Use eight timers with PIC16Fxxx microcontrollers

The need for timing in embedded programming often exceeds the small number of available hardware timers in microcontrollers. For example, the Microchip (www.microchip.com) PIC16F84A has one timer, but you can create as many as eight timers with the Timers8.inc assembly code, which you can download from the online version of this Design Idea at www.edn.com/100218dia.

Often, you need a timer while waiting for some expected lapse to expire, which blocks your program until that time has elapsed. To accomplish that task, you can use a simple routine, such as Wait_onK, .30, meaning that Timer 2 counts 300 msec. The time value is .30 ticks of 10 msec each. In this way, you can debounce input data (Listing 1).

Using Microchip’s MPLab assember, you can’t guess whether a parameter is a constant value or a variable, so you need two macros, one for each kind of circumstance. In the following explanation, use K if time is a constant and use V for variables, such as Wait_onK or Wait_onV. The same library for the TBM (timebase module) on the MCHC9S08GP32 microcontroller from Freescale Semiconductor (www.freescale.com) uses only one macro, Wait_on, under the CodeWarrior assember. The function deals with both constants and variables.

The “tmr” always stands for a constant from zero to seven. It designates the number of the timer you apply to a situation. You must always multiply time by 10, so .1 is 10 msec. You can therefore use it with times from 10 to 2560 msec. Precision is plus or minus one tick. If you need to trigger an event with a 20-msec timer, you may end with 10 msec. So use 30 msec to be safe. At the high end of the scale, you will have 2560 ± 1 msec, which is acceptable.

Timers8.inc programs the TMR0 on the PIC16F84A. You can extend beyond eight timers or use 16-bit variables, but remember that the PIC16F84A has only 68 bytes of RAM.

On some occasions, you need to start a timer but don’t need to block your program to time-out; in this case, use the SetimerK routine. You start Timer 2 to last 200 msec; doing so does not block your program. Whenever you need to know the status of your timers, test them using the TimeOut macro, TimeOut 2, Two_done, meaning that if Timer 2 has expired, you will go to the “Two_done” label. Any of your main code labels will fit. Otherwise, your program will continue executing the next instruction in sequence.

Setimer comes in two versions. SetimerK 2, .20 sets Timer 2 to count 200 msec, using constant time .20, and SetimerV 5, var sets Timer 5 to count, for example, 300 msec, using variable time var, which you should have previously loaded with .30.

You may need to employ timers in ISRs (interrupt-service routines)—for example, to debounce the interrupt pin. This situation is awkward because the routine to serve the external INT pin runs with general interrupts disabled, as usually happens in the PIC16F84A, but timer routines require you to enable interrupts. This microcontroller architecture makes it difficult to enable interrupts in ISRs. You may, however, use either ISRWait_onK or ISRWait_onV to accomplish your purpose, as in ISRWait_onK 7, 3.

This approach works in a similar way.
to its twin, the Wait_on macro, except that you can use the approach in any ISR—a nice added value for such an inexpensive microcontroller. Use it with care, however. Interrupt latency increases because you block the program in an ISR for several milliseconds with global interrupt disabled. If you choose to debounce your interrupt signal using programmed delays, you will probably encounter the same problem. If you use a specific timer number in the main program, don’t use it in the ISR.

To use the Timers8.inc library, you must include the library file and define some variables outside the timer’s code. To find the exact place to include the library and define variables, refer to the sample code. Look for « « TMR0 « «, which overemphasizes portions of the code. In particular, inspect the lines “CBLOCK” and “INCLUDE <timers8.inc>”.

Follow this plan in your program: Use the macro Init8Timers to activate the hardware and set up the eight software timers. This macro defines eight variables, from Timer 0 to Timer 7, each using one unsigned byte. Each timer ticks once every 10 msec, covering a range of 10 to 2560 msec. You need not worry about these variables, though, because the macros will handle them. A 1-byte variable, TimerFlags, has bits that represent the ready state of timers zero through seven. You need not deal with this internal variable.

To initialize a timer from zero to seven, use the Setimer macro, as in SetimerK 2, .20 (set Timer 2 to count 200 msec using a constant time of .20) or SetimerV 5, var (set Timer 5 to count 300 msec using a variable time of .30, which you previously stored in var). Setimer macros are not self-blocking; they initialize the software timers and continue. This feature comes in handy when you plan to loop, asking for several events to time out and do not need one of them to block you.

To test whether one timer has expired, use the TimeOut macro after Setimer; TimeOut 2, Two_on. If Timer 2 has expired, go to Two_on; otherwise, execute the next instruction in the sequence. TimeOut combines these macros in one: Wait_onK 2, .30. Set Timer 2 to count 300 msec using a constant time of .30 and block until time-out. Alternatively, using Wait_onK 5, var, set Timer 5 to count 300 msec using a value of .30, which you previously stored in var. Wait_on macros are self-blocking; they initialize the software timers and wait until time elapses. You can use ISRWait_on in ISRs: ISRWait_onK 6, .35. Set Timer 6 to count 350 msec using a constant time of .35 and then block.

Alternatively, you can use ISRWait_onV 5, var. Set Timer 5 to count 2000 msec using a value of .200, which you previously stored in var. ISRWait_on macros are self-blocking. You can use them in ISRs to initialize the software timers and wait until time elapses. You must include an interrupt handler; see the IntHandler in Listing 2, which you can download from the online version of this Design Idea at www.edn.com/100218dia. The library also includes TMR0ISR, Timer 0’s ISR, and the UpdTmr (update-timer) internal macro.

Each timer has a status bit, which helps when your variables have 16 bits, 24 bits, or more. When the driver detects that one multibyte timer variable reaches zero, it signals this situation by setting the timer’s status bit. That action spares you several instructions when you need to later decide whether the timer is zero. You can also use these bits as semaphores. You may start a timer with Setimer, and the hardware may interrupt you in the middle of the start-up to update your data structures, causing lots of problems. The software in this code, however, avoids touching or updating variables if the status bit is 1. Setimer begins raising the status bit and then loads the variables. If Timer 0 interrupts, it does not interfere with your data because it skips the updating process if the status bit is on. When Setimer is done, it clears the status bit, and Timer 0’s ISR will begin to update whenever a tick arrives.

This code doesn’t stop a timer before a time-out because the need never arises. If Setimer uses zero as a value for the time, it lasts for 256 10-msec ticks. If you need a 1-msec tick, you can load Timer 0 with –0.125 instead of –0.39 and use a prescaler of 8 (b’00000010’ in OPTION_REG) instead of 256 (b 0000111), which are the values this code uses. The exact time is 125×8=1000 μsec (1 msec). This approach provides a range of 1 to 256 msec.

### Tilt/fall detector has staggered thresholds

Marián Štofka, Slovak University of Technology, Bratislava, Slovakia

Measurement-and-control applications may require action based on two distinct voltage levels. Crossing a threshold can produce a warning indication, whereas reaching a higher threshold may initiate emergency action, such as a system shutdown. In a fall-detector application, an apparent decrease in gravity below a lower threshold might be a controlled displacement, but a further decrease below a second threshold might indicate an uncontrolled fall.

The circuit in Figure 1 uses a voltage divider to generate two reference voltages. Comparators and Schmitt-trigger-input NAND gates let you create two digital signals based on using reference voltages $V_{REFA}$ and $V_{REFB}$. The sample circuit drives two LEDs, but...
you can use the digital signals to drive transistors or relays, as well.

The voltage divider comprising \( R_s \), \( R_A \), and \( R_B \) sets the voltages for comparing the Z-axis output of an Analog Devices (www.analog.com) ADXL335 accelerometer (Reference 1). The higher reference voltage, \( V_{\text{REFA}} \), corresponds to the lower-threshold tilt angle, where \( \alpha_{LB} = 45^\circ \). The lower reference, with respect to the midvoltage supply minus \( V_{\text{REFB}} \), corresponds to the upper-threshold tilt angle, where \( \alpha_{UB} = 60^\circ \). If you choose a value of 100 kΩ for \( R_s \), then you can calculate \( R_A + R_B \):

\[
\frac{R_s}{R_A + R_B} = \frac{V_G}{V_G \times \cos \alpha_{TA}} - 1.
\]

The Z-axis voltage, \( V_Z \), is 300 mV, occurs when the accelerometer’s Z axis is oriented vertically. From the obtained value of \( R_A + R_B \), you can calculate \( R_B \):

\[
R_B = \frac{\cos \alpha_{TB}}{\cos \alpha_{TA}} \times (R_A + R_B).
\]

Based on the chosen values of the tilt angles, \( R_B = (R_A + R_B) / \sqrt{2} \). You can then solve for \( R_A \) from the known values of \( R_A + R_B \) as well as \( R_B \).

The AD8609 op amp’s input-bias current causes errors, but these errors are negligible because the input-bias current at room temperature is just 1 pA. The AD8609’s input offset voltage, which is typically 50 μV, also causes errors, which are negligible as well (Reference 2). The signals at the outputs of comparators IC\(_{A1} \), IC\(_{B1} \), and IC\(_{C1D} \) are ORed in NAND gates IC\(_{A2} \) and IC\(_{C2D} \), respectively. NAND gate IC\(_{C3} \) serves as an inverter, and the output of IC\(_{D+} \) is the logic output of a window comparator in which logic low appears only when the Z-axis output voltage is between \( V_{\text{REFA}} \) and \( V_{\text{REFB}} \), referenced to supply midvoltage \( V \).

Grouping the comparators into IC\(_{A1} \), IC\(_{B1} \), and IC\(_{C1D} \) pairs ensures independent detection on whether the Z axis is 0 or 180° in the vertical orientation. LED\(_1 \) and LED\(_2 \) illuminate successively upon slowly tilting the Z axis by 45 and 60° (Reference 3). Similar action occurs when you orient the Z axis steadily vertically while moving downward. LED\(_1 \)’s brightness is turned on at an apparent decrease of gravity to \( g/\sqrt{2} \). LED\(_2 \) dims, and LED\(_3 \) simultaneously illuminates when the vertical acceleration is equal to or lower than \( g/2 \). The operation of the detector is ratiometric and is therefore virtually insensitive to supply-voltage variations.

**REFERENCES**


Figure 1 An accelerometer’s Z-axis output, compared with two reference voltages, can generate two digital outputs.
Many applications, such as medical therapies, magnetic stirrers, and induction heating, call for a rotating magnetic field, which you can generate by attaching multiple permanent magnets to a dc motor. This technique involves problems, including noise and the need to maintain the moving parts. This Design Idea describes how you can instead use a microcontroller and a full-bridge driver to generate variable magnetic fields without mechanical elements. The approach requires no maintenance, does not wear out, and provides high-precision speed control. It does, however, require large cores to achieve powerful magnetic excitation.

You can excite a stationary magnetic coil with an ac current, which induces a north pole and a south pole that change at the frequency of the signal excitation. You can increase the number of poles by implementing a configuration with more magnetic coils. Figure 1 shows a practical arrangement of the coils and the typical excitation waveforms. Note that the terminals of each pair of coils connect in series opposition to always obtain magnetic fields with different polarity.

Multiple ICs can drive inductive loads. This circuit uses an L6204 dual full-bridge driver from STMicroelectronics (www.st.com). Each bridge has four power-DMOS transistors with on-resistances of 1.2Ω. A PIC16F628 microcontroller from Microchip (www.microchip.com) controls the switches of the dual-bridge driver (Figure 2). Typical waveforms show how each circuit is excited (Figure 3).

To ensure the correct driving of high-side drivers, the circuit supplies a voltage higher than the supply voltage at IC’s Pin 20. External capacitors C1 and C2, and diodes D1 and D2, use a charge-pump method to produce this voltage. You can independently control the four half-bridges by means of the IN1, IN2, IN3, IN4, ENABLE1, and ENABLE2 inputs.

The microcontroller timer’s interrupt generates the IN1 to IN4 waveforms with high precision. Using a 10-MHz oscillator crystal and fixing the postscaler to eight, the microcon-
controller’s counter increments every 3.2 μsec: \(1/(10\,\text{MHz/4}\,\text{instructions})/\text{eight}\). Taking into account that the interruptions generate when the counter overflows and the maximum count is as high as 65,535, or 16 bits, you can program the interruptions at 3.2 μsec and 210 msec: \(3.2\times65,535\).

From this wide range of interruptions, the firmware lets the user select the precharge within a small subrange of frequencies divided into 10 levels, meaning that you must vary the interruption from 49.89 to 60.45 μsec, a good range for this application. The new frequency of the interruption has a simple calculation that includes the level; the maximum frequency; and the separation between levels, which is a constant value that the operations include. You can download listings 1 and 2, which have complete C source code, from the Web version of this Design Idea at www.edn.com/100218dib.

THE FIRMWARE LETS THE USER SELECT THE PRECHARGE WITHIN A SMALL SUBRANGE OF FREQUENCIES.

Voltage reference stabilizes current sink

Sused Emmanuel, Emmanuel’s Controls, Auckland, New Zealand

Analog circuits for long-term testing of passive components, such as 0.1%-tolerant resistors or high-intensity white LEDs, often require a constant current. Using two op-amps and a voltage reference, you can develop a circuit that provides a constant-current sink with a variable setting of 0 mA to 0.99A. The circuit in Figure 1 sinks a stable current through the load. The load current is insensitive to power-supply-voltage variations. IC₁ is a voltage reference that gives a stable 5V dc. It requires 500 μA of current from the power supply. IC₂ is a National Semiconductor (www.national.com) LM324 quad op-amp. Voltage follower IC₂A buffers the reference voltage from the rest of the circuit, which increases stability.

Resistor \(R₂\) and potentiometer \(R₃\) form a variable voltage divider that reduces the 5V reference voltage to a value between 0 and 3.26V. Unity-gain amplifier IC₂D drives the base of \(Q₁\), a Darlington power transistor that has a current gain of 750, through \(R₃\). \(R₄\) and \(C₅\) form a lowpass filter that prevents oscillation. You can drive \(Q₁\) with a small base current. \(C₄\) connects between the collector and the base of \(Q₁\), adding further stability. Operating as an emitter follower, \(Q₁\) can drive an active or a passive load, such as a resistor or a high-brightness LED. \(Q₁\)'s emitter connects to a 3.3V, 5W grounded power resistor. The voltage at IC₂D’s Pin 14 sets the voltage across \(R₄\), which fixes \(Q₁\)'s emitter current. Because of \(Q₁\)'s high gain, the current in the load is effectively \(Q₁\)'s emitter current.
Figure 1 A voltage reference and two op amps provide a stable voltage to $R_5$, which provides a stable load current.