events with a regular period
  - If there is no postscaler or the postscaling factor is 1, then a **timeout** is generated immediately. Otherwise a timeout is only generated after M timer events, where M is postscaling factor.
- When a timeout occurs, and depending on the timer, it may be configured to …
  - generate an interrupt – allows a programme to take action immediately (important for real time)
  - Generate a digital signal that may initiate some action on an internal peripheral (on a microcontroller) or external peripheral

Programming and using a timer

- Configure the initial timer register value and any prescale/postscale multipliers to use
- Sometimes (if supported): configure the target period value at which the timer should be reset and a timer event generated
- Often: enable interrupts and write an interrupt handler that runs when the timeout occurs
  - The interrupt handler may also have to set/reset the timer register based on when you want the next timeout to occur
- As an alternative to timer interrupts, polling the timer is possible but not that common
  - In this case, there is a need to get the timeout (timer interrupt) flag to see if a timeout has occurred or possibly to inspect the timer register value

Timer/counter general operation

- A timer is based around a counter register typically limited to B bits
  - Therefore the minimum timer register/counter value is 0 and the maximum value is 2^B-1
- Typically
  - Timer/counter increments by one each "tick" and a timer event occurs when the counter rolls over
  - For a B-bit timer, incrementing from 2^B-1 rolls over to 0
    - e.g. For an 8 bit timer/counter, incrementing 0xFF results in 0
- Therefore, to make the timer roll-over occur $N_{rollover}$ ticks from now
  - First, ensure $N_{rollover} <= 2^B$
  - Then set the timer register to $2^B - N_{rollover}$
  - E.g. to have an 8-bit timer rollover occur in 5 ticks time, set the timer to 256 – 5 = 251. On next 5 ticks the timer will be 252, 253, 254, 255, 0 (= rollover = timer event!)

How to achieve periodic timing (software only)

- Assume we want a periodic timer event, every N_{period} ticks
  - Useful for PWM, ADC sampling, communications line sampling, etc.
- If there is no hardware reset available
  - Estimate the number of timer ticks required to call the interrupt code and reset the timer, N_{reset}
  - Set timer to rollover to $N_{period} - N_{reset}$ ticks
  - When timer rollover and an interrupt occurs, your interrupt handler code must
    - reset the timer to $N_{period} - N_{reset}$ ticks again
    - Anything else (quick) that you need to do in response to the timer interrupt
  - Main disadvantage: code must be written for this and the value set in the timer must be adjusted to allow for the time consumed by servicing the interrupt and resetting the timer

How to achieve periodic timing (hardware solution)

- If the particular timer has hardware compare and reset functionality
  - set the timer register to 0
  - Set period compare register to N_{period} - 1 ticks and configure periodic reset
  - The Timer increments on each tick as normal and is compared to period value on each tick. When it equals the value of (N_{period}) less 1, the timer is automatically reset to 0 on the next tick and an interrupt generated. (This why we set the period register to $N_{period} - 1$)
  - Your interrupt handler does not need to reset the timer and can focus on just doing anything (quick) that you need to do in response to the timer interrupt
  - Main advantages: no reset code required, and no compensation for time to reset required

Main calculations needed for timers

- B-bit timer => max num increments before expiry = 2^B
- Timer rollover/period match
  - With rollover: To expire in N ticks, start at 2^B-N
  - Timer with period register and period reset: To expire in N ticks, set timer to 0 and set period register (PR) to N-1.
- Prescaler/postscaler
  - Prescaler – defines ratio of timer increments to source clock cycles

    e.g. 1:4 or 4x means 4 source clock cycles required for every 1 timer increment, i.e. $T_{INC} = 4*T_{SRC}$
  - Post-scaler – defines ratio of timer interrupts to timer rollover or period match events

    e.g. 1:2 or 2x means 2 timer expiry/period match events required for every 1 timeout interrupt, i.e. $T_{INTR} = 2*T_{EXP}$

Main calculations needed for timers

13

- Calculate N, prescale, and postscale given desired timeout duration ($T_{INTR\text{-}desired}$)
- First determine if any pre/post-scale required
  - desiredScale = $T_{INTR} / (T_{SRC} * 2^B)$
  - If desiredScale < 1, then **N = round (T_{INTR} / T_{SRC}), prescale = postscale = 1**
  - If desiredScale >= 1, then
    - To make the timer resolution as good as possible, choose from available prescale and postscale values so that **totalScale = prescale * postscale** is the smallest integer which is **>= desiredScale**
    - Then calculate **N = round ($T_{INTR} / [T_{SRC} * prescale * postscale]$)**

13

Main calculations needed for timers

14

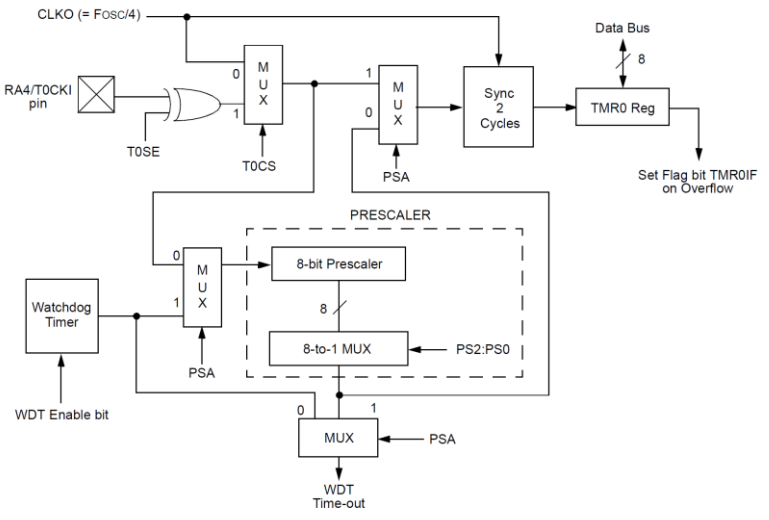
- It is common that limited timer resolution prevents us from achieving exactly the timeout or period that we want (although we can get close)
  - The minimum resolution that a timer can resolve is the effective duration of one tick (including both prescaling and postscaling factors)
- Calculate the actual timeout duration ($T_{INTR\text{-}calculated}$) given N (rollover or period), prescale, and postscale
  - $T_{EXP} = T_{SRC} * prescale * N$
  - $T_{INTR} = T_{EXP} * postscale$
    $= T_{SRC} * prescale * N * postscale$
- The percentage error in timeout is $100 * T_{INTR\text{-}calculated} / T_{INTR\text{-}desired}$

14

PIC16xxx Timers

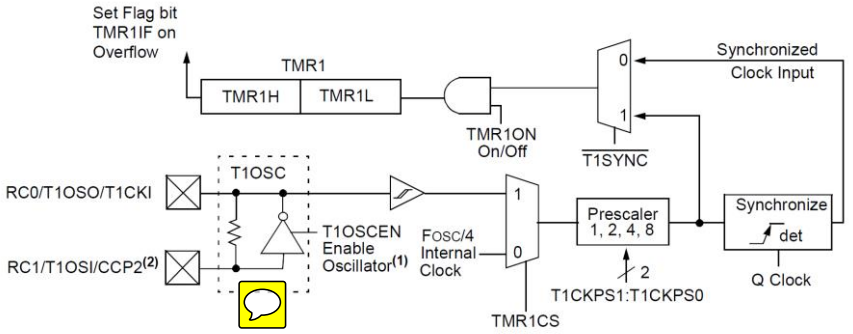
- **Timer0** is an 8 bit timer/counter with a **prescaler** that supports multipliers of 1, 2, 4, 8, 16, 32, 64, 128, 256x.
  - A Timer0 interrupt (TMR0IF) may be generated on rollover from 0xFF to 0x00.
- **Timer1** is a 16 bit (2 register) timer/counter with a prescaler that supports 1, 2, 4, 8x.
  - A Timer1 interrupt (TMR1IF) may be generated on rollover from 0xFFFF to 0x0000
  - Alternatively, to support periodic events, a period value may be specified via the two Capture/Compare/PWM (CCP) devices. When Timer1 reaches the period value, the CCP1 module can reset the timer and generate an interrupt (CCP1IF), whereas the CCP2 module can reset the timer and generate an interrupt (CCP2IF) and automatically start an A/D conversion!
- **Timer2** is an 8 bit timer designed for periodic timing. It has a prescaler with 1, 4, 16x and **postscaler** with 1,2,3,…,16x.
  - An 8 bit period register (PR) dictates the (periodic) value at which Timer2 will reset back to zero. If the postscaler value is M, then after M Timer2 resets, the Timer2 interrupt (TMR2IF) will be generated.

Timer0



Note: T0CS, T0SE, PSA, PS2:PS0 are (OPTION_REG<5:0>).

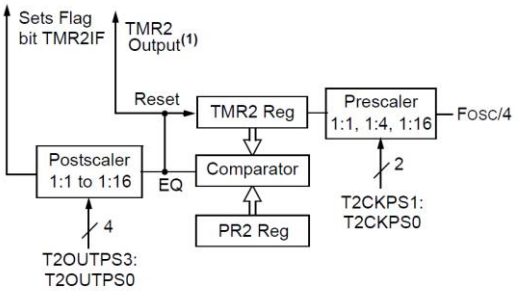
Timer1



Note 1: When the T1OSCEN bit is cleared, the inverter is turned off. This eliminates power drain.

17

Timer2



Note 1: TMR2 register output can be software selected by the SSP module as a baud clock.

18

Self test question

19

Assume F_{OSC} = 4MHz, $T_{CYC} = 4 * T_{OSC}$ and the source clock is T_{CYC}

- *$T_{OSC}=1/F_{OSC}$ and $T_{CYC} = 4 * T_{OSC}$*

Q. With Timer0 (8 bit) using T_{CYC}

- *What is the maximum duration that can be timed using 1x, 8x, and 256x prescaling?*
- *How would you time a duration of 1 ms?*
- *How would you time a duration of 1001 microsecs?*

24 November 2020

19

More self test questions

20

Q. With Timer1 (16 bit) using T_{CYC}, F_{OSC} = 8 MHz

- *What is the maximum duration that can be timed using 1x, or 8x prescaling?*
- *How would you time a duration of 1001 microsecs?*

Q. With Timer2 (8 bit) using T_{CYC}, F_{OSC} = 20 MHz

- *What is the max duration that can be timed using 1x, 16x prescaling and 1x, 16x postscaling (all combinations)?*

- *Is there any practical difference between using a prescaler and using a postscaler?*

24 November 2020

20

Exam question: Q4b, 2011-2012

On the PIC 16F877 microcontroller, Timer0 is an incrementing 8-bit timer that generates an interrupt on rollover (from 0xFF to 0x00). Timer0 also incorporates a prescaler which can be configured such that the timer increments on every 1, 2, 4, 8, 16, 32, 64, 128, or 256 instruction cycles.

Consider a system using the PIC 16F877 with an oscillator frequency of 4 MHz. The system must use its internal ADC to sample analogue input at a rate of 2000 samples/second.

1. *For this system, what is the maximum timeout period supported by Timer0 with the minimum and maximum prescaling factors applied?*
2. *Choose an appropriate prescaling factor and Timer0 start value to achieve a timeout equal to the required sampling period of this system.*

VARIATION: Same question as above but assume that it takes 2 microseconds to reset the timer in the interrupt handler

VARIATION: Same question as above but use Timer2 with prescaler AND postscaler

Self test question

VARIATION: Same question as previous worked exam question, but assume that it takes 2 microseconds to reset the timer in the interrupt handler

VARIATION: Same question as previous worked exam question, but use Timer2 with prescaler AND postscaler

Before looking at code/pseudocode to handle timers in more detail we need to quickly introduce interrupts….

Interrupt driven I/O

25

I/O devices “interrupt” CPU via an “interrupt request” (IRQ) line/pin as follows…

1. When an I/O device wants service it causes a transition on the CPU’s IRQ pin (or internal IRQ line when it is an internal peripheral)
2. CPU immediately(*) pauses main program execution
3. CPU executes Interrupt Service Routine (ISR) and ISR code services the device (e.g. by reading or writing data)
4. When ISR completes, CPU resumes program execution

Interrupt Service Routine - ISR

26

- ISR is the code that is executed to service the device which caused the interrupt
- CPU saves PC (and possibly SP) register
  - ISR must explicitly save and later restore any other registers it needs to use (called context switching) – C compiler handles this for us
- Guidelines
  - ISRs must be kept as short as possible, to minimise delays to main program and other interrupts
  - Care needs to be taken when ISR and main program share some variables (e.g. when main program and ISR both access the same data buffer)
    - Main part of program may need to disable (or mask) interrupts when accessing variables shared with ISR
  - Usually disable interrupts while executing an ISR (but it is possible to nest interrupts on some processors)

Interrupt driven I/O - analogy

- You need a colleague to do something for you (e.g. get some information)
- Go ask them to do the job (this is the I/O request)
- Unlike polling (where you wait for job to be done or else check back regularly), instead you ask your colleague to contact you when job done. This allows you to immediately continue with other work.
- Later when your colleague is done, they give you a call (this is the interrupt)
- You save your current work, perhaps marking the page in a book, or whatever. (This is pausing the main program)
- You go back to your colleagues office to pick up the job output (this is the work of the ISR and servicing the interrupt)
- When you get back to your office, you pick up your previous work where you left off (this is resuming the main program)

Example – interrupt driven timing for superloop


```
// NOTE: Use a timer to guarantee periodic superloop timing
global gTimerExpired

main()…
setup()…
loop()
  while gTimerExpired is FALSE, do nothing // i.e. wait
  set gTimerExpired = FALSE // ready to wait again
  doTimeConsumingTask()
  // no extra delay required – handled by waiting for timer

doTimeConsumingTask()…

isr() // polled interrupt service routine
  if (timer interrupt)
    // assume timer not setup for periodic reset so we must
    // explicitly reset it here
    set timer = SUPERLOOP_TICKS
    set gTimerExpired = TRUE
```

Example – timer interrupt driven background task


```
main()…
setup()…
loop()
  someTask()
  someOtherTask()
  doTimeConsumingTask()

doTimeConsumingTask()…

isr() // polled interrupt service routine
  if (timer interrupt)
    // assume timer not setup for periodic reset so we must
    // explicitly reset it here
    set timer = REQUIRED_TICKS
    doSomethingQuickInResponseToTimer()
```


Exam question: Q4b, 2011-2012 – contd.

System outline code is as shown.

The polled interrupt routine must use the timer interrupt (TMR0IF) to determine when to start an ADC conversion and use the end of ADC conversion interrupt (ADIF) to determine when processSignal should do its processing.

Write the pseudocode implementation of this polled interrupt routine, setting variables needed by the superloop functions as necessary. Ensure the timing is as specified previously. Assume that the Timer0 register is TMR0, that ADC conversion is started by setting the ADGO bit, and that we are only interested in 8-bit ADC samples from register ADRESH.

```
global gSampleToProcess = FALSE
global gSample = 0

loop():
  processSignal()
  serviceUSB()

processSignal():
  if gSampleToProcess is TRUE:
    // ready for next sample
    set gSampleToProcess = FALSE
    … // do the processing here

serviceUSB():
  … // details unimportant
```

31