University of Windsor

ELECTRICAL AND COMPUTER ENGINEERING
ELEC3270: MICROPROCESSORS
ASSIGNMENT 9: TIMERS AND INTERRUPTS

OBJECTIVE: To code interrupt service routines to effectively use MCU timers

SECTIONS:

A.	Coding an Interrupt Service Routine for Input Capture (IC)	(40%)
B.	Output Compare (OC)	(30%)
C.	Free Running Timer Overflow	(30%)

A. CODING AN INTERRUPT SERVICE ROUTINE FOR INPUT CAPTURE (IC)

From assignment 5 we have seen that delays can be introduced by having the CPU perform meaningless operations for a specific amount of time. These delays are in no way accurate as the CPU may be forced to handle an interrupt (unexpected situation) at some point. An interrupt is normally generated by a source outside of the CPU, which in the case of a MCU, can be an external circuit as well as the internal MCU systems (timers, communication systems, etc.). Interrupts generated internally in the CPU are generally referred to as exceptions; we will study these in later assignments. The CPU processes interrupts signals by running user specific code to handle them. It does this by first completing the current instruction that is in progress. Then it saves the state of the CPU (PC, register values) on the stack, and it jumps to the interrupt handler which is often referred to as the interrupt service routine (ISR). The ISR performs whatever operations are needed to service the interrupt, and once complete, a special instruction is run which restores the saved registers and PC (which were stored in the stack earlier) allowing the CPU to return to the running process prior to the interrupt signal. The time for an ISR to start and finish can vary, so we can't rely on instruction delays alone for generating accurate timing events as many things happening inside and outside the CPU can affect its overall cycle count. To solve this accurate timing problem, the MCU includes special timing hardware which results in an acceptable level of timing precision. The whole timing system is derived from the free running timer (FRT). The FRT is an internal clock in the MCU which operates similar to a wrist watch; we can set alarms or record the times in which other events occur.
For example, let's consider a 24-hour clock as it is linear throughout the day. A day begins at 00:00:00, and as every second of time passes, the clock increases by 1 second. The seconds go from 0 to 59, at which point the seconds "overflow"; it goes back to zero, and then the minutes increase, again from 0 to 59 where, just like the seconds, an overflow occurs resetting the minutes back to zero while increasing the hour, in this case from 0 to 23. After that, the clock wraps back to 00:00:00; yet another overflow. This particular overflow signifies that a new day has begun, and in some cases this isn't all that important as humans generally schedule themselves daily. For example, if someone wakes up at the same time every day and goes to sleep at the same time as well, knowing the actual day isn't all that important. The FRT is an analogy of this. It begins at 0 when the MCU is powered on, and increments every E-clock cycle, and it resets back to zero after it reaches 65535, or FFFF hexadecimal (it is a 16-bit value). If the MCU E-clock is 2 MHz, this overflow will happen about 30.5 times per second. If that amount of time is not preferable for our application, we can change the E-clock to something different, or we may not even be concerned when it overflows back to zero (just like someone with a regular sleeping schedule). But in case we are, there should be an efficient way of knowing when it happens. Polling, or always examining said value constantly, can not only be a burden on the CPU, but there is a good chance we could miss it switching back to 0 if we are busy doing something else such as examining some other information or performing an ISR. A good way for humans to know when a new day begins is to set an alarm for 00:00:00 as opposed to constantly looking at the time on the clock. The designers of the MCU used the same strategy to avoid polling; their solution was to have special hardware to do the checking, but then tell us, if we want, by

using interrupts. When the hardware detects an overflow on the FRT, we can choose to generate an interrupt thereby allowing the CPU to run an ISR so that we can know when this happens, and then do something about it. In general, we enable interrupts by setting interrupt "mask" bits to 1 in our MCU registers; using a 0 turns them off. When a mask is set, the CPU will pause the current program, then go and run the intended ISR. There are a number of hardware devices inside the MCU connected to the CPU, each of which has a fixed vector (or pointer) which instructs the CPU where to go when a particular interrupt happens (these vectors are at the top of memory, near the reset vector). In some cases there can be many possible sub-events associated with each interrupt. To solve this, the MCU designers allow us to see the results of the hardware checks, these are called interrupt "flags", a set of bits which are individually set to 1 to indicate which interrupt occurred as well as particular sub-events. It is good practice to inspect interrupt flags during an ISR to ensure we are processing the correct event. Depending on the tasks needed to be performed in an ISR, there is also a possibility that another interrupt from the same source could occur before we are done running the current ISR which could lead to cascading or nested interrupts; this can be very bad especially if we are not expecting it. To solve this, the MCU designers made it so we can control whether or not we want another interrupt to occur while we are in an ISR. When we enter an ISR, the I CCR flag is set indicated we don't wanted nested interrupts; to allow them we only need to clear the I CCR flag with a "CLI" instruction. Once we are done processing and interrupt, we need to tell the hardware which generated it that we are finished. We do this by "clearing" the corresponding interrupt flag prior to leaving the ISR; it is done by writing a "1" to the same register and bit position of the flag we inspected initially. This ensures that the CPU will not rerun an ISR after it is completed. This may sound counter intuitive to "clear" a bit by writing a "1" to it, but interrupt flags in the MCU are not normal memory. They are a special circuit in which the behaviour is different based on whether or not the action is read or write when accessed. When reading, we see the status of the flags, while when writing, we indicate which values to reset to 0. This is preferred as there may be other bits set in the flag which were generated by other interrupts. It avoids the problem of inadvertently clearing other interrupt flags.

The FRT overflow is just one of many timing interrupts, however we will use it in part C. For this part, we will first explore the input capture system. This is a feature of the MCU which can monitor a hardware line and determine the exact "time" in which it an external source switches it from a 0 to a 1, a 1 to a 0, or both. The "time" in this case is that of the free running timer (16-bit). We can therefore use this feature to determine when a signal change occurred and setup a new event based on the initial change. An ISR can be executed soon after the signal changes, but we know there can be delays from when the signal changes to when the code is actually executed. The MCU helps us by saving the FRT from its register (TCNT) into another register (TICx, where x can be between 1 and 4) at the precise moment the signal change is detected so we can use it in our ISR when it is eventually executed.

To demonstrate this in THRSim11, we will setup a virtual switch which we will use one to connect to one of the input capture lines, PA0; IC is only available on certain pins of port A. You can do this by going to the "Connect", "Switch" menu, and select "Pin PA0/IC3" on the left and be sure to press "Connect"; you can then just close the window. A new window should appear; you may have to make it larger to see the title (which shows the state of the switch) as well as move it around. Left clicking on the switch toggles its value. We can alternatively determine the state of this switch by reading PA0, or bit 0 of port A (PORTA), but as detailed earlier, polling is never a good way to do this. The IC can let us know when we can read the switch or we could just set a specific condition to generate the IC on the rising or falling edge of the input.

Rename the "**SID_9a.asm**" file appropriately and open it in THRSim11. You will notice a lot of EQUs at the top of the file. This is to instruct the assembler on the location of many of the MCU registers. The first two are the "default" values which will be used to setup the system (see the code beginning at the "start" label). "mIC3" and "iTCTL2" need to be set properly as they will be loaded into "TMSK1" and "TCTL2" respectively when the code starts; you will need to look at the reference manual to discover which bits need to be set. We need to configure IC3's behaviour with iTCTL2 so that it "captures on any edge"; set

"EDGE3B" and "EDGE3A" appropriately. The interrupt mask for IC3 must be set to 1, so "IC3I" must be set to 1 for mIC3.

Any ISRs should do what they need to as quickly as possible as there could be another interrupt coming up very soon. In this case we will use the IC3 ISR to record the time the switch was changed. We can do this by saving the 16-bit value at TIC3 into a reserved space in RAM called "itime". We will also save the state of the switch by reading PA0 ("PORTA" bit 0) and storing it in "istate"; we should turn off all the other bits (7-1) before saving it as any of those could be on as they correspond to other signals on Port A. Since a lot of interrupts can happen on an MCU, we need to flag that tells us these new values we saved are valid; we do this by setting "iflag" (another variable) to a non-zero value (0 means invalid or empty). This programming technique of saving information and setting a flag is known as a "mailbox"; we will discuss it shortly. With ISRs, we must also remember to clear the interrupt flag by writing a "1" to the correct bit position (same position as for TMSK1) in TFLG1 before we leave, otherwise the ISR will just run again. Once that is done we can then leave the ISR by issuing the RTI instruction. Essentially the ISR will put information into a mailbox ("itime" and "istate") and sets "iflag" on when there is something new in there. It is called a mailbox as it is a direct analogy to mailboxes placed at the end of your driveway near the street where a flag is raised by the mailperson delivering the message to let you know there is a new message there without you expressly opening it to look inside. We are putting information into our mailbox as processing it in the ISR would be too time consuming. Instead we will process it in our "main loop", the part of the code that repeats over and over again waiting for something important to happen, does something about it, but it can take as long as necessary to do it; this loop is not timing sensitive. If we examine the main loop, we can see it first "waits" or an interrupt to happen. As we know the "WAI" instruction prepares the system to handle an interrupt quicker by saving the CPU state before the event to increase response time. When the interrupt does happen, the ISR will run, and when the ISR leaves, the code after "WAI" will continue. We next check for the mailbox state "iflag". If it isn't zero then we know something is waiting there for us. We next print the value of the FRT as well as the value of "itime" (the value of the FRT when the switch was toggled) followed by a space and then the value of "istate" which is the new position of the switch. Here we will be able to see how long it takes from when the switch is thrown to when we can actually process the result; it isn't immediate. The FRT (TCNT) must be read using a 16-bit read (LDD, LDX, or LDY) operation, not just because it is a 16-bit load, more importantly it avoids the potential for overflow that may happen when issuing two separate 8-bit reads as the FRT may change significantly between the reads. Before finishing up, we make sure to clear the "iflag" so if on subsequent interrupts (potentially from other sources) we don't repeat this process of displaying the mailbox contents if new information isn't present in it. We then continue processing additional mailboxes (which we don't have yet, but will in part B), and then go back and wait for another interrupt. The time to write to the LCD is very quick in THRSim11, but that is because it is a simulation environment. Typically LCD controllers or serial connections require far more time to send information out to them. If this was done in an ISR, other interrupts would be delayed which could cause timing problems. It is good practice to always use ISRs, mailboxes, and main loops to process information. This is the whole basis of how the graphical user interfaces on your phone and home PC operate.

Your task is to write the core part of the IC3 ISR. Follow the comments to add code after the "ic3vect" label. The code for the ISR should be less than 15 instructions as it should run quickly.

When you run your code, the output should show the current FRT, the FRT value when you pressed the switch followed by the state of the switch (1 for on, 0 for off). The difference between these two FRT values should be about 64 decimal, indicating that it took 64 cycles from when the switch was pressed until we could begin processing the action. If the system had a 2 MHz E-Clock, this translates to 32 micro-seconds which may or may not be a long time depending on the application. Note that the FRT values shown will eventually overflow back to 0 every 65536 cycles; this can happen quickly in a real world system, but we will fix this later in part C.

B. Output Compare

We will extend the code we wrote in part A to perform an action based on toggling the switch: we will turn on and off an LED after a short but EXACT delay. We can do this using the OC feature which behaves like an alarm clock; when the OCn (where n can be 1-5) trigger register (TOCn) is equal to the FRT (TCNT), the associated output OCn can be set to 0, 1, or toggled (based on the settings of TCTL1). We need to add a virtual LED to our system. Do this by going to the "Connect", "Led" menu, and select "Pin PA4/OC4", press connect and close the window; you can resize and move then window just like the switch one earlier.
Rename the "**SID_9b.asm**" file appropriately and open it in THRSim11. You will need to setup five values at the top of the file to properly enable IC3 (the save values from part A) as well as enabling OC4. Follow the comments. In the main loop, the mailbox handler for IC3 does a few more things than in part A. When the switch is thrown we need to setup OC4. We first clear any old interrupt flags that may have been set but not cleared before; this is necessary or else the ISR handler won't run. We then need to enable the mask for OC4. This is analogous to "what do we do after the alarm"? We enable it so that our OC4 ISR runs, which when the alarm executes, we will stop future alarms as we will only do this again when the switch is thrown. Again, keep in mind the FRT resets quite often so we may not want an alarm in that current 0-65535 interval. Next, since we know when the switch was thrown (TIC3), we can easily setup the time for the LED connected to OC4 to be changed (TOC4) by using simple addition in. See the comments. The remaining part of the IC3 mailbox code now prints an "i" and the FRT value of IC3 followed by the switch state. Based on the switch state, we will set the LED to on or off. This is done by the configuration of OC4 (does the output go low or high). We have added another mailbox for OC4 which just prints an "o" followed by when the OC4 ISR ran which is stored into "otime" and enabled with "oflag".
Now onto the ISRs. Move down to the "ic3vect" label and insert your code from part A. Continue to move down to "oc4vect". Here you need to add your new ISR for OC4. Again, follow the comments carefully. When you run your code and press the switch you will see an "i" followed by the FRT value when you pressed the switch followed by the state of the switch. Exactly 16384 cycles from when the switch was thrown OC4 will happen changing the LED to the state of the switch. You will see an "o" followed by the approximate FRT time of when the OC4 ISR ran. You will notice this value is not exactly 16384 cycles from when the switch was thrown as this is the activity which happens after the output changes; typically for setting up the next OC4, again it depends on the application.

C. Free Running Timer Overflow

As mentioned early, the value of the FRT is only 16-bits which can lead to overflows happening quite quickly (in human time). For both parts A and B, since the FRT output is limiting, we don't really known when the event occurred outside of the current FRT interval. For this part of the assignment we will add in code to extend the FRT by incrementing a 16-bit value ("timerh", for timer-high) when the FRT overflows. This is can be done easily by adding in an ISR when the FRT overflows.
Rename the "**SID_9c.asm**" file appropriately and open it in THRSim11. You will need to copy your values from part B as well as add another one. A couple new variable have been added "itimeh" and "otimeh" to keep track of the higher resolution FRT. You will need to add in code in the main loop and the IC3 and OC4 ISRs to print and capture the new timer. See the comments. The ISR for TOF is simple, when it happens increment the 16-bit counter "timerh". See the comments.
When you run your code the output will be the same as part B except the FRT values will be 32-bits now and over time when you throw the switch, the upper 16-bits of the FRT should be changing thereby giving us more information as to when the switch as changed. It should be noted that this doesn't increase the resolution of the OCs. If we want to do that, it will take more work to schedule when the OCx interrupts happen based on the upper 16-bits of the new FRT, but it isn't impossible. We won't be covering that here though.
ZIP all your ASM files into "SID_9.zip" and upload it to Blackboard.