Пьезоэлектрический излучатель. Синхронизация таймеров

Мы уже научились генерировать мелодию, но сделали это весьма топорно: после запуска функции `buzzer_play()` нет никакой возможности заставить устройство замолчать не отключая питания. Все остальные операции, кроме прерываний, на устройстве так же будут не возможны. Что же делать?

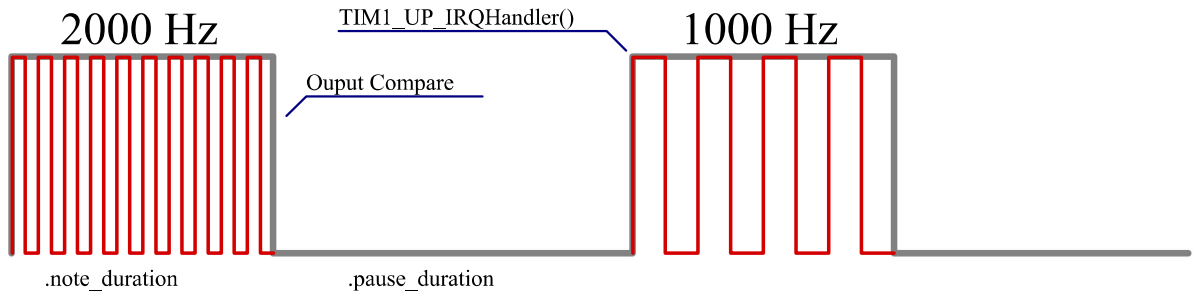
Можно добавить флаг, ориентируясь на который продолжать воспроизведение или выходить из функции. Но это плохое решение. Другой, довольно очевидный вариант настроить другой таймер и в прерывании по его переполнению переключаться с паузы на ноту, попутно изменяя параметры ШИМа и отмеряемого времени. Реализация не самая элегантная, но рабочая.

Можно немного уменьшить количество кода, убрать проверку что именно сейчас должно «звучать» — нота или пауза, ведь в STM32 таймеры можно синхронизировать — в зависимости от настроек они могут работать как в независимом (англ. master, то что мы делали раньше), так и в зависимом (англ. slave) режиме.

> Ведущий таймер может сбрасывать, запускать, останавливать или тактировать счётчик ведомого таймера.

Связать таймеры мы можем только по одному событию-триггеру. Подумаем по какому.

Пусть таймер генерирующий звук, `TIM2`, будет настроен как подчинённый, тогда другой, допустим `TIM1` будет ведущим. Так как нас интересуем ШИМ в ШИМ, то `TIM1` следует так же настроить на его генерацию, но без вывода оного на ножку МК (Output Compare, No output, PWM Mode 1). Полный период можно легко найти — это сумма `.note_duration` и `.pause_duration`, а заполнение ШИМ, как не сложно догадаться это просто `.note_duration`. У ШИМ, есть всего два события: output compare (т.е. достижение счётчиком значения из регистра `CCER3`) и переполнение. Пусть по переполнению срабатывает прерывание, в котором мы меняем параметры обоих таймеров, а вот Output Compare будет отправляться сигнал подчинённому таймеру на сброс.



Но почему мы выбрали именно `TIM1`? Вы не можете связать любой таймер с любым — чисто на физическом уровне это сложная пространственная задача: внутри чипа тоже есть дорожки. По этой причине в микроконтроллерах ST вы можете связать любой с группой других таймеров по слотам. Ниже приведена таблица из Reference Manual.

Slave TIM	ITR0 (TS = 000)	ITR1 (TS = 001)	ITR2 (TS = 010)	ITR3 (TS = 011)
TIM2	TIM1	TIM8	TIM3	TIM4
TIM3	TIM1	TIM2	TIM5	TIM4
TIM4	TIM1	TIM2	TIM3	TIM8
TIM5	TIM2	TIM3	TIM4	TIM8

Reference Manual, Table 86. TIMx Internal trigger connection

`TIM8` нет в нашем МК, а `TIM3` и `TIM4` мы используем для обработки поворота энкодера. К слову и `TIM1` мы уже успели задействовать, при длительном удержании кнопки, но так как с таймерами у нас дефицит, придётся их использовать по второму разу.

> Это непременно приведёт багу в прошивке. Что если во время проигрывания мелодии кто-то нажмёт на кнопку? Оставим обработку данной ситуации на вашей совести.

За настройку таймера в режим мастера отвечают биты `MMS` регистра `CR2`; так как мы будем использовать первый канал `TIM1` для генерации ШИМ, то в `MMS` следует записать `110`. Обратитесь к документации. Запись `1` в бит `MSM` регистра `SMCR` обеспечивает точную синхронизацию между входом и выходом триггера. Для удобства, таймер `TIM1` можно настроить на режим одиночного импульса (One Pulse Mode) — если мы дошли до последней ноты, нам не придётся его отключать.

За настройку подчинённого режима отвечает уже известная нам (энкодер) группа бит `SMS` из регистра `SMCR`. Записав `100` по приходу сигнала на TRGI таймер будет сбрасываться. Указать источник можно через группу бит `TS`. Так как нам нужен `ITR0`, в `TS` следует записать `000` (см. таблицу выше).

Преступим к реализации и для начала перепишем функцию инициализации.

```
void buzzer_init(const uint8_t vol) {
    volume = vol;

    // RCC ------------------------------------------------------------------
    RCC->APB2ENR |= RCC_APB2ENR_IOPAEN| RCC_APB2ENR_AFIOEN | RCC_APB2ENR_TIM1EN;
    RCC->APB1ENR |= RCC_APB1ENR_TIM2EN;

    // GPIOA, pin 2, alternative function, push-pull, 50 MHz ----------------
    GPIOA->CRL |= GPIO_CRL_MODE;
    GPIOA->CRL |= GPIO_CRL_CNF2_1;
    GPIOA->CRL &= ~GPIO_CRL_CNF2_0;

    // TIM2 base ------------------------------------------------------------
    TIM2->CR1 &= ~(TIM_CR1_DIR | TIM_CR1_CMS);
    TIM2->ARR = 0;
    TIM2->PSC = 63;

    // TIM2 OC --------------------------------------------------------------
    TIM2->CCMR2 &= ~TIM_CCMR2_OC3M;
    TIM2->CCMR2 |= (TIM_CCMR2_OC3M_1 | TIM_CCMR2_OC3M_2); // PWM mode 1
    TIM2->CCMR2 &= ~TIM_CCMR2_CC3S; // CC3 channel is configured as output

    // Select the Output Compare Mode
    TIM2->CCER &= ~TIM_CCER_CC3P; // low polarity level
```

```
    TIM2->CCER |= TIM_CCER_CC3E;  // enable outout state
    TIM2->CCR3 = 0;               // compare

    // Slave Mode selection: TIM2 -------------------------------------------
    TIM2->SMCR &= ~TIM_SMCR_SMS;
    TIM2->SMCR |= TIM_SMCR_SMS_2; // slave mode: reset
    TIM2->SMCR &= ~TIM_SMCR_TS;   // ITR0 TIM1 -> TIM2

    // TIM1 base ------------------------------------------------------------
    TIM1->CR1 &= ~(TIM_CR1_DIR | TIM_CR1_CMS);
    TIM1->ARR = 0;
    TIM1->PSC = 63999;
    TIM1->RCR = 0;
    TIM1->EGR |= TIM_EGR_UG; // reinit the counter

    TIM1->CR1 |= TIM_CR1_OPM; // one pulse mode

    // TIM1 OC1 -------------------------------------------------------------
    TIM1->CCMR1 &= ~TIM_CCMR1_OC1M; // Select the Output Compare Mode
    TIM1->CCMR1 |= (TIM_CCMR2_OC3M_1 | TIM_CCMR2_OC3M_2); // PWM mode 1
    TIM1->CCMR1 &= ~TIM_CCMR1_CC1S;
    TIM1->CCER |= TIM_CCER_CC1P; // low polarity level
    TIM1->CCER |= TIM_CCER_CC1E; // enable outout state
    TIM1->CR2 &= ~TIM_CR2_OIS1;  // idle low
    TIM1->CCR1 = 0;              // compare

    // Master Mode selection
    TIM1->CR2 &= ~TIM_CR2_MMS;
    TIM1->CR2 |= TIM_CR2_MMS_2; // OC1REF

    // Select the Master Slave Mode
    TIM1->SMCR |= TIM_SMCR_MSM;  // set

    TIM1->DIER |= TIM_DIER_UIE;
    NVIC_EnableIRQ(TIM1_UP_IRQn);
}
```

Так как мелодий может быть несколько, допустим банально мы хотим добавить «бип», то надо как-то давать знать функции `buzzer_play()`, что именно мы хотим проиграть. Создадим под это дело структуру.

```
// buzzer.h
typedef struct {
    uint32_t        non; // number of notes
    uint32_t         cn; // current note
    const TONE_t *notes;
} MELODY_t;
```

В поле `.non` хранится количество нот в мелодии, а в `.cn` текущая проигрываемая нота. Осталось создать нужные мелодии.

```
// buzzer.h
// list of melodies
```

```
MELODY_t beep;
MELODY_t peddler;

// buzzer.c
static const TONE_t __beep[] = {
        { 100, 0, FREQ_A_3, },
};

MELODY_t beep = {
        .non = sizeof(__beep) / sizeof(__beep[0]),
        .cn  = 0,
        .notes = __beep,
};

MELODY_t peddler = {
        .non = sizeof(__peddler) / sizeof(__peddler[0]),
        .cn  = 0,
        .notes = __peddler,
};
```

Для воспроизведения потребуется отдельная переменная, в которой мы будем хранить текущую, выбранную мелодию, а в функции воспроизведения мы просто будем её запоминать.

```
MELODY_t melody;

void buzzer_play(MELODY_t mel) {
    melody = mel;
    buzzer_turn_on();
}
```

Как только мы запустим таймеры, они сразу же начнут воспроизводить звук, согласно своим настройкам. Поэтому, чтобы избежать чего-то неожиданного, придётся записать значения в регистры `ARR` и `CCRx` значения из первой ноты. А так как в ней появились значения локальных (для модуля) переменных, то сделать её встроенной мы больше не можем, поэтому перенесём её в файл исходного кода.

```
// buzzer.h
void buzzer_turn_on(void);

// buzzer.c
void buzzer_turn_on(void) {
    SET_FREQ(melody.notes[0].note_freq);
    TIM2->CR1 |= TIM_CR1_CEN;

    TIM1->ARR  = melody.notes[0].note_duration + melody.notes[0].pause_duration;
    TIM1->CCR3 = melody.notes[0].note_duration;
    TIM1->CR1 |= TIM_CR1_CEN;
}
```

Добавим обработчик прерывания: в нём, если ноты ещё остались, то мы настраиваем ШИМ-ы на нужные параметры и перезапускаем таймер. В противном случае вызываем функцию `buzzer_timer_off()`.

```
// buzzer.h
__attribute__((always_inline)) inline void buzzer_turn_off(void) {
```

```
    TIM2->CR1 &= ~TIM_CR1_CEN;
    TIM1->CR1 &= ~TIM_CR1_CEN;
}

// stm32f10x.h
#include "buzzer.h"

void TIM1_UP_IRQHandler(void);

// stm32f10x.c
extern MELODY_t melody; // dirty

void TIM1_UP_IRQHandler(void) {
    if (++melody.cn < melody.non) {
        uint32_t freq = 1e6 / melody.notes[melody.cn].note_freq;
        TIM2->ARR  = freq - 1;     // set freq
        TIM2->CCR3 = freq / 2 - 1; // 50 %

        TIM1->CCR1 = melody.notes[melody.cn].note_duration;
        TIM1->ARR = TIM1->CCR1 + melody.notes[melody.cn].pause_duration;

        TIM1->CR1 |= TIM_CR1_CEN;
    } else {
        buzzer_turn_off();
    }

    TIM1->SR &= ~TIM_SR_UIF;
}
```

Переменную `melody`, по-хорошему следовало бы обернуть в функции `get()` / `set()`, но для ускорения работы лучше дать прямой доступ к ней. Это не очень хорошо, но в данном случае ничего страшного не произойдёт.

> Заметьте, мы не добавили регулировку громкости, сделайте это сами.

Осталось вызвать функцию воспроизведения.

```
// main.c
int main(void) {
    mcu_init(); // --> buzzer_init(127);

    buzzer_play(mario);
    while (1) {
    }
}
```

Код на можно найти на GitHub: **CMSIS**, SPL.

Назад | **Оглавление** | **Дальше**