Санкт-Петербургский политехнический университет Петра Великого

Институт металлургии, машиностроения и транспорта

Высшая школа автоматизации и робототехники

Курсовая работа

Дисциплина: Программирование на языках высокого уровня

Тема: Драйвер для работы с TMC2209 через UART протокол для RPi Pico

Выполнил студент гр. 3331506/10401 Каримуллин С.С.

Преподаватель Ананьевский М.С.

«____»_____ 2024 г.

Санкт-Петербург

2024

Содержание

Введение

Шаговые двигатели играют ключевую роль в современных системах автоматизации и робототехники благодаря своей способности точно контролировать положение. Однако, для их эффективного управления требуются специализированные драйверы, которые обеспечивают плавное и точное движение, минимизируя шум и вибрации. Одним из таких передовых решений является драйвер шаговых двигателей TMC2209 от компании Trinamic. Этот драйвер отличается высокой производительностью, надежностью и многочисленными инновационными функциями, такими как интеллектуальное управление током, режимы работы с низким уровнем шума и поддержка широкого диапазона напряжений. Введение TMC2209 в проекты на базе шаговых двигателей значительно улучшает качество и стабильность работы устройств, делая его популярным выбором среди инженеров и разработчиков.

Драйвер шаговых двигателей TMC2209 используется для управления шаговыми двигателями в различных приложениях, где требуется точное и плавное движение. Вот основные области применения и преимущества использования TMC2209:

1. **3D-принтеры**: TMC2209 обеспечивает тихую и точную работу шаговых двигателей, что критически важно для получения высококачественных отпечатков. Он снижает уровень шума и вибрации, что делает работу принтера практически бесшумной.
2. **ЧПУ станки**: В системах числового программного управления (ЧПУ) TMC2209 используется для обеспечения высокой точности позиционирования. Его функции управления током и микрошагового режима улучшают плавность и точность перемещения инструментов.
3. **Робототехника**: В робототехнических устройствах точное управление движением необходимо для выполнения сложных задач. TMC2209 позволяет управлять движением роботов с высокой степенью контроля и предсказуемости.
4. **Автоматизация и производство**: В системах автоматизации, таких как конвейеры и сборочные линии, TMC2209 помогает достичь точного контроля

движений, что повышает общую эффективность и производительность производства.

5. **Оборудование для медицинских и лабораторных исследований**: В медицине и лабораториях требуется высокая точность и надежность. TMC2209 обеспечивает эти характеристики, что делает его подходящим для управления различными приборами и устройствами.

Основные преимущества TMC2209 включают:

- **Тихая работа**: Режим StealthChop™ позволяет значительно снизить уровень шума, делая двигатель практически бесшумным.
- **Интеллектуальное управление током**: Технология SpreadCycle™ обеспечивает плавное и стабильное движение даже на высоких скоростях.
- **Энергоэффективность**: Режимы работы, такие как CoolStep™, позволяют экономить энергию, автоматически регулируя ток в зависимости от нагрузки.
- **Легкость интеграции**: Простота настройки и наличие интерфейсов UART для конфигурирования делают TMC2209 удобным в использовании даже для начинающих разработчиков.

Таким образом, TMC2209 является универсальным решением для множества приложений, где требуется точное, плавное и тихое управление шаговыми двигателями.

Написание драйвера для TMC2209 под Pico SDK важно по нескольким причинам:

1. Оптимизация под конкретное оборудование: Raspberry Pi Pico, основанный на микроконтроллере RP2040, обладает уникальными характеристиками и возможностями, такими как двойные ядра ARM Cortex-M0+ и гибкая система ввода-вывода. Создание драйвера под Pico SDK позволяет оптимально использовать эти возможности, обеспечивая эффективное управление шаговыми двигателями через TMC2209.

2. Упрощение разработки: Pico SDK предоставляет разработчикам удобные инструменты и библиотеки, упрощающие процесс разработки и интеграции. Написание драйвера для TMC2209 с использованием этого SDK снижает сложность кода и минимизирует количество потенциальных ошибок, что ускоряет разработку и тестирование приложений.

3. Повышение производительности и надежности: Поскольку Pico SDK тщательно оптимизирован для работы с аппаратными средствами Raspberry Pi Pico, драйвер для TMC2209, написанный с его использованием, может обеспечить более высокую производительность и надежность по сравнению с универсальными или плохо оптимизированными решениями.

4. Использование расширенных возможностей: Pico SDK поддерживает различные функции и протоколы, такие как SPI, I2C и UART, которые могут быть использованы для коммуникации с TMC2209. Это позволяет реализовать сложные схемы управления и мониторинга, такие как настройка параметров драйвера в реальном времени или диагностика работы шагового двигателя.

5. Сообщество и поддержка: Raspberry Pi Pico и Pico SDK имеют большое и активное сообщество разработчиков, которое постоянно предоставляет обновления, исправления ошибок и новые функции. Создание драйвера для TMC2209 под этим SDK позволяет разработчикам воспользоваться поддержкой сообщества и ускорить решение возникающих проблем.

6. Расширение возможностей приложений: Наличие драйвера для TMC2209 под Pico SDK открывает новые возможности для разработчиков, позволяя интегрировать шаговые двигатели в широкий спектр проектов — от 3D-принтеров и ЧПУ станков до робототехники и систем автоматизации.

Таким образом, написание драйвера для TMC2209 под Pico SDK обеспечивает оптимальную производительность, простоту интеграции, надежность и поддержку сообщества, что делает разработку и эксплуатацию шаговых двигателей более эффективной и удобной.

Подключение драйвера TMC2209 к RPi Pico

В ранних версиях драйверов шаговых двигателей (A4988, DRV8825, TB6600 и тд.) была реализована довольно простая система управления управления. Для управления использовались два цифровых пина микроконтроллера, STEP и DIR. DIR отвечал за направление движения мотора, например: если на пине DIR установлена логическая единица, мотор будет крутиться вперёд, а если логический ноль, то назад. STEP же отвечает за активацию шага шагового двигателя т.е., как только на пине STEP появилась логическая единица, вал двигателя сдигается на один шаг. В шаговом двигателе NEMA 17, который является наиболее популярный в 3D-принтерах, станках и робототехнике, 1 шаг равен 1.8 градуса, поэтому чтобы вал шагового двигателя совершил 1 полный оборот, необходимо сделать 200 шагов т.е. переключить пин STEP 200 раз. В некоторых драйверах есть функия дробления шага, которая позволяет добиться ещё большей точности двигателя. Например, в драйвере DRV8825 можно добиться дробления шага до 1/8 (1.8 / 8 = 0.225 градуса за один шаг), но в драйвере TMC2209 можно добиться более крупного дробления шага, вплоть до 1/256. Настроить дробление можно с помощью выставления нужных логических уровней на входах в драйвер.

Основное отличие TMC2209 от своих более древних аналогов заключается в том, что помимо наличия древнего STEP/DIR интерфейса (разработчики реализовали его лишь с целью обратной совместимости с более старыми драйверами), в драйвере реализован UART интерфейс (в TMC5160 также есть SPI интерфейс), с помощью которого можно полностью выполнять полную настройку драйвера и реализовать управления шаговым двигателем без STEP/DIR интерфейса.

Однопроводной интерфейс обеспечивает однонаправленную работу (только для настройки параметров) или двунаправленную работу для полного контроля и диагностики. Он может управляться любым стандартным микроконтроллером с UART интерфейсом. Однопроводной интерфейс можно использовать как своеобразную шину, к которой можно подключить до 4 драйверов (для этого каждому драйверу

придётся задать свой уникальный адрес, сделать это на программном уровне невозможно, придётся задавать их с помощью логических уровней на микросхему драйвера).

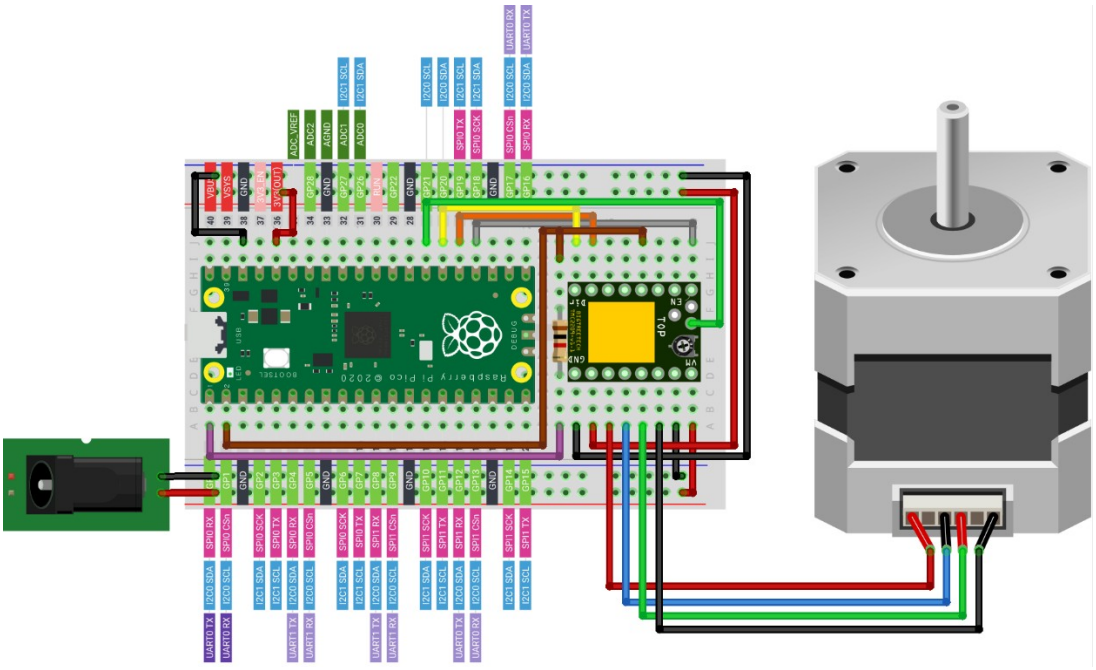

Рисунок 1 – Принципиальная схема тестового стенда

На рисунке 1 представлена схема тестового стенда с микроконтроллером RPi Pico, драйвером BTT TMC2209 V1.2 и шаговым двигателем NEMA 17. Пины UART0 TX и UART0 RX (фиолетовый и коричневый провод) служат для двунаправленного подключения по шине UART между драйвером и микроконтроллером. В целом, этого уже достаточно для полноценного управления драйвером, но мы дополнительно подлючим пины STEP, DIR, EN (отвечает за включение и выключение драйвера) и INDEX (служат для программной диагностики) для использования полного функционала драйвера. На рисунке 2 представлен собранный по схеме стенд

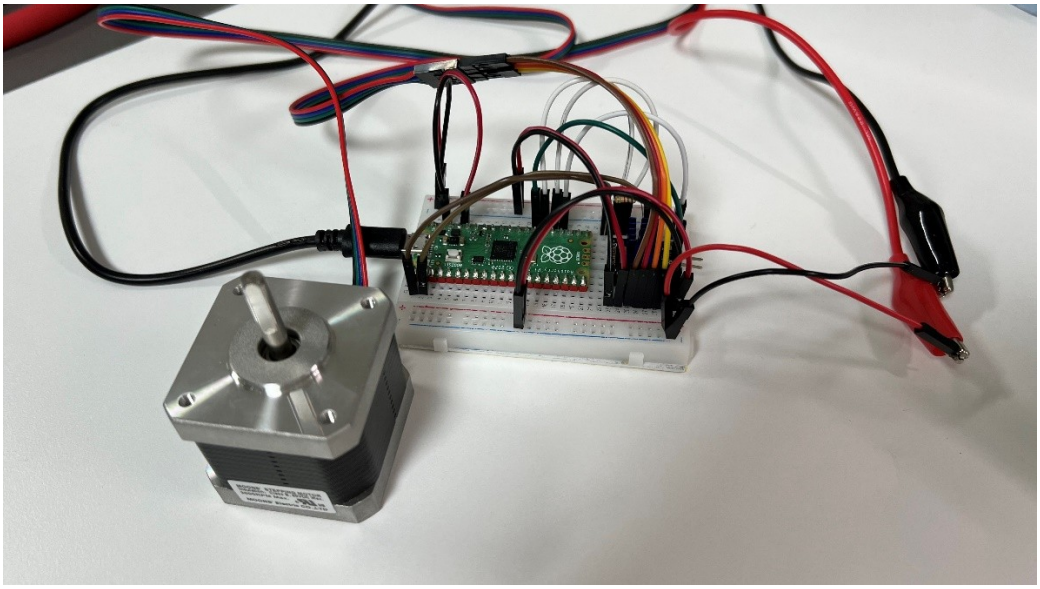

Рисунок 2 – Тестовый стенд

Помимо собранного тестового стенда, я также буду тестировать код на двух специализированных платах для 3D-принтеров, на которых уже установлен микроконтроллер, драйвера, цепи питания и прочая переферия для управления 3D-принтером. Первая плата (рисунок 3), устанавливается на печатную голову 3D-принтера и управляет подающим механизмом, нагревательными элементами, охлаждением и общается с главным контроллером по CAN шине. Вторая плата (рисунок 4), представляет из себя полноценную плату управления 3D-принтером.

Рисунок 3 – Плата контроллера печатной головы

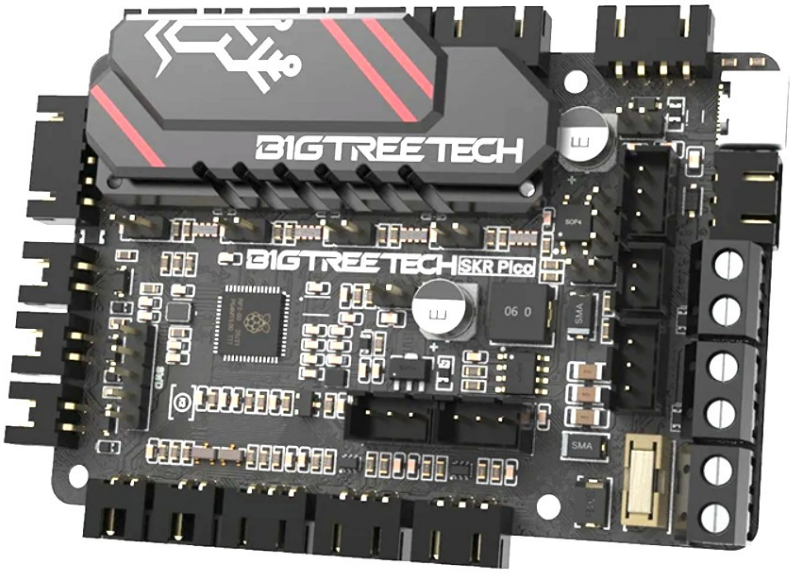

Рисунок 4 – Материнская плата BIGTREETECH BTT SKR PICO

Разработка драйвера для управления TMC2209

Для разработки драйвера обратимся к документации от фирмы Trinamic для драйвера TMC2209. Так как с подключением мы разобрались в предыдущей главе, то сразу перейдём к разделам 4 и 5, в которых есть описания интерфейса управления и карта регистров для управления драйвером. Сразу стоит обратить внимание на структуру датаграммы (рисунок 5), которая поясняет, как нужно правильно передавать информацию по протоколу.

UART WRITE ACCESS DATAGRAM STRUCTURE																				
each byte is LSB...MSB, highest byte transmitted first																				
0 ... 63																				
sync + reserved								**8 bit slave address**			**RW + 7 bit register addr.**			**32 bit data**			**CRC**		
0...7								8...15			16...23			24...55			56...63		
1	0	1	0	Reserved (don't cares but included in CRC)				*SLAVEADDR*=0...3			register address		1	data bytes 3, 2, 1, 0 (high to low byte)			CRC		
0	1	2	3	4	5	6	7	8	...	15	16	...	23	24	...	55	56	...	63

Рисунок 5 – Структура датаграммы

По датаграмме видно, что первые 8 битов сообщения представляют из себя биты синхронизации, которые остаются неизменными при записи данных. Следующие 8 битов выделены под адрес устройства, в который будут записаны данные (адрес устройства задаётся физически путём подтяжки пинов MS0 и MS1 на микросхеме

драйвера). Следующие 8 бит выделеные под указание адреса регистра, в который будут записаны данные. Следующие 32 бита это данные, которые будут записаны. Наконец, последние 8 битов используются для указания контольной суммы. Использование контрольной суммы не обязательна, но её подсчёт рекомендуется проводить после каждого отправленного и принятого сообщения. Алгоритм его подсчёта есть в документации, его необходимо добавить в программу.

C-CODE EXAMPLE FOR CRC CALCULATION

```
void swuart_calcCRC(UCHAR* datagram, UCHAR datagramLength)
{
  int i,j;
  UCHAR* crc = datagram + (datagramLength-1); // CRC located in last byte of message
  UCHAR currentByte;

  *crc = 0;
  for (i=0; i<(datagramLength-1); i++) {      // Execute for all bytes of a message
    currentByte = datagram[i];                // Retrieve a byte to be sent from Array
    for (j=0; j<8; j++) {
      if ((*crc >> 7) ^ (currentByte&0x01))   // update CRC based result of XOR operation
      {
        *crc = (*crc << 1) ^ 0x07;
      }
      else
      {
        *crc = (*crc << 1);
      }
      currentByte = currentByte >> 1;
    } // for CRC bit
  } // for message byte
}
```

Рисунок 6 – Функция подсчёта контрольной суммы.

В ходе этой курсовой работы будет реализован минимальный функционал, необходимый для успешной эксплуатации шагового двигателя, а именно:

1. Включение и выключение драйвера
2. Настройка тока двигателя на удержании
3. Настройка тока двигателя при движение
4. Изменение направления движения двигателя
5. Настройка дробления шага двигателя

Создадим класс TMC2209 с прототипами всех необходимых функций (Рисунок 7)

```
class TMC2209 {
public:
    TMC2209(uart_inst_t* uart_id, uint baud_rate, uint tx_pin, uint rx_pin, uint8_t slave_address);
    void init();
    bool write(uint8_t reg, uint32_t value);
    void setCurrent(uint8_t current);
    void setHoldCurrent(uint8_t current);
    void enableDriver();
    void disableDriver();
    void setSpeed(uint32_t speed);
    void setDirection(bool direction);
    void setMicrostepInput(bool input);
    void setMicrostepResolution(uint8_t resolution);

private:
    uint8_t calculateCrc(uint8_t* data, size_t length);
    void uartInit();
    void gpioInit();
```

Рисунок 7 – Функция подсчёта контрольной суммы.

В поле public указаны прототипы всех функций, которые необходимы для работы драйвера и которые можно использовать в программе, в том числе прототип класса TMC2209, в которые необходимо передать некоторые данные, а именно:

- uart_id — номер uart интерфейса микроконтроллера (UART0 — UART3)
- baud_rate — скорость передачи данных интерфейса (9600 — 115200 bbd)
- tx_pin — передающий пин микроконтроллера
- rx_pin — принимающий пин микроконтроллера
- slave_address — адрес драйвера (0 — 3)

Для удобства обозначения номера адреса драйвера используется итератор (рисунок 8).

```
enum slave_address
{
    DRIVER_1,
    DRIVER_2,
    DRIVER_3,
    DRIVER_4
};
```

Рисунок 8 — Итератор адреса драйвера

Для успешной работы драйвера необходимо проинициализировать uart интерфейс. Инициализация происходит в два этого — сначала необходимо настроить пины микроконтроллера, а затем сам uart интерфейс. Реализацию можно увидеть на рисунке 9.

```
void TMC2209::uartInit() {
    uart_init(uart_id, baud_rate);
    uart_set_translate_crlf((uart_inst_t *) uart_id, false);
    uart_set_format(uart_id, DATA_BITS, STOP_BITS, UART_PARITY_NONE);
    uart_set_fifo_enabled(uart_id, true);
}

void TMC2209::gpioInit(){
    gpio_set_function(tx_pin, GPIO_FUNC_UART);
    gpio_set_function(rx_pin, GPIO_FUNC_UART);
}

void TMC2209::init() {
    uartInit();
    gpioInit();
}
```

Рисунок 9 — Инициализация uart интерфейса

Дальше необходимо реализовать функцию записи данных в регистры драйвера. Для этого необходимо собрать сообщение, подсчитать его контрольную сумму и передать её по протоколу напрямую в драйвер. Реализация продемонстрирована на рисунке 10.

```
bool TMC2209::write(uint8_t regAdress, uint32_t value) {
    uint8_t frame[8];
    frame[0] = TMC2209_SYNC_BIT;
    frame[1] = slave_address;
    frame[2] = regAdress | TMC2209_WRITE_BIT;
    frame[3] = (uint8_t)(value >> 24);
    frame[4] = (uint8_t)(value >> 16);
    frame[5] = (uint8_t)(value >> 8);
    frame[6] = (uint8_t)(value >> 0);
    frame[7] = calculateCrc(frame, 7);
    uart_write_blocking(uart_id, frame, 8);
    return true;
}
```

Рисунок 10 — Запись данных в регистр

Из рисунка 10 видно, что сначала мы создаём целочисленный массив с размерностью 8 бит. В первый элемент массива записывается синхронизирующий бит, во второй элемент адрес драйвера, в третий адрес регистра, с 4 по 7 элементам записываются данные со сдвигом, в последний элемент записывается результат подсчёта контрольной суммы (функция представлена на рисунке 11).

```
uint8_t TMC2209::calculateCrc(uint8_t* data, size_t length) {
    uint8_t crc = 0;
    for (size_t i = 0; i < length; i++) {
        crc ^= data[i];
        for (int j = 0; j < 8; j++) {
            if (crc & 0x01) crc ^= 0x07;
            crc >>= 1;
        }
    }
    return crc;
}
```

Рисунок 11 - Функция подсчёта контрольной суммы

Далее приступим к реализации основных функций драйвера. Сперва это будет функция включения и выключения двигателя. Согласно документации на драйвер TMC2209 за включение и выключение двигателя отвечает регистр GSTAT, который имеет адрес 0x6D. Для включение необходимо записать в регистр логическую единицу, а для выключение нужно записать логический ноль. Реализация представлена на рисунке 12.

```
void TMC2209::enableDriver() {
    write(ADDRES_DSTAT, true);
}

void TMC2209::disableDriver() {
    write(ADDRES_DSTAT, false);
}
```

Рисунок 12 - Функция включения и выключения

Теперь реализуем главную функцию — управление двигателем. Обратимся к карте регистров в документации, чтобы узнать как это реализовать (рисунок 13).

W	0x22	24	*VACTUAL*	*VACTUAL* allows moving the motor by UART control. It gives the motor velocity in +-(2^23)-1 [µsteps / t] 0: Normal operation. Driver reacts to STEP input. /=0: Motor moves with the velocity given by *VACTUAL*. Step pulses can be monitored via INDEX output. The motor direction is controlled by the sign of *VACTUAL*.

Рисунок 13 - Карта регистра VACTUAL

Из описания становится понятно, что для управления двигателем достаточно записать значение скорости. В данном случае, это количество шагов, которые сделает двигатель за одну секунду. Также стоит учесть, что драйвер учитывает дробление шагов двигателя. Поэтому, изменяя дробления, будет меняться и скорость двигателя. Если в регистр записать нулевое значение, то двигатель остановиться либо его управлением займётся внешний пин STEP, функция которого была описана в предыдущей главе. Также, если записать в регистр значение с отрицательным знаком, то двигатель начнёт крутиться в обратном направлении. Это очень удобно, так как для смены направление вращения двигателя нам не нужно реализовывать новые функции. Однако, для наглядности, мы также реализуем другой способ инвертирования вращения двигателя. Запись значений в регистр VACTUAL представлена на рисунке 14.

```
// Allows moving the motor by UART control
// The motor direction is controlled by the sign of VACTUAL
void TMC2209::setSpeed(uint32_t speed) {
    write(ADRESS_VACTUAL, speed); // VACTUAL register
}
```

Рисунок 14 - Функция установки скорости двигателя

Второй способ изменения направления вращения двигателя — изменения бита shaft регистра GSTAT . Регистр GSTAT — один из самых главных в драйвере и отвечает за множество функций. Этот регистр мы будем использовать не только для изменения направления вращения двигателя, но и для включения UART пина и настройки дробления шага. Полная структура регистра показана на рисунке 15

General configuration registers (0x00...0x0F)					
R/W	**Addr**	**n**	***Register***	**Description */ bit names***	
				Bit	***GCONF* – Global configuration flags**
RW	0x00	10	*GCONF*	0	*I_scale_analog* (*Reset default=1*) 0: Use internal reference derived from 5VOUT 1: Use voltage supplied to VREF as current reference
				1	*internal_Rsense* (*Reset default: OTP*) 0: Operation with external sense resistors 1: Internal sense resistors. Use current supplied into VREF as reference for internal sense resistor. VREF pin internally is driven to GND in this mode.
				2	*en_SpreadCycle* (*Reset default: OTP*) 0: StealthChop PWM mode enabled (depending on velocity thresholds). Initially switch from off to on state while in stand still, only. 1: SpreadCycle mode enabled A high level on the pin SPREAD inverts this flag to switch between both chopper modes.
				3	*shaft* 1: Inverse motor direction
				4	*index_otpw* 0: INDEX shows the first microstep position of sequencer 1: INDEX pin outputs overtemperature prewarning flag (*otpw*) instead
				5	*index_step* 0: INDEX output as selected by *index_otpw* 1: INDEX output shows step pulses from internal pulse generator (toggle upon each step)
				6	*pdn_disable* 0: PDN_UART controls standstill current reduction 1: PDN_UART input function disabled. Set this bit, when using the UART interface!
				7	*mstep_reg_select* 0: Microstep resolution selected by pins MS1, MS2 1: Microstep resolution selected by MSTEP register
				8	*multistep_filt* (*Reset default=1*) 0: No filtering of STEP pulses 1: Software pulse generator optimization enabled when fullstep frequency > 750Hz (roughly). TSTEP shows filtered step time values when active.
				9	*test_mode* 0: Normal operation 1: Enable analog test output on pin ENN (pull down resistor off), ENN treated as enabled. *IHOLD*[1..0] selects the function of DCO: 0...2: T120, DAC, VDDH *Attention: Not for user, set to 0 for normal operation!*

Рисунок 15 - Полное описание регистра GCONF

Для удобства перенесём все биты регистра в отдельную структура (рисунок 16). В дальнейшем, если нам потребуется обратиться к каким то битам регистра, это будет проще.

```
//ADDRESS_GCONF register and bytes
const static uint8_t ADDRESS_GCONF = 0x00;

union GlobalConfig
{
    struct
    {
        uint32_t i_scale_analog : 1;
        uint32_t internal_rsense : 1;
        uint32_t enable_spread_cycle : 1;
        uint32_t shaft : 1;
        uint32_t index_otpw : 1;
        uint32_t index_step : 1;
        uint32_t pdn_disable : 1;
        uint32_t mstep_reg_select : 1;
        uint32_t multistep_filt : 1;
        uint32_t test_mode : 1;
        uint32_t reserved : 22;
    };
    uint32_t bytes;
};
GlobalConfig driver_config;
```

Рисунок 16 - Объявление регистра GCONF и его битов

Использование структуры намного упрощает написание кода, функция изменения направления вращения будет иметь вот такой вид (рисунок 17).

```
void TMC2209::setDirection(bool direction){
    driver_config.shaft = direction;
    write(ADDRESS_GCONF, driver_config.bytes);
}
```

Рисунок 17 - Функция инвертирования вращения

Теперь перейдём ещё к одной из главных функций драйвера — настройкам тока. Изначально, в старый драйверах ток движения и удержания выставлялся вручную путем изменения сопротивления на определённый пин драйвера. TMC2209 в целых обратной совместимости также сохранил эту функцию, но теперь ток движения и удержания можно настраивать отдельно через UART интерфейс. Для этого обратимся к документации, а именно в регистру IHOLD_IRUN (рисунок 18).

VELOCITY DEPENDENT DRIVER FEATURE CONTROL REGISTER SET (0X10...0X1F)

R/W	Addr	n	*Register*	Description */ bit names*	
				Bit	***IHOLD_IRUN* – Driver current control**
				4..0	*IHOLD* (*Reset default: OTP*) Standstill current (0=1/32 ... 31=32/32) In combination with StealthChop mode, setting *IHOLD*=0 allows to choose freewheeling or coil short circuit (passive braking) for motor stand still.
W	0x10	5 + 5 + 4	*IHOLD_IRUN*	12..8	*IRUN* (*Reset default=31*) Motor run current (0=1/32 ... 31=32/32) *Hint:* Choose sense resistors in a way, that normal *IRUN* is 16 to 31 for best microstep performance.
				19..16	*IHOLDDELAY* (*Reset default: OTP*) Controls the number of clock cycles for motor power down after standstill is detected (*stst*=1) and *TPOWERDOWN* has expired. The smooth transition avoids a motor jerk upon power down. 0: instant power down 1..15: Delay per current reduction step in multiple of 2^18 clocks

Рисунок 18 - Полное описание регистра IHOLD_IRUN

Для удобства поступим также, как и с регистром GCONF (рисунок 19)

```
union DriverCurrent
{
    struct
    {
        uint32_t ihold : 5;
        uint32_t reserved_0 : 3;
        uint32_t irun : 5;
        uint32_t reserved_1 : 3;
        uint32_t iholddelay : 4;
        uint32_t reserved_2 : 12;
    };
    uint32_t bytes;
};
DriverCurrent driver_current;
```

Рисунок 19 - Обявление регистра IHOLF_iRUN и его

Биты IHOLD и IRUN принимают 32 различных значения. В характеристиках драйвера указано, что максимальный ток на движение и удержание 3.2А. Из этого можно понять, что драйвер позволяет изменять ток в диапазоне от 0 до 3.2А с шагом в 100мА. Функции для настройка токов на движение и удержания указаны на рисунке 20.

```
void TMC2209::setCurrent(uint8_t current) {
    if(current > 3200){
        std::cout << "Error: overcurrent";
    }
    else{
        driver_current.irun = current % 100;
        write(ADRESS_IHOLD_IRUN, driver_current.bytes);
    }
}

void TMC2209::setHoldCurrent(uint8_t current) {
    if(current > 3200){
        std::cout << "Error: overcurrent";
    }
    else{
        driver_current.ihold = current % 100;
        write(ADRESS_IHOLD_IRUN, driver_current.bytes);
    }
}
```

Рисунок 20 - Функции для настройки токов на движение и удержание

Реализуем последнюю, но не менее важную функцию из списка — настройку дробления шагов. Настройку можно выполнить физически, изменяя логические уровни на определённых пинах драйвера. Но мы выполним настройку по UART интерфейсу. Для этого нам нужны биты 24-27 регистра CHOPCONF по адресу 0x6C (рисунок 21). Также стоит учесть, что для программной настройки дробления шагов необходимо отключить физическую настройку, для этого нужно записать в бит mstep_reg_select регистра GCONF (рисунок 15) логическую единицу. Так как мы заранее объявили все биты регистра, написать функцию будет несложно (рисунок 22).

27	*mres3*	*MRES*	%0000:
26	*mres2*	micro step resolution	Native 256 microstep setting.
25	*mres1*		%0001 … %1000:
24	*mres0*		128, 64, 32, 16, 8, 4, 2, FULLSTEP Reduced microstep resolution. The resolution gives the number of microstep entries per sine quarter wave. When choosing a lower microstep resolution, the driver automatically uses microstep positions which result in a symmetrical wave. Number of microsteps per step pulse = 2^*MRES* (Selection by pins unless disabled by *GCONF.mstep_reg_select*)

Рисунок 21 - Описание битов mres регистра CHOPCONF

```
void TMC2209::setMicrostepInput(bool input){
    driver_config.mstep_reg_select = input;
    write(ADDRESS_GCONF, driver_config.bytes);
}
```

Рисунок 22 - Функция выбора способа дробления

Для настройки дробления создадим итератор для удобной настройки дробления (рисунок 23), также создадим отдельную структура со всеми битами регистра CHOPCONF (рисунок 24). В конце создадим простую функцию для настройки дробления (рисунок 25).

```
enum microstep_resolution
{
    MICROSTEP_RESOLUTION_256,
    MICROSTEP_RESOLUTION_128,
    MICROSTEP_RESOLUTION_64,
    MICROSTEP_RESOLUTION_32,
    MICROSTEP_RESOLUTION_16,
    MICROSTEP_RESOLUTION_8,
    MICROSTEP_RESOLUTION_4,
    MICROSTEP_RESOLUTION_2,
    MICROSTEP_RESOLUTION_FULL_STEP
};
```

Рисунок 23 - Итератор выбора шагов дробления

```
#define ADRESS_CHOPCONF     0x6C

union ChopperConfig
{
    struct
    {
        uint32_t toff : 4;
        uint32_t hstrt : 3;
        uint32_t hend : 4;
        uint32_t reserved_0 : 4;
        uint32_t tbl : 2;
        uint32_t vsense : 1;
        uint32_t reserved_1 : 6;
        uint32_t mres : 4;
        uint32_t interpolation : 1;
        uint32_t double_edge : 1;
        uint32_t diss2g : 1;
        uint32_t diss2vs : 1;
    };
    uint32_t bytes;
};
ChopperConfig chopper_config;
```

Рисунок 24 - бит CHOPCONF и его биты

```
void TMC2209::setMicrostepResolution(uint8_t resolution){
    chopper_config.mres = resolution;
    write(ADRESS_CHOPCONF, chopper_config.bytes);
}
```

Рисунок 25 - Функция настройки дробления

На основе драйвера напишем небольшую программу для минимальной настройки и управления шаговым двигателем с помощью TMC2209 на Pico SDK (рисунок 26).

```cpp
#include <stdio.h>
#include "pico/stdlib.h"
#include "TMC2209_driver.h"

int main() {
    stdio_init_all();

    TMC2209 tmc2209(uart0, 115200, 0, 1, DRIVER_1);
    tmc2209.init();
    tmc2209.enableDriver();
    tmc2209.setHoldCurrent(500);
    tmc2209.setCurrent(1000);
    tmc2209.setMicrostepInput(true);
    tmc2209.setMicrostepResolution(MICROSTEP_RESOLUTION_16);

    bool direction = true;

    while(true){
        tmc2209.setDirection(direction);
        tmc2209.setSpeed(1000);
        sleep_ms(1000);
        direction = !direction;
    }
}
```

Рисунок 26 - Простая программа для управления двигателем

В программе реализована простейшая настройка драйвера. В бесконечном цикле двигатель крутится со скоростью 1000 шагов в секунду и меняет направление вращения каждую секунду.

Вывод

В ходе работы были изучены основные нюансы управления шаговыми двигателями, изучена документация на драйвер TMC2209, разработана и собрана электрическая схема испытательного стенда. Также на основе документации и примерах в интернете был разработан драйвер, который обеспечивает минимально необходимую настройку драйвера для успешной работы. Также были реализованы основные функции управления драйвера.