УНИВЕРСИТЕТ ИТМО

Факультет программной инженерии и компьютерной техники

Направление подготовки 09.03.04 Программная инженерия

Дисциплина «Проектирование вычислительных систем»

**Лабораторная работа №3**

Вариант 15

Студент

*Крюков А. Ю.*

*Патутин В. М.*

*P34101*

Преподаватель

*Пинкевич В. Ю.*

Санкт-Петербург, 2022 г.

Цель работы

1. Получить навыки использования прерываний от таймеров и аппаратных каналов ввода-вывода таймеров.

2. Изучить интерфейс I2C (I2C, inter-integrated circuit, т.е. межмикросхемный интерфейс) и особенности передачи данных по данному интерфейсу.

3. Изучить устройство и принципы работы контроллера интерфейса I2C в STM32 и получить навыки его программирования.

Задание лабораторной работы

Разработать программу, которая включает драйвер клавиатуры. Драйвер может быть организован одним из трех основных способов:

1. Работа с I2C по опросу (без прерываний от I2C). Опрос клавиатуры периодически выполняется из главного цикла while() в функции main(). При этом остальной код главного цикла должен отдавать управление достаточно быстро, чтобы не создавать заметных пауз в опросе клавиатуры.

2. Работа с I2C по прерываниям в главном цикле. Аналогично предыдущему, функции опроса клавиатуры вызываются из главного цикла, но используются функции, работающие в режиме прерывания. Для получения результата надо дожидаться окончания выполнения транзакций (их можно отслеживать с помощью callback-функций).

3. Работа с I2C по прерываниям в отдельном потоке. Помимо главного цикла, программа должна иметь еще один поток управления, который отвечает за опрос клавиатуры. Для этого необходимо настроить один из таймеров микроконтроллера так, чтобы он регулярно генерировал прерывания в режиме автоперезагрузки. В обработчике прерывания от данного таймера необходимо генерировать посылки для обмена по шине I2C в режиме прерываний. Таким образом, полный опрос клавиатуры требует 8 прерываний от таймера. Период между прерываниями должен обеспечивать достаточный запас времени на выполнение транзакций I2C и работу основного цикла. При этом регистрация нажатия должна происходить достаточно быстро, чтобы пользователь не чувствовал задержки отклика. Оценить время, необходимое на выполнение транзакции, достаточно просто, так как известно время передачи одного бита и количество данных. При частоте шины I2C в 400 кГц один бит передается за 2,5 мкс, а транзакция в 4 байта занимает не более 0,1 мс. Начальная инициализация устройств I2C (если требуется) может быть сделана до того, как будет запущен процесс опроса клавиатуры. При таком способе опроса зафиксированные события нажатия кнопок помещаются в программный FIFO-буфер, из которого их может считывать процесс (поток управления), реализуемый главным циклом функции main(); доступ к буферу должен быть защищен от состояния гонки в моменты модификации указателей критическими секциями с запретами прерываний (см. описание лабораторной работы №2).

Подсистема опроса клавиатуры должна удовлетворять следующим требованиям:

– реализуется защита от дребезга;

– нажатие кнопки фиксируется сразу после того, как было обнаружено, что кнопка нажата (с учетом защиты от дребезга), а не в момент отпускания кнопки; если необходимо, долгое нажатие может фиксироваться отдельно;

– кнопка, которая удерживается дольше, чем один цикл опроса, не считается повторно нажатой до тех пор, пока не будет отпущена;

– распознается и корректно обрабатывается множественное нажатие (при нажатии более чем одной или двух кнопок считается, что ни одна кнопка не нажата); – всем кнопкам назначаются коды от 1 до 12 (порядок на усмотрение исполнителей).

В главном цикле функции main() выполняется основной поток управления, который может работать в двух режимах, переключение между которыми производится по нажатию кнопки на боковой панели стенда:

– режим тестирования клавиатуры;

– прикладной режим.

Уведомление о смене режима выводится в UART.

В режиме тестирования клавиатуры программа выводит в UART коды нажатых кнопок.

В прикладном режиме программа обрабатывает нажатия кнопок и выполняет действия в соответствии с вариантом задания.

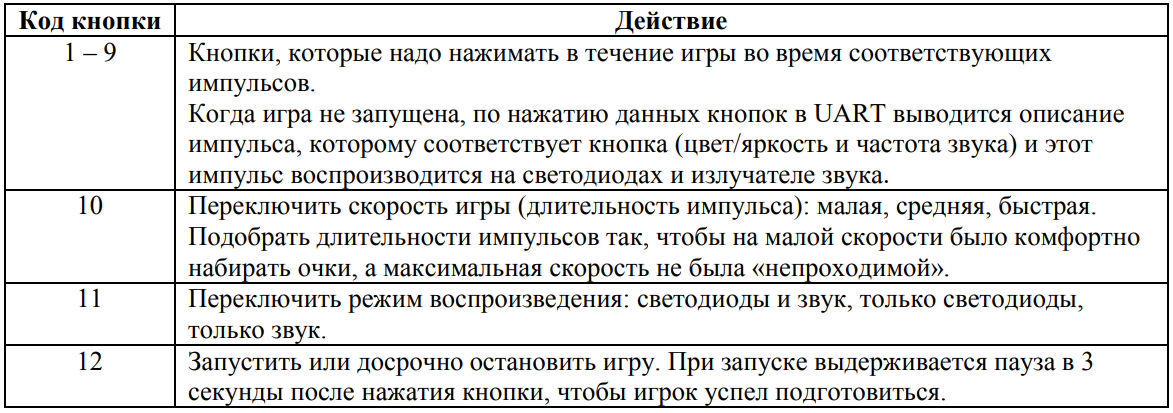
Управление излучателем звука и яркостью светодиодов должно выполняться с помощью таймеров (по прерыванию или с использованием аппаратных каналов). Блокирующее ожидание (функция HAL\_Delay()) использоваться не должно.

Вариант задания

Реализовать музыкальную ритм-игру. На светодиодах воспроизводится последовательность импульсов разного цвета и яркости, сопровождающаяся звуками, частота которых соответствует комбинации цвета и яркости (должно быть однозначное соответствие между мелодией и последовательностью световых импульсов). Во время каждого импульса игрок должен нажимать кнопки, соответствующие текущему цвету/яркости или частоте звука. Чем больше кнопок будет нажато правильно и на большей скорости игры, тем больше очков заработает игрок (система начисления очков – на усмотрение исполнителей).

Всего необходимо предусмотреть 9 видов импульсов: зеленый, желтый и красный на 20 %, 50 % и 100 % яркости. К ним следует подобрать звуки произвольных частот, легко отличимых одна от другой на слух. Предусмотреть одну стандартную последовательность импульсов длительностью не менее 20 элементов. Когда последовательность заканчивается или досрочно останавливается игроком, в UART выводится количество набранных очков и «трассировка» нажатий, где отмечены правильные и неправильные нажатия. Отсутствие нажатия считается неправильным нажатием.

По нажатию кнопок клавиатуры выполняются следующие действия:



По нажатию каждой кнопки в UART должно выводиться сообщение о том, какой импульс выбран или какой режим игры установлен.

Исходный код

#include "Game/music.h"

static char output\_buffer[256];

const sound ut = { .note = C, .color = GREEN, .brightness = FULL, .key = KEY\_1 };

const sound re = { .note = D, .color = GREEN, .brightness = HALF, .key = KEY\_2 };

const sound mi = { .note = E, .color = GREEN, .brightness = LOW, .key = KEY\_3 };

const sound fa = { .note = F, .color = RED, .brightness = FULL, .key = KEY\_4 };

const sound sol = { .note = G, .color = RED, .brightness = HALF, .key = KEY\_5 };

const sound la = { .note = A, .color = RED, .brightness = LOW, .key = KEY\_6 };

const sound si = { .note = B, .color = YELLOW, .brightness = FULL, .key = KEY\_7 };

const sound ut\_2 = { .note = C\_2, .color = YELLOW, .brightness = HALF, .key = KEY\_8 };

const sound re\_2 = { .note = D\_2, .color = YELLOW, .brightness = LOW, .key = KEY\_9 };

const sound pause = { .note = -1, .color = -1, .brightness = -1, .key = -1 };

const sound all\_sounds[] = { ut, re, mi, fa, sol, la, si, ut\_2, re\_2 };

music create\_music(size\_t size) {

music music = {};

music.size = size;

music.current = 0;

music.commands = malloc(size \* sizeof(player\_command));

return music;

}

void add\_note(music \*music, sound sound) {

if (music->current < music->size) {

player\_command \*command = &(music->commands[music->current]);

command->command = NOTE;

command->note.sound = sound;

music->current += 1;

}

}

void add\_pause(music \*melody, uint32\_t timeout) {

if (melody->current < melody->size) {

player\_command \*command = &(melody->commands[melody->current]);

command->command = PAUSE;

command->pause.timeout = timeout;

melody->current += 1;

}

}

sound find\_sound\_for\_key(key key) {

for (size\_t i = 0; i < sizeof(all\_sounds) / sizeof(all\_sounds[0]); i++) {

if (all\_sounds[i].key == key) {

return all\_sounds[i];

}

}

return pause;

}

char\* sound\_to\_string(sound sound) {

sprintf(output\_buffer, "%s / %s / %d Hz\n\r",

get\_led\_color\_name(sound.color),

get\_led\_brightness\_name(sound.brightness), sound.note);

return output\_buffer;

}

void free\_music(music melody) {

free(melody.commands);

melody.size = 0;

melody.current = 0;

}

#include "Game/logic.h"

#define PLAY\_TIME\_MS 2000

#define START\_TIMEOUT\_MS 3000

#define SPEED\_NORM 15

static void execute\_command();

static void play\_sound(sound);

static void play\_melody();

static sound get\_current\_sound();

static void change\_output\_mode();

static void toggle\_game();

static void initialize\_score();

static void add\_score(bool);

static void change\_game\_speed();

static bool lights = true;

static bool speaker = true;

static music \*melody;

static int i = 0;

static bool processing = false;

static bool pressed = false;

static uint32\_t timeout = 0;

static uint32\_t score = 0;

static uint32\_t modifier = 0;

static mode md = PLAY;

static speed sp = MEDIUM;

void set\_game\_melody(music\* m) {

melody = m;

}

void on\_key\_press(key key) {

switch(key){

case KEY\_1: case KEY\_2: case KEY\_3:

case KEY\_4: case KEY\_5: case KEY\_6:

case KEY\_7: case KEY\_8: case KEY\_9:

{

sound s = find\_sound\_for\_key(key);

print\_string(sound\_to\_string(s));

if (md == PLAY) {

pressed = true;

play\_sound(s);

} else {

sound cur = get\_current\_sound();

if (!pressed) {

add\_score(cur.key == key);

}

}

break;

}

case KEY\_A:

change\_output\_mode();

break;

case KEY\_PLUS:

change\_game\_speed();

break;

case KEY\_ENTER:

toggle\_game();

break;

case NO\_KEY: default:

break;

}

}

void on\_game\_timeout() {

if (processing) {

timeout--;

if (timeout == 0) {

if(!pressed) add\_score(false);

disable\_all\_leds();

stop\_playing();

processing = false;

if(game\_in\_process()) {

i++;

execute\_command();

}

}

}

}

bool game\_in\_process() {

return md == GAME;

}

uint32\_t get\_game\_score() {

return score;

}

static void play\_melody() {

i = -1;

timeout = START\_TIMEOUT\_MS;

initialize\_score();

pressed = true;

print\_game\_started();

processing = true;

}

static void execute\_command() {

if (i < melody->size) {

pressed = false;

player\_command command = melody->commands[i];

if (command.command == NOTE) {

sound sound = command.note.sound;

play\_sound(sound);

} else {

timeout = command.pause.timeout \* sp / SPEED\_NORM;

pressed = true;

}

processing = true;

} else {

md = PLAY;

print\_game\_finished();

}

}

static void play\_sound(sound s) {

disable\_all\_leds();

if (speaker) play\_note(s.note);

if (lights) enable\_led(s.color, s.brightness);

timeout = PLAY\_TIME\_MS \* sp / SPEED\_NORM;

processing = true;

}

static sound get\_current\_sound() {

player\_command command = melody->commands[i];

return command.command == NOTE ? command.note.sound : pause;

}

static void change\_output\_mode() {

if (speaker && lights) {

speaker = false;

print\_string("Lights only mode\n\r");

} else if(lights) {

lights = false;

speaker = true;

print\_string("Speaker only mode\n\r");

} else {

lights = true;

print\_string("Speaker and lights mode\n\r");

}

}

static void change\_game\_speed() {

switch(sp){

case FAST:

sp = SLOW;

print\_string("Slow mode\n\r");

break;

case MEDIUM:

sp = FAST;

print\_string("Fast mode\n\r");

break;

case SLOW:

sp = MEDIUM;

print\_string("Medium mode\n\r");

break;

default:

sp = MEDIUM;

break;

}

}

static void toggle\_game() {

if (md == PLAY) {

md = GAME;

play\_melody();

} else {

md = PLAY;

processing = false;

disable\_all\_leds();

stop\_playing();

print\_game\_finished();

}

}

static void initialize\_score() {

score = 0;

modifier = 1;

}

static void add\_score(bool corrent) {

if (corrent) {

print\_string("Correct guess!\n\r");

score += modifier;

modifier++;

} else {

print\_string("Wrong guess.\n\r");

modifier = 1;

}

}

#include "Utils/speaker.h"

#include "tim.h"

void play\_note(note note) {

HAL\_TIM\_PWM\_Stop(&htim1, TIM\_CHANNEL\_1);

htim1.Instance->ARR = 1.0 / note \* 1000000; //

htim1.Instance->CCR1 = htim1.Instance->ARR / 2;

HAL\_TIM\_PWM\_Start(&htim1, TIM\_CHANNEL\_1);

}

void stop\_playing() {

HAL\_TIM\_PWM\_Stop(&htim1, TIM\_CHANNEL\_1);

}

Вывод

Во время выполнения лабораторной работы мы изучили работу таймеров в STM32 и применили полученные знания на практике, разработав программу, использующую таймеры и аппаратные каналы ввода – вывода таймеров.