In this section we present Timer2A as a means to create **periodic interrupts**. The TM4C123 has six timer modules that operate in a similar manner as Timer2A, called Timer0A, Timer1A, Timer2A, Timer3A, Timer 4A and Timer 5A. The TExaS simulator, LaunchPadDLL, supports the first four of these timers. The real board grader uses Timer 5A and the TExaS oscilloscope uses Timer 4A. This means your Lab 15 game could use Timer0A, Timer1A, Timer2A and/or Timer3A to create periodic interrupts. You will find example projects for these four timers in TExaSware example folder.

VIDEO 15.6 TIMER2A

0:00 / 6:02 1.0x

JONATHAN VALVANO: Next, we want to show you

another periodic interrupt called Timer2.

Timer2 is a lot like SysTick.

It has, essentially, two registers.

One of them is a counter, and just like SysTick, it counts down.

Another one is a reload value, in our particular case it's called T-A-I-L-R,

and it's the reload value.

Just like SysTick, when this counter goes from a one to a zero,

it will set the trigger flag.

This trigger flag is going to cause an interrupt to happen.

Also like SysTick, the value we put into the reload value

will determine the frequency of the interrupts.

Different than SysTick is, this timer and this reload value,

is now a full 32 bits wide.

The code shown here on the page is the initialization code

for this Timer2 periodic interrupt.

We're going to pass in the value that we want to interrupt at.

In our particular case, since we're running at 80 MHz,

the interrupt rate will be this value, period, times 12.5 nanoseconds.

Because that's the bus period.

Like all modules, we have to turn on its clock,

and so this is the clock bit for the Timer2 register.

And after we turn on the clock, we will wait a while

for the clock to stabilize.

I have a couple of global variables that I use to debug with.

Help

This one will count the number of times the interrupt occurs,

and this is a semaphore flag which gets set every interrupt.

The 10 steps used to initialize the timer are now shown.

We disable the timer during set up, we set it up

as a periodic interrupt with a 32-bit counter, the value of period minus one

is stored into the reload value, so again, the interrupt rate

will be the value, period, times 12.5 nanoseconds.

We set the resolution to run at the bus clock, which

is going to be 12.5 nanoseconds.

I like to clear the interrupt service flag in the ritual, that's step six.

We then arm that bit for interrupts, we set its priority

by setting three bits of the priority five register,

we enable-- this is interrupt 23, and so we

will enable interrupt 23 and the nested vector interrupt controller.

Then after the entire thing is initialized,

we will then enable Timer2A.

If the main program executes at interrupt enable,

then interrupts at this period will occur periodically.

RAMESH YERRABALLI: So we saw the initialization ritual

for the interrupt service routine, so we now

have interrupts happening on Timer2A.

When an interrupt occurs, the handler, which is the Timer2A handler,

is called, and so we're going to look at what we have to do within this handler.

JONATHAN VALVANO: All right.

So the main thread will enable interrupts and then

perform some task over and over.

But when the timer goes from a one to a zero,

it will set a bit in the timer raw interrupt status.

This is the trigger flag which will cause the interrupt to occur.

So our handler, then, here, will then be executed.

Like most interrupts, we have to acknowledge that interrupt

by clearing the trigger flag.

The trigger flag happens to be in bit zero,

and to clear the Timer2A timeout flag, we

are going to write a one to the interrupt clear register.

When I write a one to bit zero, it will clear the raw interrupt status bit

zero, which is our trigger flag, acknowledging the interrupt.

Then these two steps are the body or the function

of what we want to happen in the interrupt service routine.

In this case, I'm just going to increment a counter as a debugging way

to say yes, this is running.

And then we could use a semaphore flag to signal other threads

that this interrupt occurs.

RAMESH YERRABALLI: So for example, the main program

could be checking this flag, and only when this flag is on,

it could perform a certain function.

When it's off, it's going to skip that function

and keep checking this flag again and again.

JONATHAN VALVANO: Yes.

So this is how semaphore works.

This semaphore flag here is set in the interrupt service routine,

and then the main loop will look for that flag.

If the flag is set, then it will clear the flag,

perform some function associated with the semaphore,

and this whole process is repeated over and over again

So we have two threads, the foreground

Help

The TM4C123 has six timers and each timer has two modules, as shown in Figure 15.8. In this section we will combine the two modules (Timer2A and Timer2B) into one 32-bit timer. This means the TM4C123 can have up to six 32-bit timers. In this mode the timer is configured as a 32-bit down-counter. When the timer counts from 1 to 0 it sets the trigger flag. The trigger flag is called Timer A time out raw interrupt status (TATORIS) and this flag exists as bit 0 in the **TIMER2_RIS_R** register. On the next count, the 32-bit timer is reloaded with the 32-bit value in **TIMER2_TAILR_R**. We select periodic timer mode by setting the 2-bit **TAMR** field of the **TIMER2_TAMR_R** to 0x02. In periodic mode the timer runs continuously. The timers can be used to create pulse width modulated outputs and measure pulse width, period, or frequency. For more information on the timers see Chapter 6 of Volume 2 (Embedded Systems: Real-Time Interfacing to ARM® Cortex™-M Microcontrollers, 2013.)

In Program 15.6 we will use Timer2A to trigger a periodic interrupt. The precision is 32 bits and the resolution will be the bus cycle time of 12.5 ns. This means we could trigger an interrupt as slow as every 2^{32}*12.5ns, which is 53 seconds. The interrupt period will be

(**TIMER2_TAILR_R** +1)*12.5ns

Each periodic timer module has

A clock enable bit, bit 2 in **SYSCTL_RCGCTIMER_R**
A control register, **TIMER2_CTL_R** (set to 0 to disable, 1 to enable)
A configuration register, **TIMER2_CFG_R** (set to 0 for 32-bit mode)
A mode register, **TIMER2_TAMR_R** (set to 2 for periodic mode)
A 32-bit reload register, **TIMER2_TAILR_R**
A resolution register, **TIMER2_TAPR_R** (set to 0 for 12.5ns)
An interrupt clear register, **TIMER2_ICR_R** (bit 0)
An interrupt arm bit, TATOIM, **TIMER2_IM_R** (bit 0)

A flag bit,TATORIS, **TIMER2_RIS_R** (bit 0)

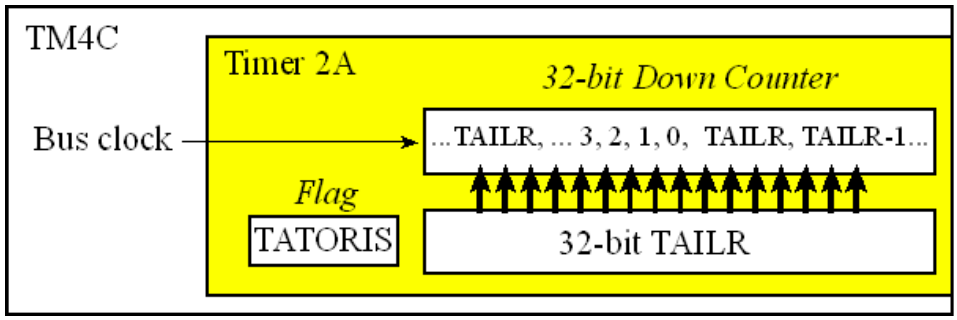

Figure 15.8 Periodic Timers on the TM4C123.

```
unsigned long TimerCount;
void Timer2_Init(unsigned long period){
  unsigned long volatile delay;
  SYSCTL_RCGCTIMER_R |= 0x04;   // 0) activate timer2
  delay = SYSCTL_RCGCTIMER_R;
  TimerCount = 0;
  TIMER2_CTL_R = 0x00000000;   // 1) disable timer2A
  TIMER2_CFG_R = 0x00000000;   // 2) 32-bit mode
  TIMER2_TAMR_R = 0x00000002;  // 3) periodic mode
  TIMER2_TAILR_R = period-1;   // 4) reload value
  TIMER2_TAPR_R = 0;           // 5) clock resolution
  TIMER2_ICR_R = 0x00000001;   // 6) clear timeout flag
  TIMER2_IMR_R = 0x00000001;   // 7) arm timeout
  NVIC_PRI5_R = (NVIC_PRI5_R&0x00FFFFFF)|0x80000000;
// 8) priority 4
  NVIC_EN0_R = 1<<23;          // 9) enable IRQ 23 in
  TIMER2_CTL_R = 0x00000001;   // 10) enable timer2A
}
// trigger is Timer2A Time-Out Interrupt
// set periodically TATORIS set on rollover
void Timer2A_Handler(void){
  TIMER2_ICR_R = 0x00000001;  // acknowledge
  TimerCount++;
// run some background stuff here
}
void Timer2A_Stop(void){
  TIMER2_CTL_R &= ~0x00000001; // disable
}
void Timer2A_Start(void){
  TIMER2_CTL_R |= 0x00000001;   // enable
}
```

Program 15.6. Periodic interrupts using Timer2A (included in C15_Timer2 starter project).

Help

Hard fault warning: The TM4C123 implements the clock bits for Timer0 Timer1, Timer2, Timer3, UART0, UART1, SSI0, SSI1 and the GPIO in two places (legacy and real). For example

SYSCTL_RCGC1_R bit 18 is the Timer2 clock (legacy to match the LM3S series)
SYSCTL_RCGCTIMER_R bit 2 is also the Timer2 clock (the real one)

The bad part is writes to SYSCTL_RCGC1_R propagate to SYSCTL_RCGCTIMER_R, but writes to SYSCTL_RCGCTIMER_R do not propagate to SYSCTL_RCGC1_R. So there is a sequence that causes a Hard fault.

```
Timer2_Init(80000);  // Program 15.5
Nokia5110_Init();    // Lab 15
```

because it breaks down to these calls

```
SYSCTL_RCGCTIMER_R |= 0x04;  // Program 15.5 (using new register)
SYSCTL_RCGC1_R |= 0x04;      // turns on SSI0, but turns off Timer2
```

1) One way to fix is by using the old register in Timer2_Init

```
SYSCTL_RCGC1_R |= 0x40000;   // Timer2 using the legacy register
```

2) Another way to fix is by initializing the new registers second

```
Nokia5110_Init();    // Lab 15
Timer2_Init(80000);  // Program 15.5
```

Help

Say you want to convert a wav file **(blah.wav**) you want to use in your game. Here are the sequence of steps you can do in Matlab (or a free alternative to Matlab from GNU called Octave – http://octave.org (http://octave.org)) to make this happen:

1. Read the file and extract the samples as well as the frequency:

[Spls, fs] = wavread('blah.wav');

2. Downsample it, to get it to the frequency you want (11.025 kHz):

Spls = downsample(Spls, round(fs/11025));

3. Adjust the samples (originals are fractions between -1 and 1) to range between 0 and 15 (4-bit)

Spls = round((Spls+ 1)* 7.5);

4. write it to a file so you can cut and paste it into your C code:

file = fopen('blah.txt', 'w');

```
fprintf(file, 'unsigned char blah[] = {');

fprintf(file, '%d,', Spls);

fprintf(file, '};\n');
```

That's it, you should have a file (called **blah.txt**) with a declaration you can cut and paste in your code. There may be an extra commas in the output that you would have to remove. You could make a convert.m file with these statements inside a subroutine and have a converter that can be invoked for conversion from matlab. (From Mitch Crooks)

CHECKPOINT 15.8

The sound data in the file **sound.c** were sampled at 11.025 kHz. To play these sounds we will trigger a timer interrupt at 11.025 kHz and output one data point to the DAC every execution of the ISR. How do we call the function **Timer2_Init** in Program 15.6?

Hide Answer

The bus clock is 80MHz, or 12.5ns/cycle. To interrupt at 11.025 kHz, the interrupt must occur every 80,000,000/11,025 cycles, which is about 7256. To make this happen we call **Timer2_Init** with the parameter 7256.

Help

In Lab 13, we had a fixed data table and varied the interrupt rate. In Lab 15 we will have a fixed interrupt rate and vary the data. To play a sound we need to specify an array of data points (see sound.c) and the number of points in the array. To start the sound we set a pointer to the beginning of the array, initialize a counter, and then arm the periodic interrupt.

```
unsigned long Index = 0;    // index into array
const unsigned char *Wave;  // pointer to array
unsigned long Count = 0;    // number of points (0 means no sound)
void Sound_Play(const unsigned char *pt, unsigned long count){
  Wave = pt;
  Index = 0;
  Count = count;
  Timer2A_Start();
}
```

In the ISR we output one point, if the count becomes zero we stop the interrupts. The data in the **sound.c** is 8 bits, so use top 4 bits

```
void Timer2A_Handler(void){
```

```
TIMER2_ICR_R = 0x00000001;  // acknowledge
  if(Count){
    DAC_Out(Wave[Index]>>4);
    Index = Index + 1;
    Count = Count - 1;
  }else{
    Timer2A_Stop()
  }
}
```

Sounds implemented this way use a lot of memory. The free version of Keil has a 32k memory limit, so you will have to make your sounds very short to be able to continue to use the free compiler.

CHECKPOINT 15.9

Estimate the ratio of the time to execute this ISR and the time between when the ISR is invoked. Is this good design of an interrupt-driven task?

Hide Answer

The ISR executes about 8 lines of C. At about 0.1us/line this ISR will execute in about 1us. 1us is small compared to the 90.7us period between interrupts. So, yes this is good.

CHECKPOINT 15.10

We now have two interrupts. The simple **Move** function analyzed in Checkpoint 15.7 only took 3us to execute. However, now assume your more complex game engine requires 100 us to execute. Assume you execute the game engine in the **SysTick_Handler**, which is triggered 30 times per second. Assume a second interrupt, **Timer2_Handler** outputs to the DAC at 11.025 Hz and takes about 1us to execute. What should be the relative priority of the two ISRs?

Hide Answer

It is more important to create sound than to create images. Our ears can hear signals up to 10 kHz but our eyes can only see up to about 15Hz. If the timer ISR has a higher priority than the SysTick ISR, then the sound output can interrupt in the middle of the game engine (which is good thing). If the timer ISR were to have an equal or lower priority than the SysTick ISR, then the sound output could be delayed during the execution of the game engine (causing a delay of up to 100us, and your ears could hear the missed sample).

EdX is a non-profit created by founding partners Harvard and MIT whose mission is to bring the best of higher education to students of all ages anywhere in the world, wherever there is Internet access. EdX's free online MOOCs are interactive and subjects include computer science, public health, and artificial intelligence.