

**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 элемента в онлайн-симуляторе Wokwi

Выполнил:
Студент группы КБ-211



Коренев Д.Н.

Принял:

Оглавление

0. Введение. Используемая терминология	3
0.1. Терминология	3
1. Добавление нового элемента	4
1.1. Файл .json	6
1.1.1. Поле "pins"	6
1.1.2. Поле "controls"	7
1.1.2. Поле "display"	8
1.2. Файл .c	9
1.2.1. API для GPIO	10
1.2.2. API для работы с аналоговыми сигналами	11
1.2.3. API для работы со временем (таймеры)	12
1.2.4. API для UART	13
1.2.5. API для I2C	14
1.2.6. API для SPI	16
1.2.7. Атрибуты	18
1.2.8. API для видеобуфера	19
2. Создание платы PCA9538	20
2.1. Тестирование	31
Приложения	34

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).

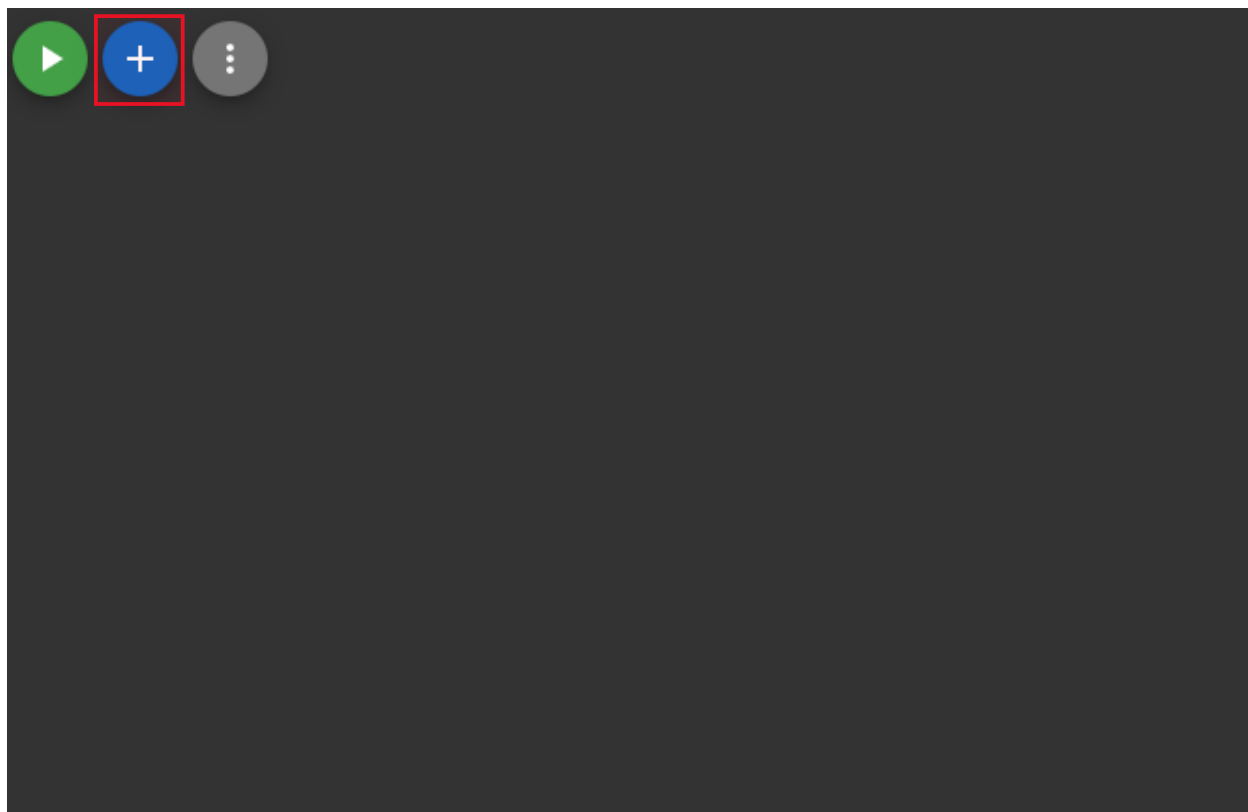


Рисунок 1. Меню схемы

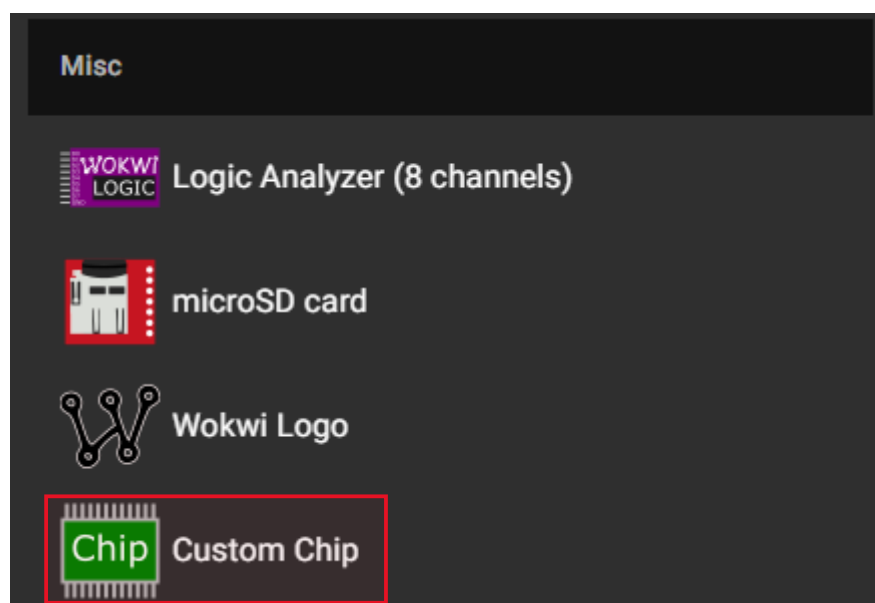


Рисунок 2. Меню добавления нового элемента

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

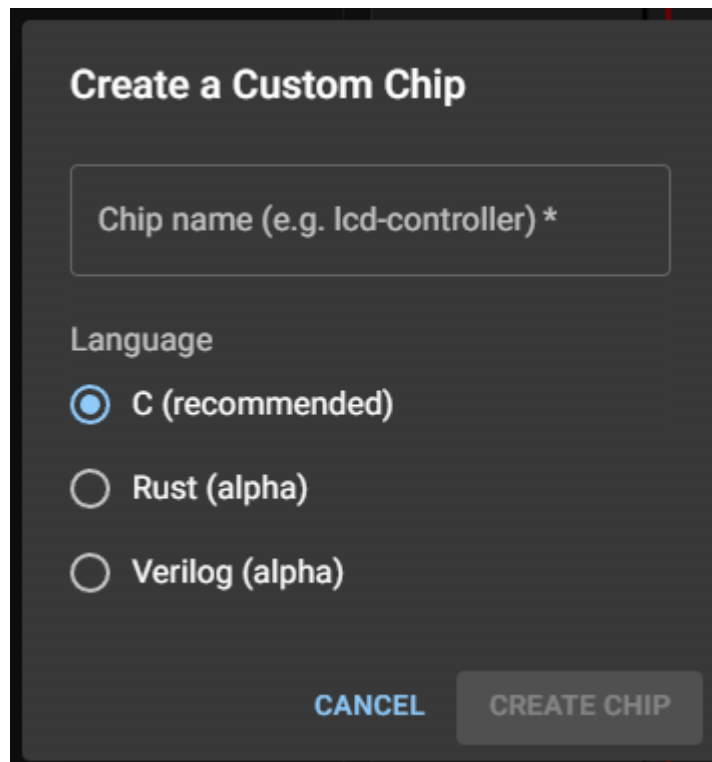


Рисунок 4. Меню создания элемента

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).

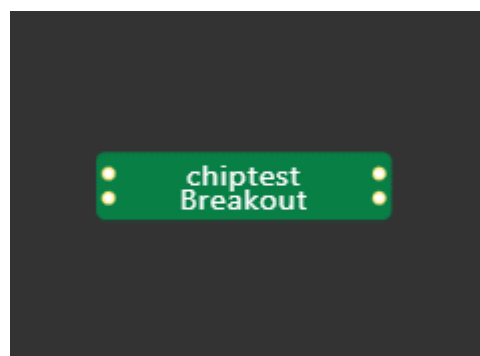


Рисунок 4. Новая плата на схеме

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).

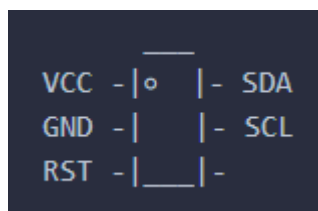


Рисунок 5. Схема примера распиновки платы

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, визуальное 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 128x64.

```
"display": {  
  "width": 128,  
  "height": 64  
},
```

Листинг 3. Пример конфигурации дисплея

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

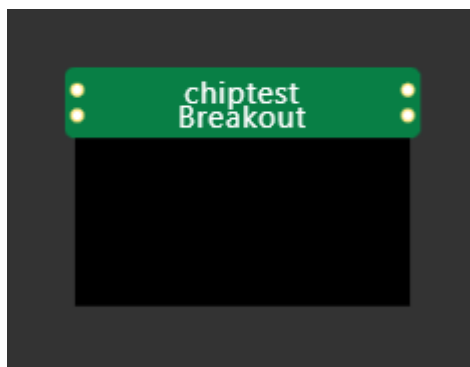


Рисунок 7. Пример дисплея на схеме

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.c"

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 <code>pin_change()</code>
-----------	--------	--

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 <code>NO_PIN</code> для отключения RX)
tx	pin_t	Вывод TX (или <code>NO_PIN</code> 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 большого 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 получаемые 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 завершённый 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>
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).

DB, DBQ, DGV, DW, OR PW PACKAGE
(TOP VIEW)

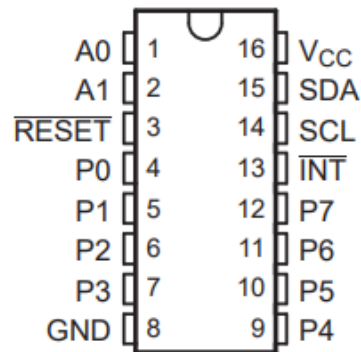


Рисунок 8. Схема подключения PCA9538

```
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:

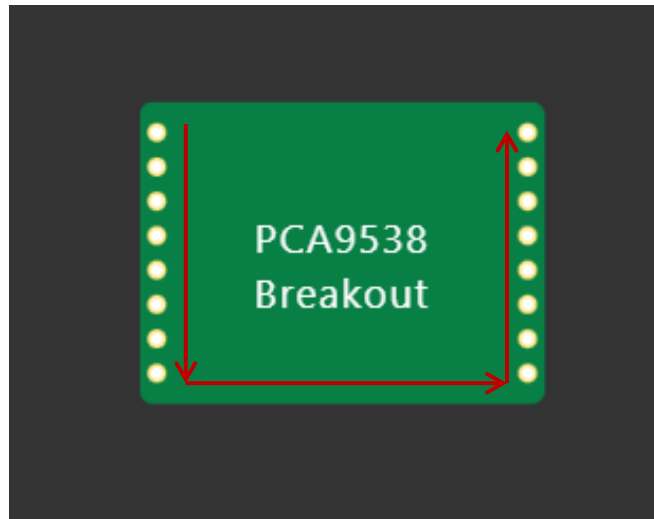


Рисунок 9. PCA9538 на схеме (стрелками указано расположение пинов, от 0-го до 15-го)

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;         // Текущая 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).

INPUTS		I ² C BUS SLAVE ADDRESS
A1	A0	
L	L	112 (decimal), 70 (hexadecimal)
L	H	113 (decimal), 71 (hexadecimal)
H	L	114 (decimal), 72 (hexadecimal)
H	H	115 (decimal), 73 (hexadecimal)

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.

```
chip-pca9538 PCA9538 initialized on I2C with address 0x70!
```

```
chip-pca9538 PCA9538 initialized on I2C with address 0x71!
```

```
chip-pca9538 PCA9538 initialized on I2C with address 0x72!
```

```
chip-pca9538 PCA9538 initialized on I2C with address 0x73!
```

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:

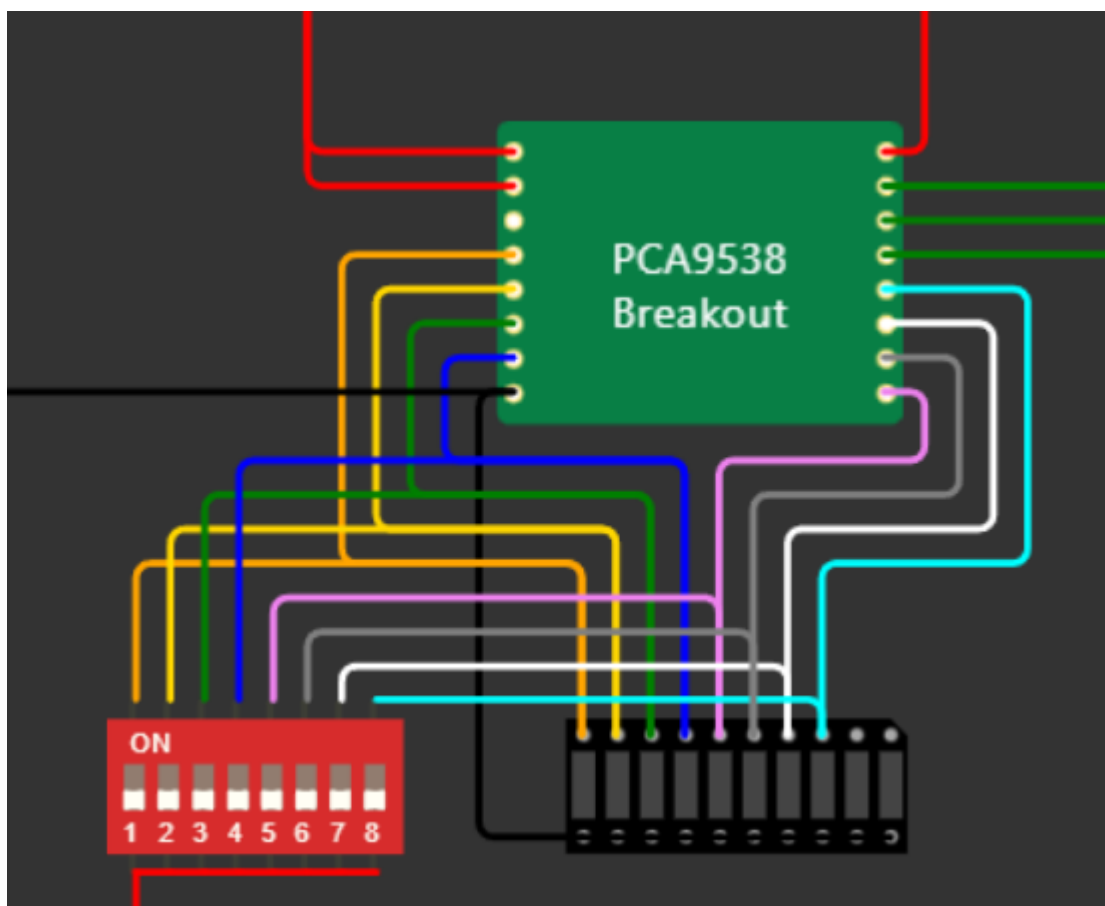


Рисунок 12. Тестовая схема для платы PCA9538

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.

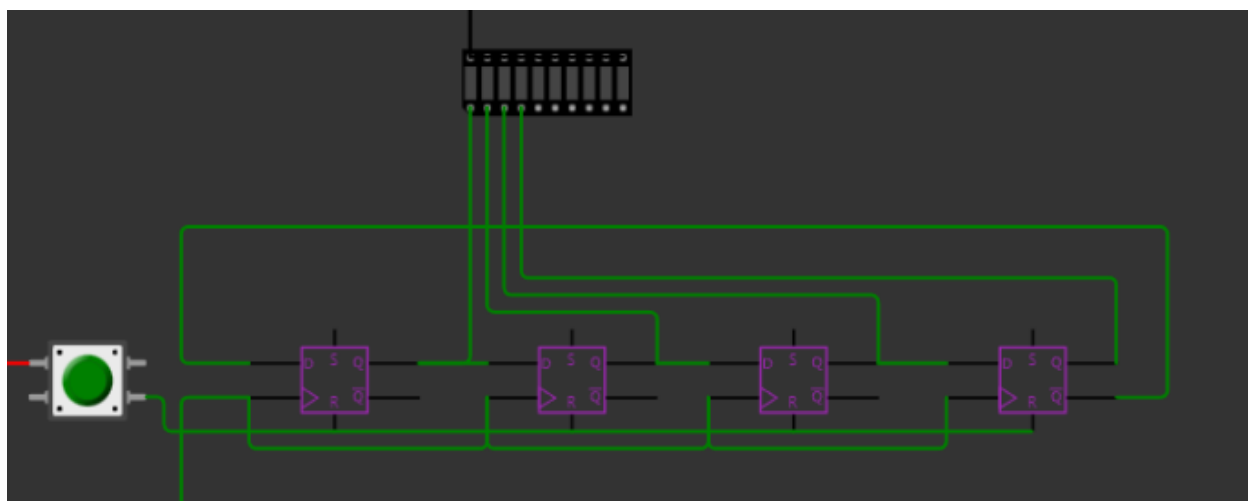


Рисунок 13. 4-х битный счетчик на D-триггерах

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

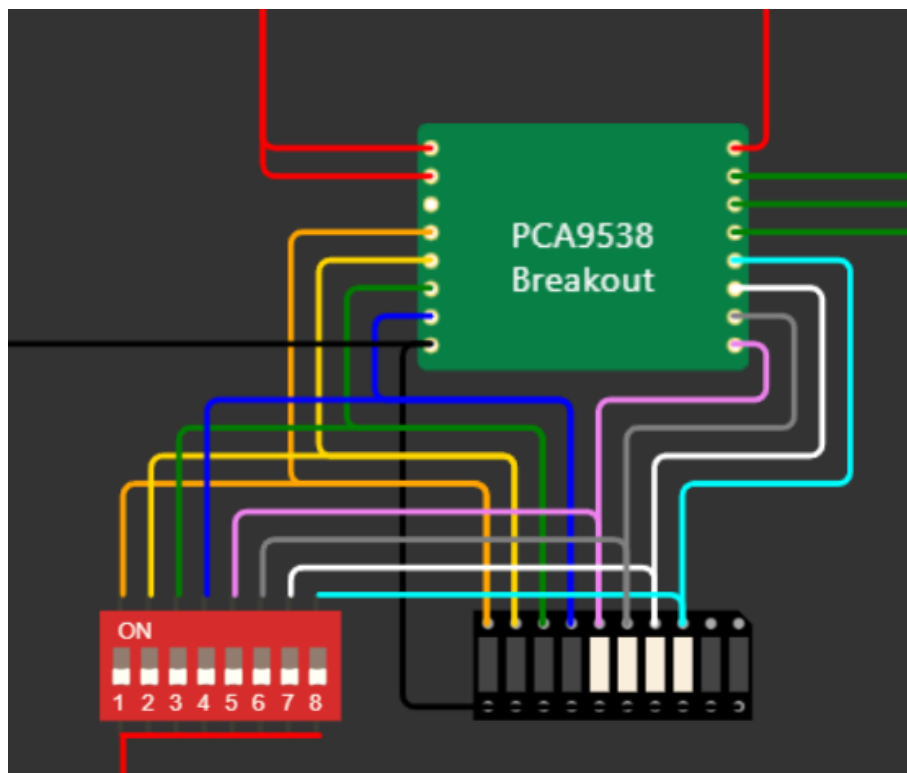


Рисунок 14. На входе платы 0b0000, на выходе 0b1111

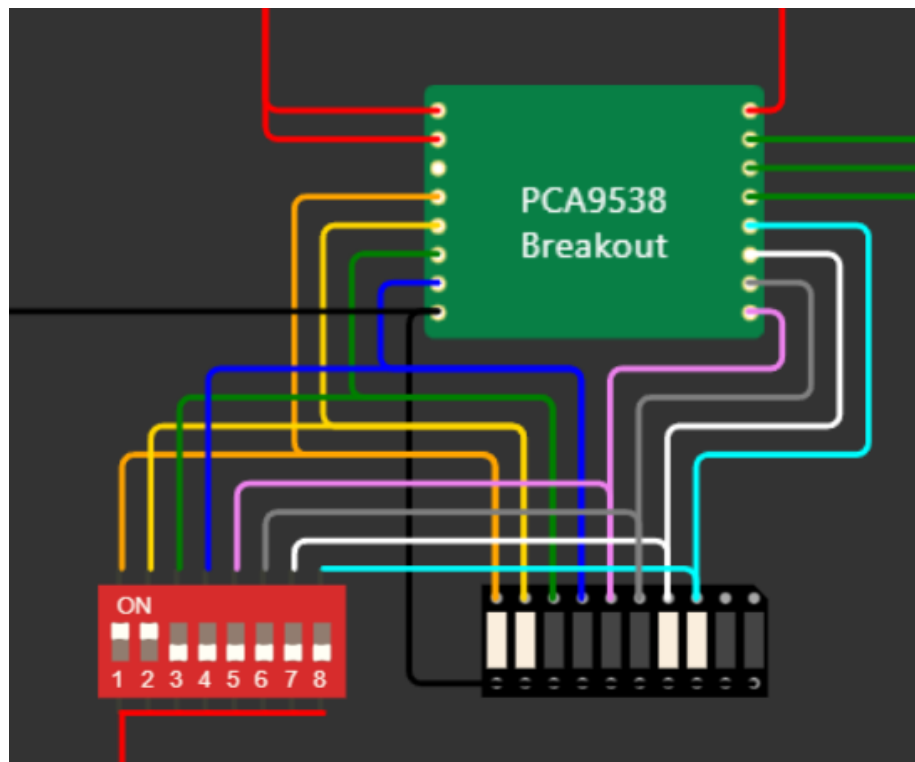


Рисунок 15. На входе платы 0b0011, на выходе 0b1100

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

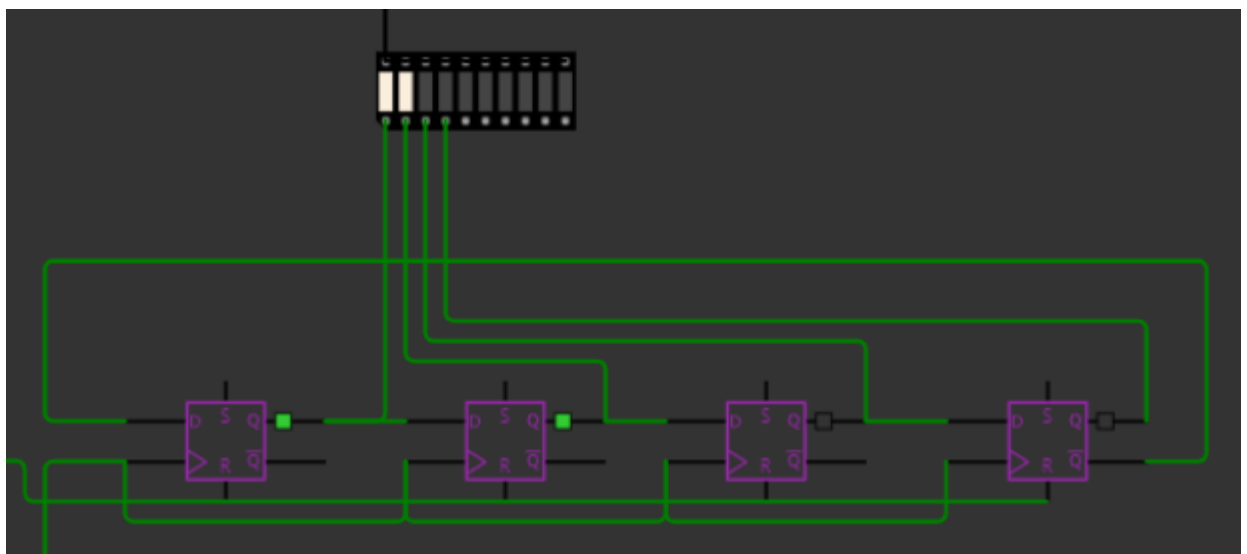


Рисунок 16. Состояние счётчика после изменений сигналов

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;         // Текущая 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
}

```