US-100 - Ultrasonic distance sensor

US-100 is one of popular distance sensor used in many projects. It has ranging distance up to 450 cm with 1 mm resolution in less then 15 degree of view angle. It also has temperature sensor to compensate the result. The US-100 is very similar to the popular HC-SR04 ultrasonic sensors, but it provides UART interface beside the Pulse Width mode.

#sensor #distance #pulse #uart

Last update: 2021-06-15 11:09:09

Table of Content

1. US-100 Ultrasonic sensor

The US-100 Ultrasonic sensor is very similar to the popular HC-SR04, and even looks the same, but has a few extra tricks:

- Can run from 3V to 5V, so don't need any logic level shifters or dividers.
- Can use in "Pulse" mode (like on HC-SR04) or in "Serial UART" mode.
- Range is about 2 cm to 450 cm away, but 10 cm to 250 cm will get the best results

Ultrasonic sensor US-100

- When the jumper is in place, use an 9600 baud UART to communicate with the sensor:
  - Send `0x55` and read back two bytes (16 bit value) that is mm distance
  - Send `0x50` to read the temperature in degrees C, in offset of -45
  - This mode can work directly with PC USB to Serial adapter, and use any raw data reader on the received byte.
- When the jumper on the back is removed, it acts like an HC-SR04 with a trigger and echo pin:
  - The width of echo pulse is the time it takes for the ultrasonic sound to travel from the sensor to the object and back.
  - No temperature data is read out using the Pulse mode

> **Debug on UART1 using Redirection**
>
> For more convenient, below labs use UART Redirection technique to use the UART1 as the debug terminal, and use standard `printf()` function to output messages.

2. Lab 1: UART mode

The UART2 interface is used to communicate with US-100. As mentioned in the US-100 specification, the UART should be at **9600** baud-rate, **8** bit, **N**o parity, and **1** bit stop. Note to use the interrupt mode to receive data.

MCU Pin	US-100 Pin
PA2 (UART2TX)	Trigger/TX
PA3 (UART2RX)	Echo/RX

Create variable to hold the states, trial counter, commands, and returned value:

```
enum {
  IDLE, WAIT_DIST, CALC_DIST, WAIT_TEMP, CALC_TEMP
};
char state = IDLE;
char try = 0;
uint16_t value = 0;
uint8_t cmd_dist[] = {0x55};
uint8_t cmd_temp[] = {0x50};
uint8_t buffer[2] = {0};
```

Then, in the main loop, process each state, note to use interrupt mode to receive data:

```
int main(void) {
    while(1) {
        if (state == IDLE) {
            // send request to measure distance
            printf("D?\n\r");
            HAL_UART_Transmit(US_100, cmd_dist, 1, HAL_MAX_DELAY);
            HAL_UART_Receive_IT(US_100, buffer, 2);
            // change state
            state = WAIT_DIST;
            try = 0;
        } else if (state == CALC_DIST) {
            // calculate distance
            value = (buffer[0] << 8) + buffer[1];
            printf("D = %d mm\n\r", value);
            // send request to get temperature
            printf("T?\n\r");
            HAL_UART_Transmit(US_100, cmd_temp, 1, HAL_MAX_DELAY);
            HAL_UART_Receive_IT(US_100, buffer, 1);
            // change state
            state = WAIT_TEMP;
            try = 0;
        } else if (state == CALC_TEMP){
            // calculate temperature
            value = buffer[0] - 45;
            printf("T = %d\n\r", value);
            // change state
            state = IDLE;
            try = 0;
        }

        HAL_Delay(100);
```

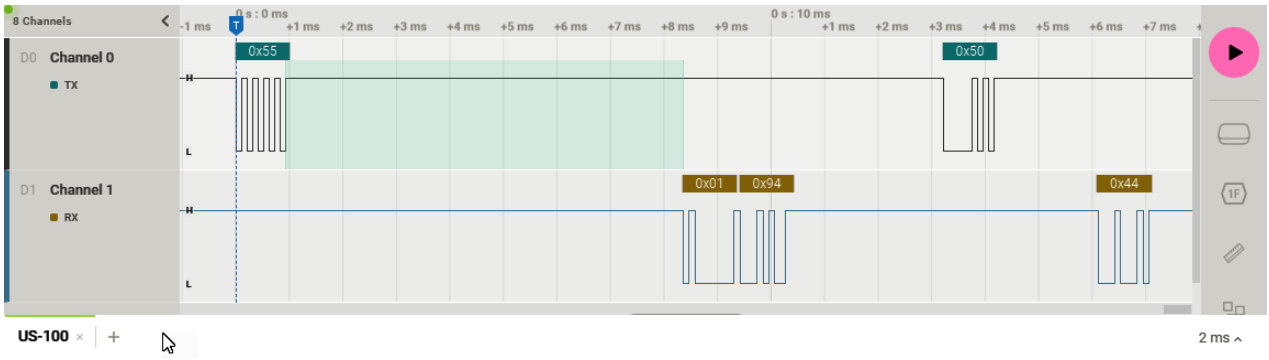
```
            // retry after 5 seconds
            if(++try >= 50) {
                printf("Re-try\n\r");
                state = IDLE;
            }
        }
    }
```

Finally, handle the interrupt callback by checking the state and set new state for the main loop:

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart) {
  if (huart == US_100) {
    if (state == WAIT_DIST) {
      state = CALC_DIST;
    } else if (state == WAIT_TEMP) {
      state = CALC_TEMP;
    }
  }
}
```

Compile and run on the board, use an logic analyser to check how fast the US-100 can response for each command, and here is the result:

- Distance response time: < 10 ms
- Temperature response time: < 5 ms



Output of US-100

And on the debug terminal, the distance and temperature are printed in decimal value:

```
T?
T = 23
D?
D = 378 mm
T?
T = 23
D?
D = 374 mm
T?
T = 23
D?
```

Print output on terminal

3. Lab 2: Pulse mode

Triggering the sensor to start operation is done by sending a short pulse to the TRIGGER pin and it should be anything wider than 5uS. It can be even a few milliseconds. The module sends an ultrasonic signal, eight pulses of 40kHz square wave from the transmitter; the echo is then picked up by the receiver and outputs a waveform with a time period proportional to the distance.

The echo response pulse corresponds to the time it takes for the ultrasonic sound to travel from the sensor to the object and back. Hence, the distance is computed as:

```
Distance = Pulse Width * Speed of Sound / 2 (m)
```

The actual speed of sound depends on the several environment factors, with temperature having most pronounced effect:

```
Speed of Sound = 331.4 + 0.6T (m/s)
```

US-100 has built-in temperature compensation, so the distance formula is reduced to:

```
Distance = Pulse width * 165.7 (m)
```

The pulse with is calculated by the timer counter divided by the counting frequency:

```
Pulse width = Timer counter / Frequency
```

Therefore, the final equation is:

```
Distance = Timer counter / Frequency * 165.7 (m)
         = Timer counter * 165700 / Frequency (mm)
```

3.1. Methods to read a pulse width.

GPIO Polling + Timer

A very basic technique is to keep polling a GPIO input pin. The MCU will keep waiting until this pin goes HIGH, then it turns ON a timer module to start counting. And keep polling the input pin until it goes LOW, then the timer is turned OFF. The timer counter value will tell the echo pulse width.

⚠ Polling the GPIO input pin is a time-wasting procedure that has a potential risk of freezing the entire system in case of sensor failure or whatever.

Ext Interrupt + Timer

An EXTI pin will be set to wait for a rasing edge to start an internal timer. After the edge is captured, that will will be set to wait for falling edge, while timer is counting. The 2^{nd} interrupt will stop the timer, and the timer counter value will tell the echo pulse width.

Timer Input Capture

Use a Timer with Input Capture mode to capture the timer value at the rising edge of the input pin, then capture the timer value at the falling edge. The different value will tell the echo pulse width.

Differential Double Input Capture

When measuring extremely short pulses, differential double IC provides accuracy and precision. Use 2 ICU channels: one triggers on the rising edge and the other triggers on the falling edge. The different value will tell the echo pulse width.

Timer Gate-Controlled

One technique that also works really well in extremely short pulse measurements is timer gate-controlled. In this specific mode, the timer is allowed to count only when the gate is activated. The gate is driven by the input pin connected to the echo pin. The timer counter value will tell the echo pulse width.

3.2. Use Input Capture

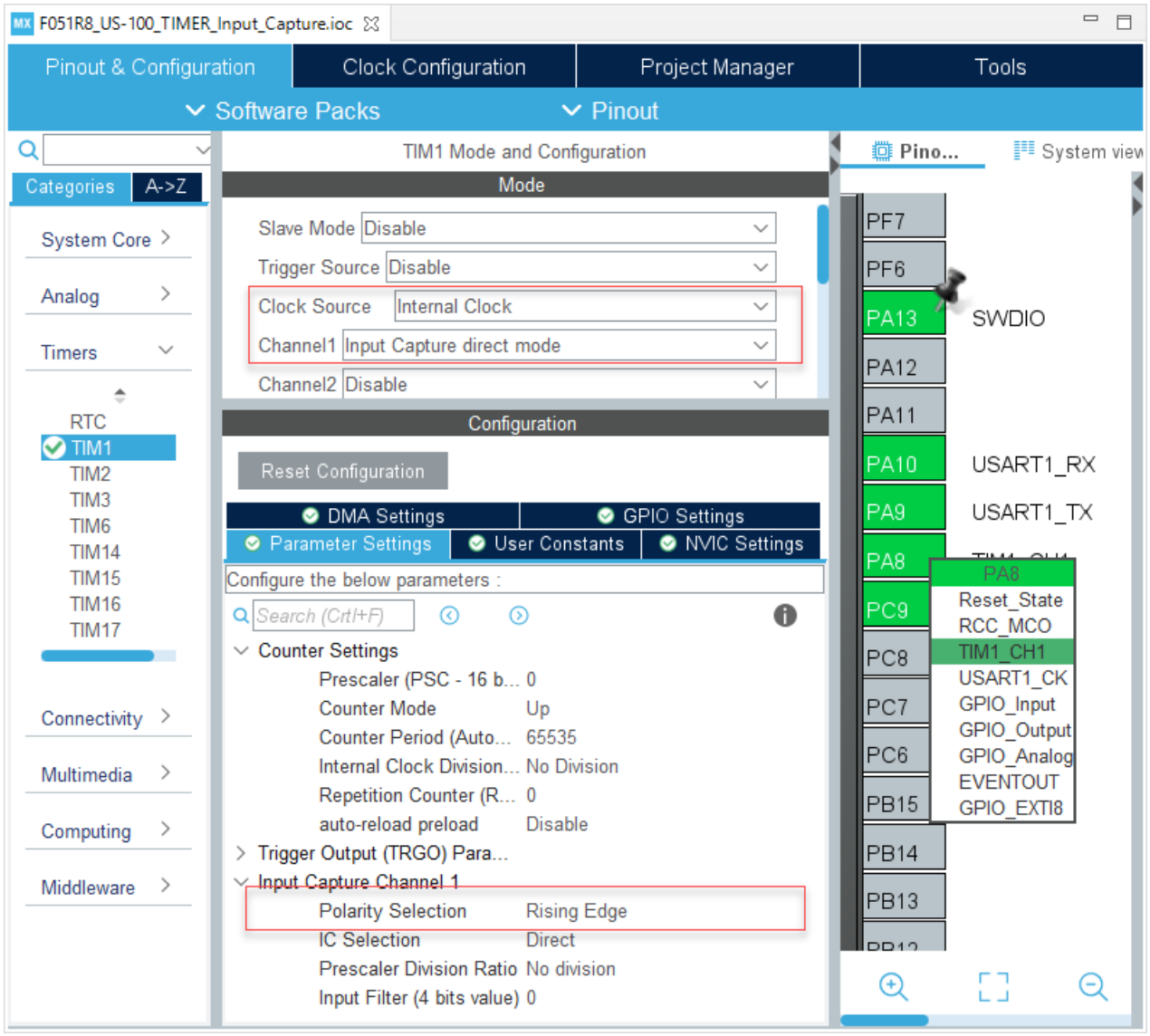
This method will use the TIM1 with Input Capture on the Channel 1. Here are steps to capture the Echo pulse:

1. Enable TIM1 using Internal Clock source, Enable Input Capture on Channel 1
2. For the best resolution, do not set the Pre-scaler, and let timers counts in full range of 16-bit
3. Set the capture edge at the Rising edge
4. Start TIM1, Start Input Capture on Channel 1 in Interrupt mode
5. When Echo pulse goes high, handle the interrupt:
    a. Save the timer counter to T1
    b. Set the capture edge at the Falling edge
6. When Echo pulse goes low, handle the interrupt:
    a. Save the timer counter to T2
        i. Different time is the width of the pulse
        ii. Distance is calculated based on the pulse width and the tick interval of the clock
    b. Set the capture edge at the Rising edge to capture another pulse

3.2.1. Enable Timer and Input Capture

Using IDE to enable the TIM1 in system peripherals:

- Select internal clock source, and choose Channel 1 to *Input Capture direct mode*, which also captures a Rising edge at startup.
- Enable interrupts for both *break, update, trigger, communication* and *capture compare*



Setup Timer with Input Capture

The pin wiring also needs to change:

MCU Pin	US-100 Pin
PA1 (Output)	Trigger/TX
PA8 (TIM1_CH1 input capture)	Echo/RX

MCU Pin	US-100 Pin
PC9 (Output)	Timer Capture
PC8 (Output)	Timer Overflow

3.2.2. Generated code

The function `MX_TIM1_Init()` is generated with below steps to setup the selected configs:

1. Initialize TIM1 Base, including Pre-scaler, Counter Period, and AutoReload
2. Select the clock source
3. Initialize Input Capture mode and its settings: polarity, edge, filter

```
static void MX_TIM1_Init(void) {
  TIM_ClockConfigTypeDef sClockSourceConfig = {0};
  TIM_MasterConfigTypeDef sMasterConfig = {0};
  TIM_IC_InitTypeDef sConfigIC = {0};

  htim1.Instance = TIM1;
  htim1.Init.Prescaler = 0;
  htim1.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim1.Init.Period = 65535;
  htim1.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim1.Init.RepetitionCounter = 0;
  htim1.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
  if (HAL_TIM_Base_Init(&htim1) != HAL_OK) {
    Error_Handler();
  }

  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim1, &sClockSourceConfig) != HAL_OK) {
    Error_Handler();
  }

  if (HAL_TIM_IC_Init(&htim1) != HAL_OK) {
    Error_Handler();
  }

  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim1, &sMasterConfig)
      != HAL_OK) { Error_Handler(); }

  sConfigIC.ICPolarity = TIM_INPUTCHANNELPOLARITY_RISING;
  sConfigIC.ICSelection = TIM_ICSELECTION_DIRECTTI;
  sConfigIC.ICPrescaler = TIM_ICPSC_DIV1;
  sConfigIC.ICFilter = 0;
  if (HAL_TIM_IC_ConfigChannel(&htim1, &sConfigIC, TIM_CHANNEL_1)
      != HAL_OK) { Error_Handler(); }
}
```

Two interrupt handlers are generated too, `TIM1_BRK_UP_TRG_COM_IRQHandler()` and `TIM1_CC_IRQHandler()` which will call to:

- `HAL_TIM_IC_CaptureCallback()` : interrupt when input capture meets the capture edge
- `HAL_TIM_PeriodElapsedCallback()` : interrupt when counter finishes one cycle (from 0 to AutoReload value)

3.2.3. User code

There are some variables to hold the system states, timer counter values, and distance value.

```
enum {
  IDLE, WAIT, CALC
};
int state = IDLE;
int try = 0;
uint32_t T1 = 0;
uint32_t T2 = 0;
uint32_t overflow = 0;
uint32_t period = 0;
uint32_t counter = 0;
uint32_t distance = 0;
char edge = 1; // raising
```

In the `main()` function, save the clock period, and start both base timer's interrupt and input capture's interrupt:

```
int main() {
    period = __HAL_TIM_GET_AUTORELOAD(&htim1);
    HAL_TIM_Base_Start_IT(&htim1); // to get PeriodElapsedCallback
    HAL_TIM_IC_Start_IT(&htim1, TIM_CHANNEL_1); // to get IC_CaptureCallback
    while(1) {...}
}
```

Save capture time

Application has to handle interrupts to save timer counter at each edge, to get overflow counter, and set correct state of system.

```
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim) {
  if (state == WAIT) {
    if (edge) {
      T1 = HAL_TIM_ReadCapturedValue(&htim1, TIM_CHANNEL_1);
      __HAL_TIM_SET_CAPTUREPOLARITY(&htim1, TIM_CHANNEL_1,
          TIM_INPUTCHANNELPOLARITY_FALLING);
      overflow = 0; // start check if timer is overflow
    } else {
      T2 = HAL_TIM_ReadCapturedValue(&htim1, TIM_CHANNEL_1);
      __HAL_TIM_SET_CAPTUREPOLARITY(&htim1, TIM_CHANNEL_1,
          TIM_INPUTCHANNELPOLARITY_RISING);
```

```
      state = CALC;
    }
    edge = !edge;
  }
  HAL_GPIO_TogglePin(LED_CC_GPIO_Port, LED_CC_Pin);
}

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
  if (state == WAIT) {
    overflow++;
  }
  HAL_GPIO_TogglePin(LED_OV_GPIO_Port, LED_OV_Pin);
}
```

Main loop

The main loop checks the current state and do corresponding actions.

> ⚠ **Overflow when calculating with big number**
>
> In Embedded system, calculating with big number can cause overflow and returns wrong result. In this tutorial, timer counter is a 32-bit number, but when it is multiplied with 165700, it can cause overflow.
>
> ```
> Distance = Timer counter / Frequency * 165.7 (m)
>          = Timer counter * 165700 / Frequency (mm)
> ```

```
while (1)
{
    if (state == IDLE) {
        // trigger
        HAL_GPIO_WritePin(TRIGGER_GPIO_Port, TRIGGER_Pin, GPIO_PIN_SET);
        HAL_Delay(1);
        HAL_GPIO_WritePin(TRIGGER_GPIO_Port, TRIGGER_Pin, GPIO_PIN_RESET);
        // change state
        state = WAIT;
        try = 0;
    } else if (state == CALC) {
        // use overflow in case pulse occurs when timer counter is overflow
        T2 += overflow * period;
        counter = T2 - T1;
        printf("T =  %lu ~ %lu us\t", counter, counter * 1000 / 48000);
        // calc. distance
        distance = counter * 1657;
        distance /= (SystemCoreClock/100);
        printf("D = %lu mm\r\n", distance);
        // change state
        state = IDLE;
        try = 0;
    }

    HAL_Delay(100);
```

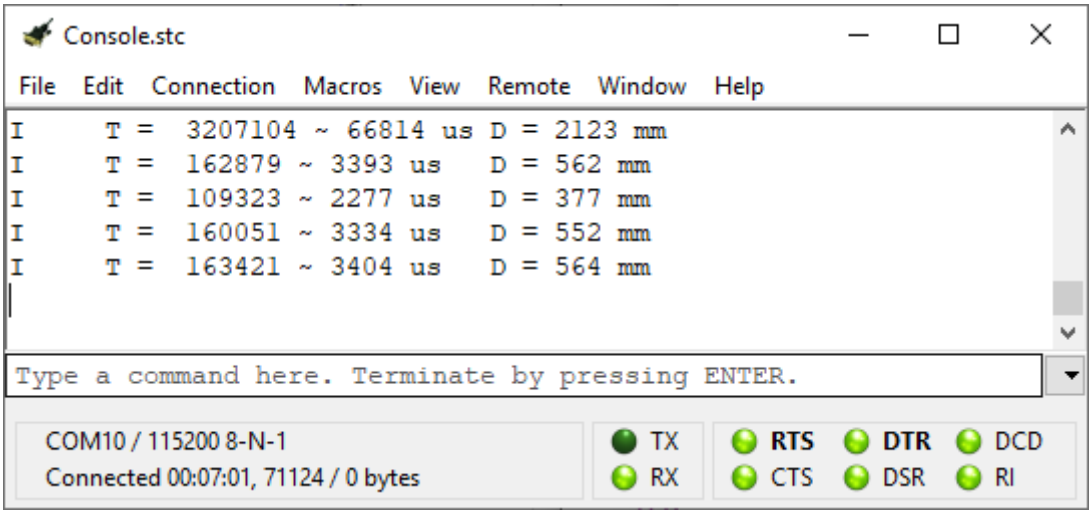
```
        if(++try > 10) {
            state = IDLE;
        }
    }
```

Toggling a pin helps showing when interrupts happen.

Here is the output seen in a logic analyser in a case that timer counter is overflow.

Output of sensor is captured in timer

Using an UART port to print the calculated distance in a terminal:



Output of calculated distance