26. Microcontroller as a frequency meter

This example demonstrates the use of timers for frequency measurement. Two timer are linked, one to count pulses and the other to define the counting interval of 1s. The frequency gets measured without the intervention of the software.

26.1. The hardware

We have been using single counters/timers (TIMs) previously. It is also possible to configure several TIMs to work together, as will be demonstrated in this experiment on measuring a frequency of a digital signal.

Consider a digital signal *X*, its frequency is simply a number of pulses appearing within a defined time interval. The block diagram of the measurement setup is given in Figure 26.1. The input signal *X* first enters the gate AND. The other input of this gate receives a timing signal *T*, which is active (low) for exactly one second. The output of the gate AND is then a gated version of the input signal *X*, and the frequency of input signal *X* is determined by counting the gated pulses.

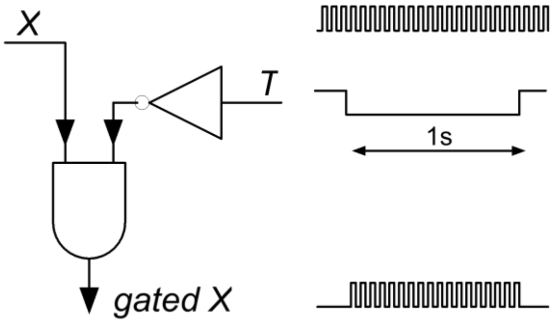


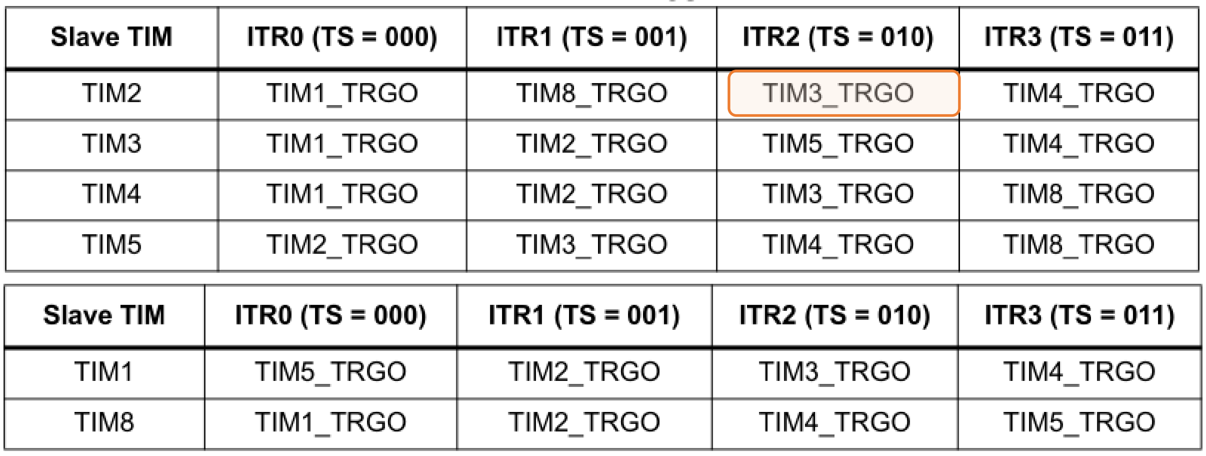
Figure 26.1: Pulses X are gated for a predefined time interval T, then counted to determine the frequency

First TIM (master) within a microcontroller can be used to define timing signal *T*, and second TIM (slave) can be used to count pulses. The missing element is the gate AND. An obvious solution is to route the output of the master TIM to a pin of a microcontroller, add a gate AND externally, and route the gated output back to a pin of the microcontroller for counting by the slave TIM. With the microcontroller STM32F407 this is not necessary, since the slave TIM can be configured in a so-called “gated mode 2” where a gate AND is already incorporated in a TIM. The details of the mode can be found in chapter 18.3.14 (“Timers and external trigger synchronization, Slave mode: External Clock mode 2”) of RM0090.

The vital information here is that every TIM has an output signal that is available to selected peripherals within the microcontroller, but cannot be routed to microcontroller pins. This signal is named TRGO (“TRiGger Output”), and can be assigned to various internal signals or events within a TIM. Additionally, every TIM has several input signals named ITR0 to ITR3, where outputs TRGO of other TIMs are available. Since there are only four ITR inputs not all outputs of other TIMs are available at every TIM. An interconnection matrix, given at Figure 26.2, describes possible connections. From this table one can deduce that TIM2 (slave in this case), for instance, can be gated with the output TRGO of TIM3 (master) by selecting ITR2 as a gating input signal to TIM2.

An arbitrary decision on TIMs used in this experiment on frequency meter was made:

- **TIM2** (slave) is used for pulse counting since it is a 32 bit counter and its input signal ETR is accessible at the BaseBoard, K406, pin 3 (PA15) and
- **TIM3** (master) is used to define the gating signal *T*: “Capture and Compare Channel 4” of this TIM can be configured in PWM mode and the output of this channel is accessible at the BaseBoard, K406, pin 6, (PA3).



Slave TIM	ITR0 (TS = 000)	ITR1 (TS = 001)	ITR2 (TS = 010)	ITR3 (TS = 011)
TIM2	TIM1_TRGO	TIM8_TRGO	TIM3_TRGO	TIM4_TRGO
TIM3	TIM1_TRGO	TIM2_TRGO	TIM5_TRGO	TIM4_TRGO
TIM4	TIM1_TRGO	TIM2_TRGO	TIM3_TRGO	TIM8_TRGO
TIM5	TIM2_TRGO	TIM3_TRGO	TIM4_TRGO	TIM8_TRGO

Slave TIM	ITR0 (TS = 000)	ITR1 (TS = 001)	ITR2 (TS = 010)	ITR3 (TS = 011)
TIM1	TIM5_TRGO	TIM2_TRGO	TIM3_TRGO	TIM4_TRGO
TIM8	TIM1_TRGO	TIM2_TRGO	TIM4_TRGO	TIM5_TRGO

Figure 26.2: Interconnection matrix for TIMs

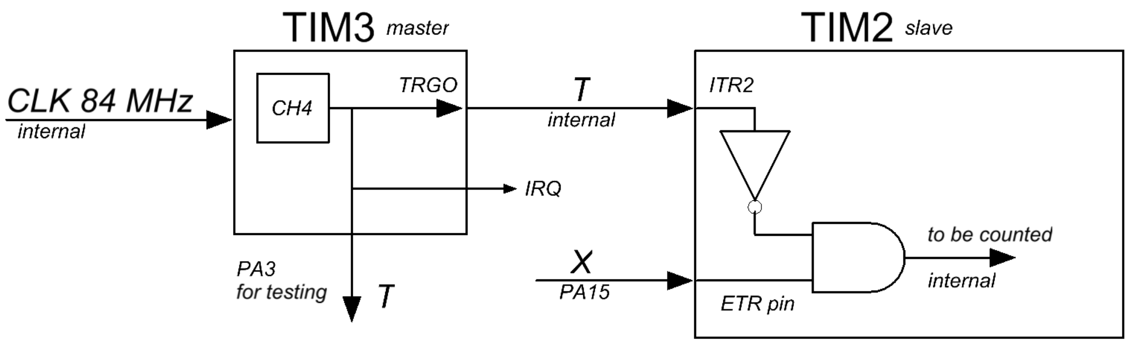


Figure 26.3: Block diagram with timers for frequency meter

The final interconnection diagram is given in Figure 26.3. The measurement cycle comprises of three actions:

a) clear TIM2 before timing signal *T* becomes active,
b) count pulses at *X* using TIM2 while signal *T* is active,
c) display the content of TIM2 on a LCD when signal *T* becomes inactive and
   repeat from a)

Actions can be reordered as depicted in the timing diagram at Figure 26.4. The timing signal *T* is periodical: it stays low for 1 s, then jumps high for (say) 100 ms, and repeats low again. The interval of 100 ms should suffice to transfer the content of TIM2 to LCD and clear TIM2.

The timing signal is generated by TIM3 working in PWM mode at Channel 4. In order to achieve such timing with a 16 bit counter the input clock signal (84 MHz) is first divided by 2000 by a pre-divider unit within timer TIM3. With this, the internal clock frequency for TIM3 becomes 42 kHz. 42000 consecutive clock pulses define a time interval of 1 s, and some more time is needed to clear TIM2 and push the result towards the LCD, altogether 1.1s. The repeat period of TIM3 is therefore set to 42000 + 4200 – 1 = 46199. Out of this period the signal *T* must be active (low) for 1 s, therefore 42000 clock pulses. From these numbers the configuration constants for TIM3 are derived.

From the timing diagram at Figure 26.4 one can see that presenting of the result and clearing of the TIM2 should commence once signal *T* jumps high. The rising edge of signal *T* can be used to trigger interrupt, and the interrupt routine should call appropriate functions to display the result on the LCD (or better still: store the result to safe place and allow the main program to display the result on the LCD) and clear TIM2. The same signal *T* is therefore used to trigger interrupt as depicted in Figure 26.3.

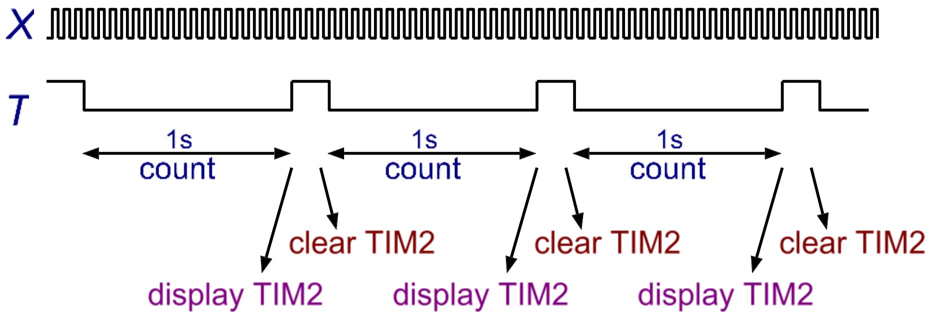


Figure 26.4: The timing diagram for frequency meter, timing is not in scale

26.2. The software – Master TIM configuration

Two blocks for configuration are needed, one per TIMer. Each block will be described separately, and will be packed into a function, and includes microprocessor pin configuration and TIM configuration. Additionally, the main program and the interrupt routine is described.

a) TIM3 - master

TIM3 PWM output from Channel 4 can be routed to port GPIOB, pin 1. This can be done with the following set of commands:

```
GPIO_InitTypeDef GPIO_InitStruct;
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);
  GPIO_InitStruct.GPIO_Pin   = GPIO_Pin_1;
  GPIO_InitStruct.GPIO_Mode  = GPIO_Mode_AF;
  GPIO_InitStruct.GPIO_Speed = GPIO_Speed_25MHz;
  GPIO_InitStruct.GPIO_OType = GPIO_OType_PP;
  GPIO_InitStruct.GPIO_PuPd  = GPIO_PuPd_NOPULL;
  GPIO_Init(GPIOB, &GPIO_InitStruct);
  GPIO_PinAFConfig(GPIOB, GPIO_PinSource1, GPIO_AF_TIM3);
```

The first line is just a declaration of a structure with parameters for port configuration, and the second line activates the port GPIOB by enabling the clock signal for it. Next five statements fill the declared structure with parameters, and then these parameters are sent to the port GPIOB by calling a function “GPIO_Init” and a reference to the data structure. Note that “GPIO_Mode_AF”, the alternate function mode is selected for this pin, since the output of the TIM3, Channel 4 is available only as alternate function. The last line selects appropriate alternate function for port GPIOB, pin 1, as the one associated with TIM3.

Next block within the same configuration function enables the time-base part of TIM3 and defines its operating parameters. This gets done by first declaring the appropriate parameter structure (first line), continues with enabling the TIM3 (second line), initializing the parameter structure (lines 3 to 7), calling the appropriate function to configure the TIM3 (line 8) and finishes with enabling the TIM3 (lines 9 and 10), as shown in the following listing.

```
TIM_TimeBaseInitTypeDef TIM_TimeBaseInitStruct;
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);
  TIM_TimeBaseInitStruct.TIM_Prescaler                = 1999;
  TIM_TimeBaseInitStruct.TIM_CounterMode              = TIM_CounterMode_Up;
  TIM_TimeBaseInitStruct.TIM_Period                   = 46199;          // period
  TIM_TimeBaseInitStruct.TIM_ClockDivision            = TIM_CKD_DIV1;
  TIM_TimeBaseInitStruct.TIM_RepetitionCounter        = 0;
  TIM_TimeBaseInit(TIM3, &TIM_TimeBaseInitStruct);
  TIM_Cmd(TIM3, ENABLE);
  TIM_InternalClockConfig(TIM3);
```

The pre-scaler division ratio is 2000, but due to the nature of TIM modules one less has to be defined here. The counter should count up, and the period of the counter is defined in line 5, again one less than the number of clock pulses per period (46200 - 1). There will be no further division of clock signal, and the repetition counter is of no interest here.

These parameters are transferred to TIM3, time-base by calling function "TIM_ TimeBaseInit" in line 8. TIM3 is then enabled in line 9, and the clock source is selected by calling the function "TIM_ InternalClockConfig" in line 10; please note that for TIM3 this call is mandatory to select the internal 84 MHz clock signal.

Next sub-block of TIM3 to be configured is the "Capture and Compare", channel 4. This block actually generates the timing signal *T* by comparing the current content of the time-base counter (already configured) by a predefined number. The configuration commences with the declaration of the appropriate parameter structure in line 1, and continues with the initialization of the data structure members in lines 2 to 9. The block ends with a call to function that transfers these parameters to the sub-block, channel 4. Statements to configure the block are given below.

```
TIM_OCInitTypeDef       TIM_OCInitStruct;
  TIM_OCInitStruct.TIM_OCMode           = TIM_OCMode_PWM1;
  TIM_OCInitStruct.TIM_OutputState      = TIM_OutputState_Enable;
  TIM_OCInitStruct.TIM_OutputNState     = TIM_OutputState_Disable;
  TIM_OCInitStruct.TIM_Pulse            = 42000;                        // interval T
  TIM_OCInitStruct.TIM_OCPolarity       = TIM_OCPolarity_Low;
  TIM_OCInitStruct.TIM_OCNPolarity      = TIM_OCPolarity_Low;
  TIM_OCInitStruct.TIM_OCIdleState      = TIM_OCIdleState_Set;
  TIM_OCInitStruct.TIM_OCNIdleState     = TIM_OCIdleState_Set;
  TIM_OC4Init(TIM3, &TIM_OCInitStruct);
```

The second line selects PWM mode for channel 4 (see the description of possible modes in RM0090). We need only 'true' output of this block, so we enable the "OutputState" member and disable the "OutputNState" member. Next we set the duration of the pulse *T* to 42000 clock cycles (1 s) in line 5, and define how this pulse should start (when the content of the time-base counter is 0) and what should happen when the time-base counter reaches the predefined value. Next four statement are used to do this: statement in line 5 states that the pulse *T* should start 'low', and statement in line 7 states that the timing signal T should jump high when the content of the time-base counter equals the predefined value. Statements in lines 6 and 8 have no meaning, since we do not use the inverted output at all. These parameters are transferred to the appropriate registers within the "Capture and Compare" block, Channel 4, by calling the function "TIM_ OC4Init" in line 9. A note to the user: oter channels of the same TIM could be configured using "TIM_ OCxInit", where "x" stands for the number of channel.

Next sub-block configures TIM3 to act as a master and selects the source of the interconnecting signal TRGO, the statements are given below. The first line configures the output signal of "Capture and Compare", Channel 4 to be the source, while the second line puts TIM3 in master mode.

```
  TIM_SelectOutputTrigger(TIM3, TIM_TRGOSource_OC4Ref);
  TIM_SelectMasterSlaveMode(TIM3, TIM_MasterSlaveMode_Enable);
```

The last sub-block in this function configures an interrupt request IRQ from TIM3, it is given bellow. The first line of this sub-block selects the moment time-base counter content equals the value written into “Capture and Compare” block, Channel 4, to be the moment of the interrupt request, and enables the IRQ output of the TIM3 module. The second line enables the corresponding input line for TIM3 to the NVIC (Nested Vectored Interrupt Controller).

```
  TIM_ITConfig(TIM3, TIM_IT_CC4, ENABLE);
  NVIC_EnableIRQ(TIM3_IRQn);
```

With these statements, the TIM3 is configured and can run and provide the required timing signal at GPIOB, pin 1. However, we do not yet have the interrupt routine for TIM3, and running the program without this routine would prevent the program from executing.

b) **TIM2 - slave**

The pulses *X* to be counted are to be connected to the ETR (External Trigger) input line of the TIM2, and this will be configured first. The ETR line can be routed to port GPIOA, pin 15, and is accessible as alternate function. The listing of the configuration for ETR line follows.

```
GPIO_InitTypeDef GPIO_InitStruct;
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);
  GPIO_InitStruct.GPIO_Pin   = GPIO_Pin_15;
  GPIO_InitStruct.GPIO_Mode  = GPIO_Mode_AF;
  GPIO_InitStruct.GPIO_Speed = GPIO_Speed_25MHz;
  GPIO_InitStruct.GPIO_PuPd  = GPIO_PuPd_DOWN;
  GPIO_Init(GPIOA, &GPIO_InitStruct);
  GPIO_PinAFConfig(GPIOA, GPIO_PinSource15, GPIO_AF_TIM2);
```

The first line is a declaration of a data structure containing parameters to be passed to the configuration function named “GPIO_Init” (line 7). The function call in line 2 enabled port GPIOA by enabling the clock for this port. Line 3 defines the pin used, and line 4 selects the mode for this pin to be the alternate function. Next two lines define the speed and termination for this pin. The last line selects the correct alternate function for pin 15 of port GPIOA and routes the signal from this pin to ETR input of TIM2.

The next block configures the counting at TIM2. The first line declares a variable containing the data structure for configuration, and the second line enables the clock for TIM2 thus turning it ON. Lines 3 to 7 are used to initialize the declared data structure. We do not need a pre-scaler, so the pre-scaling constant is set to 0. The TIM2 should count upwards, and the whole range of the counter (32 bits) should be used: the reload value for this counter is therefore set to 0xffffffff. We do not care about clock division ratio and repetition counter, so these two parameters are meaningless here. The function to distribute these parameters to corresponding registers of TIM2 is called in line 8.

```
TIM_TimeBaseInitTypeDef TIM_TimeBaseInitStruct;
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE);
  TIM_TimeBaseInitStruct.TIM_Prescaler              = 0;
  TIM_TimeBaseInitStruct.TIM_CounterMode            = TIM_CounterMode_Up;
  TIM_TimeBaseInitStruct.TIM_Period                 = 0xffffffff;
  TIM_TimeBaseInitStruct.TIM_ClockDivision          = TIM_CKD_DIV1;
  TIM_TimeBaseInitStruct.TIM_RepetitionCounter      = 0;
  TIM_TimeBaseInit(TIM2, &TIM_TimeBaseInitStruct);
```

Next sub-block configures the gating signal coming from TIM3. The function call in the first line selects gating signal to come from ITR2 input to the TIM2, this was configured as the TRGO signal from

TIM3. The second line configures the TIM2 to work in “slave” mode, effectively enabling the AND gate and thus the mode we need to make the frequency counter.

```
  TIM_SelectInputTrigger(TIM2, TIM_TS_ITR2);
  TIM_SelectSlaveMode(TIM2, TIM_SlaveMode_Gated);
```

The last sub-block configures the input ETR. We can configure one more pre-scaler (wo do not need it), triggering polarity (any will do), and filtering of the input signal *X* (we do not need any filtering). These requests are fulfilled by the execution of the function in the first line. The second line simply enables the TIM2, and counting of the input pulses *X* can commence.

```
  TIM_ETRClockMode2Config(TIM2, TIM_ExtTRGPSC_OFF, TIM_ExtTRGPolarity_NonInverted, 0);
  TIM_Cmd(TIM2, ENABLE);
```

c) Interrupt routine for TIM3

The standard shape of the interrupt routine is used. The name of the routine for TIM3 must be “TIM3_IRQHandler”, and must not receive or return any data. This routine gets executed immediately after the change of the timing signal *T* from low to high. In order for us to check this the second line in the listing bellow puts bit 8 of port GPIOE high, and the 7th line of the same listing returns this bit low. An oscilloscope can be used to check the presence of this signal; however, the port GPIOE, bit 8 must first be configured as general purpose output pin (we will not discuss this configuration here).

```
void TIM3_IRQHandler(void)       {
  GPIOE->ODR |=  BIT_8;
  TIM_ClearITPendingBit(TIM3, TIM_IT_CC4);
  Tim2Safe = TIM2->CNT;
  Tim2Flag = 1;
  TIM2->CNT = 0;
  GPIOE->ODR &= ~BIT_8;
}
```

The third statement in the listing above is important, it clears the bit for signaling the interrupt from TIM3. If this line is not here, the interrupt routine would not be called only on positive edge of the timing signal *T*, but would repeat continuously.

The interrupt routine is used to do three operations:

- Store the content of the TIM2 (the measured frequency) into a safe place
- Signal the main program that a new result is available, and
- Clear the content of TIM2.

All this is performed in lines 4 to 6. Since no data can enter or exit the interrupt routine all variables used must be declared as global.

d) Main program

The main program calls the appropriate functions described above to configure peripherals used (LCD dicplay, port GPIOE, TIM3 and TIM2). Once the configuration is done the main program enters an endless loop. Within the loop it checks the variable “Tim2Flag”: if the value is not 1 then new data is not available and the loop repeats. However, if the value of this variable is 1, then new data is available. The software then returns the value of this variable to 0 and prints the result of the measurement from the safe place to the LCD using a function “LCD_uInt32”.

```
void main (void)       {
  LCD_init();
  LCD_string("Frequency[Hz]:", 0x01);
  Init_GPIOE();
  Init_TIM3();
  Init_TIM2();
  while (1) {
    if (Tim2Flag == 1) {
      Tim2Flag = 0;
      LCD_uInt32(Tim2Safe, 0x45, 0x01);
    };
  };
}
```

A discussion. We could put the function to print the result on the LCD into the interrupt routine, in this case it is perfectly safe. However, a general rule in programming is that the interrupt routines are very important and must execute as soon as possible after the interrupt signal. This can make problems if more than one interrupt is issued simultaneously; the microcontroller cannot do both routines at the same time, so one of them will have to be postponed. If the execution of the interrupt routine takes long, than the postponement could be too long. In order to remedy the situation the execution time of an interrupt routine should be short. The user can, on the other side, wait for the result at the LCD. It is therefore wise to move slow functions as printing into the main program and periodically check if such an action is needed. This way there will be enough time available to execute interrupt routines on time.