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:

Код кнопки	Действие
1 – 9	Кнопки, которые надо нажимать в течение игры во время соответствующих импульсов. Когда игра не запущена, по нажатию данных кнопок в UART выводится описание импульса, которому соответствует кнопка (цвет/яркость и частота звука) и этот импульс воспроизводится на светодиодах и излучателе звука.
10	Переключить скорость игры (длительность импульса): малая, средняя, быстрая. Подобрать длительности импульсов так, чтобы на малой скорости было комфортно набирать очки, а максимальная скорость не была «непроходимой».
11	Переключить режим воспроизведения: светодиоды и звук, только светодиоды, только звук.
12	Запустить или досрочно остановить игру. При запуске выдерживается пауза в 3 секунды после нажатия кнопки, чтобы игрок успел подготовиться.

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.