МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное образовательное учреждение высшего образования

«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

КАФЕДРА №23

КУРСОВАЯ РАБОТА
ЗАЩИЩЕНА С ОЦЕНКОЙ

РУКОВОДИТЕЛЬ

канд. техн. наук, доцент		В. А. Ненашев
должность, уч. степень, звание	подпись, дата	инициалы, фамилия

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КУРСОВОЙ РАБОТЕ

РАЗРАБОТКА ПРОГРАММЫ ФОРМИРОВАНИЯ ШИМ ДЛЯ УПРАВЛЕНИЯ ДВИГАТЕЛЯМИ

по дисциплине: МИКРОПРОЦЕССОРНЫЕ ИНФОРМАЦИОННО-ИЗМЕРИТЕЛЬНЫЕ И УПРАВЛЯЮЩИЕ УСТРОЙСТВА

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ гр. №	2337М		А. В. Развaляев
		подпись, дата	инициалы, фамилия

Санкт-Петербург 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Современные технологии управления электрическими двигателями играют ключевую роль в развитии промышленности и автоматизации. В последние годы существенно возрос интерес к бесколлекторным и асинхронным электродвигателям, что связано с их высокой эффективностью и надежностью. В данной работе представлено программное обеспечения для управления такими двигателями.

Объектом исследования являются бесколлекторные и асинхронные двигатели, которые благодаря своей конструкционной особенности и возможностям работы с минимальными потерями энергии становятся все более востребованными в различных отраслях.

В работе также анализируются достоинства платформы STM32, использующейся для разработки программного обеспечения, что обеспечивает высокую производительность и эффективность за счет особенностей архитектуры и возможности легкого программирования. Все эти аспекты представляют значительный интерес для предприятий, стремящихся внедрить передовые технологические решения в свои производственные процессы, и подтверждают необходимость дальнейшего изучения и применения данных технологий в управлении электродвигателями.

Целью работы является разработка программного обеспечения для управления ШИМ, записи логов и обеспечения базового интерфейса взаимодействия. Задачи:

- разработать программные модули для генерации и управления ШИМ;
- реализовать механизм логирования данных;
- реализовать интерфейс для взаимодействия пользователя с системой;
- провести тестирование разработанных модулей.

1 Управление двигателями

Современные электрические двигатели широко применяются в промышленности, транспорте и бытовой технике. Их управление позволяет регулировать скорость вращения, момент, направление движения и другие параметры работы, что делает их универсальным инструментом для автоматизации. Основные подходы к управлению двигателями зависят от типа двигателя. В данной работе будет рассмотрено разомкнутое управление, бесколлекторным и синхронным или асинхронным двигателем.

Следует отметить, что схема управления двигателем электрически развязана от схемы логики микроконтроллера. Для силовой части двигателя, отвечающей за непосредственное питание и коммутацию обмоток, используются отдельные платы (инвертор). Принципиальная схема простейшего инвертора одной фазы приведена на рис. 1.

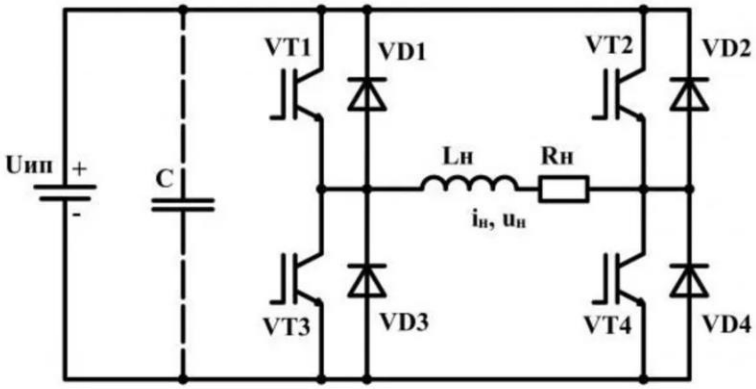


Рисунок 1 – Схема простейшего инвертора (1 фаза)

Инвертор преобразует постоянный ток в переменный за счет модуляции ширины импульсов управления ключами VD.

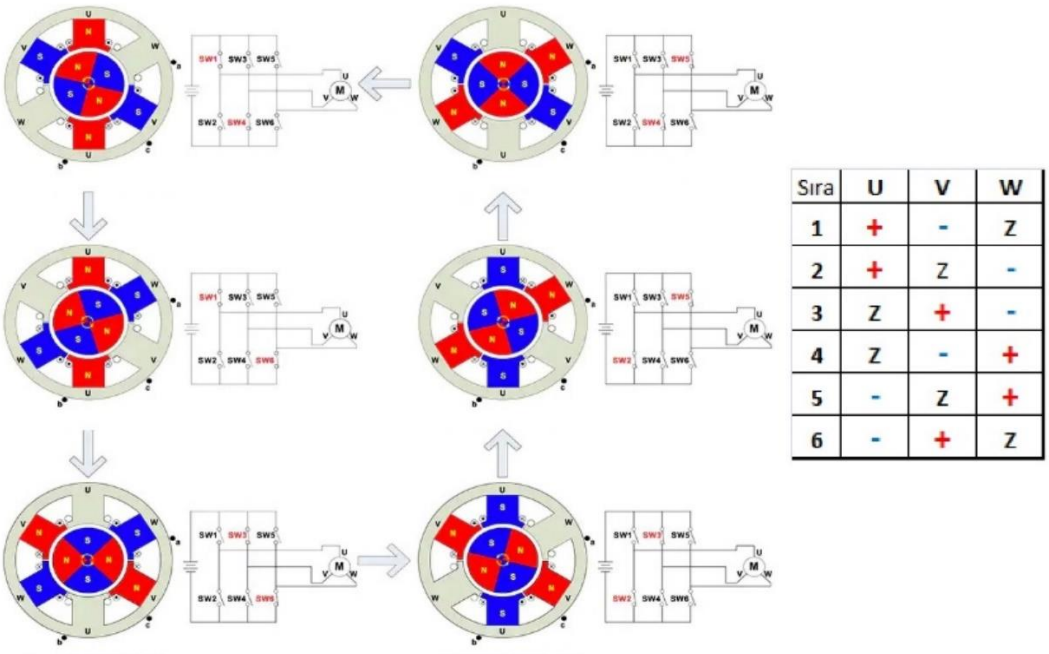
В основе работы инвертора лежит схема на силовых транзисторах (обычно IGBT или MOSFET), управляемых контроллером, который определяет последовательность и длительность коммутации фаз. В рамках исследования рассматривается только принцип формирования сигналов ШИМ, необходимых для управления транзисторами силовой схемы. Проектирование этой силовой

части не является целью данной работы.

1.1 Управление бесколлекторными двигателями постоянного тока

Управление бесколлекторным двигателем основывается на последовательном включении обмоток статора, чтобы создать вращающееся магнитное поле, которое взаимодействует с постоянными магнитами ротора. Этот процесс обеспечивает вращение ротора без механических щеток.

На рис. 2 изображен принцип управления трехфазным бесколекторным двигателем. В каждый момент времени активны две обмотки: через одну подается ток, а через другую он стекает. Третья обмотка остается отключенной. Для трехфазного двигателя существует шесть активных состояний, которые формируют одну электрическую фазу вращения. Эти состояния переключаются последовательно в зависимости от позиции ротора.



Sıra	U	V	W
1	+	-	Z
2	+	Z	-
3	Z	+	-
4	Z	-	+
5	-	Z	+
6	-	+	Z

Рисунок 2 – Состояния бесколлекторного двигателя

Частота переключения этих состояний определяет скорость вращения двигателя. Чем выше частота, тем быстрее вращается ротор.

ШИМ используется для управления мощностью, передаваемой на фазы двигателя, обеспечивая плавность работы и минимизацию потерь. Изменяя скважность сигнала ШИМ, можно контролировать среднее напряжение, подаваемое на обмотки (рис. 3).

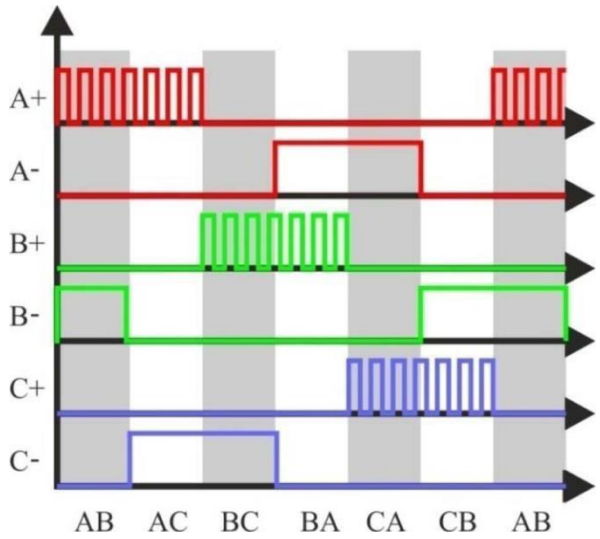


Рисунок 3 – График ШИМ для управления мощностью бесколлекторного двигателя

1.2 Управление синхронными и асинхронными двигателями

Для синхронных/асинхронных двигателей формируется синусоидальное напряжение с помощью инвертора, и частота этого напряжения определяет скорость вращения ротора. Для этого необходимо менять скважность ШИМ по синусоидальному закону. На рис. 4 изображен принцип формирования сигнала ШИМ двух ключей одного плеча (u1, u2), для получения синусоидального тока.

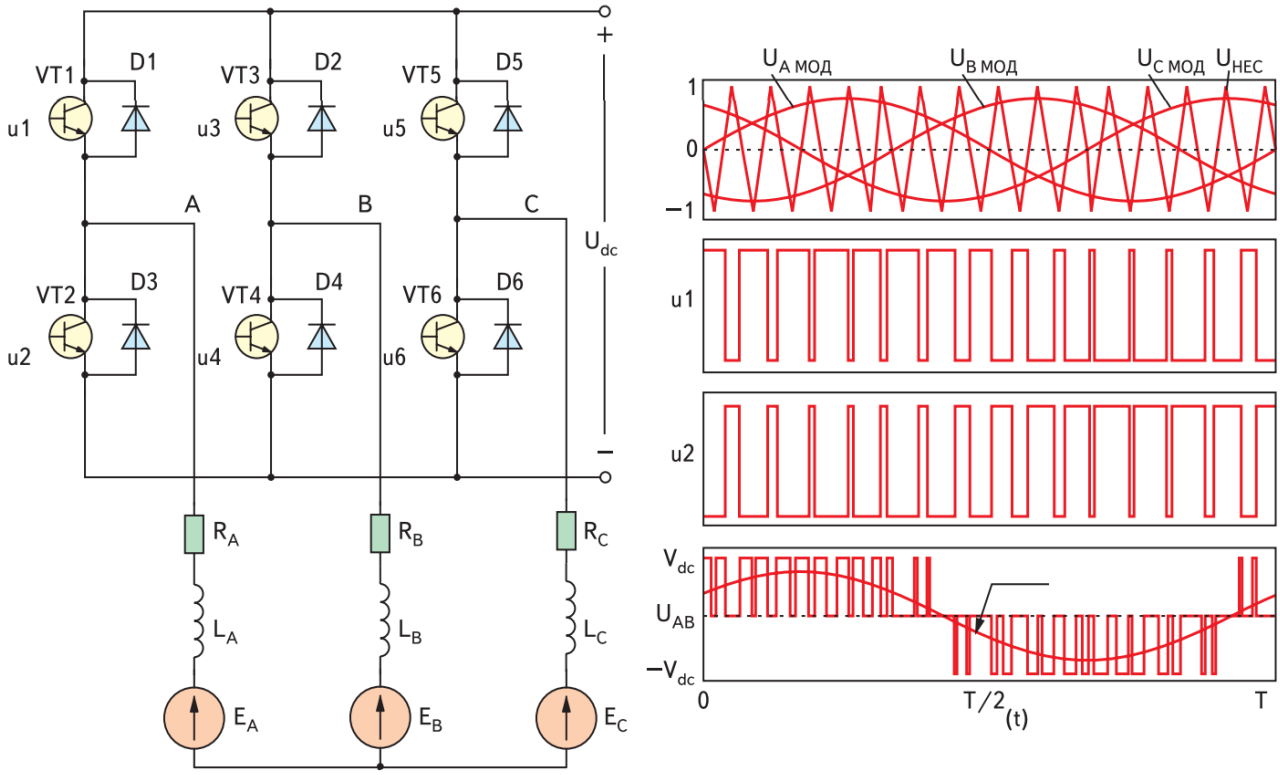


Рисунок 4 – График ШИМ для формирования синусоидального напряжения

2 Разработка программы формирования ШИМ

Выбор платформы для разработки программного обеспечения представляет собой важнейший этап в процессе создания эффективных систем управления двигателями. Одной из наиболее подходящих платформ для разработки программного обеспечения, включая алгоритмы формирования широтно-импульсной модуляции (ШИМ), является STM32. Возможность гибкой настройки таймеров для генерации ШИМ-сигналов позволяет реализовывать различные методы управления двигателями с высокой точностью.

Кроме того, STM32 поддерживает разнообразные интерфейсы для связи с другими устройствами, включая UART, SPI и CAN, что важно для обмена данными в распределенных системах. Поддержка множества встроенных функций, таких как цифровая обработка сигналов (DSP) и математические операции на аппаратном уровне, значительно ускоряет вычисления,

требующиеся для алгоритмов управления.

В результате использования платформы STM32 в сочетании с оптимизированным программным обеспечением создается надежная и мощная система управления, способная удовлетворить требования различных промышленных приложений.

2.1 Методы формирования ШИМ

Для формирования ШИМ были использованы таймеры. Они являются основным инструментом для формирования сигналов ШИМ в микроконтроллерах STM32, позволяя генерировать импульсы с заданной частотой и коэффициентом заполнения.

К таким параметрам относятся частота работы таймера, коэффициент заполнения и длительность импульсов. Эти параметры задаются через конфигурацию регистров таймера, таких как предделитель (Prescaler), регистр автоматической перезагрузки (ARR) и регистр сравнения (CCR). Настройка этих параметров позволяет добиться необходимой точности и стабильности ШИМ сигнала, что особенно важно в системах управления двигателями.

При настройке таймеров на STM32 для генерации ШИМ важно учитывать практические аспекты, такие как синхронизация таймеров для управления несколькими каналами и использование функций dead-time, предотвращающих короткое замыкание в полумостовых схемах. Т.к. встроенной функции dead-time в STM32 нет, то она реализуется программно. Эти функции имеют особое значение при работе с мощными двигателями, где требуется высокая надежность.

В программе используется 3 таймера для формирования ШИМ: TIM4, TIM3 и TIM2. Таймер TIM4 является ведущим и по нему в режиме синхронизации запускаются таймеры TIM3, TIM2. Настройка частоты и скважности таймера зависит от значений, записанных в регистрах Modbus, которые описаны в главе 2.2.

Для каждого плеча используется один таймер с двумя каналами ШИМ.

Скважность ШИМ обновляется каждый период, по прерыванию таймера TIM4. В прерывании, в зависимости от выбранного режима вызывается соответствующая функция для управления скважностью.

Поддерживаются следующие режимы:

- Постоянный одноканальный ШИМ (на любом из 6 каналов)
- Постоянный мостовой однофазный ШИМ
- Постоянный мостовой трехфазный ШИМ
- Синусоидальный одноканальный ШИМ (на любом из 6 каналов)
- Синусоидальный мостовой однофазный ШИМ
- Синусоидальный мостовой трехфазный ШИМ

Для плавного изменения параметров в программе используется ПИ-регулятор, из библиотеки CMSIS.

Т.к. данная программа реализует разомкнутое управление без обратной связи, то выход ПИ-регулятора заведен на вход. Также реализована сатурация, для ограничения скорости изменения скорости – функция.

В результате был написан модуль ШИМ (pwm.c/pwm.h), который реализует всю его работу. Он содержит функции для двух основных режимов ШИМ:

1. Постоянный ШИМ (DC) – функция **PWM_DC_Bridge_Mode()**.
2. Синусоидальный ШИМ (SINE) – функции и **PWM_Sine_Bridge_Mode()**.

Эти режимы обеспечивают формирование различных ШИМ для управления двигателями.

Так же есть режим одноканального ШИМ, который можно использовать для двигателей постоянного тока или для тестирования переключения отдельных транзисторов – функция **PWM_SingleChannel_Mode()**.

Полный листинг данного модуля приведен в приложениях А, Б.

2.1 Дистанционное управление программой

Для дистанционного управления программой было решено использовать протокол Modbus

Протокол Modbus – самый распространенный промышленный протокол для взаимодействия между полевым уровнем (измерительных устройств – датчиков) и контроллерным уровнем. Является стандартом и поддерживается почти всеми производителями промышленного оборудования.

Благодаря универсальности и открытости, стандарт позволяет интегрировать оборудование разных производителей. Modbus используется для сбора показания с датчиков, управления реле и контроллерами, мониторинга, и т.д.

В протоколе Modbus данные организованы в виде 16-битных регистров и однобитных коилов, что делает его удобным для передачи и обработки информации в системах управления. Регистры используются для хранения числовых значений, таких как параметры настройки или данные измерений, а коилы представляют собой логические состояния, которые могут быть использованы для управления или индикации. Такое разделение позволяет эффективно структурировать данные, обеспечивая простоту интеграции в системы автоматизации.

Передача данных по Modbus на STM32 реализуется через интерфейс UART.

Также при использовании Modbus важно правильно распределить регистры и коилы. Регистры могут хранить параметры ШИМ, такие как частота и коэффициент заполнения, в то время как коилы можно использовать для переключения режимов работы.

В итоге была реализована следующая структура данных Modbus:

Параметры ШИМ записываются в регистры и, если настройки обновились, таймеры STM32 перенастраиваются в соответствии с новыми настройками:

Регулируемые параметры:

- Частота синуса (Регистр №20000),
- Скважность ШИМ (Регистр №20001),
- Частота ШИМ (Регистр №20002),
- Максимальная длительность одного импульса (Регистр №20003),
- Минимальная длительность одного импульса (Регистр №20004),
- Deadtime для переключения ключей между каналами (Регистр №20005).

Режим работы ШИМ записываются в коилы и, в соответствии с которыми вызываются необходимые функции для формирования ШИМ:

- Постоянный или синусоидальный ШИМ (Коил №17),
- Мостовой ШИМ (Коил №18),
- Трехфазный ШИМ (Коил №19),
- Полярность ШИМ (Коил №20),
- Используемый канал ШИМ в одноканального режиме (Коил №21-23).

2.2 Локальное управление программой

Также было решено добавить локальное управление, через энкодер с кнопкой и дисплей.

Т.к. STM32 имеет аппаратную поддержку I2C и SPI, то удобно будет использовать дисплеи именно с такими интерфейсами

Интерфейс I2C представляет собой двухпроводной протокол передачи данных, использующий линии SDA (Serial Data) и SCL (Serial Clock). Он позволяет нескольким устройствам работать на одной шине, передавая данные в последовательном формате с подтверждением на каждом байте.

Дисплеи, работающие по I2C, имеют встроенный контроллер, который принимает команды и преобразует их в параллельный интерфейс дисплея. Благодаря этому, для управления дисплеем требуется минимум соединений, что особенно полезно при использовании микроконтроллеров с ограниченным

числом выводов.

Такие дисплеи, как правило, символьные и имеют размер 2 строки 16 символов (LCD1602) или 4 строки 20 символов (LCD2004). В данной работе использовался дисплей LCD1602.

Интерфейс SPI представляет собой стандарт для высокоскоростного обмена данными между микроконтроллерами и периферийными устройствами. Принцип работы SPI основывается на синхронной передаче данных, где используются четыре сигнальных линии: MOSI (Master Out Slave In), MISO (Master In Slave Out), SCK (Serial Clock) и SS (Slave Select).

Дисплеи с *интерфейсом SPI*, такие как LCD12864, обеспечивают быструю передачу данных благодаря своей архитектуре, что позволяет обновлять изображение на экране практически мгновенно. Также они, как правило, графические и работают на уровне отдельных пикселей, а не фиксированных символов, что делает их более гибкими в плане отображаемой информации. Также они могут быть больше по размерам (напр. LCD12864 размером 128x64 пикселей), что позволяет выводить гораздо больше информации. Тем не менее, дисплеи SPI имеют и свои недостатки: они требуют большего числа проводов по сравнению с интерфейсами, такими как I2C, что может усложнять разводку схемы.

Энкодеры представляют собой устройства, предназначенные для преобразования механического движения в электрические сигналы, которые могут быть интерпретированы системой управления. Основной принцип их работы заключается в регистрации изменений положения объекта, будь то угловое или линейное перемещение. Это достигается благодаря использованию датчиков, которые фиксируют изменения положения относительно начальной точки. В зависимости от конструкции и типа энкодера, такие сигналы могут быть либо импульсными, либо представлять собой закодированную информацию о текущем положении.

При разработке программы для задания параметров было решено

использовать угловой энкодер с кнопкой. Микроконтроллер STM32 предоставляет специализированные режимы таймеров для работы с энкодерами, что позволяет эффективно считывать угол поворота энкодера.

В результате был написан модуль для интерфейса программы (interface.c/interface.h), который содержит функции для взаимодействия пользователя с программой. Он состоит из энкодера с кнопкой и дисплея. Основные функции:

1. Обработка ввода от энкодера, включая переключение режимов и настройку параметров (грубая и тонкая настройки) – функция **Update_Encoder()**,

2. Обновление информации на дисплее, отображение текущих параметров системы – функция **Update_Encoder()**.

Энкодер с кнопкой позволяет пользователю выбирать и настраивать параметры системы:

- Однократное нажатие кнопки – выбор параметра для регулирования,
- Прокрутка – при выбранном параметре его регулирование, иначе прокрутка меню параметров,
- Двукратное нажатие кнопки – переключение между выбором режима ШИМ и регулированием его параметров.

Дисплей обеспечивает обратную связь, отображая текущие настройки и состояния. Есть поддержка LCD дисплея по интерфейсу I2C (LCD1602) и SPI (LCD12864).

Полный листинг данного модуля приведен в приложениях В, Г.

2.3 Управление параметрами ШИМ и логирование данных

Т.к. таймер ШИМ имеет очень высокую и непостоянную частоту, то было решено добавить еще один управляющий таймер, который будет перенастраивать ШИМ и а также заниматься логированием данных.

В результате был написан модуль для контроля программы (control.c/control.h), который настраивает управляющий таймер на определенную

частоту.

В прерывании этого таймера проверяются настройки, заданные в Modbus и если они обновляются – перенастраиваются таймеры).

Модуль также содержит функции для сбора и записи данных о работе ШИМ:

1. Запись текущих параметров в массив логов – функция **Fill_Logs_with_Data()**,
2. Обновление параметров логирования (частота логирования, объем логированных данных) – функция **Update_Params_For_Log()**.

Логи включают в себя следующие данные:

- Текущую скважность ШИМ
- Визуализация скважности ШИМ (в трех периодах)
- Счетчик (пила) для проверки корректно ли работает логирование

Полный листинг данного модуля приведен в приложениях Д, Е.

3 Демонстрация работы программы

Видео с демонстрацией управления ШИМ и считыванием логированных данным с МК через Modbus, (приложение на ПК) приведено здесь: https://disk.yandex.ru/i/7S2R2Ckg_3fXMw.

Видео с демонстрацией переключения режимов ШИМ с помощью энкодера и отображения текущего режима на дисплее LCD12864 приведено здесь: https://disk.yandex.ru/i/pxXePqLXaXEKkA.

Видео с регулированием параметров ШИМ с помощью энкодера и отображения текущих параметров на дисплее LCD1602 приведено здесь: https://disk.yandex.ru/i/xKvLAbo4s5YlPw.

График работы ПИ-регулятора приведен на рис. 5.

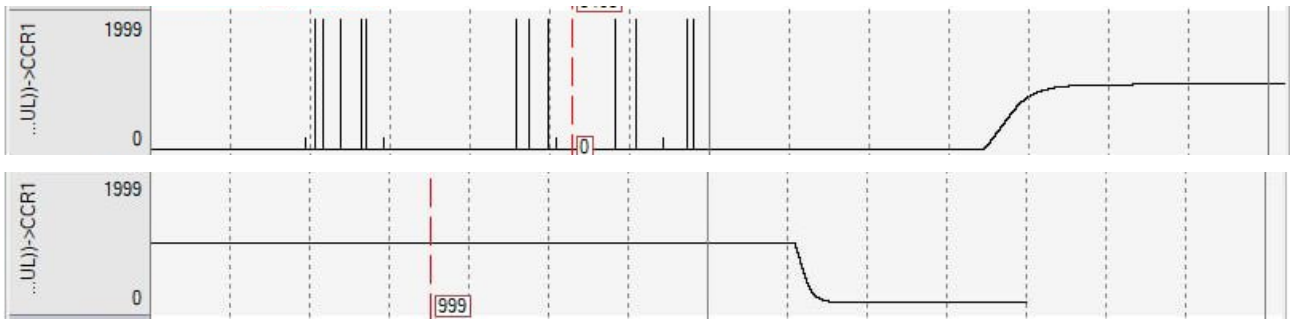


Рисунок 5 – Демонстрация регулирования скважности ШИМ (черная линия): набор и сброс частоты

4 Тестирование на бесколлекторном двигателе

Данная программа также была протестирована на реальном бесколлекторном двигателе. Видео демонстрирующее его управление приведено здесь (разгон до 70 об/с при частоте ШИМ 2 кГц): https://disk.yandex.ru/i/eEsR5Wqq7ed-LA.

Максимальная частота вращения до которой удалось разогнать двигатель составила 400 об/с (при частоте ШИМ 40 кГц).

ЗАКЛЮЧЕНИЕ

В ходе данной курсовой работы была разработана программа формирования ШИМ для управления различными типами двигателей с использованием микроконтроллера STM32. Разработанное ПО успешно выполняет поставленные задачи, обеспечивая управление ШИМ-сигналами и логирование данных в реальном времени.

Особое внимание уделено программной реализации формирования ШИМ с использованием таймеров STM32, что позволило продемонстрировать их практическую применимость. Кроме того, в работе были рассмотрены дополнительные модули, такие как интерфейс Modbus, дисплеи для отображения информации и энкодер для выбора режимов работы. Эти модули обеспечивают взаимодействие пользователя с системой и повышают её функциональность

Практическая значимость данной работы заключается в создании универсального подхода к управлению двигателями с использованием ШИМ. Реализованная программа может быть адаптирована для различных типов двигателей, что делает её полезной для применения в системах автоматизации, робототехнике и других областях. Кроме того, использование интерфейса Modbus и дисплеев для вывода информации позволяет интегрировать разработанную систему в более сложные решения, обеспечивая удобство управления и мониторинга

Программа обладает модульной структурой, что облегчает добавление новых функции. Например, добавление датчиков скорости для формирование обратной связи. Или добавление векторного управление двигателями.

ПРИЛОЖЕНИЕ А

Листинг pwm.c

```
/**
    ******************************************************************************
    * @file     pwm.c
    * @brief    Модуль для реализации PWM.
    ******************************************************************************/
#include "pwm.h"
#include "rng.h"

PWM_HandleTypeDef hpwm1;
PWM_SlaveHandleTypeDef hpwm2;
PWM_SlaveHandleTypeDef hpwm3;
uint32_t sin_table[SIN_TABLE_SIZE_MAX];
unsigned                ActiveChannelSHDW_Master;
float                   DeadTimeCnt_Master;
unsigned                ActiveChannelSHDW_Slave2;
float                   DeadTimeCnt_Slave2;
unsigned                ActiveChannelSHDW_Slave3;
float                   DeadTimeCnt_Slave3;
float shiftPHASE1= 0.6666666666;
float shiftPHASE2= 0.3333333333;

/**
    * @brief        First set up of PWM Two Channel.
    * @details  Первый инит ШИМ. Заполняет структуры и инициализирует таймер
для генерации синуоидального ШИМ.
    *                   Скважность ШИМ меняется по закону синусоиды, каждый
канал генерирует свой полупериод синуса (от -1 до 0 И от 0 до 1)
    *                   ШИМ генерируется на одном канале.
    * @note     This called from main
    */
void PWM_Sine_FirstInit(void)
{
    hpwm1.sPWM_Config.PWM_Mode = (PWM_ModeCoilsTypeDef *)&coils_regs[0];
    hpwm1.sPWM_Config.PWM_Settings = (PWM_ModeRegsTypeDef *)&pwm_ctrl[0];

    // PWM RAMP INIT
    hpwm1.pDuty_Table_Origin = SIN_TABLE_ORIGIN;

    // SINE RAMP PARAMS
    hpwm1.hramp[PWM_RampSineHz_Ind].pid.Ki = PWM_RAMP_SPEED;
    hpwm1.hramp[PWM_RampSineHz_Ind].limMax = 656;
    hpwm1.hramp[PWM_RampSineHz_Ind].limMin = 0;
    hpwm1.hramp[PWM_RampSineHz_Ind].limMaxInt = 20;
    hpwm1.hramp[PWM_RampSineHz_Ind].limMinInt = -20;
    hpwm1.hramp[PWM_RampSineHz_Ind].SampleT = (1.0f/HZ_TIMER_CTRL);
    PWM_Ramp_UpdateParams(&hpwm1.hramp[PWM_RampSineHz_Ind], 1);
//
    // DUTY RAMP PARAMS
    hpwm1.hramp[PWM_RampPWMDuty_Ind].pid.Ki = PWM_RAMP_SPEED;
    hpwm1.hramp[PWM_RampPWMDuty_Ind].limMax = 100;
    hpwm1.hramp[PWM_RampPWMDuty_Ind].limMin = 0;
    hpwm1.hramp[PWM_RampPWMDuty_Ind].limMaxInt = 20;
    hpwm1.hramp[PWM_RampPWMDuty_Ind].limMinInt = -20;
    hpwm1.hramp[PWM_RampPWMDuty_Ind].SampleT = (1.0f/HZ_TIMER_CTRL);
    PWM_Ramp_UpdateParams(&hpwm1.hramp[PWM_RampPWMDuty_Ind], 1);

    // PWM CONTROL PINS INIT
```

```
    GPIO_InitTypeDef GPIO_InitStruct = {0};
    GPIO_Clock_Enable(GPIOC);

    GPIO_InitStruct.Pin = GPIO_PIN_0;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_PULLUP;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
    GPIOC->ODR|=1<<0;//подача 0 разрешает . Аппаратное разрешение и запрет не
подачу шим . Сигнал идет на  логический элемент & 74AC08d

    GPIO_InitStruct.Pin = GPIO_PIN_1;
  GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
  GPIO_InitStruct.Pull = GPIO_PULLUP;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);  //кнопка для вкл аппаратно шим
74AC08d


    //---------PWM TIMER1 INIT------------
    // channels settings
    hpwm1.sConfigPWM.OCMode = TIM_OCMODE_PWM1;
  hpwm1.sConfigPWM.Pulse = 0;
  hpwm1.sConfigPWM.OCPolarity = TIM_OCPOLARITY_LOW;
  hpwm1.sConfigPWM.OCFastMode = TIM_OCFAST_DISABLE;


    // tim1 settings
    hpwm1.stim.htim.Instance =  TIMER_PWM1_INSTANCE;
//  hpwm1.stim.htim.Init.CounterMode =  TIM_COUNTERMODE_DOWN;
    hpwm1.stim.htim.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
    hpwm1.stim.sTimMode =           TIM_IT_MODE;
    hpwm1.stim.sTimFreqHz =         HZ_TIMER_PWM;
    hpwm1.stim.sTickBaseUS =        PROJSET.TIM_PWM_TICKBASE;
    hpwm1.stim.sTimAHBFreqMHz = PROJSET.TIM_PWM_AHB_FREQ;
    hpwm1.GPIOx =                           TIMER_PWM1_GPIOx;
    hpwm1.GPIO_PIN_X1 =                 PROJSET.TIM_PWM1_GPIO_PIN_X1;
    hpwm1.GPIO_PIN_X2 =                 PROJSET.TIM_PWM1_GPIO_PIN_X2;
    hpwm1.sPWM_Channel1 =           PROJSET.TIM_PWM1_TIM_CHANNEL1;
    hpwm1.sPWM_Channel2 =           PROJSET.TIM_PWM1_TIM_CHANNEL2;
    hpwm1.hpwm2 = (void *)&hpwm2;
    hpwm1.hpwm3 = (void *)&hpwm3;

    TIM_Base_Init(&hpwm1.stim);
    TIM_Output_PWM_Init(&hpwm1.stim.htim, &hpwm1.sConfigPWM,
hpwm1.sPWM_Channel1, hpwm1.GPIOx, hpwm1.GPIO_PIN_X1);
    TIM_Output_PWM_Init(&hpwm1.stim.htim, &hpwm1.sConfigPWM,
hpwm1.sPWM_Channel2, hpwm1.GPIOx, hpwm1.GPIO_PIN_X2);


    // PWM SLAVES INIT
    hpwm2.hMasterPWM = &hpwm1;
    hpwm2.stim = hpwm1.stim;
    hpwm2.stim.htim.Instance =  (TIM_TypeDef *)PROJSET.TIM_PWM2_INSTANCE;
    hpwm2.GPIOx =                           (GPIO_TypeDef
*)PROJSET.TIM_PWM2_GPIOx;
    hpwm2.GPIO_PIN_X1 =                 PROJSET.TIM_PWM2_GPIO_PIN_X1;
    hpwm2.GPIO_PIN_X2 =                 PROJSET.TIM_PWM2_GPIO_PIN_X2;
    hpwm2.sPWM_Channel1 =           PROJSET.TIM_PWM2_TIM_CHANNEL1;
    hpwm2.sPWM_Channel2 =           PROJSET.TIM_PWM2_TIM_CHANNEL2;
```

```
    //hpwm2.Duty_Shift_Ratio = shiftPHASE1;
    hpwm2.Duty_Shift_Ratio = shiftPHASE1;

    hpwm3.hMasterPWM = &hpwm1;
    hpwm3.stim = hpwm1.stim;
    hpwm3.stim.htim.Instance =  (TIM_TypeDef *)PROJSET.TIM_PWM3_INSTANCE;
    hpwm3.GPIOx =                              (GPIO_TypeDef
*)PROJSET.TIM_PWM3_GPIOx;
    hpwm3.GPIO_PIN_X1 =                 PROJSET.TIM_PWM3_GPIO_PIN_X1;
    hpwm3.GPIO_PIN_X2 =                 PROJSET.TIM_PWM3_GPIO_PIN_X2;
    hpwm3.sPWM_Channel1 =           PROJSET.TIM_PWM3_TIM_CHANNEL1;
    hpwm3.sPWM_Channel2 =           PROJSET.TIM_PWM3_TIM_CHANNEL2;

    //hpwm3.Duty_Shift_Ratio = shiftPHASE2;
    hpwm3.Duty_Shift_Ratio = shiftPHASE2;

    PWM_SynchInit();
    PWM_SlavePhase_Init(&hpwm2);
    PWM_SlavePhase_Init(&hpwm3);

    //----------TIMERS START-------------
    HAL_TIM_OC_Start(&hpwm1.stim.htim, TIM_CHANNEL_3);
    HAL_TIM_PWM_Start(&hpwm1.stim.htim, hpwm1.sPWM_Channel1); // PWM channel 1
    HAL_TIM_PWM_Start(&hpwm1.stim.htim, hpwm1.sPWM_Channel2); // PWM channel 2
    HAL_TIM_Base_Start_IT(&hpwm1.stim.htim); // timer for PWM
}


void PWM_SynchInit(void)
{
    TIM_MasterConfigTypeDef sMasterConfig = {0};
    TIM_OC_InitTypeDef          sConfigOC = {0};

    //--------MASTER SYNCH INIT----------
    sMasterConfig.MasterOutputTrigger = TIM_TRGO_OC3REF;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
    sConfigOC.OCMode = TIM_OCMODE_ACTIVE;
  sConfigOC.Pulse = 0;
  sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;

  HAL_TIMEx_MasterConfigSynchronization(&hpwm1.stim.htim, &sMasterConfig);
  HAL_TIM_OC_ConfigChannel(&hpwm1.stim.htim, &sConfigOC, TIM_CHANNEL_3);

}

/**
    * @brief        PWM Handler.
    * @param        hpwm - указатель на хендл ШИМ.
    * @details  Управляет скважность ШИМ в режиме PWM_TABLE.
    * @note         This called from TIM_PWM_Handler
    */
void PWM_Handler(PWM_HandleTypeDef *hpwm)
{
    uint16_t rotate_ind_A;
    uint16_t rotate_ind_B;
    uint16_t rotate_ind_C;
    // rotate pwm
    rotate_ind_A = PWM_Get_Duty_Table_Ind(hpwm, hpwm->stim.sTimFreqHz);
    if (PWM_Get_Mode(hpwm, PWM_PHASE_MODE))
    {
```

```
        rotate_ind_B =
PWM_SlavePhase_Calc_TableInd(PWM_Set_pSlaveHandle(hpwm,hpwm2), rotate_ind_A);
        rotate_ind_C =
PWM_SlavePhase_Calc_TableInd(PWM_Set_pSlaveHandle(hpwm,hpwm3), rotate_ind_A);
    }
    if((hpwm->PWM_Sine_Hz == 0 &&
(PWM_Get_Mode(&hpwm1,PWM_BRIDGE_MODE|PWM_DC_MODE) != PWM_DC_MODE)) || // if
SINE HZ = 0  in SINE mode or BRIDGE mode (DC = 0, BRIDGE = 1)
        (hpwm->PWM_Duty == 0 && PWM_Get_Mode(&hpwm1,PWM_DC_MODE)) )     // if
DUTY = 0 in DC MODE
    {       //reset all channels
        PWM_Set_Compare1(hpwm, 0);  // reset first channel
        PWM_Set_Compare2(hpwm, 0);  // reset second channel
        PWM_Set_Compare1(PWM_Set_pSlaveHandle(hpwm,hpwm2), 0);  // reset first
channel
        PWM_Set_Compare2(PWM_Set_pSlaveHandle(hpwm,hpwm2), 0);  // reset
second channel
        PWM_Set_Compare1(PWM_Set_pSlaveHandle(hpwm,hpwm3), 0);  // reset first
channel
        PWM_Set_Compare2(PWM_Set_pSlaveHandle(hpwm,hpwm3), 0);  // reset
second channel
        return;
    }
    //--------BRIDGE MODE DISABLE--------
    if(PWM_Get_Mode(&hpwm1,PWM_BRIDGE_MODE) == 0)
    {
        //-----------PWM DC MODE-------------
        if(PWM_Get_Mode(&hpwm1,PWM_DC_MODE))
            PWM_SingleChannel_Mode(hpwm, rotate_ind_A, 0);
        //------------SINUS MODE-------------
        else
            PWM_SingleChannel_Mode(hpwm, rotate_ind_A, 1);
    }
    //---------BRIDGE MODE ENABLE--------
    else
    {
        if(PWM_Get_Mode(&hpwm1,PWM_DC_MODE))
        //-----------PWM DC MODE-------------
            PWM_DC_Bridge_Mode(hpwm, rotate_ind_A, rotate_ind_B,
rotate_ind_C);
        //------------SINUS MODE-------------
        else
            PWM_Sine_Bridge_Mode(hpwm, rotate_ind_A, rotate_ind_B,
rotate_ind_C);
    }

    //-----CHECK CHANNELS FOR ERRORS-----
    uint16_t min_duty = PWM_Calc_Min_Duty(hpwm);
    uint16_t max_duty = PWM_Calc_Max_Duty(hpwm);
    // IF FIRST CHANNEL IS ACRIVE
    if(PWM_Get_Compare1(hpwm) != 0)
    {
        // Duty shoud be bigger or equeal than min duration
        if (PWM_Get_Compare1(hpwm)<min_duty)
            PWM_Set_Compare1(hpwm, min_duty);
        // Duty shoud be less or equeal than ARR-min duration
        if (PWM_Get_Compare1(hpwm)>max_duty)
            PWM_Set_Compare1(hpwm, max_duty);
    }
    // IF SECOND CHANNEL IS ACRIVE
    else if(PWM_Get_Compare2(hpwm) != 0)
```

```
    {
    // Duty shoud be bigger or equeal than min duration
    if (PWM_Get_Compare2(hpwm)<min_duty)
        PWM_Set_Compare2(hpwm, min_duty);
    // Duty shoud be less or equeal than ARR
    if (PWM_Get_Compare2(hpwm)>max_duty)
        PWM_Set_Compare2(hpwm, max_duty);
    }
    // IF BOTH CHANNEL IS ACRIVE
    if((PWM_Get_Compare1(hpwm) != 0) && (PWM_Get_Compare2(hpwm) != 0))
    {
        // Only one channel shoud be active so disable all
        PWM_Set_Compare1(hpwm, 0);
        PWM_Set_Compare2(hpwm, 0);
    }
    PWM_SlavePhase_Check_Channels(PWM_Set_pSlaveHandle(hpwm,hpwm2));
    PWM_SlavePhase_Check_Channels(PWM_Set_pSlaveHandle(hpwm,hpwm3));

    if(hpwm->PWM_DeadTime)
    {
        PWM_CreateDeadTime(hpwm, &DeadTimeCnt_Master,
&ActiveChannelSHDW_Master);
        PWM_SlavePhase_CreateDeadTime(PWM_Set_pSlaveHandle(hpwm,hpwm2),
&DeadTimeCnt_Slave2, &ActiveChannelSHDW_Slave2);
        PWM_SlavePhase_CreateDeadTime(PWM_Set_pSlaveHandle(hpwm,hpwm3),
&DeadTimeCnt_Slave3, &ActiveChannelSHDW_Slave3);
    }


}

/**
    * @brief    Form PWM on Single Channel (DC or Sine PWM). Slave PWM
disabled.
    * @param    hspwm               - указатель на хендл слейв ШИМ.
    * @param    rotate_ind_A    - индекс таблицы для Мастер ШИМ.
    * @param    SineOrDC            - Флаг: скважность постоянная или
меняющаяся по закону синусна.
    */
void PWM_SingleChannel_Mode(PWM_HandleTypeDef *hpwm, uint16_t rotate_ind_A,
uint8_t SineOrDC)
{
    PWM_SlaveHandleTypeDef *hspwm2 = PWM_Set_pSlaveHandle(hpwm,hpwm2);
    PWM_SlaveHandleTypeDef *hspwm3 = PWM_Set_pSlaveHandle(hpwm,hpwm3);

//  PWM_Set_Compare1(hpwm, 0);  // reset first channel
//  PWM_Set_Compare2(hpwm, 0);  // reset second channel
//  PWM_Set_Compare1(PWM_Set_pSlaveHandle(hpwm,hpwm2), 0);  // reset first
channel
//  PWM_Set_Compare2(PWM_Set_pSlaveHandle(hpwm,hpwm2), 0);  // reset second
channel
//  PWM_Set_Compare1(PWM_Set_pSlaveHandle(hpwm,hpwm3), 0);  // reset first
channel
//  PWM_Set_Compare2(PWM_Set_pSlaveHandle(hpwm,hpwm3), 0);  // reset second
channel


    switch(PWM_Get_Mode(hpwm, PWM_ACTIVECHANNEL_MODE))
    {
        case PWM_ACTIVECHANNEL_A0:
            if(SineOrDC)
```

```
                PWM_Set_Duty_From_Table(hpwm, hpwm->sPWM_Channel1,
rotate_ind_A);
            else
                PWM_Set_Duty_From_Value(hpwm, hpwm->sPWM_Channel1);     // set
first channel
            PWM_Set_Compare2(hpwm, 0);  // reset second channel
            PWM_Set_Compare1(PWM_Set_pSlaveHandle(hpwm,hpwm2), 0);  // reset
first channel
            PWM_Set_Compare2(PWM_Set_pSlaveHandle(hpwm,hpwm2), 0);  // reset
second channel
            PWM_Set_Compare1(PWM_Set_pSlaveHandle(hpwm,hpwm3), 0);  // reset
first channel
            PWM_Set_Compare2(PWM_Set_pSlaveHandle(hpwm,hpwm3), 0);  // reset
second channel
            break;

        case PWM_ACTIVECHANNEL_A1:
            if(SineOrDC)
                PWM_Set_Duty_From_Table(hpwm, hpwm->sPWM_Channel2,
rotate_ind_A);
            else
                PWM_Set_Duty_From_Value(hpwm, hpwm->sPWM_Channel2);     // set
second channel
            PWM_Set_Compare1(hpwm, 0);  // reset first channel
            PWM_Set_Compare1(PWM_Set_pSlaveHandle(hpwm,hpwm2), 0);  // reset
first channel
            PWM_Set_Compare2(PWM_Set_pSlaveHandle(hpwm,hpwm2), 0);  // reset
second channel
            PWM_Set_Compare1(PWM_Set_pSlaveHandle(hpwm,hpwm3), 0);  // reset
first channel
            PWM_Set_Compare2(PWM_Set_pSlaveHandle(hpwm,hpwm3), 0);  // reset
second channel
            break;

        case PWM_ACTIVECHANNEL_B0:
            if(SineOrDC)
                PWM_Set_SlaveDuty_From_Table(hspwm2, hspwm2->sPWM_Channel1,
rotate_ind_A);
            else
                PWM_Set_SlaveDuty_From_Value(hspwm2, hspwm2->sPWM_Channel1);
// set first channel
            PWM_Set_Compare1(hpwm, 0);  // reset first channel
            PWM_Set_Compare2(hpwm, 0);  // reset second channel
            PWM_Set_Compare2(PWM_Set_pSlaveHandle(hpwm,hpwm2), 0);  // reset
second channel
            PWM_Set_Compare1(PWM_Set_pSlaveHandle(hpwm,hpwm3), 0);  // reset
first channel
            PWM_Set_Compare2(PWM_Set_pSlaveHandle(hpwm,hpwm3), 0);  // reset
second channel
            break;

        case PWM_ACTIVECHANNEL_B1:
            if(SineOrDC)
                PWM_Set_SlaveDuty_From_Table(hspwm2, hspwm2->sPWM_Channel2,
rotate_ind_A);
            else
                PWM_Set_SlaveDuty_From_Value(hspwm2, hspwm2->sPWM_Channel2);
// set second channel
            PWM_Set_Compare1(hpwm, 0);  // reset first channel
            PWM_Set_Compare2(hpwm, 0);  // reset second channel
```

```
            PWM_Set_Compare1(PWM_Set_pSlaveHandle(hpwm,hpwm2), 0);  // reset
first channel
            PWM_Set_Compare1(PWM_Set_pSlaveHandle(hpwm,hpwm3), 0);  // reset
first channel
            PWM_Set_Compare2(PWM_Set_pSlaveHandle(hpwm,hpwm3), 0);  // reset
second channel
            break;

        case PWM_ACTIVECHANNEL_C0:
            if(SineOrDC)
                PWM_Set_SlaveDuty_From_Table(hspwm3, hspwm3->sPWM_Channel1,
rotate_ind_A);
            else
                PWM_Set_SlaveDuty_From_Value(hspwm3, hspwm3->sPWM_Channel1);
// set first channel
            PWM_Set_Compare1(hpwm, 0);  // reset first channel
            PWM_Set_Compare2(hpwm, 0);  // reset second channel
            PWM_Set_Compare1(PWM_Set_pSlaveHandle(hpwm,hpwm2), 0);  // reset
first channel
            PWM_Set_Compare2(PWM_Set_pSlaveHandle(hpwm,hpwm2), 0);  // reset
second channel
            PWM_Set_Compare2(PWM_Set_pSlaveHandle(hpwm,hpwm3), 0);  // reset
second channel
            break;

        case PWM_ACTIVECHANNEL_C1:
            if(SineOrDC)
                PWM_Set_SlaveDuty_From_Table(hspwm3, hspwm3->sPWM_Channel2,
rotate_ind_A);
            else
                PWM_Set_SlaveDuty_From_Value(hspwm3, hspwm3->sPWM_Channel2);
// set second channel
            PWM_Set_Compare1(hpwm, 0);  // reset first channel
            PWM_Set_Compare2(hpwm, 0);  // reset second channel
            PWM_Set_Compare1(PWM_Set_pSlaveHandle(hpwm,hpwm2), 0);  // reset
first channel
            PWM_Set_Compare2(PWM_Set_pSlaveHandle(hpwm,hpwm2), 0);  // reset
second channel
            PWM_Set_Compare1(PWM_Set_pSlaveHandle(hpwm,hpwm3), 0);  // reset
first channel
            break;

        default:
            PWM_Set_Compare1(hpwm, 0);  // reset first channel
            PWM_Set_Compare2(hpwm, 0);  // reset second channel
            PWM_Set_Compare1(PWM_Set_pSlaveHandle(hpwm,hpwm2), 0);  // reset
first channel
            PWM_Set_Compare2(PWM_Set_pSlaveHandle(hpwm,hpwm2), 0);  // reset
second channel
            PWM_Set_Compare1(PWM_Set_pSlaveHandle(hpwm,hpwm3), 0);  // reset
first channel
            PWM_Set_Compare2(PWM_Set_pSlaveHandle(hpwm,hpwm3), 0);  // reset
second channel
            break;

    }


}

/**
```

```
    * @brief     Form Bridge Sine PWM. Forming Slave PWM included.
    * @param     hspwm                - указатель на хендл слейв ШИМ.
    * @param     rotate_ind_A     - индекс таблицы для Мастер ШИМ.
    * @param     rotate_ind_B     - индекс таблицы для Слейв ШИМ 2.
    * @param     rotate_ind_C     - индекс таблицы для Слейв ШИМ 3.
    * @note      Индекс для свейл ШИМ расчитывается внутри функции.
    */
void PWM_Sine_Bridge_Mode(PWM_HandleTypeDef *hpwm, uint16_t rotate_ind_A,
uint16_t rotate_ind_B, uint16_t rotate_ind_C)
{
    if(hpwm->PWM_MaxPulseDur - hpwm->PWM_MinPulseDur > 2)
    {
        //----------MASTER PWM-----------
        // SET PHASE 1 (A)
        int Duty = PWM_Scaled_Element(hpwm, PWM_Get_Table_Element_Signed(hpwm,
rotate_ind_A));
        // если это первая полуволна
        if(Duty > 0) {
            PWM_Set_Compare1(hpwm, Duty+PWM_Calc_Min_Duty(hpwm));   // set
first channel
            PWM_Set_Compare2(hpwm, 0);      // reset second channel
        }
        // если это вторая полуволна
        else {
            PWM_Set_Compare1(hpwm, 0);      // reset first channel
            PWM_Set_Compare2(hpwm, (-Duty)+PWM_Calc_Min_Duty(hpwm));    // set
second channel
        }

        //----------SLAVE PWMs-----------
        if (PWM_Get_Mode(hpwm, PWM_PHASE_MODE))
        {
            // SET PHASE 2 (B)
            PWM_SlaveHandleTypeDef *hspwm = PWM_Set_pSlaveHandle(hpwm,hpwm2);
            Duty = PWM_Scaled_Element(hpwm, PWM_Get_Table_Element_Signed(hpwm,
rotate_ind_B));
            // если это первая полуволна
            if(Duty > 0) {
                PWM_Set_Compare1(hspwm, Duty+PWM_Calc_Min_Duty(hspwm-
>hMasterPWM)); // set first channel
                PWM_Set_Compare2(hspwm, 0);         // reset second channel
            }
            // если это вторая полуволна
            else {
                PWM_Set_Compare1(hspwm, 0);         // reset first channel
                PWM_Set_Compare2(hspwm, (-Duty)+PWM_Calc_Min_Duty(hspwm-
>hMasterPWM));      // set second channel
            }

            // SET PHASE 3 (C)
            hspwm = PWM_Set_pSlaveHandle(hpwm,hpwm3);
            Duty = PWM_Scaled_Element(hpwm, PWM_Get_Table_Element_Signed(hpwm,
rotate_ind_C));
            // если это первая полуволна
            if(Duty > 0) {
                PWM_Set_Compare1(hspwm, Duty+PWM_Calc_Min_Duty(hspwm-
>hMasterPWM)); // set first channel
                PWM_Set_Compare2(hspwm, 0);         // reset second channel
            }
            // если это вторая полуволна
            else {
```

```
                PWM_Set_Compare1(hspwm, 0);         // reset first channel
                PWM_Set_Compare2(hspwm, (-Duty)+PWM_Calc_Min_Duty(hspwm-
>hMasterPWM));      // set second channel
            }
        }
        else
        {
            PWM_Set_Compare1(PWM_Set_pSlaveHandle(hpwm,hpwm2), 0);  // reset
first channel
            PWM_Set_Compare2(PWM_Set_pSlaveHandle(hpwm,hpwm2), 0);  // reset
second channel
            PWM_Set_Compare1(PWM_Set_pSlaveHandle(hpwm,hpwm3), 0);  // reset
first channel
            PWM_Set_Compare2(PWM_Set_pSlaveHandle(hpwm,hpwm3), 0);  // reset
second channel
        }
    }
    else
    {
        PWM_Set_Compare1(hpwm, 0);      // reset first channel
        PWM_Set_Compare2(hpwm, 0);      // reset second channel
        PWM_Set_Compare1(PWM_Set_pSlaveHandle(hpwm,hpwm2), 0);  // reset first
channel
        PWM_Set_Compare2(PWM_Set_pSlaveHandle(hpwm,hpwm2), 0);  // reset
second channel
        PWM_Set_Compare1(PWM_Set_pSlaveHandle(hpwm,hpwm3), 0);  // reset first
channel
        PWM_Set_Compare2(PWM_Set_pSlaveHandle(hpwm,hpwm3), 0);  // reset
second channel
    }
}



/**
    * @brief    Form Bridge DC PWM. Forming Slave PWM included.
    * @param    hspwm               - указатель на хендл слейв ШИМ.
    * @param    rotate_ind_A    - индекс таблицы для Мастер ШИМ.
    * @param    rotate_ind_B    - индекс таблицы для Слейв ШИМ 2.
    * @param    rotate_ind_C    - индекс таблицы для Слейв ШИМ 3.
    * @note     Индекс для свейл ШИМ расчитывается внутри функции.
    */
void PWM_DC_Bridge_Mode(PWM_HandleTypeDef *hpwm, uint16_t rotate_ind_A,
uint16_t rotate_ind_B, uint16_t rotate_ind_C)
{
        //----------MASTER PWM-----------
        // SET PHASE 1 (A)
        int Duty = PWM_Get_Table_Element_Signed(hpwm, rotate_ind_A);
        // если это первая полуволна
        if(Duty > 0) {
            PWM_Set_Duty_From_Value(hpwm, hpwm->sPWM_Channel1);     // set
first channel
            PWM_Set_Compare2(hpwm, 0);      // reset second channel
        }
        // если это вторая полуволна
        else {
            PWM_Set_Compare1(hpwm, 0);      // reset first channel
            PWM_Set_Duty_From_Value(hpwm, hpwm->sPWM_Channel2);     // set
second channel
        }
```

```
        //----------SLAVE PWMs-----------
        if (PWM_Get_Mode(hpwm, PWM_PHASE_MODE))
        {
            // SET PHASE 2 (B)
            PWM_SlaveHandleTypeDef *hspwm = PWM_Set_pSlaveHandle(hpwm,hpwm2);
            Duty = PWM_Get_Table_Element_Signed(hpwm, rotate_ind_B);
            // если это первая полуволна
            if(Duty > 0) {
                PWM_Set_SlaveDuty_From_Value(hspwm, hspwm->sPWM_Channel1);  //
set first channel
                PWM_Set_Compare2(hspwm, 0);         // reset second channel
            }
            // если это вторая полуволна
            else {
                PWM_Set_Compare1(hspwm, 0);         // reset first channel
                PWM_Set_SlaveDuty_From_Value(hspwm, hspwm->sPWM_Channel2);  //
set second channel
            }

            // SET PHASE 3 (C)
            hspwm = PWM_Set_pSlaveHandle(hpwm,hpwm3);
            Duty = PWM_Get_Table_Element_Signed(hpwm, rotate_ind_C);
            // если это первая полуволна
            if(Duty > 0) {
                PWM_Set_SlaveDuty_From_Value(hspwm, hspwm->sPWM_Channel1);  //
set first channel
                PWM_Set_Compare2(hspwm, 0);         // reset second channel
            }
            // если это вторая полуволна
            else {
                PWM_Set_Compare1(hspwm, 0);         // reset first channel
                PWM_Set_SlaveDuty_From_Value(hspwm, hspwm->sPWM_Channel2);  //
set second channel
            }
        }
        else
        {
            PWM_Set_Compare1(PWM_Set_pSlaveHandle(hpwm,hpwm2), 0);  // reset
first channel
            PWM_Set_Compare2(PWM_Set_pSlaveHandle(hpwm,hpwm2), 0);  // reset
second channel
            PWM_Set_Compare1(PWM_Set_pSlaveHandle(hpwm,hpwm3), 0);  // reset
first channel
            PWM_Set_Compare2(PWM_Set_pSlaveHandle(hpwm,hpwm3), 0);  // reset
second channel
        }
}


/**
    * @brief        Update PWM parameters.
    * @param        hpwm - указатель на хендл ШИМ.
    * @details  Проверка надо ли обновлять параметры ШИМ, и если надо -
обновляет их.
    * @note         This called from TIM_CTRL_Handler
    */
void PWM_Update_Params(PWM_HandleTypeDef *hpwm)
{
    unsigned UpdateModeParams = 0;
    unsigned UpdateLog = 0;
```

```
    static float PWM_Sine_setpoint;
    static float PWM_Duty_setpoint;

    // READ PWM_SINE_HZ
    if(PWM_Sine_setpoint != int_to_percent(pwm_ctrl[R_PWM_CTRL_PWM_SINE_HZ]))
    {
        PWM_Sine_setpoint = int_to_percent(pwm_ctrl[R_PWM_CTRL_PWM_SINE_HZ]);
        if((PWM_Sine_setpoint == 0) ||      // if zero - force reset pwm value
            (hpwm->PWM_Sine_Hz != hpwm->PWM_Sine_Hz ) )     // or if
PWM_Sine_Hz isnt in range
        {
            PWM_Ramp_Reset(&hpwm->hramp[PWM_RampSineHz_Ind]);
        }
        // update logs params
        UpdateLog = 1;
    }
        // READ DUTY_BRIDGE
    if(PWM_Duty_setpoint != pwm_ctrl[R_PWM_CTRL_DUTY_BRIDGE])
    {
        PWM_Duty_setpoint = int_to_percent(pwm_ctrl[R_PWM_CTRL_DUTY_BRIDGE]);
        if((PWM_Duty_setpoint == 0) ||      // if zero - force reset pwm value
            (hpwm->PWM_Duty != hpwm->PWM_Duty ) )       // or if PWM_Duty isnt
in range
        {
            PWM_Ramp_Reset(&hpwm->hramp[PWM_RampPWMDuty_Ind]);
        }
        // update mode params
        UpdateModeParams = 1;
        // update logs params
        UpdateLog = 1;
    }

    // UPDATE RAMP SINE HZ
    PWM_Ramp_ControlValue(&hpwm->hramp[PWM_RampSineHz_Ind], &hpwm-
>PWM_Sine_Hz, PWM_Sine_setpoint);
    // UPDATE RAMP
    PWM_Ramp_ControlValue(&hpwm->hramp[PWM_RampPWMDuty_Ind], &hpwm->PWM_Duty,
PWM_Duty_setpoint);

    // READ PWM_ACTIVECHANNEL_MODE
    uint16_t activechannesls =  (MB_Read_Coil_Local(&coils_regs[0],
COIL_PWM_ACTIVECHANNEL))|

(MB_Read_Coil_Local(&coils_regs[0], COIL_PWM_ACTIVECHANNEL+1) << 1)|

(MB_Read_Coil_Local(&coils_regs[0], COIL_PWM_ACTIVECHANNEL+2) << 2);
    if(PWM_Get_Mode(hpwm, PWM_ACTIVECHANNEL_MODE) != (activechannesls <<
PWM_ACTIVECHANNEL_MODE_Pos))
    {
        uint16_t tmpmode = hpwm->sPWM_Mode&(~PWM_ACTIVECHANNEL_MODE);
        tmpmode |= activechannesls << PWM_ACTIVECHANNEL_MODE_Pos;

        hpwm->sPWM_Mode = tmpmode;


        // update mode params
        UpdateModeParams = 1;
        // update logs params
        UpdateLog = 1;
    }
```

```
    // READ PWM_DC_MODE
    if(PWM_Get_Mode(hpwm, PWM_DC_MODE) != (MB_Read_Coil_Local(&coils_regs[0],
COIL_PWM_DC_MODE) << PWM_DC_MODE_Pos))
    {
        if(MB_Read_Coil_Local(&coils_regs[0], COIL_PWM_DC_MODE))
        {
            hpwm->sPWM_Mode |= PWM_DC_MODE;
        }
        else
        {
            hpwm->sPWM_Mode &= ~PWM_DC_MODE;
        }
        // update mode params
        UpdateModeParams = 1;
        // update logs params
        UpdateLog = 1;
    }

    // READ PWM_BRIDGE_MODE
    if(PWM_Get_Mode(hpwm, PWM_BRIDGE_MODE) !=
(MB_Read_Coil_Local(&coils_regs[0], COIL_PWM_BRIDGE_MODE) <<
PWM_BRIDGE_MODE_Pos))
    {
        if(MB_Read_Coil_Local(&coils_regs[0], COIL_PWM_BRIDGE_MODE))
        {
            hpwm->sPWM_Mode |= PWM_BRIDGE_MODE;
        }
        else
        {
            hpwm->sPWM_Mode &= ~PWM_BRIDGE_MODE;
        }
        // update mode params
        UpdateModeParams = 1;
        // update logs params
        UpdateLog = 1;
    }


    // READ PWM_PHASE_MODE
    if(PWM_Get_Mode(hpwm, PWM_PHASE_MODE) !=
(MB_Read_Coil_Local(&coils_regs[0], COIL_PWM_PHASE_MODE) <<
PWM_PHASE_MODE_Pos))
    {
        if(MB_Read_Coil_Local(&coils_regs[0], COIL_PWM_PHASE_MODE))
        {
            hpwm->sPWM_Mode |= PWM_PHASE_MODE;
        }
        else
        {
            hpwm->sPWM_Mode &= ~PWM_PHASE_MODE;
        }
        // update mode params
        UpdateModeParams = 1;
        // update logs params
        UpdateLog = 1;
    }



    // READ PWM_POLARITY
```

```
    uint16_t polarity_temp = MB_Read_Coil_Local(&coils_regs[0],
COIL_PWM_POLARITY);
    if(((hpwm->stim.htim.Instance->CCER&TIM_CCER_CC1P) || (hpwm-
>stim.htim.Instance->CCER&TIM_CCER_CC3P)) != (polarity_temp))
    {
        __HAL_TIM_SET_CAPTUREPOLARITY(&hpwm->stim.htim, hpwm->sPWM_Channel1,
polarity_temp<<1);
        __HAL_TIM_SET_CAPTUREPOLARITY(&hpwm->stim.htim, hpwm->sPWM_Channel2,
polarity_temp<<1);

        PWM_SlaveHandleTypeDef *shpwm = PWM_Set_pSlaveHandle(hpwm,hpwm2);
        __HAL_TIM_SET_CAPTUREPOLARITY(&shpwm->stim.htim, shpwm->sPWM_Channel1,
polarity_temp<<1);
        __HAL_TIM_SET_CAPTUREPOLARITY(&shpwm->stim.htim, shpwm->sPWM_Channel2,
polarity_temp<<1);

        shpwm = PWM_Set_pSlaveHandle(hpwm,hpwm3);
        __HAL_TIM_SET_CAPTUREPOLARITY(&shpwm->stim.htim, shpwm->sPWM_Channel1,
polarity_temp<<1);
        __HAL_TIM_SET_CAPTUREPOLARITY(&shpwm->stim.htim, shpwm->sPWM_Channel2,
polarity_temp<<1);


    }



    // READ TABLE_SIZE
    if(hpwm->Duty_Table_Size != pwm_ctrl[R_PWM_CTRL_SIN_TABLE_SIZE])
    {
        hpwm->Duty_Table_Size = PWM_Fill_Sine_Table(&hpwm1,
pwm_ctrl[R_PWM_CTRL_SIN_TABLE_SIZE]);
        pwm_ctrl[R_PWM_CTRL_SIN_TABLE_SIZE] = hpwm->Duty_Table_Size;
    }



    // READ MIN PULSE DURATION
    if(hpwm->PWM_MinPulseDur != pwm_ctrl[R_PWM_CTRL_MIN_PULSE_DUR])
    {
        hpwm->PWM_MinPulseDur = pwm_ctrl[R_PWM_CTRL_MIN_PULSE_DUR];
        // update mode params
        UpdateModeParams = 1;
        // update logs params
        UpdateLog = 1;
    }
    // READ MAX PULSE DURATION
    if((hpwm->PWM_MaxPulseDur != pwm_ctrl[R_PWM_CTRL_MAX_PULSE_DUR]) &&
        (pwm_ctrl[R_PWM_CTRL_MAX_PULSE_DUR] <= PWM_Get_Autoreload(hpwm)) &&
        (pwm_ctrl[R_PWM_CTRL_MAX_PULSE_DUR] != 0))
    {
        hpwm->PWM_MaxPulseDur = pwm_ctrl[R_PWM_CTRL_MAX_PULSE_DUR];
        // update mode params
        UpdateModeParams = 1;
        // update logs params
        UpdateLog = 1;
    }
    else if (((hpwm->PWM_MaxPulseDur != PWM_Get_Autoreload(hpwm)) &&
(pwm_ctrl[R_PWM_CTRL_MAX_PULSE_DUR] == 0)) ||
                    ((hpwm->PWM_MaxPulseDur != PWM_Get_Autoreload(hpwm)) &&
(pwm_ctrl[R_PWM_CTRL_MAX_PULSE_DUR] > PWM_Get_Autoreload(hpwm))))
```

```
    {
        hpwm->PWM_MaxPulseDur = PWM_Get_Autoreload(hpwm);
        // update mode params
        UpdateModeParams = 1;
    }


    // READ DEAD TIME
    if(hpwm->PWM_DeadTime != pwm_ctrl[R_PWM_CTRL_DEAD_TIME])
    {
        hpwm->PWM_DeadTime = pwm_ctrl[R_PWM_CTRL_DEAD_TIME];
    }


    // UPDATE PWM PARAMS
    if(UpdateModeParams)
    {
        // UPDATE DUTY TABLE SCALE
        PWM_Update_DutyTableScale(hpwm);


        // update logs params
        UpdateLog = 1;
    }

    // UPDATE LOG PARAMS
    if(UpdateLog)
    {
        // set logs params
        Set_Log_Params();
    }
}

/**
    * @brief        PID for ramp.
    * @param        hramp           - указатель на хендл PID-регулятора.
    * @param        PID_Output  - указатель на переменную для выхода
регулятора.
    * @param        PID_Input   - задание на ПИД-регулятор.
    * @details  ПИД-регулятор, который управляет скважностью ШИМ и не дает её
изменяться резко.
    */
void PWM_Ramp_ControlValue(PWM_RampHandleTypeDef *hramp, float *PID_Output,
float PID_Input)
{
    float out;
    float err = PID_Input-*PID_Output;

    /* Limiting the accelerate */
    if (err > hramp->limMaxInt)
        err = hramp->limMaxInt;
    else if (err < hramp->limMinInt)
        err = hramp->limMinInt;

    out = arm_pid_f32(&hramp->pid, err);

    /* Limiting the accelerate */
    if (out > hramp->limMax)
        out = hramp->limMax;
    else if (out < hramp->limMin)
        out = hramp->limMin;
```

```
    /* Apply the PID-regulating */
    if(PID_Input != 0)
    {
        PWM_Ramp_UpdateParams(hramp, 0);
        *PID_Output = out;
    }
    else // if input = 0 - disable pid and set 0
    {
        PWM_Ramp_Reset(hramp);
        *PID_Output = 0;
    }
}

/**
    * @brief        Initialize PID params for ramp.
    * @param        hramp       - указатель на хендл PID-регулятора.
    * @param        flagReset - флаг для сброса ПИД-регулятора (сброс
регулирования, не параметров).
    * @details  Обновляет параметры A0, A1, A2 для регилуривания @ref PID
Motor Control
    */
void PWM_Ramp_UpdateParams(PWM_RampHandleTypeDef *hramp, int32_t flagReset)
{
//  hramp->pid.Ki
    arm_pid_init_f32(&hramp->pid, flagReset);
}

/**
    * @brief        Reset PID for ramp.
    * @param        hramp - указатель на хендл PID-регулятора.
    * @details  Сбрасывает переменные ПИД-регулятора, чтобы потом его
запустить "с чистого листа".
    */
void PWM_Ramp_Reset(PWM_RampHandleTypeDef *hramp)
{
    arm_pid_reset_f32(&hramp->pid);
}


/**
    * @brief        reInitialization of PWM TIM.
    * @param        hpwm - указатель на хендл ШИМ.
    * @details  Перенастраивает таймер согласно принятным настройкам в
pwm_ctrl.
    */
void PWM_Sine_ReInit(PWM_HandleTypeDef *hpwm)
{
    Trace_PWM_reInit_Enter();
    TIM_Base_MspDeInit(&hpwm->stim.htim);
    hpwm1.stim.sTickBaseUS = TIMER_PWM_TICKBASE;
    hpwm1.sConfigPWM.OCPolarity = MB_Read_Coil_Local(&coils_regs[0],
COIL_PWM_POLARITY) << TIM_CCER_CC1P_Pos;
    TIM_Base_Init(&hpwm->stim);
    TIM_Output_PWM_Init(&hpwm->stim.htim, &hpwm->sConfigPWM, hpwm-
>sPWM_Channel1, hpwm->GPIOx, hpwm->GPIO_PIN_X1);
    TIM_Output_PWM_Init(&hpwm->stim.htim, &hpwm->sConfigPWM, hpwm-
>sPWM_Channel2, hpwm->GPIOx, hpwm->GPIO_PIN_X2);


    PWM_SynchInit();
```

```
    PWM_SlavePhase_reInit(&hpwm2);
    PWM_SlavePhase_reInit(&hpwm3);

    PWM_Update_DutyTableScale(hpwm);


    //----------TIMERS START-------------
    HAL_TIM_OC_Start(&hpwm1.stim.htim, TIM_CHANNEL_3);
    HAL_TIM_PWM_Start(&hpwm1.stim.htim, hpwm->sPWM_Channel1); // PWM channel 1
    HAL_TIM_PWM_Start(&hpwm1.stim.htim, hpwm->sPWM_Channel2); // PWM channel 2
    HAL_TIM_Base_Start_IT(&hpwm1.stim.htim); // timer for PWM

    Trace_PWM_reInit_Exit();
}
/**
    * @brief        Getting ind for Duty Table.
    * @param        hpwm - указатель на хендл ШИМ.
    * @param        FreqTIM - частота таймера ШИМ.
    * @details  Рассчитывает индекс для таблицы скважностей.
    *                       PWM_Sine_Hz в hpwm - частота с которой эта таблица
должна выводиться на ШИМ
    */
uint32_t PWM_Get_Duty_Table_Ind(PWM_HandleTypeDef *hpwm, float FreqTIM)
{
    float sine_ind_step;
    // calc ind for sin table
    if(hpwm->PWM_Sine_Hz != 0) // if there some frequency
    {
        sine_ind_step = hpwm->Duty_Table_Size/(FreqTIM/hpwm->PWM_Sine_Hz);
        hpwm->Duty_Table_Ind += sine_ind_step;
    }
    else
        return hpwm->Duty_Table_Size;

    // overflow check
    if(hpwm->Duty_Table_Ind >= hpwm->Duty_Table_Size)
        hpwm->Duty_Table_Ind -= hpwm->Duty_Table_Size;
    if(hpwm->Duty_Table_Ind >= hpwm->Duty_Table_Size)
        hpwm->Duty_Table_Ind = 0;

    // if its too big (e.g. inf)
    if((hpwm->Duty_Table_Ind >= 0xFFFF) ||
        (hpwm->Duty_Table_Ind != hpwm->Duty_Table_Ind)) // in nan case
        hpwm->Duty_Table_Ind = 0;

    return hpwm->Duty_Table_Ind;

}



/**
    * @brief    Create Dead Time when switches channels.
    * @param    hpwm - указатель на хендл ШИМ.
    * @param    LocalDeadTimeCnt        - указатель на переменную для
отсчитывания дедтайма.
    * @param    LocalActiveChannel  - указатель на переменную для отслеживания
смены канала.
    */
void PWM_CreateDeadTime(PWM_HandleTypeDef *hpwm, float *LocalDeadTimeCnt,
unsigned *LocalActiveChannel)
```

```
{
    // get current active channel
    hpwm->fActiveChannel = (PWM_Get_Compare2(hpwm) != 0); // if channel two is
active - write 1, otherwise - 0
    // when channels are swithed and no dead time currently active
    if(*LocalActiveChannel != hpwm->fActiveChannel)
    {   // update active channel
        *LocalActiveChannel = hpwm->fActiveChannel;
        // set deadtime
        *LocalDeadTimeCnt = hpwm->PWM_DeadTime;
        Trace_PWM_DeadTime_Enter();
    }
    // decrement dead time
    *LocalDeadTimeCnt -= (PWM_Get_Autoreload(hpwm)+1)*hpwm->stim.sTickBaseUS;
    if(*LocalDeadTimeCnt > 0) // if dead time is still active
    { // reset all channels
        // reset channels
        PWM_Set_Compare1(hpwm, 0);
        PWM_Set_Compare2(hpwm, 0);
    }
    else // if dead time is done
    { // set it to zero
        *LocalDeadTimeCnt = 0;
        Trace_PWM_DeadTime_Exit();
    }
}

/**
    * @brief        Filling table with one period of sinus values.
    * @param        hpwm - указатель на хендл ШИМ.
    * @param        table_size - размер таблицы.
    * @details  Формирует таблицу синусов размером table_size.
    */
uint32_t PWM_Fill_Sine_Table(PWM_HandleTypeDef *hpwm, uint32_t table_size)
{
    if((hpwm == NULL) || (hpwm->pDuty_Table_Origin == NULL)  || (table_size ==
0))
    {
        return 0;
    }
    if (table_size > SIN_TABLE_SIZE_MAX)
        table_size = SIN_TABLE_SIZE_MAX;


    hpwm->Duty_Table_Size = table_size;
    float pi_step = 2*M_PI/(hpwm->Duty_Table_Size);
    float pi_val = 0;
    float sin_koef = 0;
    uint32_t sin_val = 0;

    // fill table with sinus
    for(int i = 0; i < hpwm->Duty_Table_Size; i++)
    {
        // rotate pi
        pi_val += pi_step;
        // calc sin value
        sin_koef = (float)0xFFFF;
        sin_val = (sin(pi_val)+1)*sin_koef/2;
        sin_table[i] = sin_val;
    }
    // fill rest of table with zeros
```

```
    for(int i = hpwm->Duty_Table_Size; i < SIN_TABLE_SIZE_MAX; i++)
        sin_table[i] = 0;

    // if second channel is enabled
    PWM_Update_DutyTableScale(hpwm);

    return hpwm->Duty_Table_Size;
}

/**
    * @brief        Calc and update new Duty Table Scale.
    * @param        hpwm - указатель на хендл ШИМ.
    * @details  Используется, когда изменяется значение регистра ARR.
    */
void PWM_Update_DutyTableScale(PWM_HandleTypeDef *hpwm)
{
            // UPDATE DUTY TABLE SCALE
        if(PWM_Get_Mode(hpwm, PWM_BRIDGE_MODE)) // if second channel is
enabled
        {
            hpwm->Duty_Table_Scale = PWM_Calc_Duty_Scale(&hpwm1, 0x8000);
        }
        else
        {
            hpwm->Duty_Table_Scale = PWM_Calc_Duty_Scale(&hpwm1, 0xFFFF);
        }
    // for case if min pulse dur is too big and scale is negative
        if (hpwm->Duty_Table_Scale < 0)
            hpwm->Duty_Table_Scale = 1;
}

//-------------------------------------------------------------------
//----------------------THREEPHASE FUNCTIONS-------------------------
/**
    * @brief        Initialization of Slave PWM TIM.
    * @param        hspwm - указатель на хендл слейв ШИМ.
    * @details  Вызывает функции инициализации и включения слейв ШИМ.
    */
void PWM_SlavePhase_Init(PWM_SlaveHandleTypeDef *hspwm)
{
    TIM_Base_Init(&hspwm->stim);
    TIM_Output_PWM_Init(&hspwm->stim.htim, &hspwm->hMasterPWM->sConfigPWM,
hspwm->sPWM_Channel1, hspwm->GPIOx, hspwm->GPIO_PIN_X1);
    TIM_Output_PWM_Init(&hspwm->stim.htim, &hspwm->hMasterPWM->sConfigPWM,
hspwm->sPWM_Channel2, hspwm->GPIOx, hspwm->GPIO_PIN_X2);

    TIM_MasterConfigTypeDef sMasterConfig = {0};
    TIM_SlaveConfigTypeDef  sSlaveConfig = {0};
    //--------SLAVE SYNCH INIT----------
  sSlaveConfig.SlaveMode = TIM_SLAVEMODE_TRIGGER;
  sSlaveConfig.InputTrigger = TIM_TS_ITR3;
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_OC2REF;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;

  HAL_TIM_SlaveConfigSynchro(&hpwm2.stim.htim, &sSlaveConfig);
  HAL_TIMEx_MasterConfigSynchronization(&hpwm2.stim.htim, &sMasterConfig);

  HAL_TIM_SlaveConfigSynchro(&hpwm3.stim.htim, &sSlaveConfig);
  HAL_TIMEx_MasterConfigSynchronization(&hpwm3.stim.htim, &sMasterConfig);
    //----------TIMERS START------------
    HAL_TIM_Base_Start(&hspwm->stim.htim);
```

```
    HAL_TIM_PWM_Start(&hspwm->stim.htim, hspwm->sPWM_Channel1); // PWM channel
1
    HAL_TIM_PWM_Start(&hspwm->stim.htim, hspwm->sPWM_Channel2); // PWM channel
2

    if(PWM_Get_Mode(hspwm->hMasterPWM, PWM_PHASE_MODE) == 0) // if three phase
disabled
    {
        PWM_Set_Compare1(hspwm, 0); // reset first channel
        PWM_Set_Compare2(hspwm, 0); // reset second channel
    }
}
/**
    * @brief    reInitialization of Slave PWM TIM.
    * @param    hspwm - указатель на хендл слейв ШИМ.
    */
void PWM_SlavePhase_reInit(PWM_SlaveHandleTypeDef *hspwm)
{
    PWM_Slave_CopyTimSetting(hspwm, sTimFreqHz);
    PWM_Slave_CopyTimSetting(hspwm, sTickBaseUS);
    PWM_Slave_CopyTimSetting(hspwm, sTickBasePrescaler);
    TIM_Base_MspDeInit(&hspwm->stim.htim);

    PWM_SlavePhase_Init(hspwm);
}

/**
    * @brief    Set Duty from table on Slave PWM at one channel by sin_ind of
the Master PWM.
    * @param    hspwm       - указатель на хендл слейв ШИМ.
    * @param    sin_ind     - индекс таблицы для Мастер ШИМ.
    * @note     Индекс для свейл ШИМ расчитывается в самой функции.
    */
uint16_t PWM_SlavePhase_Calc_TableInd(PWM_SlaveHandleTypeDef *hspwm, uint16_t
rotate_ind_Phase)
{
    // if three phase enables
        if(hspwm->Duty_Shift_Ratio > 0)
            rotate_ind_Phase += hspwm->hMasterPWM->Duty_Table_Size*hspwm-
>Duty_Shift_Ratio;
        else
            rotate_ind_Phase += hspwm->hMasterPWM->Duty_Table_Size*(1+hspwm-
>Duty_Shift_Ratio);

        // overflow check
        if(rotate_ind_Phase > hspwm->hMasterPWM->Duty_Table_Size)
            rotate_ind_Phase -= hspwm->hMasterPWM->Duty_Table_Size;
        if(rotate_ind_Phase > hspwm->hMasterPWM->Duty_Table_Size)
            rotate_ind_Phase = 0;;


        return rotate_ind_Phase;
//
//      PWM_Set_SlaveDuty_From_Table(hspwm, sin_ind);                       //
set first channel
}


/**
    * @brief        Check is all Slave channels works properly.
    * @param        hspwm - указатель на хендл слейв ШИМ.
```

```
    * @details  Проверка работает ли только один из каналов, и проверка чтобы
CCRx <= ARR
    * @details  В мастере проверка происходит напрямую в PWM_Handler.
    */
void PWM_SlavePhase_Check_Channels(PWM_SlaveHandleTypeDef *hspwm)
{
    uint16_t min_duty = PWM_Calc_Min_Duty(hspwm->hMasterPWM);
    uint16_t max_duty = PWM_Calc_Max_Duty(hspwm->hMasterPWM);
    // IF FIRST CHANNEL IS ACRIVE
    if(PWM_Get_Compare1(hspwm) != 0)
    {
        // Duty shoud be bigger or equeal than min duration
        if (PWM_Get_Compare1(hspwm)<min_duty)
            PWM_Set_Compare1(hspwm, min_duty);
        // Duty shoud be less or equeal than ARR-min duration
        if (PWM_Get_Compare1(hspwm)>max_duty)
            PWM_Set_Compare1(hspwm, max_duty);
    }
    // IF SECOND CHANNEL IS ACRIVE
    else if(PWM_Get_Compare2(hspwm) != 0)
    // Duty shoud be bigger or equeal than min duration
    if (PWM_Get_Compare2(hspwm)<min_duty)
        PWM_Set_Compare2(hspwm, min_duty);
    // Duty shoud be less or equeal than ARR
    if (PWM_Get_Compare2(hspwm)>max_duty)
        PWM_Set_Compare2(hspwm, max_duty);
    // IF BOTH CHANNEL IS ACRIVE
    if((PWM_Get_Compare1(hspwm) != 0) && (PWM_Get_Compare2(hspwm) != 0))
    {
        // Only one channel shoud be active so disable all
        PWM_Set_Compare1(hspwm, 0);
        PWM_Set_Compare2(hspwm, 0);
    }
}

/**
    * @brief    Create Dead Time for Slave PWM when switches channels.
    * @param    hspwm                          - указатель на хендл слейв
ШИМ.
    * @param    LocalDeadTimeCnt        - указатель на переменную для
отсчитывания дедтайма.
    * @param    LocalActiveChannel  - указатель на переменную для отслеживания
смены канала.
    * @note     Аналог функции PWM_CreateDeadTime но для слейв ШИМов.
    */
void PWM_SlavePhase_CreateDeadTime(PWM_SlaveHandleTypeDef *hspwm, float
*LocalDeadTimeCnt, unsigned *LocalActiveChannel)
{
    // get current active channel
    hspwm->fActiveChannel = (PWM_Get_Compare2(hspwm) != 0); // if channel two
is active - write 1, otherwise - 0
    // when channels are swithed and no dead time currently active
    if(*LocalActiveChannel != hspwm->fActiveChannel)
    {   // update active channel
        *LocalActiveChannel = hspwm->fActiveChannel;
        // set deadtime
        *LocalDeadTimeCnt = hspwm->hMasterPWM->PWM_DeadTime;
        Trace_PWM_DeadTime_Enter();
    }
    // decrement dead time
```

```
    *LocalDeadTimeCnt -= (PWM_Get_Autoreload(hspwm)+1)*hspwm->hMasterPWM-
>stim.sTickBaseUS;
    if(*LocalDeadTimeCnt > 0) // if dead time is still active
    { // reset all channels
        // reset channels
        PWM_Set_Compare1(hspwm, 0);
        PWM_Set_Compare2(hspwm, 0);
    }
    else // if dead time is done
    { // set it to zero
        *LocalDeadTimeCnt = 0;
        Trace_PWM_DeadTime_Exit();
    }
}
//---------------------------------------------------------------------
//-------------------------HANDLERS FUNCTIONS--------------------------
//----------------PWM TIMER-----------------
#if (PWM_MASTER_TIM_NUMB == 1) || (PWM_MASTER_TIM_NUMB == 10) // choose
handler for TIM
void TIM1_UP_TIM10_IRQHandler(void)
#elif (PWM_MASTER_TIM_NUMB == 2)
void TIM2_IRQHandler(void)
#elif (PWM_MASTER_TIM_NUMB == 3)
void TIM3_IRQHandler(void)
#elif (PWM_MASTER_TIM_NUMB == 4)
void TIM4_IRQHandler(void)
#elif (PWM_MASTER_TIM_NUMB == 5)
void TIM5_IRQHandler(void)
#elif (PWM_MASTER_TIM_NUMB == 6)
void TIM6_DAC_IRQHandler(void)
#elif (PWM_MASTER_TIM_NUMB == 7)
void TIM7_IRQHandler(void)
#elif (PWM_MASTER_TIM_NUMB == 8) || (PWM_MASTER_TIM_NUMB == 13)
void TIM8_UP_TIM13_IRQHandler(void)
#elif (PWM_MASTER_TIM_NUMB == 1) || (PWM_MASTER_TIM_NUMB == 9)
void TIM1_BRK_TIM9_IRQHandler(void)
#elif (PWM_MASTER_TIM_NUMB == 1) || (PWM_MASTER_TIM_NUMB == 11)
void TIM1_TRG_COM_TIM11_IRQHandler(void)
#elif (PWM_MASTER_TIM_NUMB == 8) || (PWM_MASTER_TIM_NUMB == 12)
void TIM8_BRK_TIM12_IRQHandler(void)
#elif (PWM_MASTER_TIM_NUMB == 8) || (PWM_MASTER_TIM_NUMB == 14)
void TIM8_TRG_COM_TIM14_IRQHandler(void)
#endif
{
    /* TIM_PWM_Handler */
    Trace_PWM_TIM_Enter();
  HAL_TIM_IRQHandler(&hpwm1.stim.htim);
    PWM_Handler(&hpwm1);

    Trace_PWM_TIM_Exit();
}
```

ПРИЛОЖЕНИЕ Б

Листинг pwm.h

```
/**
    ******************************************************************************
    * @file     pwm.h
    * @brief    Заголовочный файл для модуля реализации ШИМ.
    *
    * @defgroup PWM
    * @brief        PWM stuff
    *
    ******************************************************************************/
#ifndef __PWM_H_
#define __PWM_H_
#include "control.h"



extern uint32_t sin_table[SIN_TABLE_SIZE_MAX];

/**
    * @brief    Transform int "XXxx" to float "XX.xx".
    * @param    _int_   - число для преобразования.
    * @return _val_ - преобразованное число.
    */
#define int_to_percent(_int_)       ((float)_int_/100)

////////////////////////////////////////////////////////////////////
//////////////////////////////---DEFINES---///////////////////////////
//-----------------------------PWM HANDLE-----------------------------//
/**
    * @addtogroup PWM_HANDLE_DEFINES
    * @ingroup      PWM
    * @brief            Defines for access and processing elements of handle
structure
    @{
    */

/**
    * @brief    Calc duration of maximum pulse in ticks.
    * @param    _hpwm_              - указатель на хендл pwm.
    * @return _val_                 - количество тиков кратчайшего импульса.
    */
#define PWM_Calc_Max_Duty(_hpwm_)
(((_hpwm_)->PWM_MaxPulseDur)/(_hpwm_)->stim.sTickBaseUS)

/**
    * @brief    Calc duration of minimum pulse in ticks.
    * @param    _hpwm_              - указатель на хендл pwm.
    * @return _val_                 - количество тиков кратчайшего импульса.
    */
#define PWM_Calc_Min_Duty(_hpwm_)
(((_hpwm_)->PWM_MinPulseDur)/(_hpwm_)->stim.sTickBaseUS)

/**
    * @brief    Calc Scale Koef for Table & AUTORELOAD REGISTER
    * @param    _hpwm_              - указатель на хендл pwm.
    * @param    _scale_             - верхняя граница диапазона значений.
    * @return _koef_                - коэффициент для масштабирования.
```

```
    * @details  Данный макрос рассчитывает коэффициент для приведения значений
с диапазоном [0,_scale_]
                        к регистру автозагрузки с диапазоном [min, max], где
min, max - минимальная и максимальная длительность импульса в тактах.
    */
#define PWM_Calc_Duty_Scale(_hpwm_, _scale_)
((float)(PWM_Calc_Max_Duty(_hpwm_)-PWM_Calc_Min_Duty(_hpwm_))/(_scale_))

/**
    * @brief    Get Table Element Scaled corresponding to TIM ARR register
    * @param    _hpwm_              - указатель на хендл pwm.
    * @param    _ind_               - номер элемента из таблицы скважностей.
    * @return _val_                 - масштабированный под регистры таймера
значение.
    */
#define PWM_Get_Table_Element_Unsigned(_hpwm_,_ind_)
(*((_hpwm_)->pDuty_Table_Origin+_ind_))

/**
    * @brief    Get Table Element Shifted corresponding to TIM ARR register
    * @param    _hpwm_              - указатель на хендл pwm.
    * @param    _ind_               - номер элемента из таблицы скважностей.
    * @return _val_                 - масштабированный под регистры таймера
значение.
    * @details  По сути такая же как PWM_Get_Table_Element_Unsigned но
добавляется сдвиг вниз на одну амплитуду для учитывания знака.
                        (если точнее, то сдвиг добавляется для компенсации
сдвига вверх, для хранения в unsigned)
    * @note     0x8000*(_hpwm_)->Duty_Table_Scale - т.к. первая полуволна
находится в диапазоне (0x8000-0xFFFF) вычитаем константу 0x8000 с
масштабированием
    */
#define PWM_Get_Table_Element_Signed(_hpwm_,_ind_)
((int)PWM_Get_Table_Element_Unsigned(_hpwm_,_ind_)-0x8000)

/**
    * @brief    Scale Table Element corresponding to TIM ARR register
    * @param    _hpwm_              - указатель на хендл pwm.
    * @param    _element_       - число которое надо скейлнуть.
    * @details  Если задана минимальная длительность импульса в тактах n,
                        то регистр ARR заполняется так, что диапазон его
значений будет [n, ARR-n]
    */
#define PWM_Scaled_Element(_hpwm_, _element_)
((_element_)*((_hpwm_)->Duty_Table_Scale))
/**
    * @brief    Create pointer to slave PWM from pointer to void in
PWM_HandleTypeDef.
    * @param    _hpwm_              - указатель на хендл pwm.
    * @param    _slavepwm_      - имя слейв pwm.
    * @return _pslavepwm_   - указатель на структуру PWM_SlaveHandleTypeDef.
    */
#define PWM_Set_pSlaveHandle(_hpwm_,_slavepwm_)
((PWM_SlaveHandleTypeDef *)_hpwm_->_slavepwm_)
/**
    * @brief    Copy setting from master TIM_SettingsTypeDef to slave
TIM_SettingsTypeDef.
    * @param    _hpwm_              - указатель на хендл pwm.
    * @param    _set_               - имя настройки.
    */
```

```
#define PWM_Slave_CopyTimSetting(_hspwm_, _set_)
((_hspwm_)->stim._set_ = (_hspwm_)->hMasterPWM->stim._set_)

/** PWM_HANDLE_DEFINES
  * @}
  */
//-----------------------------------------------------------------//

//---------------------------TIMER REGS----------------------------//
/**
    * @addtogroup PWM_TIMER_REGS_DEFINES
    * @ingroup      PWM
    * @brief            Defines for access to timers registers and processing
handle structure
    @{
    */

/**
    * @brief    Get PWM autoreload value (max duty value).
    * @param    _hpwm_              - указатель на хендл pwm.
    */
#define PWM_Get_Autoreload(_hpwm_)
__HAL_TIM_GET_AUTORELOAD(&((_hpwm_)->stim.htim))

/**
    * @brief    Get PWM Counter value.
    * @param    _hpwm_              - указатель на хендл pwm.
    */
#define PWM_Get_Counter(_hpwm_)
__HAL_TIM_GET_COUNTER(&((_hpwm_)->stim.htim))
/**
    * @brief    Set PWM Counter value.
    * @param    _hpwm_              - указатель на хендл pwm.
    * @param    _val_               - значение, которое нужно записать в TIM-
>CNT.
    */
#define PWM_Set_Counter(_hpwm_, _val_)
__HAL_TIM_SET_COUNTER(&((_hpwm_)->stim.htim), (_val_))

/**
* @brief    Get PWM Duty from channel 1.
    * @param    _hpwm_              - указатель на хендл pwm.
    * @param    _val_               - значение, которое нужно записать в
Compare.
    */
#define PWM_Get_Compare1(_hpwm_)
__HAL_TIM_GET_COMPARE(&((_hpwm_)->stim.htim), (_hpwm_)->sPWM_Channel1)
/**
* @brief    Get PWM Duty from channel 2.
    * @param    _hpwm_              - указатель на хендл pwm.
    * @param    _val_               - значение, которое нужно записать в
Compare.
    */
#define PWM_Get_Compare2(_hpwm_)
__HAL_TIM_GET_COMPARE(&((_hpwm_)->stim.htim), (_hpwm_)->sPWM_Channel2)

/**
    * @brief    Set PWM Duty on Channel 1.
    * @param    _hpwm_              - указатель на хендл pwm.
    * @param    _val_               - значение, которое нужно записать в
Compare.
```

```
    */
#define PWM_Set_Compare1(_hpwm_, _val_)
__HAL_TIM_SET_COMPARE(&((_hpwm_)->stim.htim), (_hpwm_)->sPWM_Channel1,
(_val_))
/**
    * @brief    Set PWM Duty on Channel 2.
    * @param    _hpwm_              - указатель на хендл pwm.
    * @param    _val_               - значение, которое нужно записать в
Compare.
    */
#define PWM_Set_Compare2(_hpwm_, _val_)
__HAL_TIM_SET_COMPARE(&((_hpwm_)->stim.htim), (_hpwm_)->sPWM_Channel2,
(_val_))

/**
    * @brief    Set PWM Duty From PWM_Sine_Hz Percent
    * @param    _hpwm_              - указатель на хендл pwm.
    * @param    _channel_       - канал для выставления скважности.
    */
#define PWM_Set_Duty_From_Value(_hpwm_, _channel_)
__HAL_TIM_SET_COMPARE(&((_hpwm_)->stim.htim), _channel_, ((_hpwm_)-
>PWM_Duty/100)*(PWM_Get_Autoreload(_hpwm_)+1))

/**
    * @brief    Set PWM Duty From table
    * @param    _hpwm_              - указатель на хендл pwm.
    * @param    _channel_       - канал для выставления скважности.
    * @param    _ind_               - номер элемента из таблицы скважностей.
    */
#define PWM_Set_Duty_From_Table(_hpwm_, _channel_, _ind_)
__HAL_TIM_SET_COMPARE(&((_hpwm_)->stim.htim), _channel_, PWM_Scaled_Element(
_hpwm_, PWM_Get_Table_Element_Unsigned((_hpwm_), (_ind_))+1) +
PWM_Calc_Min_Duty(_hpwm_))

/**
    * @brief    Set PWM Duty From PWM_Sine_Hz Percent
    * @param    _hpwm_              - указатель на хендл pwm.
    * @param    _channel_       - канал для выставления скважности.
    */
#define PWM_Set_SlaveDuty_From_Value(_hpwm_, _channel_)
__HAL_TIM_SET_COMPARE(&((_hpwm_)->stim.htim), _channel_, ((_hpwm_)-
>hMasterPWM->PWM_Duty/100)*(PWM_Get_Autoreload(_hpwm_)+1))

/**
    * @brief    Set PWM Duty From table
    * @param    _hpwm_              - указатель на хендл pwm.
    * @param    _channel_       - канал для выставления скважности.
    * @param    _ind_               - номер элемента из таблицы скважностей.
    */
#define PWM_Set_SlaveDuty_From_Table(_hpwm_, _channel_, _ind_)
__HAL_TIM_SET_COMPARE(&((_hpwm_)->stim.htim), _channel_,
PWM_Scaled_Element((_hpwm_)->hMasterPWM,
PWM_Get_Table_Element_Unsigned((_hpwm_)->hMasterPWM, (_ind_))+1) +
PWM_Calc_Min_Duty((_hpwm_)->hMasterPWM))

/** TIMER REGS
  * @}
  */
//-------------------------------------------------------------------//
```

```
//--------------------------MODE DEFINES---------------------------//
/**
    * @addtogroup PWM_MODE_DEFINES
    * @ingroup      PWM
    * @brief            Defines for bits of @ref PWM_ModeTypeDef
    @{
    */
#define PWM_DC_MODE_Pos                          (0)     ///< Position of DC
MODE bit in @ref PWM_ModeTypeDef
#define PWM_BRIDGE_MODE_Pos                  (1)     ///< Position of BRIDGE
MODE bit in @ref PWM_ModeTypeDef
#define PWM_PHASE_MODE_Pos                   (2)     ///< Position of PHASE
MODE bit in @ref PWM_ModeTypeDef
#define PWM_ACTIVECHANNEL_MODE_Pos  (3)     ///< Position of ACTIVECHANNEL
bits in @ref PWM_ModeTypeDef

/**
    * @brief        Bit for Sine/DC Mode
    * @par          Modes:
    * @details  - 0 - PWM has Sine form duty,
    * @details  - 1 - PWM has DC duty
    */
#define PWM_DC_MODE                                  (1<<(PWM_DC_MODE_Pos))

/**
    * @brief        Bit for Bridge Mode
    * @par          Modes:
    * @details  - 0 - PWM generated only on one channel ( @ref
PWM_ACTIVECHANNEL_MODE)
    * @details  - 1 - PWM generated at two channels of each phase ( @ref
PWM_PHASE_MODE)
    */
#define PWM_BRIDGE_MODE                          (1<<(PWM_BRIDGE_MODE_Pos))

/**
    * @brief        Bit for 3-Phase Mode
    * @par          Modes:
    * @details  - 0 - PWM generated only at phase "A"
    * @details  - 1 - PWM generated at three phase "A", "B", "C"
    */
#define PWM_PHASE_MODE                           (1<<(PWM_PHASE_MODE_Pos))

/**
    * @brief        Bit for 3-Phase Mode
    * @par          Modes:
    * @details  - 0 - PWM doesnt generated
    * @details  - 1-6 - PWM generated on corresponting channel №
"PWM_ACTIVECHANNEL_MODE"
    */
#define PWM_ACTIVECHANNEL_MODE          (0x7<<(PWM_ACTIVECHANNEL_MODE_Pos)) //
0 - One channel PWM doesnt generated, 1 - PWM generated on corresponting
channel № "PWM_ACTIVECHANNEL_MODE"
/**
    * @brief        Mode: PWM generation disabled
    */
#define PWM_ACTIVECHANNEL_DISABLE       (0x0<<(PWM_ACTIVECHANNEL_MODE_Pos))
/**
    * @brief        Mode: PWM generated on channel 1 (Phase A channel 0)
    */
#define PWM_ACTIVECHANNEL_A0
(0x1<<(PWM_ACTIVECHANNEL_MODE_Pos))
```

```
/**
    * @brief        Mode: PWM generated on channel 2 (Phase A channel 1)
    */
#define PWM_ACTIVECHANNEL_A1
(0x2<<(PWM_ACTIVECHANNEL_MODE_Pos))
/**
    * @brief        Mode: PWM generated on channel 3 (Phase B channel 0)
    */
#define PWM_ACTIVECHANNEL_B0
(0x3<<(PWM_ACTIVECHANNEL_MODE_Pos))
/**
    * @brief        Mode: PWM generated on channel 4 (Phase B channel 1)
    */
#define PWM_ACTIVECHANNEL_B1
(0x4<<(PWM_ACTIVECHANNEL_MODE_Pos))
/**
    * @brief        Mode: PWM generated on channel 5 (Phase C channel 0)
    */
#define PWM_ACTIVECHANNEL_C0
(0x5<<(PWM_ACTIVECHANNEL_MODE_Pos))
/**
    * @brief        Mode: PWM generated on channel 6 (Phase C channel 1)
    */
#define PWM_ACTIVECHANNEL_C1
(0x6<<(PWM_ACTIVECHANNEL_MODE_Pos))

/**
    * @brief        Read current PWM mode from handle
    */
#define PWM_Get_Mode(_hpwm_, _mode_)     ((_hpwm_)->sPWM_Mode&(_mode_))

/**
    * @brief    Structure for PWM modes.
    */
typedef enum
{
    // SINE MODES
    PWM_SINE_SINGLE =       0,
/*!< set pwm duty from table with PWM_Sine_Hz Hz */
    PWM_SINE_BRIDGE =       PWM_BRIDGE_MODE,
/*!< set pwm duty from table with PWM_Sine_Hz Hz on two channels (positive and
negative halfes) */
    PWM_SINE_3PHASE =       PWM_BRIDGE_MODE|PWM_PHASE_MODE,
/*!< set pwm table duty on three phase with PWM_Sine_Hz Hz */
    // DC MODES
    PWM_DC_SINGLE =             PWM_DC_MODE,
/*!< set pwm duty PWM_Sine_Hz (in percent) on one channel ( @ref
PWM_ACTIVECHANNEL_MODE) */
    PWM_DC_BRIDGE =             PWM_DC_MODE|PWM_BRIDGE_MODE,
/*!< set pwm duty PWM_Duty (in percent) on two channel with PWM_Sine_Hz Hz */
    PWM_DC_3PHASE =             PWM_DC_MODE|PWM_BRIDGE_MODE|PWM_PHASE_MODE,
/*!< set pwm PWM_Duty duty on three phase with PWM_Sine_Hz Hz, with requested
shift */

    // Variables of signel channel modes
    PWM_SINE_SINGLE_A0 =    PWM_ACTIVECHANNEL_A0,
/*!< set sine pwm on Phase A channel 0 */
    PWM_SINE_SINGLE_A1 =    PWM_ACTIVECHANNEL_A1,
/*!< set sine pwm on Phase A channel 1 */
    PWM_SINE_SINGLE_B0 =    PWM_ACTIVECHANNEL_B0,
/*!< set sine pwm on Phase B channel 0 */
```

```
    PWM_SINE_SINGLE_B1 =    PWM_ACTIVECHANNEL_B1,
/*!< set sine pwm on Phase B channel 1 */
    PWM_SINE_SINGLE_C0 =    PWM_ACTIVECHANNEL_C0,
/*!< set sine pwm on Phase C channel 0 */
    PWM_SINE_SINGLE_C1 =    PWM_ACTIVECHANNEL_C1,
/*!< set sine pwm on Phase C channel 1 */
    PWM_SINE_BRIDGE_A0 =    PWM_DC_MODE|PWM_ACTIVECHANNEL_A0,
/*!< set DC pwm on Phase A channel 0) */
    PWM_SINE_BRIDGE_A1 =    PWM_DC_MODE|PWM_ACTIVECHANNEL_A1,
/*!< set DC pwm on Phase A channel 1) */
    PWM_SINE_BRIDGE_B0 =    PWM_DC_MODE|PWM_ACTIVECHANNEL_B0,
/*!< set DC pwm on Phase B channel 0) */
    PWM_SINE_BRIDGE_B1 =    PWM_DC_MODE|PWM_ACTIVECHANNEL_B1,
/*!< set DC pwm on Phase B channel 1) */
    PWM_SINE_BRIDGE_C0 =    PWM_DC_MODE|PWM_ACTIVECHANNEL_C0,
/*!< set DC pwm on Phase C channel 0) */
    PWM_SINE_BRIDGE_C1 =    PWM_DC_MODE|PWM_ACTIVECHANNEL_C1,
/*!< set DC pwm on Phase C channel 1) */
}PWM_ModeTypeDef;

/** PWM_MODE_DEFINES
  * @}
  */
//-------------------------------------------------------------------//


//-----------------------HANDLE STRUCTURES-------------------------//
/**
    * @addtogroup PWM_HANDLE_STRUCTURES
    * @ingroup      PWM
    * @brief            Structures for handling PWM
    @{
    */

/**
    * @brief    Structure for PWM modbus coils.
    */
typedef struct // PWM_ModeCoilsTypeDef
{
    unsigned                reserved:1;                     /*!< reserved bits
*/
    unsigned                DC:1;                                   /*!<
Sine/DC mode*/
    unsigned                BRIDGE:1;                           /*!< Bridge
mode */
    unsigned                PHASE:1;                            /*!< minimum
pulse duration for PWM in us*/
    unsigned                POLARITY:1;                     /*!< minimum pulse
duration for PWM in us*/
    unsigned                ACTIVECHANNEL:3;            /*!< minimum pulse
duration for PWM in us*/


}PWM_ModeCoilsTypeDef;

/**
    * @brief    Structure for PWM modbus registers.
    */
typedef struct // PWM_ModeRegsTypeDef
{
```

```
    uint16_t                                    Sine_Hz;
/*!< frequency of sine */
    uint16_t                                    Duty;
/*!< duty of dc pwm */
    uint16_t                                    TIMFreqHz;                  /*!<
frequency of pwm timer */
    uint16_t                                    MaxPulseDur;                /*!<
maximum pulse duration for PWM in us*/
    uint16_t                                    MinPulseDur;                /*!<
minimum pulse duration for PWM in us*/
    uint16_t                                    DeadTime;
/*!< dead-Time between switches half waves (channels) in us */
    uint16_t                                    TickBasePrescaler;  /*!< tickbase
prescaler for pwm timer */
    uint16_t                                    TableSize;                  /*!<
size of sine table */

}PWM_ModeRegsTypeDef;

/**
    * @brief    Structure for PWM config.
    */
typedef struct // PWM_ConfigTypeDef
{
    PWM_ModeCoilsTypeDef        *PWM_Mode;                  /*!< Mode of PWM:
modbus coils */
    PWM_ModeRegsTypeDef         *PWM_Settings;          /*!< Parameters of
PWM: modbus registers */
}PWM_ConfigTypeDef;

/**
    * @brief    Handle for ramp.
    */
typedef struct {
    /* Controller gains */
    arm_pid_instance_f32 pid;   /*!< Parameters of PID-regulator */

    /* Output limits */
    float limMin;                           /*!< Limit of PID-regulator: max
rate of increase */
    float limMax;                           /*!< Limit of PID-regulator: max
rate of decrease */

    /* Integrator limits */
    float limMinInt;                    /*!< Limit of integrator: max rate of
increase */
    float limMaxInt;                    /*!< Limit of integrator: max rate of
decrease */

    /* Sample time (in seconds) */
    float SampleT;                          /*!< Sample time of regulator (usualy
1/TIM_Hz that calls function of update PID-regulator) */
} PWM_RampHandleTypeDef;
#define PWM_PID_state_prevOut   1   ///< index of handle of ramp for PWM Sine
Hz
#define PWM_PID_state_Input         0   ///< index of handle of ramp for PWM
Sine Hz
#define PWM_PID_state_Output        2   ///< index of handle of ramp for PWM
Sine Hz
#define PWM_RampSineHz_Ind      0       ///< index of handle of ramp for PWM
Sine Hz
```

```
#define PWM_RampPWMDuty_Ind     1           ///< index of handle of ramp for
PWM Duty

/**
    * @brief    Handle for PWM.
    * @note     Prefixes: h - handle, s - settings, f - flag
    */
typedef struct // PWM_HandleTypeDef
{
    /* PWM VARIABLES */
    PWM_ConfigTypeDef           sPWM_Config;                    /*!< PWM
modbus config */
    PWM_ModeTypeDef                 sPWM_Mode;                      /*!< PWM
current mode */
    float                                   PWM_Sine_Hz;
/*!< Current frequency */
    float                                   PWM_Duty;
/*!< Current duty for DC bridge mode */
    uint16_t                                PWM_MaxPulseDur;            /*!<
Current maximum pulse duration for PWM in tickbase*/
    uint16_t                                PWM_MinPulseDur;            /*!<
Current minimum pulse duration for PWM in tickbase*/
    uint16_t                                PWM_DeadTime;
/*!< Current dead-Time between switches half waves (channels) in us */
    PWM_RampHandleTypeDef       hramp[2];                       /*!<
Handle for ramp */
    uint8_t                                 PWM_enHardware;             /*!<
Enable PWM output on transistor */

    /* SETTINGS FOR TIMER */
    TIM_SettingsTypeDef         stim;                           /*!<
Settings for TIM */
    TIM_OC_InitTypeDef          sConfigPWM;                     /*!< Settings
for PWM oc channel */
    TIM_OC_InitTypeDef          sConfigTrigger;             /*!< Settings for
Trigger oc channel */
    unsigned                                fActiveChannel;             /*!<
Flag for active oc channel: 0 - first channel, 1 - second channel */
    uint16_t                                sPWM_Channel1;              /*!<
Instance of first channel */
    uint16_t                                sPWM_Channel2;              /*!<
Instance of second channel */

    /* VARIABLES FOR TABLE DUTY PARAMETERS */
    uint32_t                                *pDuty_Table_Origin;    /*!<
Pointer to table of pwm duties */
    uint32_t                                Duty_Table_Size;            /*!<
Size of duty table */
    float                                   Duty_Table_Ind;             /*!<
Current ind of duty table */
    float                                   Duty_Table_Scale;           /*!<
Scale for TIM ARR register */

    /* SETTIGNS FOR PWM OUTPUT */
    GPIO_TypeDef                    *GPIOx;                         /*!<
GPIO port for PWM output */
    uint32_t                            GPIO_PIN_X1;                    /*!<
GPIO pin for PWM output */
    uint32_t                            GPIO_PIN_X2;                    /*!<
GPIO pin for PWM output (in bridge mode) */
```

```
    /* SLAVES PWM */
    void                                        *hpwm2;
/*!< Pointer to handle of slave PWM 2 */
    void                                        *hpwm3;
/*!< Pointer to handle of slave PWM 3 */

}PWM_HandleTypeDef;
extern PWM_HandleTypeDef hpwm1;                                  ///< Master
PWM handle

/**
    * @brief    Handle for Slave PWM.
    * @note     Prefixes: h - handle, s - settings, f - flag
    */
typedef struct // PWM_SlaveHandleTypeDef
{
    /* MASTER PWM*/
    PWM_HandleTypeDef           *hMasterPWM;                     /*!< master
pwm handle */

    /* SETTINGS FOR TIMER */
    TIM_SettingsTypeDef         stim;                                /*!<
slave tim handle */
    unsigned                                fActiveChannel;          /*!<
flag for active oc channel: 0 - first channel, 1 - second channel */
    uint16_t                                sPWM_Channel1;           /*!<
instance of first channel */
    uint16_t                                sPWM_Channel2;           /*!<
instance of second channel */

    /* VARIABLES FOR TABLE DUTY PARAMETERS */
    float                                   Duty_Shift_Ratio;        /*!<
Ratio of table shift: 0.5 shift - shift = Table_Size/2  */

    /* SETTIGNS FOR PWM OUTPUT */
    GPIO_TypeDef                    *GPIOx;                          /*!<
GPIO port for PWM output */
    uint32_t                            GPIO_PIN_X1;                 /*!<
GPIO pin for PWM output */
    uint32_t                            GPIO_PIN_X2;                 /*!<
GPIO pin for PWM output (second half wave) */
}PWM_SlaveHandleTypeDef;
extern PWM_SlaveHandleTypeDef hpwm2;                          ///< Slave PWM
handle
extern PWM_SlaveHandleTypeDef hpwm3;                          ///< Slave PWM
handle
/** PWM_HANDLE_STRUCTURES
  * @}
  */
//-------------------------------------------------------------------//

//--------------------------------PWM FUNCTIONS------------------------------
---
/**
    * @addtogroup PWM_FUNCTIONS
    * @ingroup      PWM
    * @brief            Function for control and configuring PWM
    */

// MASTER PWM FUNCTIONS
/**
```

```
    * @addtogroup PWM_MAIN_FUNCTIONS
    * @ingroup      PWM_FUNCTIONS
    * @brief            Function for controling PWM
    * @{
    */
/* PWM Handler */
void PWM_Handler(PWM_HandleTypeDef *hpwm);
/* Form PWM on Single Channel (DC or Sine PWM). Slave PWM disabled */
void PWM_SingleChannel_Mode(PWM_HandleTypeDef *hpwm, uint16_t rotate_ind_A,
uint8_t SineOrDC);
/* Form Bridge Sine PWM. Forming Slave PWM included */
void PWM_Sine_Bridge_Mode(PWM_HandleTypeDef *hpwm, uint16_t rotate_ind_A,
uint16_t rotate_ind_B, uint16_t rotate_ind_C);
/* Form Bridge DC PWM. Forming Slave PWM included */
void PWM_DC_Bridge_Mode(PWM_HandleTypeDef *hpwm, uint16_t rotate_ind_A,
uint16_t rotate_ind_B, uint16_t rotate_ind_C);
/* Getting ind for Duty Table */
uint32_t PWM_Get_Duty_Table_Ind(PWM_HandleTypeDef *hpwm, float FreqTIM);
/* Create Dead Time when switches channels */
void PWM_CreateDeadTime(PWM_HandleTypeDef *hpwm, float *LocalDeadTimeCnt,
unsigned *LocalActiveChannel);
/** PWM_MAIN_FUNCTIONS
  * @}
  */


// SLAVE PWM FUNCTIONS
/**
    * @addtogroup PWM_SLAVE_FUNCTIONS
    * @ingroup      PWM_FUNCTIONS
    * @brief            Function for controling slaves PWM
    * @{
    */
/* Set Duty from table on Slave PWM at one channel by sin_ind of the Master
PWM */
uint16_t PWM_SlavePhase_Calc_TableInd(PWM_SlaveHandleTypeDef *hspwm, uint16_t
sin_ind);
/* Check is all Slave channels works properly */
void PWM_SlavePhase_Check_Channels(PWM_SlaveHandleTypeDef *hspwm);
/* Create Dead Time for Slave PWM when switches channels */
void PWM_SlavePhase_CreateDeadTime(PWM_SlaveHandleTypeDef *hspwm, float
*LocalDeadTimeCnt, unsigned *LocalActiveChannel);
/** PWM_SLAVE_FUNCTIONS
  * @}
  */

/**
    * @addtogroup PWM_CONFIG_FUNCTIONS
    * @ingroup      PWM_FUNCTIONS
    * @brief            Function for initializing and re-configuring PWM
    * @{
    */
/* Update PWM parameters */
void PWM_Update_Params(PWM_HandleTypeDef *hpwm);
/* PID for ramp. */
void PWM_Ramp_ControlValue(PWM_RampHandleTypeDef *hramp, float *PID_Output,
float PID_Input);
/* Update PID parameters */
void PWM_Ramp_UpdateParams(PWM_RampHandleTypeDef *hramp, int32_t flagReset);
/* Reset PID for ramp */
void PWM_Ramp_Reset(PWM_RampHandleTypeDef *hramp);
```

```
/* Filling table with one period of sinus values */
uint32_t PWM_Fill_Sine_Table(PWM_HandleTypeDef *hpwm, uint32_t table_size);

//-----------------------------------init func-----------------------------------
---
/* First set up of PWM Two Channel */
void PWM_Sine_FirstInit(void);
void PWM_SynchInit(void);
/* reInitialization of PWM TIM */
void PWM_Sine_ReInit(PWM_HandleTypeDef *hpwm);
/* Initialization of Slave PWM TIM */
void PWM_SlavePhase_Init(PWM_SlaveHandleTypeDef *hspwm);
/* reInitialization of Slave PWM TIM */
void PWM_SlavePhase_reInit(PWM_SlaveHandleTypeDef *hspwm);
/* Calc and update new Duty Table Scale */
void PWM_Update_DutyTableScale(PWM_HandleTypeDef *hpwm);
/** PWM_CONFIG_FUNCTIONS
  * @}
  */

//-------------------------------------------------------------------//

#endif // __PWM_H_
```

ПРИЛОЖЕНИЕ В

Листинг interface.c

```
/**
    ******************************************************************************
    * @file     interface.c
    * @brief    Модуль для реализации интерфейса (енкодер и LCD дисплей)
    ******************************************************************************/
    #include "interface.h"
#include "pwm.h"

TIM_SettingsTypeDef TIM_ENCODER = {0};
Menu_TypeDef MENU;

extern I2C_HandleTypeDef hi2c1;
SPI_SettingsTypeDef sspi_disp;

extern PWM_HandleTypeDef hpwm1;


Interface_HandleTypeDef hinterface;

//-----------------ENCODER FUNCTIONS---------------//
/**
    * @brief    Update Encoder.
    * @param    hinterface - handle for interface.
    * @note     Encoder control menu of parameters and their values.
    * @note     This function call all functions below in section @ref
INTERFACE_ENCODER_FUNCTIONS.
    */
void Update_Encoder(Interface_HandleTypeDef *hinterface)
{
    // если энкодера не в режиме точной настройки)
    if(hinterface->FineChanging == 0)
    {
        // Нажатие кнопки
        if((~hinterface->henc.GPIOx->IDR)&hinterface->henc.GPIO_PIN_SW)
        {
            msDelay(50); // задержка для дребезга
            if((~hinterface->henc.GPIOx->IDR)&hinterface->henc.GPIO_PIN_SW) //
если кнопка действительно нажата
            {
                // Первый клик
                if(HAL_GetTick() - hinterface->Switch_prevTick > hinterface-
>DoubleClick_Timeout)   // слишком большая задержка после предыдущего нажатия
                {
                    Encoder_SingleClick(hinterface);
                }
                // Дабл клик
                else if(HAL_GetTick() - hinterface->Switch_prevTick <
hinterface->DoubleClick_Timeout)  // слишком маленькая задержка после
предыдущего нажатия
                {
                    Encoder_DoubleClick(hinterface);
                }
            }
        }
    }
    // если екодер в режиме точной настройки)
    if(hinterface->FineChanging == 1)
```

```
    {
        if(((~hinterface->henc.GPIOx->IDR)&hinterface->henc.GPIO_PIN_SW) == 0)
// ожидание отпускания кнопки
        {
            msDelay(50);
            if(((~hinterface->henc.GPIOx->IDR)&hinterface->henc.GPIO_PIN_SW)
== 0)               // если кнопка действительно отпущена
            {
                hinterface->FineChanging = 0;    // отключение режима точной
настройки
            }
        }
    }

    // запись считанных данных с енкодера
    Encoder_UpdateVarsFromEncoder(hinterface);
}

void Encoder_SingleClick(Interface_HandleTypeDef *hinterface)
{
    hinterface->Switch_prevTick = HAL_GetTick(); // сохраняем время нажатия
для регистрации долгого удержания кнопки если оно будет
    hinterface->Encoder_Shdw = hinterface->henc.htim->Instance->CNT;
    while((~hinterface->henc.GPIOx->IDR)&hinterface->henc.GPIO_PIN_SW)  //
ожидаем её отпускания
    {
        // если долгое удержание
        if(Encoder_HoldingSwitch(hinterface))
        {
            return; // выход из функции
        }
        // если кнопка не отпускается, и енкодер начинает регулироваться
        if(Encoder_HoldingAndRotateEncoder(hinterface))
        {
            return;  // выход из функции
        }
        msDelay(1);
    }
    // когда кнопка отпущена
    Encoder_SwitchRelease(hinterface);
}

void Encoder_DoubleClick(Interface_HandleTypeDef *hinterface)
{
    // переключение на другое меню
    hinterface->MenuNumber++;
    if(hinterface->MenuNumber > COILS_MENU)
    {
        hinterface->MenuNumber = 0;
    }
    hinterface->StartDispItem = 0;
    hinterface->CurrentSelection = NOTHING_SELECTED;
//  if (hinterface->MenuNumber == 0)
//      hinterface->CurrentSelection = NOTHING_SELECTED;
//  else if (hinterface->MenuNumber == 1)
//      hinterface->CurrentSelection = NOTHING_COIL_SELECTED;
}

/**
    * @brief    Holding Switch action.
    * @param    hinterface - handle for interface.
```

```
    * @note     This called when switch is holding for 5 second.
    */
uint8_t Encoder_HoldingSwitch(Interface_HandleTypeDef *hinterface)
{
    if(HAL_GetTick() - hinterface->Switch_prevTick > 5000)          // если
долго удержание - 5 секунд
    {
        #ifdef LCD_I2C // запрос на реинициализацю дисплея
        hinterface->hlcd.LCD_REINIT = 1;
        #endif
        return 1;
// 1 для выхода из цикла
    }
    return 0;
// 0 чтобы остаться в цикле
}

/**
    * @brief    Holding Switch and Rotate encoder action.
    * @param    hinterface - handle for interface.
    * @note     This called when switch is holding and encoder starts
rotating.
    */
uint8_t Encoder_HoldingAndRotateEncoder(Interface_HandleTypeDef *hinterface)
{
    if (hinterface->Encoder_Shdw != hinterface->henc.htim->Instance->CNT)   //
если кнопка не отпускается, и енкодер начинает регулироваться
    {
        hinterface->FineChanging = 1;
// переходим в режим точной настройки
        return 1;
// 1 для выхода из цикла
    }
    return 0;
// 0 чтобы остаться в цикле
}

/**
    * @brief    Release Switch action.
    * @param    hinterface - handle for interface.
    * @note     This called when switch is clicked .
    */
void Encoder_SwitchRelease(Interface_HandleTypeDef *hinterface)
{
    uint32_t *displayingitems;
    if(hinterface->MenuNumber == REGISTERS_MENU)
        displayingitems = &hinterface->Displaying.Regs.all;
    else if(hinterface->MenuNumber == COILS_MENU)
        displayingitems = &hinterface->Displaying.Coils.all;


    if(hinterface->FineChanging == 0)   // если энкодер не перешел в режим
точной настройки)
    {
        hinterface->Switch_prevTick = HAL_GetTick(); // сохраняем время
отжатия для регистрации даблклика если он будет
//                  hinterface->CurrentSelection++; // переключаем параметр
для настройки
        do{
                hinterface->CurrentSelection++;
                // проверка на конец списка меню
```

```
                if(hinterface->MenuNumber == REGISTERS_MENU)
                {   // в меню регистров переключаем до размера меню регистров
                    if(hinterface->CurrentSelection > Menu_Regs_Size)
                    {
                        // сброс обратно до меню
                        hinterface->CurrentSelection = NOTHING_SELECTED;
                        break;
                    }
                }
                else if(hinterface->MenuNumber == COILS_MENU) // COILS_MENU
                {   // в меню коилов переключаем до размера меню коилов
                    if(hinterface->CurrentSelection - NOTHING_COIL_SELECTED >
Menu_Coil_Size)
                    {
                        // сброс обратно до меню
                        hinterface->CurrentSelection = NOTHING_COIL_SELECTED;
                        break;
                    }
                }
        }
        // пропускаем все настройки меню, которые не отображаются н дисплее
        while((*displayingitems&(1<<(hinterface->CurrentSelection-1))) == 0);
    }
}

/**
    * @brief    Update params variables from encoder rotate.
    * @param    hinterface - handle for interface.
    * @note     This function call @ref Encoder_SetUint16Value.
    */
void Encoder_UpdateVarsFromEncoder(Interface_HandleTypeDef *hinterface)
{
    static uint16_t coef_value;
    if(hinterface->FineChanging == 0) // если енкодер не регулируется
        coef_value = 100;
    else
        coef_value = 2;

    hinterface->henc.Encoder_Diff = (int32_t)hinterface->henc.htim->Instance-
>CNT - hinterface->Encoder_Shdw;
    if(hinterface->FineChanging == 0)
    {

    }

    int32_t tmpreg;
    uint8_t scrolmenu_flag = 0;
    if(hinterface->henc.Encoder_Diff != 0)
    {
        if(hinterface->MenuNumber == REGISTERS_MENU)
        {
            switch(hinterface->CurrentSelection)
            {
                case PWM_SINE_HZ_SELECTED:
                    Encoder_SetUint16Value(&hinterface->henc,
&hpwm1.sPWM_Config.PWM_Settings->Sine_Hz, coef_value, 0, 0xFFFF);
                    break;

                case PWM_DUTY_SELECTED:
                    Encoder_SetUint16Value(&hinterface->henc,
&hpwm1.sPWM_Config.PWM_Settings->Duty, coef_value, 0, 10000);
```

```
                    break;

                case PWM_HZ_SELECTED:
                    Encoder_SetUint16Value(&hinterface->henc,
&hpwm1.sPWM_Config.PWM_Settings->TIMFreqHz, coef_value, 0, 0xFFFF);
                    break;

                case PWM_MAXPULSE_SELECTED:
                    // if max duration in equal min duration bot not zero: set
minimum as 0. for "if" statement later
                    if((hpwm1.sPWM_Config.PWM_Settings->MaxPulseDur <=
hpwm1.PWM_MinPulseDur) &&
                       (hpwm1.sPWM_Config.PWM_Settings->MaxPulseDur != 0) )
                       Encoder_SetUint16Value(&hinterface->henc,
&hpwm1.sPWM_Config.PWM_Settings->MaxPulseDur, ((coef_value>50)? coef_value/10
: coef_value), 0, 0xFFFF);
                    // if max duration in some other value: set minimum as min
duration
                    else
                        Encoder_SetUint16Value(&hinterface->henc,
&hpwm1.sPWM_Config.PWM_Settings->MaxPulseDur, ((coef_value>50)? coef_value/10
: coef_value), hpwm1.PWM_MinPulseDur, 0xFFFF);

                    // if max duration are less than min duration: set max
duration to defalut (zero)
                    if(hpwm1.sPWM_Config.PWM_Settings->MaxPulseDur <
hpwm1.PWM_MinPulseDur)
                       hpwm1.sPWM_Config.PWM_Settings->MaxPulseDur = 0; //
automcaticaly calc max duration corresponding to ARR register
                    break;

                case PWM_MINPULSE_SELECTED:
                    Encoder_SetUint16Value(&hinterface->henc,
&hpwm1.sPWM_Config.PWM_Settings->MinPulseDur, ((coef_value>50)? coef_value/10
: coef_value), 0, hpwm1.PWM_MaxPulseDur);
                    break;

                case PWM_DEADTIME_SELECTED:
                    Encoder_SetUint16Value(&hinterface->henc,
&hpwm1.sPWM_Config.PWM_Settings->DeadTime, coef_value, 0, 0xFFFF);
                    break;

                case PWM_TICKBASEPRESC_SELECTED:
                    Encoder_SetUint16Value(&hinterface->henc,
&hpwm1.sPWM_Config.PWM_Settings->TickBasePrescaler, 1, 0, 999);
                    break;

                case NOTHING_SELECTED:
                        if(hinterface->henc.Encoder_Diff > 0)
                            hinterface->StartDispItem += 1;
                        else if(hinterface->henc.Encoder_Diff < 0)
                            hinterface->StartDispItem -= 1;
                        if(hinterface->StartDispItem < 0)
                            hinterface->StartDispItem = 0;
                    break;

                default: scrolmenu_flag = 1; break;
            }
        }
        else if (hinterface->MenuNumber == COILS_MENU)
        {
```

```
            switch(hinterface->CurrentSelection)
            {
                case PWM_COIL_DC_SELECTED:
                    if(hinterface->henc.Encoder_Diff)
                        hpwm1.sPWM_Config.PWM_Mode->DC =
~hpwm1.sPWM_Config.PWM_Mode->DC;
                    break;


                case PWM_COIL_BRIDGE_SELECTED:
                    if(hinterface->henc.Encoder_Diff)
                        hpwm1.sPWM_Config.PWM_Mode->BRIDGE =
~hpwm1.sPWM_Config.PWM_Mode->BRIDGE;
                    break;

                case PWM_COIL_PHASE_SELECTED:
                    if(hinterface->henc.Encoder_Diff)
                        hpwm1.sPWM_Config.PWM_Mode->PHASE =
~hpwm1.sPWM_Config.PWM_Mode->PHASE;
                    break;


                case PWM_COIL_POLARITY_SELECTED:
                    if(hinterface->henc.Encoder_Diff)
                        hpwm1.sPWM_Config.PWM_Mode->POLARITY =
~hpwm1.sPWM_Config.PWM_Mode->POLARITY;
                    break;

                case PWM_COIL_ACTIVECHANNEL_SELECTED:
                    tmpreg = hpwm1.sPWM_Config.PWM_Mode->ACTIVECHANNEL;
                    tmpreg += (hinterface->henc.Encoder_Diff);
                    if (tmpreg < 0)
                        hpwm1.sPWM_Config.PWM_Mode->ACTIVECHANNEL = 0;
                    else if (tmpreg > 6)
                        hpwm1.sPWM_Config.PWM_Mode->ACTIVECHANNEL = 6;
                    else if (hinterface->henc.Encoder_Diff > 0)
                        hpwm1.sPWM_Config.PWM_Mode->ACTIVECHANNEL++;
                    else if (hinterface->henc.Encoder_Diff < 0)
                        hpwm1.sPWM_Config.PWM_Mode->ACTIVECHANNEL--;
                    break;

                default: scrolmenu_flag = 1; break;
            }
        }
        if(scrolmenu_flag)
        {
            if(hinterface->henc.Encoder_Diff > 0)
                hinterface->StartDispItem += 1;
            else if(hinterface->henc.Encoder_Diff < 0)
                hinterface->StartDispItem -= 1;
            if(hinterface->StartDispItem < 0)
                hinterface->StartDispItem = 0;
        }
    }
    hinterface->Encoder_Shdw = hinterface->henc.htim->Instance->CNT;
}

/**
    * @brief    Set uint16_t value from from encoder rotate.
    * @param    hinterface - handle for interface.
    * @note     Target value is ranged between min_value and max_value.
```

```
    */
void Encoder_SetUint16Value(TIM_EncoderTypeDef *henc, uint16_t *TargetValue,
uint16_t coef, int32_t min_value, int32_t max_value)
{
    int32_t tmpreg;
    tmpreg = *TargetValue;
    tmpreg += ((henc->Encoder_Diff*coef)/2);
    if (tmpreg < min_value)
        *TargetValue = min_value;
    else if (tmpreg > max_value)
        *TargetValue = max_value;
    else
        *TargetValue = tmpreg;
}

/**
    * @brief    Set uint8_t value from from encoder rotate.
    * @param    hinterface - handle for interface.
    * @note     @ref Encoder_SetUint16Value.
    */
void Encoder_SetUint8Value(TIM_EncoderTypeDef *henc, uint8_t *TargetValue,
uint16_t coef, int32_t min_value, int32_t max_value)
{
    int32_t tmpreg;
    tmpreg = *TargetValue;
    tmpreg += ((henc->Encoder_Diff*coef)/2);
    if (tmpreg < min_value)
        *TargetValue = min_value;
    else if (tmpreg > max_value)
        *TargetValue = max_value;
    else
        *TargetValue = tmpreg;
}



//---------------LCD DISPLAY FUNCTIONS--------------//
/**
    * @brief    Update Active Items in Menu Structure for displaying.
    * @param    hinterface - handle for interface.
    * @note     Update corresponding to current mode and stop displaying
unnecessary items.
    * @note     This called from StartMainTask
    */
void Update_ActiveMenuItems(Interface_HandleTypeDef *hinterface)
{
    if(hinterface->MenuNumber == REGISTERS_MENU)
    {
//      RegsMenu_HandleTypeDef *RegsMenu = (RegsMenu_HandleTypeDef
*)&hinterface->Menu.Regs;

        if(hpwm1.sPWM_Config.PWM_Mode->BRIDGE || (hpwm1.sPWM_Config.PWM_Mode-
>DC == 0))
            hinterface->Menu.Regs.Items.PWM_Sine_Hz = 1;
        else
            hinterface->Menu.Regs.Items.PWM_Sine_Hz = 0;

        if(hpwm1.sPWM_Config.PWM_Mode->DC)
            hinterface->Menu.Regs.Items.PWM_Duty = 1;
        else
            hinterface->Menu.Regs.Items.PWM_Duty = 0;
```

```
        hinterface->Menu.Regs.Items.PWM_Hz = 1;
        hinterface->Menu.Regs.Items.PWM_MaxPulseDur = 1;
        hinterface->Menu.Regs.Items.PWM_MinPulseDur = 1;

        if(hpwm1.sPWM_Config.PWM_Mode->BRIDGE)
            hinterface->Menu.Regs.Items.PWM_DeadTime = 1;
        else
            hinterface->Menu.Regs.Items.PWM_DeadTime = 0;

        hinterface->Menu.Regs.Items.PWM_TickBasePresc = 1;
    }
    else if(hinterface->MenuNumber == COILS_MENU)
    {
//      CoilsMenu_HandleTypeDef *CoilsMenu = (CoilsMenu_HandleTypeDef
*)&hinterface->Menu[COILS_MENU];

        hinterface->Menu.Coils.Items.PWM_DC = 1;
        hinterface->Menu.Coils.Items.PWM_BRIDGE = 1;

        if(hpwm1.sPWM_Config.PWM_Mode->BRIDGE)
        {
            hinterface->Menu.Coils.Items.PWM_PHASE = 1;
            hinterface->Menu.Coils.Items.PWM_ACTIVECHANNEL = 0;
        }
        else
        {
            hinterface->Menu.Coils.Items.PWM_PHASE = 0;
            hinterface->Menu.Coils.Items.PWM_ACTIVECHANNEL = 1;
        }

        hinterface->Menu.Coils.Items.PWM_POLARITY = 1;
    }
}

/**
    * @brief    Update LCD display.
    * @param    hinterface - handle for interface.
    * @note     LCD displaying parameters of PWM.
    * @note     This call all functions below in section @ref LCD DISPLAY
FUNCTIONS.
    */
void Update_LCDDisplay(Interface_HandleTypeDef *hinterface)
{

    // clear display
    #ifdef LCD_I2C
    LCD_Check(&hinterface->hlcd);
    LCD_Send_CMD(&hinterface->hlcd, 0x01);
    msDelay(hinterface->LCD_ClearDelay);
    #endif
    #ifdef LCD_SPI
    SPIDisp_Clean_Buffer_Frame();
    #endif

    hinterface->disp_cnt = 0 - hinterface->StartDispItem;

    if(hinterface->MenuNumber == 0)
        hinterface->disp_cnt = LCD_DisplayMenuRegisters(hinterface);
    else
        hinterface->disp_cnt = LCD_DisplayMenuCoils(hinterface);
```

```
  vTaskDelayUntil(&hinterface->LCD_prevUpdateTick, hinterface-
>LCD_UpdateDelay);
    #ifdef LCD_SPI
    // update frame
    SPIDisp_Update();
    #endif

    // if more items can be showed item displayed - decrease start menu item
    if((hinterface->StartDispItem > 0) && (hinterface->disp_cnt < hinterface-
>hlcd.LCD_Rows))
    {
            hinterface->StartDispItem -= 1;
    }

}



/**
    * @brief    Display all coils menu at LCD Display.
    * @param    hinterface  - handle for interface.
    * @return   disp_cnt        - how many string is displayed.
    * @note     This call LCD_PrintString and LCD_UpdateString.
    */
uint8_t LCD_DisplayMenuCoils(Interface_HandleTypeDef *hinterface)
{
    uint8_t disp_flag = 0;
    uint8_t line = 0;
    for(int i = 0; i < Menu_Coil_Size; i++)
    {
        switch(hinterface->Menu.Coils.all&(1<<i))
        {
            // delete item from displaying if it isnt displaying
            case 0: disp_flag = 0; break;

            case Menu_PWM_DC:
                disp_flag = LCD_PrintString(hinterface, PWM_COIL_DC_SELECTED,
Print_PWM_DC);
                break;


            case Menu_PWM_BRIDGE:
                disp_flag = LCD_PrintString(hinterface,
PWM_COIL_BRIDGE_SELECTED, Print_PWM_BRIDGE);
                break;


            case Menu_PWM_PHASE:
                disp_flag = LCD_PrintString(hinterface,
PWM_COIL_PHASE_SELECTED, Print_PWM_PHASE);
                break;


            case Menu_PWM_POLARITY:
                disp_flag = LCD_PrintString(hinterface,
PWM_COIL_POLARITY_SELECTED, Print_PWM_POLARITY);
                break;


            case Menu_PWM_ACTIVECHANNEL:
```

```
                disp_flag = LCD_PrintString(hinterface,
PWM_COIL_ACTIVECHANNEL_SELECTED, Print_PWM_ACTIVECHANNEL);
                break;
        }
        disp_flag = LCD_UpdateString(hinterface, disp_flag);    // update
string

        // update displaying items srtuct
        if(disp_flag)
            hinterface->Displaying.Coils.all |= (1<<i);
        else
            hinterface->Displaying.Coils.all &= ~(1<<i);

    }

    return hinterface->disp_cnt;
}

/**
    * @brief    Display all registers menu at LCD Display.
    * @param    hinterface  - handle for interface.
    * @return   disp_cnt        - how many string is displayed.
    * @note     This call LCD_PrintString and LCD_UpdateString.
    */
uint8_t LCD_DisplayMenuRegisters(Interface_HandleTypeDef *hinterface)
{
    uint8_t disp_flag = 0;
    uint8_t line = 0;
    for(int i = 0; i < Menu_Regs_Size; i++)
    {
        switch(hinterface->Menu.Regs.all&(1<<i))
        {
            // delete item from displaying if it isnt displaying
            case 0: disp_flag = 0; break;

            case Menu_PWM_Sine_Hz:
                disp_flag = LCD_PrintString(hinterface, PWM_SINE_HZ_SELECTED,
Print_PWM_Sine_Hz);
                break;

            case Menu_PWM_Duty:
                disp_flag = LCD_PrintString(hinterface, PWM_DUTY_SELECTED,
Print_PWM_Duty);
                break;

            case Menu_PWM_Hz:
                disp_flag = LCD_PrintString(hinterface, PWM_HZ_SELECTED,
Print_PWM_Hz);
                break;

            case Menu_PWM_MaxPulseDur:
                disp_flag = LCD_PrintString(hinterface, PWM_MAXPULSE_SELECTED,
Print_PWM_MaxPulseDur);
                break;

            case Menu_PWM_MinPulseDur:
                disp_flag = LCD_PrintString(hinterface, PWM_MINPULSE_SELECTED,
Print_PWM_MinPulseDur);
                break;

            case Menu_PWM_DeadTime:
```

```
                disp_flag = LCD_PrintString(hinterface, PWM_DEADTIME_SELECTED,
Print_PWM_DeadTime);
                break;

            case Menu_PWM_TickBasePresc:
                disp_flag = LCD_PrintString(hinterface,
PWM_TICKBASEPRESC_SELECTED, Print_PWM_TickBasePresc);
                break;
        }
        disp_flag = LCD_UpdateString(hinterface, disp_flag);    // update
string

        // update displaying items srtuct
        if(disp_flag)
            hinterface->Displaying.Regs.all |= (1<<(i+NOTHING_SELECTED));
        else
            hinterface->Displaying.Regs.all &= ~(1<<(i+NOTHING_SELECTED));
    }
    return hinterface->disp_cnt;
}


/**
    * @brief    Print string with menu item.
    * @param    hinterface          - handle for interface.
    * @param    SELECTED_DEFINE - define for choosing print string as
currently selected, or as regular.
    * @param    Print_Function  - function to print string with item to char
array @ref Print_PWM_Sine_Hz.
    * @return   disp_flag       - flag is string displaying or not.
    * @note     This call @ref Print_PWM_Sine_Hz or smth function like that.
    */
uint8_t LCD_PrintString(Interface_HandleTypeDef *hinterface, unsigned
SELECTED_DEFINE, void (*Print_Function)(Interface_HandleTypeDef *, uint8_t))
{
    uint8_t disp_flag = 0;
    hinterface->disp_cnt += 1;

    if(hinterface->disp_cnt > 0)
    {
        disp_flag = 1;
        if (hinterface->CurrentSelection == SELECTED_DEFINE)
            Print_Function(hinterface, 1);
        else
            Print_Function(hinterface, 0);
    }
    else
    {
        disp_flag = 0;
    }
    return disp_flag;
}

/**
    * @brief    Update string.
    * @param    hinterface  - handle for interface.
    * @param    disp_flag   - flag to display string or not.
    * @return   disp_flag       - flag is string displaying or not.
    * @note     For I2C display: Display string.
    * @note     For SPI display: print in frame buffer. Display update happens
in Update_LCDDisplay.
```

```
    */
uint8_t LCD_UpdateString(Interface_HandleTypeDef *hinterface, uint8_t
disp_flag)
{
    // if there something to display
    if(hinterface->disp_cnt <= hinterface->hlcd.LCD_Rows)
    {
        if(disp_flag)
        {
            #ifdef LCD_I2C
            LCD_SetString(&hinterface->hlcd, hinterface->disp_cnt-1);
            LCD_Send_STRING(&hinterface->hlcd, hinterface->LCD_String);
            #endif

            #ifdef LCD_SPI
            static uint8_t line = 0;
            // reset line to zero if its first item displaying
            if (hinterface->disp_cnt == 1)
                line = 0;
            SPIDisp_Output_String(0, line, hinterface->LCD_String, 0, 0);
            line += 10;
            #endif
        }
    }
    else
    {
        disp_flag = 0;
    }
    return disp_flag;
}

//------------------PRINT FUNCTIONS-----------------//
void Print_PWM_Sine_Hz(Interface_HandleTypeDef *hinterface, uint8_t
selected_flag)
{
    if(selected_flag)
        snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1, ">Sine:
%d.%02d Hz", pwm_ctrl[R_PWM_CTRL_PWM_SINE_HZ]/100,
pwm_ctrl[R_PWM_CTRL_PWM_SINE_HZ]%100);
    else
        snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1, "Sine:
%d.%02d Hz", pwm_ctrl[R_PWM_CTRL_PWM_SINE_HZ]/100,
pwm_ctrl[R_PWM_CTRL_PWM_SINE_HZ]%100);
}

void Print_PWM_Duty(Interface_HandleTypeDef *hinterface, uint8_t
selected_flag)
{
    if(selected_flag)
        snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1, ">Duty:
%d.%02d%%", pwm_ctrl[R_PWM_CTRL_DUTY_BRIDGE]/100,
pwm_ctrl[R_PWM_CTRL_DUTY_BRIDGE]%100);
    else
        snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1, "Duty:
%d.%02d%%", pwm_ctrl[R_PWM_CTRL_DUTY_BRIDGE]/100,
pwm_ctrl[R_PWM_CTRL_DUTY_BRIDGE]%100);
}
void Print_PWM_Hz(Interface_HandleTypeDef *hinterface, uint8_t selected_flag)
{
    if(selected_flag)
```

```
        snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1, ">PWM:
%d Hz", pwm_ctrl[R_PWM_CTRL_PWM_HZ]);
    else
        snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1, "PWM:
%d Hz", pwm_ctrl[R_PWM_CTRL_PWM_HZ]);
}

void Print_PWM_MaxPulseDur(Interface_HandleTypeDef *hinterface, uint8_t
selected_flag)
{
    #ifdef LCD_I2C
    if(hpwm1.stim.sTickBasePrescaler > 1)
    {
        if(selected_flag)
            snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
">Max: %d us (%d)", hpwm1.PWM_MaxPulseDur/hpwm1.stim.sTickBasePrescaler,
hpwm1.PWM_MaxPulseDur);
        else
            snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
"Max: %d us (%d)", hpwm1.PWM_MaxPulseDur/hpwm1.stim.sTickBasePrescaler,
hpwm1.PWM_MaxPulseDur);
    }
    else
    {
        if(selected_flag)
            snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
">Max: %d us", hpwm1.PWM_MaxPulseDur);
        else
            snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
"Max: %d us", hpwm1.PWM_MaxPulseDur);
    }
    #endif
    #ifdef LCD_SPI
    if(hpwm1.stim.sTickBasePrescaler > 1)
    {
        if(selected_flag)
            snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
">Max dur: %d us (%d)",
(uint32_t)hpwm1.PWM_MaxPulseDur/hpwm1.stim.sTickBasePrescaler,
hpwm1.PWM_MaxPulseDur);
        else
            snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
"Max dur: %d us (%d)",
(uint32_t)hpwm1.PWM_MaxPulseDur/hpwm1.stim.sTickBasePrescaler,
hpwm1.PWM_MaxPulseDur);
    }
//  else if(hpwm1.stim.sTickBasePrescaler > 10)
//  {
//      if(selected_flag)
//          snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
">Max dur: %d 10ns (%d)",
(uint32_t)hpwm1.PWM_MaxPulseDur*100/hpwm1.stim.sTickBasePrescaler,
hpwm1.PWM_MaxPulseDur);
//      else
//          snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
"Max dur: %d 10ns (%d)",
(uint32_t)hpwm1.PWM_MaxPulseDur*100/hpwm1.stim.sTickBasePrescaler,
hpwm1.PWM_MaxPulseDur);
//  }
//  else if(hpwm1.stim.sTickBasePrescaler > 1)
//  {
```

```
//      if(selected_flag)
//          snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
">Max dur: %d 100ns (%d)",
(uint32_t)hpwm1.PWM_MaxPulseDur*10/hpwm1.stim.sTickBasePrescaler,
hpwm1.PWM_MaxPulseDur);
//      else
//          snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
"Max dur: %d 100ns (%d)",
(uint32_t)hpwm1.PWM_MaxPulseDur*10/hpwm1.stim.sTickBasePrescaler,
hpwm1.PWM_MaxPulseDur);
//  }
    else
    {
        if(selected_flag)
            snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
">Max dur: %d us", hpwm1.PWM_MaxPulseDur);
        else
            snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
"Max dur: %d us", hpwm1.PWM_MaxPulseDur);
    }
    #endif
}
void Print_PWM_MinPulseDur(Interface_HandleTypeDef *hinterface, uint8_t
selected_flag)
{
    #ifdef LCD_I2C
    if(hpwm1.stim.sTickBasePrescaler > 1)
    {
        if(selected_flag)
            snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
">Min: %d ns",
(uint32_t)hpwm1.PWM_MinPulseDur*1000/hpwm1.stim.sTickBasePrescaler);
        else
            snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
"Min: %d ns",
(uint32_t)hpwm1.PWM_MinPulseDur*1000/hpwm1.stim.sTickBasePrescaler);
    }
    else
    {
        if(selected_flag)
            snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
">Min: %d us", hpwm1.PWM_MinPulseDur);
        else
            snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
"Min: %d us", hpwm1.PWM_MinPulseDur);
    }
    #endif

    #ifdef LCD_SPI
    if(hpwm1.stim.sTickBasePrescaler > 1)
    {
        if(selected_flag)
            snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
">Min dur: %d us (%d)",
(uint32_t)hpwm1.PWM_MinPulseDur/hpwm1.stim.sTickBasePrescaler,
hpwm1.PWM_MinPulseDur);
        else
            snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
"Min dur: %d us (%d)",
(uint32_t)hpwm1.PWM_MinPulseDur/hpwm1.stim.sTickBasePrescaler,
hpwm1.PWM_MinPulseDur);
```

```
    }
//  else if(hpwm1.stim.sTickBasePrescaler > 10)
//  {
//      if(selected_flag)
//          snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
">Min dur: %d 10ns (%d)",
(uint32_t)hpwm1.PWM_MinPulseDur*100/hpwm1.stim.sTickBasePrescaler,
hpwm1.PWM_MinPulseDur);
//      else
//          snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
"Min dur: %d 10ns (%d)",
(uint32_t)hpwm1.PWM_MinPulseDur*100/hpwm1.stim.sTickBasePrescaler,
hpwm1.PWM_MinPulseDur);
//  }
//  else if(hpwm1.stim.sTickBasePrescaler > 1)
//  {
//      if(selected_flag)
//          snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
">Min dur: %d 100ns (%d)",
(uint32_t)hpwm1.PWM_MinPulseDur*10/hpwm1.stim.sTickBasePrescaler,
hpwm1.PWM_MinPulseDur);
//      else
//          snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
"Min dur: %d 100ns (%d)",
(uint32_t)hpwm1.PWM_MinPulseDur*10/hpwm1.stim.sTickBasePrescaler,
hpwm1.PWM_MinPulseDur);
//  }
    else
    {
        if(selected_flag)
            snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
">Min dur: %d us", hpwm1.PWM_MinPulseDur);
        else
            snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
"Min dur: %d us", hpwm1.PWM_MinPulseDur);
    }
    #endif
}


void Print_PWM_DeadTime(Interface_HandleTypeDef *hinterface, uint8_t
selected_flag)
{
    #ifdef LCD_I2C
    if(selected_flag)
        snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
">DeadT: %d us", pwm_ctrl[R_PWM_CTRL_DEAD_TIME]);
    else
        snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1, "DeadT:
%d us", pwm_ctrl[R_PWM_CTRL_DEAD_TIME]);
    #endif

    #ifdef LCD_SPI
    if(selected_flag)
        snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1, ">Dead
Time: %d us", pwm_ctrl[R_PWM_CTRL_DEAD_TIME]);
    else
        snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1, "Dead
Time: %d us", pwm_ctrl[R_PWM_CTRL_DEAD_TIME]);
    #endif
```

```
}


void Print_PWM_TickBasePresc(Interface_HandleTypeDef *hinterface, uint8_t
selected_flag)
{
    #ifdef LCD_I2C
    if(selected_flag)
        snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1, ">Tick
Presc: %d", pwm_ctrl[R_PWM_CTRL_TICKBASE_PRESCALER]);
    else
        snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1, "Tick
Presc: %d", pwm_ctrl[R_PWM_CTRL_TICKBASE_PRESCALER]);
    #endif

    #ifdef LCD_SPI
    if(selected_flag)
        snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
">TickBase Prescaler: %d", pwm_ctrl[R_PWM_CTRL_TICKBASE_PRESCALER]);
    else
        snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
"TickBase Prescaler: %d", pwm_ctrl[R_PWM_CTRL_TICKBASE_PRESCALER]);
    #endif

}



void Print_PWM_DC(Interface_HandleTypeDef *hinterface, uint8_t selected_flag)
{
    if(selected_flag)
    {
        if(hpwm1.sPWM_Config.PWM_Mode->DC)
            snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
">Mode: DC");
        else
            snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
">Mode: Sine");
    }
    else
    {
        if(hpwm1.sPWM_Config.PWM_Mode->DC)
            snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
"Mode: DC");
        else
            snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
"Mode: Sine");
    }
}


void Print_PWM_BRIDGE(Interface_HandleTypeDef *hinterface, uint8_t
selected_flag)
{
    if(selected_flag)
    {
        if(hpwm1.sPWM_Config.PWM_Mode->BRIDGE)
            snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
">Bridge: Bridge");
        else
```

```
            snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
">Bridge: Single");
    }
    else
    {
        if(hpwm1.sPWM_Config.PWM_Mode->BRIDGE)
            snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
"Bridge: Bridge");
        else
            snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
"Bridge: Single");
    }
}


void Print_PWM_PHASE(Interface_HandleTypeDef *hinterface, uint8_t
selected_flag)
{
    if(selected_flag)
    {
        if(hpwm1.sPWM_Config.PWM_Mode->PHASE)
            snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
">Phase: 3-Phase");
        else
            snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
">Phase: One Phase");
    }
    else
    {
        if(hpwm1.sPWM_Config.PWM_Mode->PHASE)
            snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
"Phase: 3-Phase");
        else
            snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
"Phase: One Phase");
    }
}

void Print_PWM_POLARITY(Interface_HandleTypeDef *hinterface, uint8_t
selected_flag)
{
    if(selected_flag)
    {
        if(hpwm1.sPWM_Config.PWM_Mode->POLARITY)
            snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
">Polarity: Neg");
        else
            snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
">Polarity: Pos");
    }
    else
    {
        if(hpwm1.sPWM_Config.PWM_Mode->POLARITY)
            snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
"Polarity: Neg");
        else
            snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1,
"Polarity: Pos");
    }
}
```

```
void Print_PWM_ACTIVECHANNEL(Interface_HandleTypeDef *hinterface, uint8_t
selected_flag)
{
    if(selected_flag)
        snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1, ">PWM
channel: %d", hpwm1.sPWM_Config.PWM_Mode->ACTIVECHANNEL);
    else
        snprintf(hinterface->LCD_String, hinterface->hlcd.LCD_Width+1, "PWM
channel: %d", hpwm1.sPWM_Config.PWM_Mode->ACTIVECHANNEL);
}
/**
    * @brief    First initialization of interface struct and necessary
peripheral.
    * @note     This called from StartMainTask
    */
void EncoderFirstInit(void)
{
    hinterface.DoubleClick_Timeout = 200;
        // tim settings
    TIM_ENCODER.htim.Instance = TIMER_ENCODER_INSTANCE;
    TIM_ENCODER.htim.Init.Period = 0xFFFF;
    TIM_ENCODER.htim.Init.Prescaler = 0;
    TIM_ENCODER.sTimMode = TIM_IT_MODE;
//  TIM_ENCODER.sTickBaseUS = TIMER_ENCODER_TICKBASE;
//  TIM_ENCODER.sTimAHBFreqMHz = TIMER_ENCODER_AHB_FREQ;
//  TIM_ENCODER.sTimFreqHz = 0;

    TIM_Base_Init(&TIM_ENCODER);

    hinterface.henc.sConfig.EncoderMode = TIM_ENCODERMODE_TI2;
  hinterface.henc.sConfig.IC1Polarity = TIM_INPUTCHANNELPOLARITY_FALLING;
  hinterface.henc.sConfig.IC1Selection = TIM_ICSELECTION_DIRECTTI;
  hinterface.henc.sConfig.IC1Prescaler = TIM_ICPSC_DIV1;
  hinterface.henc.sConfig.IC1Filter = 5;
  hinterface.henc.sConfig.IC2Polarity = TIM_INPUTCHANNELPOLARITY_FALLING;
  hinterface.henc.sConfig.IC2Selection = TIM_ICSELECTION_DIRECTTI;
  hinterface.henc.sConfig.IC2Prescaler = TIM_ICPSC_DIV1;
  hinterface.henc.sConfig.IC2Filter = 5;
    hinterface.henc.GPIOx                = TIMER_ENCODER_PORT;
    hinterface.henc.GPIO_PIN_TI1    = TIMER_ENCODER_PIN1;
    hinterface.henc.GPIO_PIN_TI2    = TIMER_ENCODER_PIN2;
    hinterface.henc.GPIO_PIN_SW = TIMER_ENCODER_PIN_SW;
    TIM_Encoder_Init(&hinterface.henc, &TIM_ENCODER.htim);

    HAL_TIM_Base_Start(&TIM_ENCODER.htim);
}


/**
    * @brief    First initialization of SPI LCD.
    * @note     This called from Interface_FirstInit
    */
void SPILCD_FirstInit(void)
{
  sspi_disp.hspi.Instance = SPI1;
  sspi_disp.hspi.Init.Mode = SPI_MODE_MASTER;
  sspi_disp.hspi.Init.Direction = SPI_DIRECTION_2LINES;
  sspi_disp.hspi.Init.DataSize = SPI_DATASIZE_8BIT;
  sspi_disp.hspi.Init.CLKPolarity = SPI_POLARITY_LOW;
  sspi_disp.hspi.Init.CLKPhase = SPI_PHASE_1EDGE;
  sspi_disp.hspi.Init.NSS = SPI_NSS_SOFT;
```

```
  sspi_disp.hspi.Init.BaudRatePrescaler = SPI_BAUDRATEPRESCALER_4;
  sspi_disp.hspi.Init.FirstBit = SPI_FIRSTBIT_MSB;
  sspi_disp.hspi.Init.TIMode = SPI_TIMODE_DISABLE;
  sspi_disp.hspi.Init.CRCCalculation = SPI_CRCCALCULATION_DISABLE;

    sspi_disp.CLK_GPIOx = GPIOA;
    sspi_disp.CLK_PIN = GPIO_PIN_5;
    sspi_disp.MISO_GPIOx = GPIOA;
    sspi_disp.MISO_PIN = GPIO_PIN_6;
    sspi_disp.MOSI_GPIOx = GPIOA;
    sspi_disp.MOSI_PIN = GPIO_PIN_7;

    SPI_Base_Init(&sspi_disp);
    SPIDisp_Init();
}


/**
    * @brief    First initialization of LCD (SPI/I2C corresponding to define).
    * @note     This called from Interface_FirstInit
    */
void LCD_FirstInit(void)
{
    hinterface.LCD_prevUpdateTick = xTaskGetTickCount();

    #ifdef LCD_I2C
    hinterface.hlcd.hi2c = &hi2c1;
    hinterface.LCD_ClearDelay = 10;
    hinterface.LCD_UpdateDelay = 250;
    LCD_Init(&hinterface.hlcd, NumbOfLines, NumbOfSymb);
    #endif

    #ifdef LCD_SPI
    hinterface.hlcd.hspi = &sspi_disp.hspi;
    hinterface.hlcd.LCD_Width = NumbOfSymb;
    hinterface.hlcd.LCD_Rows = NumbOfLines;
    hinterface.LCD_UpdateDelay = 100;
    SPILCD_FirstInit();
    #endif
}
/**
    * @brief    First initialization of Encoder Timer.
    * @note     This called from Interface_FirstInit
    */
void Interface_FirstInit(void)
{
    LCD_FirstInit();
    EncoderFirstInit();
}
```

ПРИЛОЖЕНИЕ Г

Листинг interface.h

```
/**
    ******************************************************************************
    * @file     interface.h
    * @brief    Заголовочный файл модуля реализации интерфейса (енкодер и LCD
дисплей).
    *
    * @defgroup INTERFACE
    * @brief        Interface stuff
    *
    ******************************************************************************/
#ifndef __INTERFACE_H_
#define __INTERFACE_H_

#include "periph_general.h"
#include "i2c_lcd.h"
#include "gmg12864_r9ofg.h"

//#define LCD_I2C
#define LCD_SPI

#ifdef LCD_I2C
    #define NumbOfSymb  16
    #define NumbOfLines 2
    #define LCD_HandleTypeDef       LCDI2C_HandleTypeDef
#endif
#ifdef LCD_SPI
    #define NumbOfSymb  30
    #define NumbOfLines 6
    #define LCD_HandleTypeDef       LCDSPI_HandleTypeDef
#endif

/**
    * @addtogroup INTERFACE_MENU_DEFINES
    * @ingroup      INTERFACE
    * @brief            Defines for menu
    * @{
    */

/**
    * @brief Struct for "selected" values for each coils for PWM.
    */
typedef enum
{
    NOTHING_SELECTED,                           /*!< nothing selected to
change */
    PWM_SINE_HZ_SELECTED,                   /*!< SINE HZ selected to change */
    PWM_DUTY_SELECTED,                      /*!< DUTY selected to change */
    PWM_HZ_SELECTED,                            /*!< PWM HZ selected to change
*/
    PWM_MAXPULSE_SELECTED,              /*!< MAX PULSE selected to change */
    PWM_MINPULSE_SELECTED,              /*!< MIN PULSE DC selected to change
*/
    PWM_DEADTIME_SELECTED,              /*!< DEADTIME selected to change */
    PWM_TICKBASEPRESC_SELECTED,     /*!< TICKBASEPRESCALER selected to change
*/
}CurrentSelectionRegs_TypeDef;
```

```
/**
    * @brief Struct for "selected" values for each coils for PWM.
    */
typedef enum
{
    NOTHING_COIL_SELECTED,                      /*!< nothing selected to
change */
    PWM_COIL_DC_SELECTED,                           /*!< COIL DC selected to
change */
    PWM_COIL_BRIDGE_SELECTED,               /*!< COIL BRIDGE selected to
change */
    PWM_COIL_PHASE_SELECTED,                    /*!< COIL PHASE selected to
change */
    PWM_COIL_POLARITY_SELECTED,             /*!< COIL POLARITY selected to
change */
    PWM_COIL_ACTIVECHANNEL_SELECTED,    /*!< COIL ACTIVECHANNEL selected to
change */
}CurrentSelectionCoil_TypeDef;

/**
    * @brief Struct with indexes for each menu.
    */
typedef enum
{
    REGISTERS_MENU = 0,     /*!< Registers menu is choosed */
    COILS_MENU,                     /*!< Coils menu is choosed */
}MenuIndex_TypeDef;


/**
    * @brief        Struct for registers menu of settings for PWM.
    * @details  If item is showed - bit is setted. If item isnt showed - bit
is reseted
    */
typedef union
{
    uint32_t all; /*!< Value to access all bits at once */
    /**
        * @brief        Struct of PWM params with separated bits
        */
    struct
    {
    unsigned                                            PWM_Sine_Hz:1;
    unsigned                                            PWM_Duty:1;
    unsigned                                            PWM_Hz:1;
    unsigned                                            PWM_MaxPulseDur:1;
    unsigned                                            PWM_MinPulseDur:1;
    unsigned                                            PWM_DeadTime:1;
    unsigned                                            PWM_TickBasePresc:1;
    }Items;
}RegsMenu_HandleTypeDef;
/**
    * @brief        Struct for coils menu of settings for PWM.
    * @details  If item is showed - bit is setted. If item isnt showed - bit
is reseted
    */
typedef union
{
    uint32_t all; /*!< Value to access all bits at once */
    /**
        * @brief        Struct of PWM params with separated bits
```

```
        */
    struct
    {
    unsigned                                            PWM_DC:1;
    unsigned                                            PWM_BRIDGE:1;
    unsigned                                            PWM_PHASE:1;
    unsigned                                            PWM_POLARITY:1;
    unsigned                                            PWM_ACTIVECHANNEL:1;
    }Items;
}CoilsMenu_HandleTypeDef;

/**
    * @brief        Struct with all settings for PWM: coils and registers.
    */
typedef struct
{
    RegsMenu_HandleTypeDef      Regs;       /*!< PWM parametrs in modbus
registers */
    CoilsMenu_HandleTypeDef     Coils;  /*!< PWM parametrs in modbus coils */
}Menu_TypeDef;

#define Menu_Regs_Size      7                               ///< size of @ref
RegsMenu_HandleTypeDef
#define Menu_Coil_Size      5                               ///< size of @ref
RegsMenu_HandleTypeDef

#define Menu_PWM_Sine_Hz                (1<<0)      ///< position of
PWM_Sine_Hz in              @ref RegsMenu_HandleTypeDef
#define Menu_PWM_Duty                       (1<<1)      ///< position of
PWM_Duty in                     @ref RegsMenu_HandleTypeDef
#define Menu_PWM_Hz                         (1<<2)      ///< position of
PWM_Hz in                       @ref RegsMenu_HandleTypeDef
#define Menu_PWM_MaxPulseDur        (1<<3)      ///< position of
PWM_MaxPulseDur in      @ref RegsMenu_HandleTypeDef
#define Menu_PWM_MinPulseDur        (1<<4)      ///< position of
PWM_MinPulseDur in      @ref RegsMenu_HandleTypeDef
#define Menu_PWM_DeadTime               (1<<5)      ///< position of
PWM_DeadTime in             @ref RegsMenu_HandleTypeDef
#define Menu_PWM_TickBasePresc  (1<<6)      ///< position of PWM_TickBasePresc
in @ref RegsMenu_HandleTypeDef

#define Menu_PWM_DC                         (1<<0)      ///< position of
PWM_DC in                       @ref CoilsMenu_HandleTypeDef
#define Menu_PWM_BRIDGE                 (1<<1)      ///< position of
PWM_BRIDGE in               @ref CoilsMenu_HandleTypeDef
#define Menu_PWM_PHASE                  (1<<2)      ///< position of PWM_PHASE
in                @ref CoilsMenu_HandleTypeDef
#define Menu_PWM_POLARITY               (1<<3)      ///< position of
PWM_POLARITY in             @ref CoilsMenu_HandleTypeDef
#define Menu_PWM_ACTIVECHANNEL  (1<<4)      ///< position of PWM_ACTIVECHANNEL
in @ref CoilsMenu_HandleTypeDef

/** INTERFACE_MENU_DEFINES
  * @}
  */

/**
    * @addtogroup INTERFACE_HANDLE_DEFINES
    * @ingroup      INTERFACE
    * @brief            Defines for menu
    * @{
```

```
    */
typedef struct
{
    LCD_HandleTypeDef                       hlcd;
    TIM_EncoderTypeDef                  henc;

    // menu vars
    Menu_TypeDef                                Menu;
    Menu_TypeDef                                Displaying;
    uint32_t                                        CurrentSelection;
    MenuIndex_TypeDef                       MenuNumber;
    unsigned                                        FineChanging:1;
    int8_t                                      StartDispItem;
    int8_t                                      disp_cnt;

    // lcd vars
    char
LCD_String[NumbOfSymb+1];
    portTickType                                LCD_prevUpdateTick;
    uint16_t                                        LCD_UpdateDelay;
    uint8_t                                     LCD_ClearDelay;

    // encoder vars
    uint32_t                                        Encoder_Shdw;
    uint32_t                                        Switch_Shdw;
    uint32_t                                        Switch_prevTick;
    uint32_t                                        DoubleClick_Timeout;
}Interface_HandleTypeDef;
extern Interface_HandleTypeDef hinterface;

/** INTERFACE_HANDLE_DEFINES
  * @}
  */



//---------------------------------FUNCTIONS-----------------------------------
/**
    * @addtogroup INTERFACE_FUNCTIONS
    * @ingroup      INTERFACE
    * @brief            Function for controling interface and menu
    */


//-----------------ENCODER FUNCTIONS---------------//
/**
    * @addtogroup INTERFACE_ENCODER_FUNCTIONS
    * @ingroup      INTERFACE_FUNCTIONS
    * @brief            Function for controling encoder
    * @{
    */

/* Update Encoder */
void Update_Encoder(Interface_HandleTypeDef *hinterface);
/* Start Single Click Actions */
void Encoder_SingleClick(Interface_HandleTypeDef *hinterface);
/* Start Double Click Actions */
void Encoder_DoubleClick(Interface_HandleTypeDef *hinterface);
/* Holding Switch action */
uint8_t Encoder_HoldingSwitch(Interface_HandleTypeDef *hinterface);
/* Holding Switch and Rotate encoder action */
```

```
uint8_t Encoder_HoldingAndRotateEncoder(Interface_HandleTypeDef *hinterface);
/* Release Switch action */
void Encoder_SwitchRelease(Interface_HandleTypeDef *hinterface);
/* Update params variables from encoder rotate */
void Encoder_UpdateVarsFromEncoder(Interface_HandleTypeDef *hinterface);
/* Set uint16_t value from from encoder rotate */
void Encoder_SetUint16Value(TIM_EncoderTypeDef *henc, uint16_t *TargetValue,
uint16_t coef, int32_t min_value, int32_t max_value);
/* Set uint8_t value from from encoder rotate */
void Encoder_SetUint8Value(TIM_EncoderTypeDef *henc, uint8_t *TargetValue,
uint16_t coef, int32_t min_value, int32_t max_value);

/** INTERFACE_ENCODER_FUNCTIONS
  * @}
  */



//---------------LCD DISPLAY FUNCTIONS--------------//
/**
    * @addtogroup INTERFACE_LCD_FUNCTIONS
    * @ingroup      INTERFACE_FUNCTIONS
    * @brief            Function for controling lcd display
    * @{
    */
/* Update Active Items in Menu Structure for displaying */
void Update_ActiveMenuItems(Interface_HandleTypeDef *hinterface);
/* Update LCD display */
void Update_LCDDisplay(Interface_HandleTypeDef *hinterface);
/* Display all coils menu at LCD Display */
uint8_t LCD_DisplayMenuCoils(Interface_HandleTypeDef *hinterface);
/* Display all registers menu at LCD Display */
uint8_t LCD_DisplayMenuRegisters(Interface_HandleTypeDef *hinterface);
/* Print string with menu item */
uint8_t LCD_PrintString(Interface_HandleTypeDef *hinterface, unsigned
SELECTED_DEFINE, void (*Print_Function)(Interface_HandleTypeDef *, uint8_t));
/* Update string */
uint8_t LCD_UpdateString(Interface_HandleTypeDef *hinterface, uint8_t
disp_flag);

/** INTERFACE_LCD_FUNCTIONS
  * @}
  */

//------------------PRINT FUNCTIONS-----------------//
/**
    * @addtogroup INTERFACE_PRINT_FUNCTIONS
    * @ingroup      INTERFACE_FUNCTIONS
    * @brief    Function for print string in char array.
    * @param    hinterface      - handle for interface
    * @param    selected_flag - flag to print string as currently selected, or
as regular
    * @note     Char array: hinterface->LCD_String
    * @{
    */
void Print_PWM_Sine_Hz(Interface_HandleTypeDef *hinterface, uint8_t
selected_flag);
void Print_PWM_Hz(Interface_HandleTypeDef *hinterface, uint8_t selected_flag);
void Print_PWM_Duty(Interface_HandleTypeDef *hinterface, uint8_t
selected_flag);
```

```
void Print_PWM_MaxPulseDur(Interface_HandleTypeDef *hinterface, uint8_t
selected_flag);
void Print_PWM_MinPulseDur(Interface_HandleTypeDef *hinterface, uint8_t
selected_flag);
void Print_PWM_DeadTime(Interface_HandleTypeDef *hinterface, uint8_t
selected_flag);
void Print_PWM_TickBasePresc(Interface_HandleTypeDef *hinterface, uint8_t
selected_flag);
void Print_PWM_DC(Interface_HandleTypeDef *hinterface, uint8_t selected_flag);
void Print_PWM_BRIDGE(Interface_HandleTypeDef *hinterface, uint8_t
selected_flag);
void Print_PWM_PHASE(Interface_HandleTypeDef *hinterface, uint8_t
selected_flag);
void Print_PWM_POLARITY(Interface_HandleTypeDef *hinterface, uint8_t
selected_flag);
void Print_PWM_ACTIVECHANNEL(Interface_HandleTypeDef *hinterface, uint8_t
selected_flag);

/** INTERFACE_PRINT_FUNCTIONS
  * @}
  */


//--------------INITIALIZATION FUNCTIONS------------//
/**
    * @addtogroup INTERFACE_INIT_FUNCTIONS
    * @ingroup      INTERFACE_FUNCTIONS
    * @brief            Function for initialize interface and menu
    * @{
    */


/* First initialization of interface struct and necessary peripheral */
void Interface_FirstInit(void);
/* First initialization of SPI LCD */
void SPILCD_FirstInit(void);
/* First initialization of LCD (SPI/I2C corresponding to define) */
void LCD_FirstInit(void);
/* First initialization of Encoder Timer */
void EncoderFirstInit(void);

/** INTERFACE_INIT_FUNCTIONS
  * @}
  */

#endif // __INTERFACE_H_
```

ПРИЛОЖЕНИЕ Д

Листинг control.c

```
/**
    ******************************************************************************
    * @file     control.c
    * @brief    Модуль для управления ШИМ и записи логов.
    ******************************************************************************/
#include "pwm.h"

TIM_SettingsTypeDef TIM_CTRL = {0};

// variables for filling arrays
int Numb_Of_Peroids = 2;            // number of periods
int Samples_Per_Peroid = 0;     // how many samples in one period
int Size_Of_Log = 0;                    // size of written data to log
int log_ind = 0;                            // index of log arrays
int cnt_to_cnt_log = 0;             // counter for log_cnt

int sine_ind_prev = 0;

/**
    * @brief    Filling logs.
    * @note     Заполнение логов: синус, шим, пила.
    * @note     This called from TIM_CTRL_Handler
    */
void Fill_Logs_with_Data(void)
{
    // calc pwm duty from timer
    float PWM_Duty;
    if(PWM_Get_Mode(&hpwm1, PWM_DC_MODE) == 0) // if sinus need to be written
    {
        if(PWM_Get_Mode(&hpwm1, PWM_BRIDGE_MODE)) // if its signed sine mode
(two channels)
        {
            if(hpwm1.Duty_Table_Ind < hpwm1.Duty_Table_Size/2)  // first half
get from channel 1
                PWM_Duty =
(((float)PWM_Get_Compare1(&hpwm1))/(PWM_Get_Autoreload(&hpwm1)))+1;
            else
// second half get from channel 2
                PWM_Duty = 1-
(((float)PWM_Get_Compare2(&hpwm1))/(PWM_Get_Autoreload(&hpwm1)));
        }
        else // if its unsigned sine mode (single channel)
        {   // just get current pwm duty
            PWM_Duty =
((float)PWM_Get_Compare1(&hpwm1)/PWM_Get_Autoreload(&hpwm1));
        }
    }
    else // if its dc pwm mode
    { // just get current pwm duty
        if(PWM_Get_Mode(&hpwm1, PWM_BRIDGE_MODE)) // if its second channels
mode
            PWM_Duty =
((float)PWM_Get_Compare2(&hpwm1)/PWM_Get_Autoreload(&hpwm1));
        else
// if its first channel mode
            PWM_Duty =
((float)PWM_Get_Compare1(&hpwm1)/PWM_Get_Autoreload(&hpwm1));
```

```
    }



    // WRITE SINUS TO WHOLE ARRAY
//  sine_log[log_ind] = sin_val;
    if(PWM_Get_Mode(&hpwm1,PWM_DC_MODE) == 0)    // in table mode write PWM
Duty (write sine) with scale 1/2 from sin table max value (0xFFFF/2)
        sine_log[log_ind] = PWM_Duty*(0x8000-1);
    else
// in dc mode write PWM Duty (write sine)
        sine_log[log_ind] = 0;



    // WRITE PWM
    if(PWM_Get_Mode(&hpwm1,PWM_DC_MODE)) // in DC mode
    {
        // write 1 - if log_ind < Size_Of_Period*PWM_Dury
        // write 0 - otherwise
        pwm_log[log_ind] = (log_ind%(Size_Of_Log/Numb_Of_Peroids) <
(Size_Of_Log/Numb_Of_Peroids+1)*hpwm1.PWM_Sine_Hz/100)? 1: 0;
    }
    else // in table mode
    {
        // write fill whole pwm array at one interrupt
        int PWM_Period_End_Ind = (Size_Of_Log/Numb_Of_Peroids);
        int PWM_Step_End_Ind;
        if(PWM_Get_Mode(&hpwm1,PWM_BRIDGE_MODE))
            PWM_Step_End_Ind = PWM_Period_End_Ind*fabs(PWM_Duty-1);
        else
            PWM_Step_End_Ind = PWM_Period_End_Ind*PWM_Duty;
        for(int i = 0; i <= PWM_Step_End_Ind; i++)
        {
            for (int j = 0; j < Numb_Of_Peroids; j++)
                pwm_log[i+j*PWM_Period_End_Ind] = 1;
        }
        for(int i = PWM_Step_End_Ind+1; i < PWM_Period_End_Ind; i++)
            for (int j = 0; j < Numb_Of_Peroids; j++)
                pwm_log[i+j*PWM_Period_End_Ind] = 0;
    }

    // WRITE COUNTER
    cnt_log[log_ind] = (uint16_t)(hpwm1.PWM_Sine_Hz*100);
    cnt_to_cnt_log++;
    if(cnt_to_cnt_log>=Size_Of_Log/2)
        cnt_to_cnt_log = 0;

    // INCREMENT AND RESET COUNTER
    log_ind++;
    if(PWM_Get_Mode(&hpwm1,PWM_DC_MODE) == 0) // if its PWM table mode
    {
        // SYNCHRONIZE PERIOD OF SIN IN LOG
        // (это надо, чтобы данные не съезжали из-за несинхронизированного
периода)

        // wait until period ended
        if(log_ind>Size_Of_Log-1) // if logs are filled
        {
            if((unsigned)hpwm1.Duty_Table_Ind < sine_ind_prev) // and if new
period started
```

```
            {
                log_ind = 0; // reset counter
                sine_ind_prev = (unsigned)hpwm1.Duty_Table_Ind;
            }
        }
        // update prev variable only if log currently writing
        else
            sine_ind_prev = (unsigned)hpwm1.Duty_Table_Ind;
    }
    else // if its PWM DC mode
    {
        // if logs are filled
        if(log_ind>Size_Of_Log-1)
            log_ind = 0;
    }


    // if its overflow log array size - reset log_ind
    if(log_ind>LOG_SIZE-1)
    {
        log_ind = 0;
        sine_ind_prev = (unsigned)hpwm1.Duty_Table_Ind;
    }
}

/**
    * @brief    Update log parameters.
    * @note     Проверка надо ли обновлять параметры логов, и если надо -
обновляет их.
    * @note     This called from TIM_CTRL_Handler
    */
void Update_Params_For_Log(void)
{
    unsigned UpdateLog = 0;

    // READ NUMB OF PERIOD IN LOGS
    if(Numb_Of_Peroids != log_ctrl[R_LOG_CTRL_LOG_PWM_NUMB])
    {
        Numb_Of_Peroids = log_ctrl[R_LOG_CTRL_LOG_PWM_NUMB];
        // update logs params
        UpdateLog = 1;
    }
    // READ SIZE OF LOGS
    if(Size_Of_Log != log_ctrl[R_LOG_CTRL_LOG_SIZE])
    {
        Size_Of_Log = log_ctrl[R_LOG_CTRL_LOG_SIZE];
        // update logs params
        UpdateLog = 1;
    }

    // UPDATE LOG PARAMS
    if(UpdateLog)
    {
        // set logs params
        Set_Log_Params();
    }
}

/**
    * @brief    Set up log parameters.
    * @note     Устанавливает настройки логов и проверяет их на корректность.
```

```
    */
void Set_Log_Params(void)
{
    // SET LOG PARAMS
    log_ind = 0;
    Samples_Per_Peroid = TIM_CTRL.sTimFreqHz/hpwm1.PWM_Sine_Hz;

    if(Size_Of_Log > LOG_SIZE) // if its too much data in log
    {
        Numb_Of_Peroids = (LOG_SIZE/Samples_Per_Peroid);
        log_ctrl[R_LOG_CTRL_LOG_SIZE] = Numb_Of_Peroids;
        Size_Of_Log = Numb_Of_Peroids*Samples_Per_Peroid;
    }

    // clear logs arrays
    for(int i = Size_Of_Log; i < LOG_SIZE; i++)
    {
        sine_log[i] = 0;
        pwm_log[i] = 0;
        cnt_log[i] = 0;
    }

}

/**
    * @brief    reInitialization of control timer.
    * @note     Перенастраивает таймер согласно принятным настройкам в
log_ctrl.
    * @note     This called from main while
    */
void Control_Timer_ReInit(TIM_SettingsTypeDef *stim)
{
    TIM_Base_MspDeInit(&stim->htim);
    hpwm1.stim.sTickBaseUS = PROJSET_MEM->TIM_CTRL_TICKBASE;
    TIM_Base_Init(stim);

    HAL_TIM_Base_Start_IT(&stim->htim); // timer for sinus
  HAL_NVIC_SetPriority(TIM8_BRK_TIM12_IRQn, 1, 1);
}
/**
    * @brief    First initialization of Control Timer.
    * @note     Первый управляющего таймера. Таймер записывает логи и
обновляет параметры ШИМ.
    * @note     This called from main
    */
void Control_Timer_FirstInit(void)
{
    //-------CONTROL TIMER INIT----------
    // tim settings
    TIM_CTRL.htim.Instance = TIMER_CTRL_INSTANCE;
    TIM_CTRL.sTimMode = TIM_IT_MODE;
    TIM_CTRL.sTickBaseUS = PROJSET.TIM_CTRL_TICKBASE;
    TIM_CTRL.sTimAHBFreqMHz = PROJSET.TIM_CTRL_AHB_FREQ;
    TIM_CTRL.sTimFreqHz = HZ_TIMER_CTRL;

    TIM_Base_Init(&TIM_CTRL);
  HAL_NVIC_SetPriority(TIM8_TRG_COM_TIM14_IRQn, 1, 1);

    HAL_TIM_Base_Start_IT(&TIM_CTRL.htim); // timer for sinus
```

```
    // FILL TIME ARRAY WITH TIME
    for(int i = 0; i < R_TIME_LOG_QNT; i++)
        time_log[i] = i;

}
//-----------------------------------------------------------------------
//-------------------------HANDLERS FUNCTIONS----------------------------
//-------------CONTROL TIMER---------------
void TIM8_UP_TIM13_IRQHandler(void)
{
    /* TIM_CTRL_Handler */
    Trace_CTRL_TIM_Enter();
  HAL_TIM_IRQHandler(&TIM_CTRL.htim);

    Fill_Logs_with_Data();
    Update_Params_For_Log();
    PWM_Update_Params(&hpwm1);

    WriteSettingsToMem();

    Trace_CTRL_TIM_Exit();
}
```

ПРИЛОЖЕНИЕ Е

Листинг control.h

```
/**
    ******************************************************************************
    * @file     control.h
    * @brief    Заголовочный файл для модуля управления ШИМ и записи логов.
    ******************************************************************************/
    #ifndef __CONTROL_H_
#define __CONTROL_H_

#include "periph_general.h"
#include "modbus.h"
#include "math.h"
//#include "settings.h"

#define M_PI            3.14159265358979323846  /* pi */

extern TIM_SettingsTypeDef TIM_CTRL;

//----------------------this called from TIM_CTRL_Handler()--------------------
---
/**
    * @brief    Update log parameters.
    * @note     Проверка надо ли обновлять параметры логов, и если надо -
обновляет их.
    * @note     This called from TIM_CTRL_Handler
    */
void Update_Params_For_Log(void);
/**
    * @brief    Filling logs.
    * @note     заполнение логов: синус, шим, пила.
    * @note     this called from TIM_CTRL_Handler
    */
void Fill_Logs_with_Data(void);
/**
    * @brief    Set up log parameters.
    * @note     Устанавливает настройки логов и проверяет их на корректность.
    */
void Set_Log_Params(void);
// this called from main while(1)
/**
    * @brief    reInitialization of control timer.
    * @param    stim - указатель на настройки таймера.
    * @note     Перенастраивает таймер согласно принятным настройкам в
log_ctrl.
    * @note     This called from main while
    */
void Control_Timer_ReInit(TIM_SettingsTypeDef *stim);
/**
    * @brief    First initialization of Control Timer.
    * @note     Первый управляющего таймера. Таймер записывает логи и
обновляет параметры ШИМ.
    * @note     This called from main
    */
void Control_Timer_FirstInit(void);
/**
    * @brief    First initialization of Encoder Timer.
    * @note     This called from main
    */
```

```
void EncoderFirstInit(void);

#endif // __CONTROL_H_
```