**Федеральное государственное бюджетное образовательное учреждение высшего образования "Белгородский государственный технологический университет им. В.Г. Шухова"**

Кафедра программного обеспечения вычислительной техники и

автоматизированных систем.

**Добавление своего эл****емента в онлайн-симуляторе Wokwi**

Выполнил:

Студент группы КБ-211

Коренев Д.Н.

Принял:

Шамраев А.А.

**Оглавление**

[0. Введение. Используемая терминология 2](#_Toc151330399)

[0.1. Терминология 2](#_Toc151330400)

[1. Добавление нового элемента 2](#_Toc151330401)

[1.1. Файл .json 2](#_Toc151330402)

[1.1.1. Поле “pins” 2](#_Toc151330403)

[1.1.2. Поле “controls” 2](#_Toc151330404)

[1.1.2. Поле “display” 2](#_Toc151330405)

[1.2. Файл .c 2](#_Toc151330406)

[1.2.1. API для GPIO 2](#_Toc151330407)

[1.2.2. API для работы с аналоговыми сигналами 2](#_Toc151330408)

[1.2.3. API для работы со временем (таймеры) 2](#_Toc151330409)

[1.2.4. API для UART 2](#_Toc151330410)

[1.2.5. API для I2C 2](#_Toc151330411)

[1.2.6. API для SPI 2](#_Toc151330412)

[1.2.7. Атрибуты 2](#_Toc151330413)

[1.2.8. API для видеобуфера 2](#_Toc151330414)

[2. Создание платы PCA9538 2](#_Toc151330415)

[2.1. Тестирование 2](#_Toc151330416)

[Приложения 2](#_Toc151330417)

0. Введение. Используемая терминология

Прежде чем перейти к основной теме, я хочу извиниться за возможные трудности в понимании написанного. Из-за моего билингвизма и некоторой устоявшейся терминологии в моей жизни, я введу некоторые местные термины, которые буду использовать в дальнейшем для простоты восприятия моих мыслей. Спасибо за ваше терпение и внимание.

0.1. Терминология

Плата – элемент, который мы добавляем (breakout-плата).

Схема – все элементы, размещенные в окне симуляции.

Пин – вывод платы.

API (Application Programming Interface) – программный интерфейс.

Callback-функция (колбэк-функция) или обратный вызов - это функция, переданная в другую функцию в качестве аргумента, которая затем вызывается по завершению какого-либо действия.

1. Добавление нового элемента

Для добавления нового элемента в Wokwi можно воспользоваться встроенным методом “Custom Chip”, либо добавить его вручную (просто добавить все файлы в проект).

Рассмотрим первый способ. Для этого необходимо в меню добавления нового элемента на схему выбрать пункт “Custom Chip” (рисунки 1-2).

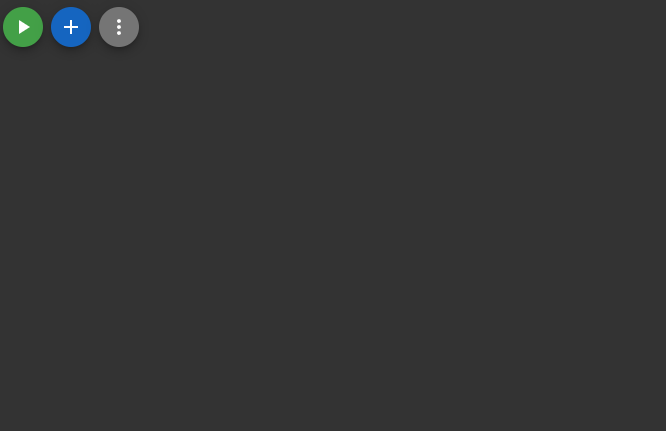


Рисунок 1. Меню схемы

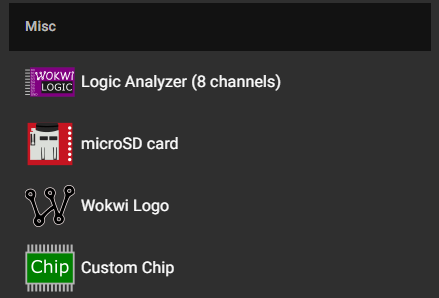


Рисунок 2. Меню добавления нового элемента

В открывшемся меню (рисунок 3) выбираем название будущего элемента и язык, на котором будет написан его скрипт работы.

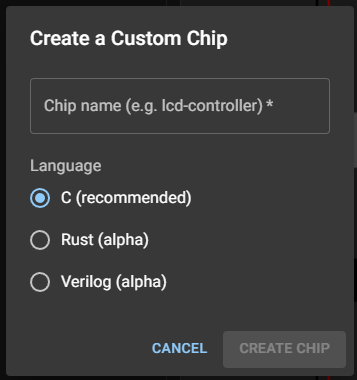


Рисунок 4. Меню создания элемента

Для нашего примера возьмем язык C. Я выбрал название “chiptest”. После нажатия на кнопку “Create Chip” на схеме появится новая плата (рисунок 4) и два файла с расширениями “.json” (листинг 1) и “.c” (листинг 4).



Рисунок 4. Новая плата на схеме

1.1. Файл .json

{

"name": "chiptest",

"author": "КБ-211 Коренев Д.Н.",

"pins": [

"VCC",

"GND",

"IN",

"OUT"

],

"controls": []

}

Листинг 1. Файл “chiptest.chip.json”

В этом файле указываются параметры платы. В таблице 1 приведено описание каждого поля.

|  |  |  |
| --- | --- | --- |
| **Поле** | **Тип данных** | **Описание** |
| name | string | Имя платы. Оно будет также отображаться в редакторе схемы |
| author | string | Имя автора |
| pins | array of strings | Список пинов (подключений) платы |
| controls | array of objects | (опционально) Список способов взаимодействия с платой |
| display | object | (опционально) Конфигурация дисплея на плате |

Таблица 1. Описание полей “.json” файла

Рассмотрим некоторые поля подробнее.

1.1.1. Поле “pins”

Это поле содержит имена пинов на плате, начиная с номера 1. Если необходимо пропустить несколько пинов и оставить их пустыми (например, если нужно создать плату с пинами только на левой стороне), можно использовать пустую строку (“”) в качестве имени пина.

Например "pins": ["VCC", "GND", "RST", "", "SCL", "SDA"], создаст плату с таким видом (рисунок 5).

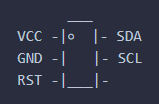


Рисунок 5. Схема примера распиновки платы

1.1.2. Поле “controls”

Данное поле предоставляет пользователю возможность взаимодействовать с платой во время симуляции. Например, датчик температуры может иметь элемент управления, который позволит пользователю установить текущую температуру.

Элементов управления может быть несколько (поле представляет собой массив).

Это поле должно содержать следующие параметры (таблица 2).

|  |  |  |
| --- | --- | --- |
| **Поле** | **Тип данных** | **Описание** |
| id | string | Идентификатор. Рекомендуется использования стиля наименования camelCase |
| label | string | Имя параметра, которое будет показываться пользователю |
| type | string | Тип управления. На данный момент доступно только значение “range” (диапазон), для которого формируется слайдер |
| min | number | Минимальное значение |
| max | number | Максимальное значение |
| step | number | Размер шага слайдера |

Таблица 2. Описание параметров поля “controls”

Давайте для примера добавим поле для указания температуры.

"controls": [

{

"id": "temperature",

"label": "Temperature, °C",

"type": "range",

"min": -100,

"max": 500,

"step": 1

}

]

Листинг 2. Пример поля температуры

Данный код позволит указать параметр температуры при нажатии на плату (при запущенной симуляции).



Рисунок 6. Пример ввода параметра

1.1.2. Поле “display”

Это поле позволяет прикрепить дисплей к нашей плате. Используя его, можно реализовать собственный LCD, OLED или e-paper дисплей, либо показывать состояние платы (рисовать графики, визуально отображать параметры).

Это поле должно содержать следующие параметры (таблица 3).

|  |  |  |
| --- | --- | --- |
| **Поле** | **Тип данных** | **Описание** |
| width | number | Ширина дисплея, в пикселях |
| height | number | Высота дисплея, в пикселях |

Таблица 3. Описание параметров поля “controls”

Для примера создадим дисплей 128х64.

"display": {

"width": 128,

"height": 64

},

Листинг 3. Пример конфигурации дисплея

Эта конфигурация прикрепит к нашей плате такой дисплей (рисунок 7).

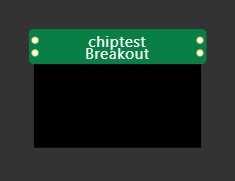


Рисунок 7. Пример дисплея на схеме

1.2. Файл .c

// Wokwi Custom Chip - For docs and examples see:

// https://docs.wokwi.com/chips-api/getting-started

//

// SPDX-License-Identifier: MIT

// Copyright 2023 КБ-211 Коренев Д.Н.

#include "wokwi-api.h"

#include <stdio.h>

#include <stdlib.h>

typedef struct {

// TODO: Put your chip variables here

} chip\_state\_t;

void chip\_init() {

chip\_state\_t \*chip = malloc(sizeof(chip\_state\_t));

// TODO: Initialize the chip, set up IO pins, create timers, etc.

printf("Hello from custom chip!\n");

}

Листинг 4. Файл “chiptest.chip.с”

В этом файле прописывается вся логика работы платы.

Постараюсь описать весь функционал, но добавление собственных плат еще не вышло из стадии бета-тестирования на данный момент (ноябрь 2023). На случай обновлений у Wokwi есть официальная документация по их API на английском языке: <https://docs.wokwi.com/chips-api/getting-started>.

1.2.1. API для GPIO

Этот API позволяет взаимодействовать с GPIO (цифровыми пинами).

**pin\_t pin\_init(const char \*name, uint32\_t mode)**

Параметры для mode:

1. INPUT – цифровой вход;

2. INPUT\_PULLUP – цифровой вход, подтяжка к плюсу;

3. INPUT\_PULLDOWN – цифровой вход, подтяжка к земле;

4. OUTPUT – цифровой вывод;

5. OUTPUT\_LOW – цифровой вывод, устанавливает значение LOW;

6. OUTPUT\_HIGH – цифровой вывод, устанавливает значение HIGH;

7. ANALOG – аналоговый режим, подробнее в пункте API работы с аналоговыми сигналами.

Важно: pin\_init() может быть вызван только в chip\_init() ! Для изменения режима пинов можно воспользоваться функцией pin\_mode().

**void pin\_mode(pin\_t pin, uint32\_t mode)**

Устанавливает mode для пина. Параметры такие же, как и в pin\_init().

**void pin\_write(pin\_t pin, uint32\_t value)**

Устанавливает значение для пина. Можно использовать LOW и HIGH.

**uint32\_t pin\_read(pin\_t pin)**

Возвращает состояние пина (LOW или HIGH).

**bool pin\_watch(pin\_t pin, pin\_watch\_config\_t \*config)**

Следит за изменением значения указанного пина. Структура конфигурации состоит из:

|  |  |  |
| --- | --- | --- |
| **Поле** | **Тип данных** | **Описание** |
| edge | uint32\_t | За каким состоянием следит пин (возрастающий, убывающий, оба) |
| pin\_change | callback | Вызывается, когда состояние пина изменяется |
| user\_data | void \* | Данные, которые будут переданы как первый аргумент в pin\_change() |

Таблица 4. Описание параметров конфигурации

Для поля edge можно использовать следующие значения:

1. BOTH – любое изменение значения;

2. FALLING – изменение с HIGH на LOW;

3. RISING – изменение с LOW на HIGH.

Можно начать только одно отслеживание для каждого пина. Функция возвращает TRUE, если отслеживание успешно начато, или FALSE в случае, если отслеживание уже начато для данного пина.

Сигнатура callback-функции должна иметь следующий вид:

void chip\_pin\_change(void \*user\_data, pin\_t pin, uint32\_t value) {

// value должно быть HIGH или LOW

}

Листинг 5. Сигнатура callback-функции pin\_watch()

Пример использования:

const pin\_watch\_config\_t watch\_config = {

.edge = FALLING,

.pin\_change = chip\_pin\_change,

.user\_data = chip,

};

pin\_watch(pin, &watch\_config)

Листинг 6. Пример использования pin\_watch()

**void pin\_watch\_stop(pin\_t pin)**

Останавливает отслеживание пина.

1.2.2. API для работы с аналоговыми сигналами

**float pin\_adc\_read(pin\_t pin)**

Измеряет текущее напряжение на указанном пине и возвращает его. Пин должен быть установлен в режим ANALOG, иначе возвращенное значение непредсказуемо.

**void pin\_dac\_write(pin\_t pin, float voltage)**

Устанавливает напряжение на указанном пине.

На текущий момент времени для всех виртуальных АЦП опорное напряжение равно 5В (независимо от контроллера), так что напряжение в 0В всегда будет минимальным значением, а напряжение 5В будет считаться максимальным (например, как 1023 на Arduino).

Этот метод может быть вызван в любом режиме пина, но значение будет установлено только после изменения режима работы на ANALOG.

Пример создания платы с аналоговым пином от самих Wokwi: <https://wokwi.com/projects/330112801381024338>.

1.2.3. API для работы со временем (таймеры)

**uint64\_t get\_sim\_nanos()**

Возвращает текущее виртуальное время симуляции в наносекундах.

**timer\_t timer\_init(timer\_config\_t \*config)**

Инициализирует новый таймер. Возвращает идентификатор таймера. Вызовите timer\_start() для старта таймера, и определите callback-функцию chip\_timer\_event(), чтобы отслеживать события таймера.

Структура timer\_config\_t должна иметь следующие поля:

|  |  |  |
| --- | --- | --- |
| **Поле** | **Тип данных** | **Описание** |
| callback | callback | Вызывается при срабатывании таймера |
| user\_data | void \* | Данные, которые будут переданы как первый аргумент в callback-функцию |

Таблица 5. Описание параметров конфигурации

Сигнатура callback-функции должна иметь следующий вид:

void chip\_timer\_callback(void \*user\_data) {

}

Листинг 7. Сигнатура callback-функции timer\_init()

Важно: timer\_init() может быть вызван только из chip\_init().

**void timer\_start(uint32\_t timer\_id, uint32\_t micros, bool repeat)**

Устанавливает расписание таймера по ID. Аргумент micros определяет сколько микросекунд пройдет перед тем как будет вызвана функция chip\_timer\_event().

Если repeat установлен в FALSE, таймер будет вызван один раз. Если TRUE, таймер будет повторяться каждый установленный период времени до тех пор, пока не будет вызван timer\_stop() или пока конфигурация не будет изменена через timer\_start().

**void timer\_start\_ns(uint32\_t timer\_id, uint64\_t nanos, bool repeat)**

Функция похожа на timer\_start(), но время указывается в наносекундах. Предпочтительно не использовать эту функцию для улучшения производительности.

**void timer\_stop(uint32\_t timer\_id)**

Останавливает таймер. Если таймер еще не был запущен, он не запустится до тех пор, пока функция timer\_start() не будет вызвана снова.

1.2.4. API для UART

**uart\_dev\_t uart\_init (const uart\_config\_t \*config)**

Инициализация UART. Аргумент config определяет контакты, конфигурацию и обратные вызовы для устройства UART. Он содержит следующие поля:

|  |  |  |
| --- | --- | --- |
| **Поле** | **Тип данных** | **Описание** |
| rx | pin\_t | Вывод RX (или NO\_PIN для отключения RX) |
| tx | pin\_t | Вывод TX (или NO\_PIN для отключения TX) |
| baud\_rate | uint32\_t | Скорость передачи данных в бодах (например, 115200) |
| rx\_data | callback | Вызывается для каждого байта, полученного на выводе RX |
| write\_done | callback | Вызывается, когда передача данных по выводу TX завершена |
| user\_data | void \* | Данные, которые будут переданы в качестве первого аргумента в callback-функцию |

Таблица 6. Описание параметров конфигурации

Оба пункта rx\_data, write\_done являются необязательными. Все они используют указатель user\_data в качестве своего первого аргумента.

Пример:

static void on\_uart\_rx\_data(void \*user\_data, uint8\_t byte) {

// `byte` — это байт, полученный на пине "RX"

}

static uint8\_t on\_uart\_write\_done(void \*user\_data) {

// Можно записать фрагменты данных для передачи через uart\_write().

}

// ...

const uart\_config\_t uart1 = {

.tx = pin\_init("TX", INPUT\_PULLUP),

.rx = pin\_init("RX", INPUT),

.baud\_rate = 115200,

.rx\_data = on\_uart\_rx\_data,

.write\_done = on\_uart\_write\_done,

.user\_data = chip,

};

Листинг 8. Пример uart\_init()

**bool uart\_write (uart\_dev\_t uart, uint8\_t \*buffer, uint32\_t count)**

Записывает байты из памяти, на которую указывает буфер, на данное устройство UART.

Возвращает TRUE при успешном выполнении или FALSE, если устройство UART уже занято передачей данных из предыдущего вызова uart\_write() (новые данные передаваться не будут).

Передача данных начинается после выхода из функции uart\_write(). Как только Wokwi завершает передачу данных, вызывается callback write\_done() (из структуры uart\_config\_t, которую вы передали uart\_init).

Пример создания платы с UART от самих Wokwi: <https://wokwi.com/projects/333638144389808723>

1.2.5. API для I2C

Чтобы создать устройство I2C, сначала вызовите i2c\_init, передав структуру i2c\_config\_t. Эта структура определяет контакты SCL/SDA, адрес устройства I2C и callback-функции подключения/чтения/записи/отключения.

**i2c\_dev\_t i2c\_init(i2c\_config\_t \*config)**

Инициализирует устройство I2C. Аргумент config определяет пины, адрес и callback-функции для устройства I2C. Он содержит следующие поля:

|  |  |  |
| --- | --- | --- |
| **Поле** | **Тип данных** | **Описание** |
| address | uint32\_t | Слушает запросы, соответствующие заданному адресу I2C (7-разрядный). Чтобы прослушивать все запросы, установите значение 0 |
| sda | pin\_t | Пин SDA |
| scl | pin\_t | Пин SCL |
| connect | callback | Вызывается, когда плата адресована по шине I2C |
| read | callback | Вызывается, когда микроконтроллер хочет считать байт данных с вашей платы |
| write | callback | Вызывается, когда микроконтроллер записывает байт в вашу плату |
| disconnect | callback | Вызывается, когда микроконтроллер отключается от вашей платы |
| user\_data | void \* | Данные, которые будут переданы в первом аргументе callback-функции |

Таблица 7. Описание параметров конфигурации

Все callback-функции опциональны. Они все используют указатель на user\_data в качестве первого аргумента.

Важно: i2c\_init может быть вызван только из chip\_init(). Не рекомендуется вызывать его позже.

Пример и сигнатуры callback-функций:

bool on\_i2c\_connect(void \*user\_data, uint32\_t address, bool read) {

// `address` - 7-битовый адрес, полученный на I2C шине

// `read` - определяет это запрос чтения (TRUE) или записи (FALSE)

return true; // TRUE == ACK, FALSE == NACK

}

uint8\_t on\_i2c\_read(void \*user\_data) {

return 0; // Байт, который будет возвращен микроконтроллеру

}

bool on\_i2c\_write(void \*user\_data, uint8\_t data) {

// `data` - байт, полученный от микроконтроллера

return true; // TRUE == ACK, FALSE == NACK

}

void on\_i2c\_disconnect(void \*user\_data) {

// Метод опциональный. Полезен когда нужно знать, что соединение было разорвано

}

static const i2c\_config\_t i2c1 {

.address = 0x22,

.scl = pin\_init("SCL", INPUT\_PULLUP),

.sda = pin\_init("SDA", INPUT\_PULLUP),

.connect = on\_i2c\_connect,

.read = on\_i2c\_read,

.write = on\_i2c\_write,

.disconnect = on\_i2c\_disconnect,

.user\_data = chip,

};

Листинг 9. Сигнатура и пример callback-функции i2c\_init()

1.2.6. API для SPI

Чтобы создать SPI интерфейс устройства, необходимо вызвать функцию spi\_init(), передав ей структуру настроек spi\_config\_t. Эта структура определяет пины CLOCK, MOSI, MISO и callback-функцию done.

**spi\_dev\_t spi\_init(spi\_config\_t \*config)**

Инициализирует интерфейс устройства SPI. Аргумент config определяет пины, режим и обратные вызовы для устройства SPI. Он содержит следующие поля:

|  |  |  |
| --- | --- | --- |
| **Поле** | **Тип данных** | **Описание** |
| sck | pin\_t | Пин тактирования |
| mosi | pin\_t | Пин MOSI |
| miso | pin\_t | Пин MISO |
| mode | uint32\_t | Режим SPI: 0, 1, 2 или 3 (по умолчанию: 0) |
| done | callback | Вызывается при завершении транзакции SPI |
| user\_data | void \* | Данные, которые будут переданы в первом аргументе callback-функции done |

Таблица 8. Описание параметров конфигурации

Важно: API не поддерживает пины CS/SS: пользователь должен выбрать/отменить выбор интерфейса SPI, вызывая spi\_start() и spi\_stop().

Важно: spi\_init может быть вызван только из chip\_init(). Не рекомендуется вызывать его позже.

Пример структуры config:

const spi\_config\_t spi1 = {

.sck = pin\_init("SCK", INPUT),

.mosi = pin\_init("MOSI", INPUT),

.miso = pin\_init("MISO", INPUT),

.mode = 0,

.done = chip\_spi\_done,

.user\_data = chip,

};

Листинг 10. Пример структуры config

**void spi\_start(spi\_dev\_t spi, uint8\_t \*buffer, uint32\_t count)**

Запускает транзакцию SPI, отправляя и получая количество байт в/из заданного буфера.

Обычно вы будете прослушивать пин CS (chip select) с помощью pin\_watch. Вызывайте spi\_start(), когда вывод CS становится LOW, и spi\_stop(), когда вывод CS становится HIGH.

При создании устройства, передающего большие объемы данных (например, жидкокристаллический дисплей), рекомендуется использовать буфер большого размера (несколько килобайт). Имитатор может использовать буфер большего размера для оптимизации передачи SPI, управляемой DMA, и ускорения моделирования.

Для простых устройств, которые передают небольшие объемы данных, вы можете использовать однобайтовый буфер и обрабатывать каждый байт по мере его поступления в обратном вызове done.

**void spi\_stop(spi\_dev\_t spi)**

Останавливает интерфейс SPI. Обычно этот метод вызывают, когда вывод CS становится HIGH.

**Callback-функция done**

Сигнатура функции выглядит так:

static void chip\_spi\_done(void \*user\_data, uint8\_t \*buffer, uint32\_t count) {

// 1. Обрабатываем получаемы данные (опционально)

// 2. Если пин CS все еще LOW, откладываем следующую транзакцию SPI до вызова

// spi\_start

}

Листинг 11. Сигнатура callback-функции spi\_stop()

Обратный вызов done выполняется, когда транзакция SPI завершается: либо когда буфер, предоставленный spi\_start, заполнен, либо когда был вызван spi\_stop. Буфер содержит полученные данные (это тот же буфер, который был передан spi\_start), а count - это количество переданных байтов (или 0, если spi\_stop был вызван до того, как был передан полный байт).

Ваш завершенный обратный вызов должен проверить состояние вывода CS, и если он все еще низкий, он должен снова вызвать spi\_start(), чтобы получить следующий фрагмент данных от микроконтроллера.

Пример создания платы с SPI от самих Wokwi: <https://wokwi.com/projects/330669951756010068>.

1.2.7. Атрибуты

Атрибуты - это входные параметры, которые пользователь может задать в diagram.json. Вы также можете определить раздел элементов управления controls в файле .chip.json, чтобы позволить пользователю редактировать эти параметры в интерактивном режиме во время моделирования. Это особенно полезно для входных данных датчиков (например, температуры, влажности и т.д.).

Рекомендации по названию атрибутов от Wokwi:

* Используйте camelCase для имен атрибутов
* Используйте американское английское написание (например, color, а не colour).

**uint32\_t attr\_init(const char \*name, uint32\_t default\_value)**

Определяет новый целочисленный атрибут с заданным именем. Значение по умолчанию будет использоваться, когда пользователь не определяет значение для атрибута в diagram.json (в разделе attrs).

Функция возвращает дескриптор атрибута, доступ к которому можно получить с помощью attr\_read().

Важно: attr\_init может быть вызван только из chip\_init(). Не вызывайте его позже.

**uint32\_t attr\_init\_float(const char \*name, float default\_value)**

Определяет новый атрибут с плавающей запятой (точкой) с заданным именем. Смотрите attr\_init() для получения дополнительной информации.

Важно: attr\_init\_float может быть вызван только из chip\_init(). Не вызывайте его позже.

**uint32\_t attr\_read(uint32\_t attr)**

Возвращает текущее значение атрибута. attr должен быть допустимым дескриптором атрибута, ранее возвращенным функцией attr\_init().

**float attr\_read\_float(uint32\_t attr)**

Возвращает текущее значение атрибута. attr должен быть допустимым дескриптором атрибута, ранее возвращенным функцией attr\_init\_float().

1.2.8. API для видеобуфера

Используйте этот API для реализации дисплеев (LCD, OLED, e-paper и т.д.). Размер отображения определен в файле .chip.json. В буфере кадров используется 32 бита на пиксель. Пиксели сохраняются в формате RGBA. Общий размер буфера равен pixel\_width \* pixel\_height \* 4 байта.

**buffer\_t framebuffer\_init(uint32\_t pixel\_width, uint32\_t pixel\_height)**

Возвращает буфер кадров для текущего платы и размеры в пикселях (ширина/высота) буфера кадров.

Переменные для размеров необходимо передать через указатели (/референсы), например:

chip->framebuffer = framebuffer\_init(&chip->width, &chip->height);

Важно: framebuffer\_init может быть вызван только из chip\_init(). Не вызывайте его позже.

**void buffer\_write(buffer\_t buffer, uint32\_t offset, void \*data, uint32\_t data\_len)**

Копирует data\_len байтов из data в буфер кадров с заданным смещением.

**void buffer\_read(buffer\_t buffer, uint32\_t offset, void \*data, uint32\_t data\_len)**

Копирует data\_len байтов с заданным смещением буфера кадров в data.

Пример использования видеобуфера от самих Wokwi:

1. [Базовый пример: https://wokwi.com/projects/330503863007183442](Базовый%20пример:%20https://wokwi.com/projects/330503863007183442)

2. SSD1306 I2C OLED: <https://wokwi.com/projects/371050937178768385>

3. IL9163 128x128 Color LCD: <https://wokwi.com/projects/333332561949360723>

2. Создание платы PCA9538

Выполним все те же операции, что и в пункте 1. На выходе получим следующее:

pca9538.chip.json

{

"name": "PCA9538",

"author": "Kseen715",

"pins": [

"VCC",

"GND",

"IN",

"OUT"

]

}

Листинг 12. Полученный файл pca9538.chip.json

pca9538.chip.c

// SPDX-License-Identifier: MIT

// Copyright 2023 Kseen715

#include "wokwi-api.h"

#include <stdio.h>

#include <stdlib.h>

typedef struct {

// TODO: Put your chip variables here

} chip\_state\_t;

void chip\_init() {

chip\_state\_t \*chip = malloc(sizeof(chip\_state\_t));

// TODO: Initialize the chip, set up IO pins, create timers, etc.

printf("Hello from custom chip!\n");

}

Листинг 13. Полученный файл pca9538.chip.c

Для начала добавим все необходимые пины на плату. Будем пользоваться схемой из документации (рисунок 8).

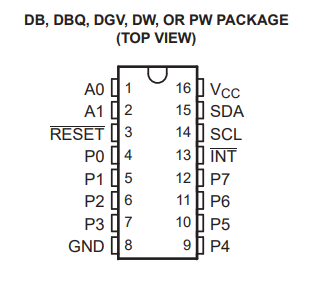


Рисунок 8. Схема подключения PCA9538

pca9538.chip.json

{

"name": "PCA9538",

"author": "Kseen715",

"pins": [

"A0",

"A1",

"RESET",

"P0",

"P1",

"P2",

"P3",

"GND",

"P4",

"P5",

"P6",

"P7",

"INT",

"SCL",

"SDA",

"VCC"

]

}

Листинг 14. Распиновка PCA9538 в коде json

Получим следующую плату на нашей схеме:

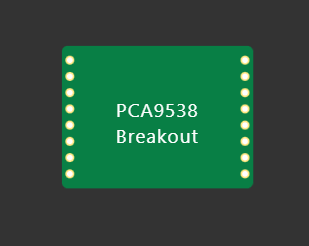


Рисунок 9. PCA9538 на схеме (стрелками указано расположение пинов, от 0-го до 15-го)

Создадим структуру нашей платы в файле pca9538.chip.c.

pca9538.chip.c

...

#include <stdlib.h>

typedef struct {

pin\_t PIN\_VCC;

pin\_t PIN\_SDA;

pin\_t PIN\_SCL;

pin\_t PIN\_INT;

pin\_t PIN\_GND;

pin\_t PIN\_RESET;

pin\_t PIN\_A0;

pin\_t PIN\_A1;

pin\_t PIN\_P0;

pin\_t PIN\_P1;

pin\_t PIN\_P2;

pin\_t PIN\_P3;

pin\_t PIN\_P4;

pin\_t PIN\_P5;

pin\_t PIN\_P6;

pin\_t PIN\_P7;

uint8\_t buffer; // Буфер данных.

uint8\_t address; // I2C адрес.

uint8\_t command; // Текущяя команда.

uint8\_t config\_dup; // Копия конфигурации,

// т.к. мы не можем читать ее.

uint8\_t polarity\_dup; // Копия полярностей,

// т.к. мы не можем читать их.

uint8\_t last\_input; // Последнее обнаруженное

// состояние вводов.

} chip\_state\_t;

void chip\_init() {

...

Листинг 15. Структура PCA9538 в коде

Инициализируем все пины на вход (кроме INT).

pca9538.chip.c

...

void chip\_init() {

chip\_state\_t \*chip = malloc(sizeof(chip\_state\_t));

chip->PIN\_VCC = pin\_init("VCC", INPUT\_PULLDOWN);

chip->PIN\_SCL = pin\_init("SCL", INPUT\_PULLDOWN);

chip->PIN\_SDA = pin\_init("SDA", INPUT\_PULLDOWN);

chip->PIN\_INT = pin\_init("INT", OUTPUT);

chip->PIN\_GND = pin\_init("GND", INPUT\_PULLDOWN);

chip->PIN\_RESET = pin\_init("RESET", INPUT\_PULLDOWN);

chip->PIN\_A0 = pin\_init("A0", INPUT\_PULLDOWN);

chip->PIN\_A1 = pin\_init("A1", INPUT\_PULLDOWN);

chip->PIN\_P0 = pin\_init("P0", INPUT\_PULLDOWN);

chip->PIN\_P1 = pin\_init("P1", INPUT\_PULLDOWN);

chip->PIN\_P2 = pin\_init("P2", INPUT\_PULLDOWN);

chip->PIN\_P3 = pin\_init("P3", INPUT\_PULLDOWN);

chip->PIN\_P4 = pin\_init("P4", INPUT\_PULLDOWN);

chip->PIN\_P5 = pin\_init("P5", INPUT\_PULLDOWN);

chip->PIN\_P6 = pin\_init("P6", INPUT\_PULLDOWN);

chip->PIN\_P7 = pin\_init("P7", INPUT\_PULLDOWN);

chip->polarity\_dup = 0x00; // Все с прямой полярностью.

chip->config\_dup = 0xFF; // Все на ввод.

chip->buffer = 0;

chip->last\_input = 0;

chip->command = 0xFF; // Будем считать, что в таком

// состоянии плата ждет команды.

}

Листинг 16. Инициализация пинов

Добавим инициализацию адреса платы на основе состояния пинов A0-1. Схему возьмем из документации (рисунок 10).

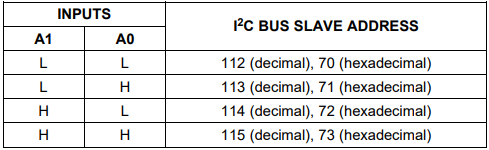


Рисунок 10. Схема определения адреса PCA9538

pca9538.chip.c

...

bool i2c\_address\_init(chip\_state\_t \*chip){

uint32\_t a0 = pin\_read(chip->PIN\_A0);

uint32\_t a1 = pin\_read(chip->PIN\_A1);

uint8\_t mode = 0;

mode |= (a0 << 0) | (a1 << 1);

switch (mode) {

case 0b00:

chip->address = 0x70;

break;

case 0b01:

chip->address = 0x71;

break;

case 0b10:

chip->address = 0x72;

break;

case 0b11:

chip->address = 0x73;

break;

default:

printf("PCA9538: Error! Wrong address configuration.\n");

return false;

}

return true;

}

...

Листинг 17. Определение адреса на шине I2C (часть 1 из 2)

pca9538.chip.c

...

chip->last\_input = 0;

chip->command = 0xFF; // Будем считать, что в таком

// состоянии плата ждет команды.

i2c\_address\_init(chip);

printf("PCA9538 initialized on I2C with address 0x%02X!\n", chip->address);

...

Листинг 18. Определение адреса на шине I2C (часть 2 из 2)

Инициализируем плату на шине I2C.

pca9538.chip.c

...

i2c\_address\_init(chip);

const i2c\_config\_t i2c\_config = {

.user\_data = chip,

.address = chip->address,

.scl = chip->PIN\_SCL,

.sda = chip->PIN\_SDA,

.connect = on\_i2c\_connect,

.read = on\_i2c\_read,

.write = on\_i2c\_write,

.disconnect = on\_i2c\_disconnect,

};

i2c\_init(&i2c\_config);

if (CHIP\_ON) {

printf("PCA9538 initialized on I2C with address 0x%02X!\n", chip->address);

}

...

Листинг 19. Инициализация платы на шине I2C

Теперь наша плата умеет получать адрес на шине I2C.









Рисунок 11. Вывод в консоль отладки адреса платы в зависимости от пинов A0-1

Для удобства создадим несколько макросов и вспомогательных функций.

pca9538.chip.c

...

#define PCA9538\_IN\_REG 0x00

#define PCA9538\_OUT\_REG 0x01

#define PCA9538\_POL\_REG 0x02

#define PCA9538\_CONF\_REG 0x03

#define CHIP\_ON (pin\_read(chip->PIN\_VCC)==HIGH)

...

Листинг 20. Вспомогательные макросы

pca9538.chip.c

...

bool getNbit(uint8\_t byte, uint8\_t bit){

return (bool)(byte & (0b1 << bit));

}

uint8\_t shiftBit(uint8\_t byte, uint8\_t bit){

return (byte & (0b1 << bit));

}

...

Листинг 21. Вспомогательные функции

Теперь начнем реализовывать функции для I2C API:

static bool on\_i2c\_connect(void \*user\_data, uint32\_t address, bool connect);

static uint8\_t on\_i2c\_read(void \*user\_data);

static bool on\_i2c\_write(void \*user\_data, uint8\_t data);

static void on\_i2c\_disconnect(void \*user\_data);

Сначала простые, которые нам не особо важны (мы никак не обрабатываем моменты подключения и отключения от шины):

pca9538.chip.c

...

bool on\_i2c\_connect(void \*user\_data, uint32\_t address, bool connect) {

return true;

}

...

Листинг 22. Функция on\_i2c\_connect()

и

pca9538.chip.c

...

void on\_i2c\_disconnect(void \*user\_data) {

// Do nothing

}

...

Листинг 23. Функция on\_i2c\_disconnect()

Теперь приступим к обработке выходных сигналов.

pca9538.chip.c

...

bool on\_i2c\_write(void \*user\_data, uint8\_t data) {

chip\_state\_t \*chip = user\_data;

if (CHIP\_ON) {

#ifdef DEBUG

printf("Sent to I2C^0x%02X: ", chip->address);

printf("0x%02X (", data);

printBits(1, &data);

printf(")\n");

#endif // DEBUG

chip->buffer = data;

update\_state(chip);

return true; // ACK

}

return false; // NACK

}

...

Листинг 24. Обработчик выходных сигналов. Функция on\_i2c\_write()

Основную логику я вынес в отдельную функцию, чтобы не загромождать код.

Сначала нам необходимо получить байт с командой, а затем данные, поэтому мы и инициализировали поле chip->command значением 0xFF, что отличается от любой существующей команды нашей платы. Как только мы можем распознать команду, мы переходим к ее исполнению.

Команду чтения из платы Wokwi обрабатывает отдельно, поэтому пишем там простую заглушку. При записи учитываем только те пины, которые настроены на запись и не забываем про настройку инверсии. Команда настройки инверсии полностью обрабатывается нами, поэтому просто записываем ее. Команда конфигурации пинов интуитивно понятна.

pca9538.chip.c

...

static void update\_state(chip\_state\_t \*chip)

{

if (chip->command == 0xFF){

// Записываем команду

chip->command = chip->buffer;

} else {

// Записываем данные

switch (chip->command) {

case PCA9538\_IN\_REG: // 0x00

chip->command = 0xFF;

break;

case PCA9538\_OUT\_REG: // 0x01

if (getNbit(chip->config\_dup, 0) == 0)

pin\_write(chip->PIN\_P0, (getNbit(chip->polarity\_dup, 0)

^ getNbit(chip->buffer, 0)) ? HIGH : LOW);

if (getNbit(chip->config\_dup, 1) == 0)

pin\_write(chip->PIN\_P1, (getNbit(chip->polarity\_dup, 1)

^ getNbit(chip->buffer, 1)) ? HIGH : LOW);

if (getNbit(chip->config\_dup, 2) == 0)

pin\_write(chip->PIN\_P2, (getNbit(chip->polarity\_dup, 2)

^ getNbit(chip->buffer, 2)) ? HIGH : LOW);

if (getNbit(chip->config\_dup, 3) == 0)

pin\_write(chip->PIN\_P3, (getNbit(chip->polarity\_dup, 3)

^ getNbit(chip->buffer, 3)) ? HIGH : LOW);

if (getNbit(chip->config\_dup, 4) == 0)

pin\_write(chip->PIN\_P4, (getNbit(chip->polarity\_dup, 4)

^ getNbit(chip->buffer, 4)) ? HIGH : LOW);

if (getNbit(chip->config\_dup, 5) == 0)

pin\_write(chip->PIN\_P5, (getNbit(chip->polarity\_dup, 5)

^ getNbit(chip->buffer, 5)) ? HIGH : LOW);

if (getNbit(chip->config\_dup, 6) == 0)

pin\_write(chip->PIN\_P6, (getNbit(chip->polarity\_dup, 6)

^ getNbit(chip->buffer, 6)) ? HIGH : LOW);

if (getNbit(chip->config\_dup, 7) == 0)

pin\_write(chip->PIN\_P7, (getNbit(chip->polarity\_dup, 7)

^ getNbit(chip->buffer, 7)) ? HIGH : LOW);

chip->command = 0xFF;

break;

case PCA9538\_POL\_REG: // 0x02

chip->polarity\_dup = chip->buffer;

chip->command = 0xFF;

break;

case PCA9538\_CONF\_REG: // 0x03

pin\_mode(chip->PIN\_P0, getNbit(chip->buffer, 0)

? INPUT\_PULLDOWN : OUTPUT);

pin\_mode(chip->PIN\_P1, getNbit(chip->buffer, 1)

? INPUT\_PULLDOWN : OUTPUT);

pin\_mode(chip->PIN\_P2, getNbit(chip->buffer, 2)

? INPUT\_PULLDOWN : OUTPUT);

pin\_mode(chip->PIN\_P3, getNbit(chip->buffer, 3)

? INPUT\_PULLDOWN : OUTPUT);

pin\_mode(chip->PIN\_P4, getNbit(chip->buffer, 4)

? INPUT\_PULLDOWN : OUTPUT);

pin\_mode(chip->PIN\_P5, getNbit(chip->buffer, 5)

? INPUT\_PULLDOWN : OUTPUT);

pin\_mode(chip->PIN\_P6, getNbit(chip->buffer, 6)

? INPUT\_PULLDOWN : OUTPUT);

pin\_mode(chip->PIN\_P7, getNbit(chip->buffer, 7)

? INPUT\_PULLDOWN : OUTPUT);

chip->config\_dup = chip->buffer;

chip->command = 0xFF;

break;

default:

printf("PCA9538: Error! Wrong command.\n");

}

}

}

...

Листинг 25. Обработчик выходных сигналов. Функция update\_state()

Теперь обработка чтения с пинов платы. Здесь мы опять учитываем то, настроен ли пин на чтение и его инвертированность. Также запоминаем состояние всех пинов, для того чтобы при изменении любого пина в режиме чтения вызывать прерывания на пине INT.

pca9538.chip.c

...

uint8\_t on\_i2c\_read(void \*user\_data) {

chip\_state\_t \*chip = user\_data;

if (CHIP\_ON) {

chip->buffer = 0;

if (getNbit(chip->config\_dup, 7) == 1)

chip->buffer |= pin\_read(chip->PIN\_P7)

^ getNbit(chip->polarity\_dup, 7);

chip->buffer<<=1;

if (getNbit(chip->config\_dup, 6) == 1)

chip->buffer |= pin\_read(chip->PIN\_P6)

^ getNbit(chip->polarity\_dup, 6);

chip->buffer<<=1;

if (getNbit(chip->config\_dup, 5) == 1)

chip->buffer |= pin\_read(chip->PIN\_P5)

^ getNbit(chip->polarity\_dup, 5);

chip->buffer<<=1;

if (getNbit(chip->config\_dup, 4) == 1)

chip->buffer |= pin\_read(chip->PIN\_P4)

^ getNbit(chip->polarity\_dup, 4);

chip->buffer<<=1;

if (getNbit(chip->config\_dup, 3) == 1)

chip->buffer |= pin\_read(chip->PIN\_P3)

^ getNbit(chip->polarity\_dup, 3);

chip->buffer<<=1;

if (getNbit(chip->config\_dup, 2) == 1)

chip->buffer |= pin\_read(chip->PIN\_P2)

^ getNbit(chip->polarity\_dup, 2);

chip->buffer<<=1;

if (getNbit(chip->config\_dup, 1) == 1)

chip->buffer |= pin\_read(chip->PIN\_P1)

^ getNbit(chip->polarity\_dup, 1);

chip->buffer<<=1;

if (getNbit(chip->config\_dup, 0) == 1)

chip->buffer |= pin\_read(chip->PIN\_P0)

^ getNbit(chip->polarity\_dup, 0);

if (chip->last\_input != chip->buffer){

chip->last\_input = chip->buffer;

pin\_write(chip->PIN\_INT, HIGH);

pin\_write(chip->PIN\_INT, LOW);

}

#ifdef DEBUG

printf("Sent from I2C^0x%02X: ", chip->address);

printf("0x%02X (", chip->buffer);

printBits(1, &chip->buffer);

printf(")\n");

#endif // DEBUG

chip->command = 0xFF;

return chip->buffer;

}

return 0;

}

...

Листинг 26. Обработчик входных сигналов. Функция on\_i2c\_read()

2.1. Тестирование

Для тестирования собрал такую схему:

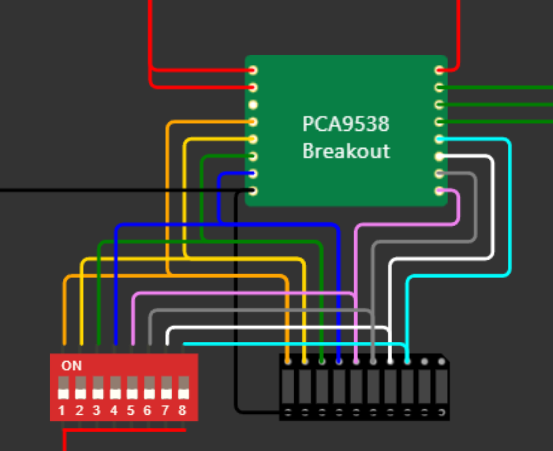


Рисунок 12. Тестовая схема для платы PCA9538

DIP-переключатель слева будет использоваться для подачи сигналов на входы платы P0-7, а LED-индикатор для отображения выходных сигналов с пинов P0-7.

Для подсчета сигналов прерываний INT соберем рядом простой 4-х битный счетчик на D-триггерах.

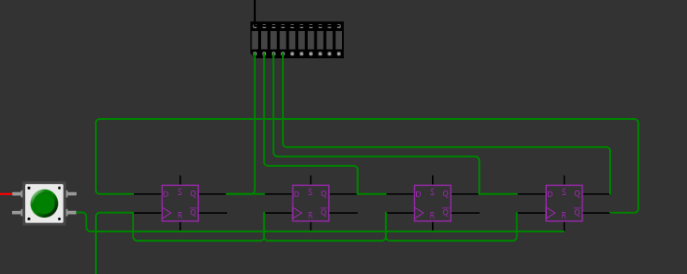


Рисунок 13. 4-х битный счетчик на D-триггерах

В скетче контроллера напишем простую программу, которая будет устанавливать пины P0-3 на вход (с инверсией) и пины P4-7 на выход. Будем подавать первые 4 бита (входные) на вторые 4 бита данных (выходные).

sketch.ino

#include <Arduino.h>

#include "pca95xx.h"

void setup()

{

pca95xx\_configure(0x73, 0x0F, 0x0F);

}

void loop()

{

pca95xx\_out(0x73, pca95xx\_in(0x73) << 4);

delay(200);

}

Листинг 27. Код sketch.ino

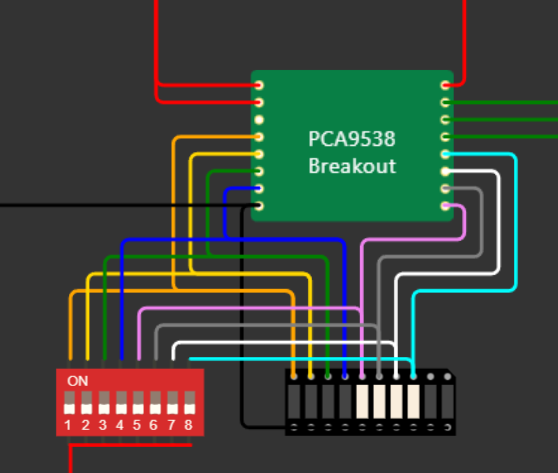


Рисунок 14. На входе платы 0b0000, на выходе 0b1111

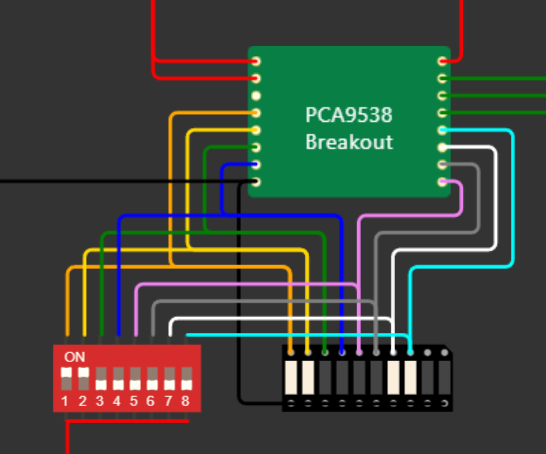
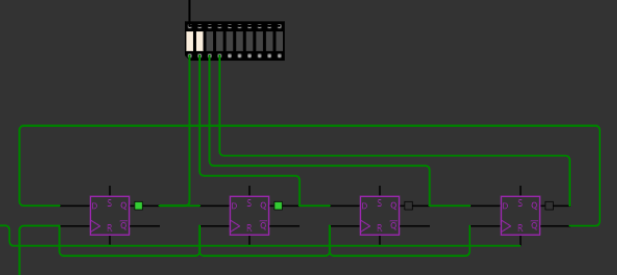


Рисунок 15. На входе платы 0b0011, на выходе 0b1100

Входной сигнал изменился два раза, прерывание сработало так же два раза, что отразилось на счетчике.

 Рисунок 16. Состояние счётчика после изменений сигналов

Тестирование показало исправность логики работы платы в соответствии с документацией от Texas Instruments Incorporated.

Проект Wokwi с тестовой схемой: <https://wokwi.com/projects/381564992805272577>

Мой репозиторий GitHub с кодом платы: <https://github.com/Kseen715/wokwi-pca9538>

Приложения

Приложение 1. Полный код файла pca9538.chip.json

pca9538.chip.c

{

"name": "PCA9538",

"author": "Kseen715",

"pins": [

"A0",

"A1",

"RESET",

"P0",

"P1",

"P2",

"P3",

"GND",

"P4",

"P5",

"P6",

"P7",

"INT",

"SCL",

"SDA",

"VCC"

]

}

Приложение 2. Полный код файла pca9538.chip.c

pca9538.chip.c

// SPDX-License-Identifier: MIT

// Copyright 2023 Kseen715

#include "wokwi-api.h"

#include <stdio.h>

#include <stdlib.h>

#ifndef DEBUG

// #define DEBUG

// #undef DEBUG

#endif

#define PCA9538\_IN\_REG 0x00

#define PCA9538\_OUT\_REG 0x01

#define PCA9538\_POL\_REG 0x02

#define PCA9538\_CONF\_REG 0x03

#define CHIP\_ON (pin\_read(chip->PIN\_VCC)==HIGH)

void printBits(size\_t const size, void \*ptr)

{

unsigned char \*b = (unsigned char\*) ptr;

unsigned char byte;

int i, j;

printf("0b");

for (i = size-1; i >= 0; i--) {

for (j = 7; j >= 0; j--) {

byte = (b[i] >> j) & 1;

printf("%u", byte);

}

}

}

typedef struct {

pin\_t PIN\_VCC;

pin\_t PIN\_SDA;

pin\_t PIN\_SCL;

pin\_t PIN\_INT;

pin\_t PIN\_GND;

pin\_t PIN\_RESET;

pin\_t PIN\_A0;

pin\_t PIN\_A1;

pin\_t PIN\_P0;

pin\_t PIN\_P1;

pin\_t PIN\_P2;

pin\_t PIN\_P3;

pin\_t PIN\_P4;

pin\_t PIN\_P5;

pin\_t PIN\_P6;

pin\_t PIN\_P7;

uint8\_t buffer; // Буфер данных.

uint8\_t address; // I2C адрес.

uint8\_t command; // Текущяя команда.

uint8\_t config\_dup; // Копия конфигурации,

// т.к. мы не можем читать ее.

uint8\_t polarity\_dup; // Копия полярностей,

// т.к. мы не можем читать их.

uint8\_t last\_input; // Последнее обнаруженное

// состояние вводов.

} chip\_state\_t;

static bool on\_i2c\_connect(void \*user\_data, uint32\_t address, bool connect);

static uint8\_t on\_i2c\_read(void \*user\_data);

static bool on\_i2c\_write(void \*user\_data, uint8\_t data);

static void on\_i2c\_disconnect(void \*user\_data);

bool i2c\_address\_init(chip\_state\_t \*chip){

uint32\_t a0 = pin\_read(chip->PIN\_A0);

uint32\_t a1 = pin\_read(chip->PIN\_A1);

uint8\_t mode = 0;

mode |= (a0 << 0) | (a1 << 1);

switch (mode) {

case 0b00:

chip->address = 0x70;

break;

case 0b01:

chip->address = 0x71;

break;

case 0b10:

chip->address = 0x72;

break;

case 0b11:

chip->address = 0x73;

break;

default:

printf("PCA9538: Error! Wrong address configuration.\n");

return false;

}

return true;

}

void chip\_init() {

chip\_state\_t \*chip = malloc(sizeof(chip\_state\_t));

chip->PIN\_VCC = pin\_init("VCC", INPUT\_PULLDOWN);

chip->PIN\_SCL = pin\_init("SCL", INPUT\_PULLDOWN);

chip->PIN\_SDA = pin\_init("SDA", INPUT\_PULLDOWN);

chip->PIN\_INT = pin\_init("INT", OUTPUT);

chip->PIN\_GND = pin\_init("GND", INPUT\_PULLDOWN);

chip->PIN\_RESET = pin\_init("RESET", INPUT\_PULLDOWN);

chip->PIN\_A0 = pin\_init("A0", INPUT\_PULLDOWN);

chip->PIN\_A1 = pin\_init("A1", INPUT\_PULLDOWN);

chip->PIN\_P0 = pin\_init("P0", INPUT\_PULLDOWN);

chip->PIN\_P1 = pin\_init("P1", INPUT\_PULLDOWN);

chip->PIN\_P2 = pin\_init("P2", INPUT\_PULLDOWN);

chip->PIN\_P3 = pin\_init("P3", INPUT\_PULLDOWN);

chip->PIN\_P4 = pin\_init("P4", INPUT\_PULLDOWN);

chip->PIN\_P5 = pin\_init("P5", INPUT\_PULLDOWN);

chip->PIN\_P6 = pin\_init("P6", INPUT\_PULLDOWN);

chip->PIN\_P7 = pin\_init("P7", INPUT\_PULLDOWN);

chip->polarity\_dup = 0x00; // Все с прямой полярностью.

chip->config\_dup = 0xFF; // Все на ввод.

chip->buffer = 0;

chip->last\_input = 0;

chip->command = 0xFF; // Будем считать, что в таком

// состоянии плата ждет команды.

i2c\_address\_init(chip);

const i2c\_config\_t i2c\_config = {

.user\_data = chip,

.address = chip->address,

.scl = chip->PIN\_SCL,

.sda = chip->PIN\_SDA,

.connect = on\_i2c\_connect,

.read = on\_i2c\_read,

.write = on\_i2c\_write,

.disconnect = on\_i2c\_disconnect,

};

i2c\_init(&i2c\_config);

if (CHIP\_ON) {

printf("PCA9538 initialized on I2C with address 0x%02X!\n", chip->address);

}

}

bool getNbit(uint8\_t byte, uint8\_t bit){

return (bool)(byte & (0b1 << bit));

}

uint8\_t shiftBit(uint8\_t byte, uint8\_t bit){

return (byte & (0b1 << bit));

}

bool on\_i2c\_connect(void \*user\_data, uint32\_t address, bool connect) {

return true;

}

uint8\_t on\_i2c\_read(void \*user\_data) {

chip\_state\_t \*chip = user\_data;

if (CHIP\_ON) {

chip->buffer = 0;

if (getNbit(chip->config\_dup, 7) == 1)

chip->buffer |= pin\_read(chip->PIN\_P7)

^ getNbit(chip->polarity\_dup, 7);

chip->buffer<<=1;

if (getNbit(chip->config\_dup, 6) == 1)

chip->buffer |= pin\_read(chip->PIN\_P6)

^ getNbit(chip->polarity\_dup, 6);

chip->buffer<<=1;

if (getNbit(chip->config\_dup, 5) == 1)

chip->buffer |= pin\_read(chip->PIN\_P5)

^ getNbit(chip->polarity\_dup, 5);

chip->buffer<<=1;

if (getNbit(chip->config\_dup, 4) == 1)

chip->buffer |= pin\_read(chip->PIN\_P4)

^ getNbit(chip->polarity\_dup, 4);

chip->buffer<<=1;

if (getNbit(chip->config\_dup, 3) == 1)

chip->buffer |= pin\_read(chip->PIN\_P3)

^ getNbit(chip->polarity\_dup, 3);

chip->buffer<<=1;

if (getNbit(chip->config\_dup, 2) == 1)

chip->buffer |= pin\_read(chip->PIN\_P2)

^ getNbit(chip->polarity\_dup, 2);

chip->buffer<<=1;

if (getNbit(chip->config\_dup, 1) == 1)

chip->buffer |= pin\_read(chip->PIN\_P1)

^ getNbit(chip->polarity\_dup, 1);

chip->buffer<<=1;

if (getNbit(chip->config\_dup, 0) == 1)

chip->buffer |= pin\_read(chip->PIN\_P0)

^ getNbit(chip->polarity\_dup, 0);

if (chip->last\_input != chip->buffer){

chip->last\_input = chip->buffer;

pin\_write(chip->PIN\_INT, HIGH);

pin\_write(chip->PIN\_INT, LOW);

}

#ifdef DEBUG

printf("Sent from I2C^0x%02X: ", chip->address);

printf("0x%02X (", chip->buffer);

printBits(1, &chip->buffer);

printf(")\n");

#endif // DEBUG

chip->command = 0xFF;

return chip->buffer;

}

return 0;

}

static void update\_state(chip\_state\_t \*chip)

{

if (chip->command == 0xFF){

// Записываем команду

chip->command = chip->buffer;

} else {

// Записываем данные

switch (chip->command) {

case PCA9538\_IN\_REG: // 0x00

chip->command = 0xFF;

break;

case PCA9538\_OUT\_REG: // 0x01

if (getNbit(chip->config\_dup, 0) == 0)

pin\_write(chip->PIN\_P0, (getNbit(chip->polarity\_dup, 0)

^ getNbit(chip->buffer, 0)) ? HIGH : LOW);

if (getNbit(chip->config\_dup, 1) == 0)

pin\_write(chip->PIN\_P1, (getNbit(chip->polarity\_dup, 1)

^ getNbit(chip->buffer, 1)) ? HIGH : LOW);

if (getNbit(chip->config\_dup, 2) == 0)

pin\_write(chip->PIN\_P2, (getNbit(chip->polarity\_dup, 2)

^ getNbit(chip->buffer, 2)) ? HIGH : LOW);

if (getNbit(chip->config\_dup, 3) == 0)

pin\_write(chip->PIN\_P3, (getNbit(chip->polarity\_dup, 3)

^ getNbit(chip->buffer, 3)) ? HIGH : LOW);

if (getNbit(chip->config\_dup, 4) == 0)

pin\_write(chip->PIN\_P4, (getNbit(chip->polarity\_dup, 4)

^ getNbit(chip->buffer, 4)) ? HIGH : LOW);

if (getNbit(chip->config\_dup, 5) == 0)

pin\_write(chip->PIN\_P5, (getNbit(chip->polarity\_dup, 5)

^ getNbit(chip->buffer, 5)) ? HIGH : LOW);

if (getNbit(chip->config\_dup, 6) == 0)

pin\_write(chip->PIN\_P6, (getNbit(chip->polarity\_dup, 6)

^ getNbit(chip->buffer, 6)) ? HIGH : LOW);

if (getNbit(chip->config\_dup, 7) == 0)

pin\_write(chip->PIN\_P7, (getNbit(chip->polarity\_dup, 7)

^ getNbit(chip->buffer, 7)) ? HIGH : LOW);

chip->command = 0xFF;

break;

case PCA9538\_POL\_REG: // 0x02

chip->polarity\_dup = chip->buffer;

chip->command = 0xFF;

break;

case PCA9538\_CONF\_REG: // 0x03

pin\_mode(chip->PIN\_P0, getNbit(chip->buffer, 0)

? INPUT\_PULLDOWN : OUTPUT);

pin\_mode(chip->PIN\_P1, getNbit(chip->buffer, 1)

? INPUT\_PULLDOWN : OUTPUT);

pin\_mode(chip->PIN\_P2, getNbit(chip->buffer, 2)

? INPUT\_PULLDOWN : OUTPUT);

pin\_mode(chip->PIN\_P3, getNbit(chip->buffer, 3)

? INPUT\_PULLDOWN : OUTPUT);

pin\_mode(chip->PIN\_P4, getNbit(chip->buffer, 4)

? INPUT\_PULLDOWN : OUTPUT);

pin\_mode(chip->PIN\_P5, getNbit(chip->buffer, 5)

? INPUT\_PULLDOWN : OUTPUT);

pin\_mode(chip->PIN\_P6, getNbit(chip->buffer, 6)

? INPUT\_PULLDOWN : OUTPUT);

pin\_mode(chip->PIN\_P7, getNbit(chip->buffer, 7)

? INPUT\_PULLDOWN : OUTPUT);

chip->config\_dup = chip->buffer;

chip->command = 0xFF;

break;

default:

printf("PCA9538: Error! Wrong command.\n");

}

}

}

bool on\_i2c\_write(void \*user\_data, uint8\_t data) {

chip\_state\_t \*chip = user\_data;

if (CHIP\_ON) {

#ifdef DEBUG

printf("Sent to I2C^0x%02X: ", chip->address);

printf("0x%02X (", data);

printBits(1, &data);

printf(")\n");

#endif // DEBUG

chip->buffer = data;

update\_state(chip);

return true; // ACK

}

return false; // NACK

}

void on\_i2c\_disconnect(void \*user\_data) {

// Do nothing

}