МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное автономное образовательное учреждение высшего образования
«САНКТ–ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

ДОПУСТИТЬ К ЗАЩИТЕ
Заведующий кафедрой №23

доктор тех. наук., проф.		А. Р. Бестугин
должность, уч. степень, звание	подпись, дата	инициалы, фамилия

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

на тему Разработка модели микроконтроллера в MATLAB для отладки и тестирования программы управления устройством плавного пуска двигателя

выполнена Разваляевым Алексеем Викторовичем
фамилия, имя, отчество студента в творительном падеже

по направлению подготовки 11.04.04 (код) Электроника и наноэлектроника (наименование направления)

(наименование направления)

направленности 01 (код) Системы сбора, обработки (наименование направленности)
и отображения информации

(наименование направленности)

Студент группы № 2337М _____ (подпись, дата) А. В. Разваляев (инициалы, фамилия)

Руководитель

доц. каф. 23, д.т.н., доц.		А. Л. Ляшенко
должность, уч. степень, звание	подпись, дата	инициалы, фамилия

Санкт-Петербург 2025

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное автономное образовательное учреждение высшего образования
«САНКТ–ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

УТВЕРЖДАЮ
Заведующий кафедрой №23

доктор тех. наук., проф.		А. Р. Бестугин
должность, уч. степень, звание	подпись, дата	инициалы, фамилия

ЗАДАНИЕ НА ВЫПОЛНЕНИЕ МАГИСТЕРСКОЙ ДИССЕРТАЦИИ

студенту группы № 2337М Разваляеву Алексею Викторовичу
(фамилия, имя, отчество)

на тему Разработка модели микроконтроллера в MATLAB для отладки и тестирования программы управления устройством плавного пуска двигателя

утвержденную приказом ГУАП от _____ № _____

Цель исследования: Разработка программной модели микроконтроллера STM32 и анализ её применимости для отладки микроконтроллерных приложений в среде MATLAB/Simulink

Задачи исследования: Обеспечить интеграцию С-кода микроконтроллера в MATLAB, Исследовать адекватность работы модели микроконтроллера, Оценить ограничения и условия применимости симуляции для отладки встроенных программ

Содержание диссертации (основные разделы): Асинхронные двигатели, Разработка модели микроконтроллера в MATLAB, Разработка микроконтроллерных приложений в MATLAB, Моделирование устройства плавного пуска

Срок сдачи диссертации «____» _____ 2025

Руководитель

доц. каф. 23, д.т.н., доц.		А. Л. Ляшенко
должность, уч. степень, звание	подпись, дата	инициалы, фамилия

Задание принял к исполнению

студент группы № 2337М		А. В. Разваляев
	подпись, дата	инициалы, фамилия

РЕФЕРАТ

Отчет 77 с., 33 рис., 1 табл., 7 источн., 3 прил.

УПП – УСТРОЙСТВО ПЛАВНОГО ПУСКА, MATLAB, МК – МИКРОКОНТРОЛЛЕРЫ, ОТЛАДКА ИСХОДНОГО КОДА, SIMULINK

Объектом исследования является микроконтроллерная программа устройствами плавного пуска асинхронных электродвигателей.

Цель работы – исследование возможности отладки микроконтроллерных программ средствами MATLAB/Simulink.

В ходе работы были проанализированы особенности программной реализации периферии микроконтроллера, разработана модель микроконтроллера с поддержкой ключевых периферийных устройств и обеспечено взаимодействие с моделью силовой схемы через интерфейсы Simscape. Также была проверена корректность модели в сравнении с реальным микроконтроллером

Также на примере алгоритма управления устройством плавного пуска продемонстрирована возможность применения разработанной модели для отладки и тестирования управляющих программ.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Современные системы управления электроприводами всё чаще базируются на микроконтроллерах, которые реализуют сложные алгоритмы регулирования и плавного пуска. Такие алгоритмы особенно актуальны для асинхронных электродвигателей – наиболее распространённого типа электрических машин, применяемого в промышленности, энергетике и бытовом секторе. Несмотря на свои преимущества, такие как надёжность, простота конструкции и низкая стоимость, асинхронные двигатели в момент запуска характеризуются значительными переходными процессами. Резкие скачки токов и крутящего момента приводят к перегрузкам в электросети, сокращению срока службы оборудования и снижению общей устойчивости системы.

Для снижения отрицательных эффектов переходных режимов применяются устройства плавного пуска и преобразователи частоты. Они позволяют формировать напряжение и частоту, изменяющиеся по заданному закону, обеспечивая контролируемый разгон и торможение двигателя. Однако эффективность этих средств напрямую зависит от точности реализуемых алгоритмов управления и отладки параметров. Современная практика проектирования требует не только моделирования электрической части, но и полной симуляции управляющего программного обеспечения. Это позволяет проводить тестирование и настройку алгоритмов до внедрения их на реальное устройство, что особенно важно при разработке встроенных систем управления.

Тем не менее, стандартные средства моделирования в MATLAB и Simulink предоставляют ограниченные возможности для интеграции кода, написанного для микроконтроллеров. Отладка таких алгоритмов традиционно осуществляется уже на уровне “железа”, что требует наличия оборудования, замедляет цикл разработки и усложняет воспроизводимость результатов. Кроме того, отдельное моделирование программной и аппаратной части

системы может приводить к расхождениям между симулируемым и реальным алгоритмами.

В данной работе рассматривается решение этой проблемы путём разработки программной модели микроконтроллера STM32, предназначенной для исполнения пользовательского C-кода в среде MATLAB. Модель позволяет реализовать связку между кодом, используемым в реальных микроконтроллерах, и средой моделирования Simulink, что даёт возможность запускать, тестировать и отлаживать управляющие программы без использования физического микроконтроллера. При этом моделируется как логика работы кода, так и взаимодействие с внешними компонентами – например, электрической частью системы пуска, реализованной в Simulink.

Разработка такой модели решает сразу несколько инженерных задач: сокращение времени отладки, повышение воспроизводимости тестов, возможность работы без целевого оборудования, а также обеспечение точного соответствия симулируемой логики реальному программному коду. Предлагаемый подход обеспечивает непрерывность между программной реализацией и её проверкой в контексте моделируемой системы. Таким образом, он может быть полезен как при исследовании сложных переходных процессов, так и при разработке встроенных систем управления электроприводами и других задач автоматизации.

Целью настоящей работы является разработка программной модели микроконтроллера STM32 в MATLAB, предназначенной для пошаговой отладки встроенного кода управления и его интеграции в модель управляемой системы в среде Simulink. Разработанная модель позволяет значительно упростить тестирование управляющих алгоритмов, выявлять логические ошибки и некорректную работу периферии ещё на этапе симуляции, без необходимости в физическом оборудовании, что делает процесс проектирования более гибким, быстрым и надёжным.

В настоящей работе впервые предложен и реализован интегрированный подход к моделированию устройств, включающий совместное представление

электрической части и программного обеспечения микроконтроллера в среде MATLAB. Разработанная программная модель микроконтроллера обеспечивает возможность детального тестирования и отладки управляющих алгоритмов в условиях, максимально приближенных к реальным, без необходимости использования физического оборудования. Использование данного инструмента предполагает базовое владение средой MATLAB/Simulink, понимание принципов работы микроконтроллеров, знание языка программирования С и общее представление о структуре программ.

Новизна исследования заключается в создании комплексной симуляционной платформы, которая позволяет объединить физическую модель переходных процессов асинхронного двигателя с моделями управления, реализованными на уровне микроконтроллерного программного кода. Такой подход расширяет возможности анализа систем плавного пуска, позволяя не только исследовать динамику электропривода, но и проводить всестороннюю оценку работы программных алгоритмов управления.

В ходе исследования будет продемонстрирован процесс разработки алгоритма в виде программного кода для микроконтроллера для устройства плавного пуска (УПП). Будут рассмотрены принципы построения УПП и особенности их взаимодействия с асинхронными двигателями, а также создана функциональная модель устройства в MATLAB. Комплексное моделирование как объекта управления, так и управляющей системы обеспечивает всесторонний анализ поведения всей системы.

Методологическую основу работы составляет применение математического моделирования и средств MATLAB для описания переходных процессов, построения схем управления и оценки эффективности работы УПП. Благодаря встроенным инструментам симуляции и отладки программ, MATLAB предоставляет удобную возможность пройти весь путь разработки – от создания модели до проверки работы управляющей программы – в одной среде.

Первая глава посвящена анализу методов пуска асинхронных двигателей, включая традиционный прямой пуск, схемы снижения пускового тока. В главе рассматриваются особенности переходных процессов и влияние различных схем на электромеханические характеристики двигателя. Особое внимание уделено принципам работы устройств плавного пуска с микроконтроллерным управлением, что создаёт основу для последующего изучения алгоритмов управления и их моделирования в среде MATLAB.

Вторая глава посвящена созданию модели микроконтроллера в среде MATLAB, которая служит инструментом для симуляции и отладки встроенных программ. В данной главе рассматриваются основные принципы управления выполнением программного кода микроконтроллера в Simulink, а также методы моделирования периферийных устройств.

В третьей главе продемонстрирован процесс разработки микроконтроллерных приложений в среде MATLAB с использованием предложенной модели МК. Особое внимание уделяется интеграции программного кода с моделью и этапам отладки с использованием возможностей MATLAB.

В четвертой главе представлено моделирование процесса пуска асинхронного двигателя с использованием устройства плавного пуска (УПП). Сначала проанализирован режим прямого пуска, далее рассмотрена работа двигателя при пуске через УПП. Также было произведено сравнение работы реального микроконтроллера и его модели в MATLAB.

1 АСИНХРОННЫЕ ДВИГАТЕЛИ

Асинхронные электродвигатели являются одним из наиболее широко используемых типов электрических машин в промышленности и коммунальном хозяйстве благодаря своей надёжности, простоте конструкции и доступной стоимости. Основными конструктивными элементами асинхронного двигателя являются статор и ротор, каждый из которых играет ключевую роль в процессе преобразования электрической энергии в механическую. Устройство асинхронного двигателя схематично изображено на рисунке 1 [1].

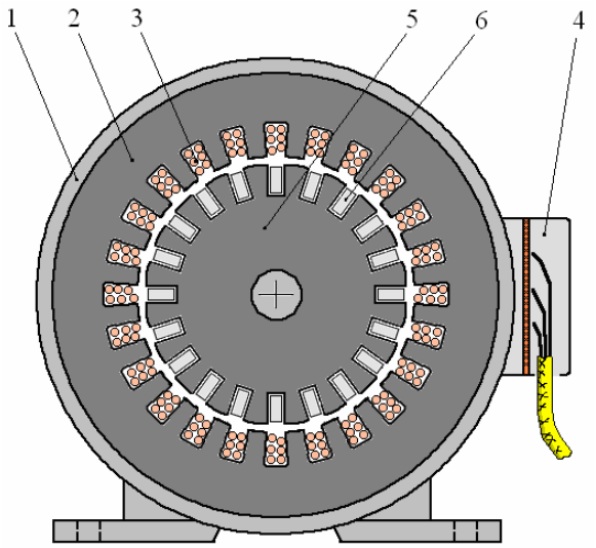


Рисунок 1 – Конструкция асинхронного двигателя

1-станина; 2-сердечник статора; 3-обмотка статора; 4-клеммная коробка; 5-сердечник ротора; 6-стержни обмотки ротора

Статор представляет собой неподвижную часть машины и состоит из магнитопровода с размещёнными в нём обмотками. К обмоткам статора подключается трёхфазное переменное напряжение, в результате чего создаётся вращающееся магнитное поле. Это поле индуцирует токи в роторе – вращающейся части двигателя, находящейся внутри статора. Взаимодействие индуцированных токов с магнитным полем приводит к появлению электромагнитного момента, в результате чего ротор начинает вращаться.

Особенностью асинхронного двигателя является то, что скорость вращения ротора всегда немного отстаёт от скорости вращения магнитного поля статора – это явление называется скольжением и объясняет происхождение термина “асинхронный”. Если ротор вращается с скоростью магнитного поля статора – токи в роторе не индуцируются, и, как следствие, двигатель не развивает момент и замедляется.

Существуют два основных типа асинхронных двигателей. С короткозамкнутым ротором, где роторные проводники замкнуты между собой при помощи контактных колец, образуя своего рода “беличье колесо” (наиболее распространённый тип). С фазным ротором, в котором роторные обмотки подключены к внешним элементам управления (резисторам или тиристорным модулям), что позволяет регулировать пусковые и динамические характеристики двигателя.

Работа асинхронного двигателя сопровождается динамическими процессами, особенно наиболее выраженными при изменении режимов работы. Такие процессы называют переходными – они представляют собой процессы, происходящие во времени при переходе системы из одного установившегося состояния в другое, например, при пуске, остановке или резком изменении нагрузки. В этот момент в системе наблюдаются резкие изменения тока, напряжения, скорости вращения и электромагнитного момента.

Причины возникновения переходных процессов могут быть различными:

- управляющие воздействия, такие как включение или отключение двигателя, а также изменение заданной скорости вращения;
- возмущающие воздействия, например, внезапное изменение нагрузки на валу, что вызывает нарушение установившегося режима.

Переходные процессы обусловлены как электромагнитной инерционностью (в первую очередь – наличием индуктивностей обмоток, ограничивающих скорость изменения токов), так и механической инерцией

подвижных масс, соединённых с валом двигателя. При запуске двигателя происходит переход от состояния покоя к вращению, в результате чего токи и электромагнитные моменты могут кратковременно значительно превышать свои номинальные значения.

В частности, пусковой ток в обмотках статора может превышать номинальный в 5–7 раз, а электромагнитный момент – быть в 2–3 раза выше установившегося. При реверсировании двигателя (изменении направления вращения) эти величины могут возрастать ещё сильнее – вплоть до 12–18 раз. Более того, переходные моменты могут принимать отрицательные значения, вызывая торможение ротора и создавая значительные динамические нагрузки на механическую часть привода.

Дополнительным фактором, усиливающим переходные процессы, является остаточное магнитное поле, сохраняющееся в магнитопроводе двигателя после его отключения. Это остаточное намагничивание может привести к усилению индукции токов в момент следующего включения, увеличивая пиковые значения тока и момента, что критично при частых циклах пуска и останова.

Таким образом, для эффективной и безопасной работы асинхронных электродвигателей, особенно в условиях переменных нагрузок, необходим тщательный анализ переходных процессов. Это требует применения различных методов управления пуском и торможением, в том числе тех, которые основаны на алгоритмически сложных схемах регулирования и позволяют смягчать динамические проявления переходных состояний.

1.1 МЕТОДЫ ПУСКА АСИНХРОННЫХ ДВИГАТЕЛЕЙ

Прямой пуск (или пуск с полным напряжением) представляет собой наиболее простой и широко распространённый способ запуска асинхронного двигателя. При таком подходе на статор двигателя сразу подаётся полное напряжение сети, что приводит к резкому возрастанию тока до величин, в 5–8 раз превышающих номинальные значения. Электромагнитный момент при этом также достигает максимума, но из-за высокой инерции механической части он не успевает обеспечить быструю раскрутку ротора, что удлиняет время пуска и увеличивает тепловую нагрузку на обмотки.

Основной проблемой традиционного прямого включения двигателя в сеть является высокий пусковой ток, который может в несколько раз превышать номинальный. Это приводит к перегрузкам питающей сети, износу коммутационной аппаратуры и сокращению срока службы двигателя. Для снижения негативных эффектов разработано множество методов пуска, каждый из которых по-своему влияет на характер переходных процессов. Их правильный выбор позволяет адаптировать электропривод под конкретные условия эксплуатации и требования к надёжности и энергоэффективности.

Характерной чертой прямого пуска является его высокая эффективность в плане быстрого выхода двигателя на рабочий режим, однако он приемлем лишь в условиях, когда сеть обладает высокой мощностью, а двигатель не испытывает больших механических нагрузок в момент запуска. В противном случае возникают кратковременные падения напряжения и возможны срывы пуска. Также из-за резкого нарастания тока возникает сильное электромагнитное взаимодействие между проводниками, способное повредить элементы распределительной системы.

Для смягчения переходных процессов, возникающих при запуске, применяются различные схемы пуска, позволяющие ограничить пусковой ток, уменьшить механические удары и продлить срок службы оборудования. Эти методы отличаются как по технической реализации, так и по степени влияния на характеристики двигателя в переходном режиме [2].

Пуск с пониженным напряжением (через автотрансформатор или реостат). Снижение напряжения, подаваемого на статор двигателя на этапе пуска, позволяет существенно ограничить пусковой ток. Один из способов реализации – использование автотрансформатора. При этом на двигатель подаётся часть фазного напряжения (обычно 50–80 %), что позволяет уменьшить пусковой ток пропорционально квадрату поданного напряжения. Однако одновременно с током снижается и электромагнитный момент, что может привести к неустойчивому пуску, особенно при наличии нагрузки на валу.

Альтернативным методом является включение сопротивления в цепь статора или ротора. В случае фазного ротора это осуществляется довольно просто: через контактные кольца включаются дополнительные резисторы, которые постепенно шунтируются по мере разгона двигателя. Это позволяет обеспечить плавное нарастание тока и момента, улучшив стабильность пуска.

Подобный способ плавного пуска имеет очевидные недостатки проблематичность автоматизации, работа контакторов не привязывается к реальному значению тока, они либо переключаются вручную, либо перебираются с помощью реле времени автоматически. Также усложнен пуск под нагрузкой, так как крутящий момент асинхронного двигателя пропорционален квадрату напряжения питания, снижение напряжения в момент пуска в два раза приведет к снижению крутящего момента в четыре раза.

Пуск по схеме “звезд-треугольник”. Схема “звезда–треугольник” основана на изменении способа соединения обмоток статора в процессе пуска. Основная идея использования такого способа пуска состоит в том, что в начальный момент разгона двигателя, его обмотки соединены “звездой”, что обеспечивает пониженный ток. По истечении определенного времени подключение меняется на “треугольник”, что обеспечит полный ток и крутящий момент. При подключении по схеме “треугольник” напряжение на каждой обмотке двигателя соответствует напряжению в сети.

Этот способ особенно популярен благодаря своей простоте и эффективности. Он не требует дорогостоящих компонентов и обеспечивает приемлемый баланс между токовыми и моментными характеристиками. Преимуществом пуска также является, то, что некоторые трехфазные двигатели на низкое напряжение с мощностью выше 5 кВт рассчитывают на напряжение 400 В при включении по схеме “треугольник” (Δ) или на 690 В при включении по схеме “звезда” (Y). Такая схема включения дает возможность производить пуск двигателя при меньшем напряжении. При пуске двигателя по схеме “звезда-треугольник” удается уменьшить пусковой ток, до 1/3 от тока прямого пуска от сети. Пуск по схеме “звезда-треугольник” особенно подходит для механизмов с большими маховыми массами, когда нагрузка набрасывается уже после разгона двигателя до номинальной скорости.

Недостатком пуска асинхронного двигателя переключением “звезда-треугольник” является то, что при пуске двигателя переключением “звезда-треугольник” происходит также снижение пускового момента, приблизительно на 33%. Данный метод можно использовать только для трехфазных асинхронных двигателей, которые имеют возможность подключения по схеме “треугольник”. В таком варианте существует опасность переключения на “треугольник” при слишком низкой частоте вращения, что вызовет рост тока до такого же уровня, что и ток при прямом пуске. Во время переключения со звезды на треугольник асинхронный электродвигатель может быстро снизить скорость вращения, для увеличения которой также потребуется резкое увеличение тока.

Частотный пуск. Наиболее современным и гибким способом пуска является использование преобразователей частоты (ПЧ). ПЧ – техническое устройство, предназначенное для преобразования сетевых параметров на входе в выходные другой частоты. Производители предлагают устройства широкого частотного диапазона. Преобразователи частоты применяются при управлении асинхронными электродвигателями, регулируя скорость

вращения ротора. Преобразователь активно используются в системах плавного пуска и управления асинхронных электродвигателей

Принцип действия частотного преобразователя основан на предварительном выпрямлении входного переменного напряжения с последующей его фильтрацией и преобразованием в постоянное. Затем это напряжение с помощью инвертора и широтно-импульсной модуляции вновь преобразуется в переменное, но уже с заданными частотой и амплитудой. На этапе пуска преобразователь задаёт низкое напряжение с пониженной частотой, что позволяет плавно разогнать двигатель от нуля до требуемой скорости, не вызывая скачков тока и механических перегрузок. Такой подход существенно снижает износ оборудования и увеличивает срок его службы.

Основным преимуществом частотного пуска является высокая степень управляемости. Пользователь может задать профиль разгона и торможения в соответствии с характеристиками конкретного механизма, вплоть до нелинейных зависимостей, например экспоненциального нарастания скорости. Это особенно важно при пуске под нагрузкой, когда требуется контролируемое нарастание крутящего момента. При этом пусковой ток может быть ограничен на уровне, близком к номинальному значению двигателя, что минимизирует влияние пуска на электросеть и исключает необходимость в дополнительных устройствах защиты.

Кроме того, применение частотных преобразователей открывает возможность для более широкого управления рабочим процессом электропривода: изменения направления вращения, удержания скорости при колебаниях нагрузки, реализации торможения с рекуперацией энергии и т. д [3]. Таким образом, частотный пуск выходит за рамки просто способа запуска двигателя и становится частью интеллектуальной системы управления.

Тем не менее, данный способ пуска не лишён недостатков. Главным ограничением является высокая стоимость ПЧ по сравнению с более простыми средствами управления, такими как схемы “звезда–треугольник” или автотрансформаторные пускатели. Кроме того, инверторная природа

преобразователя вносит в токовые сигналы высшие гармоники, которые могут вызывать дополнительный нагрев обмоток двигателя, акустический шум, повышенные потери и электромагнитные помехи. Эти проблемы частично решаются установкой фильтров и использованием двигателей с улучшенной изоляцией, но это повышает общую стоимость системы.

*Устройства плавного пуска***.** Устройства плавного пуска представляют собой электронные системы на базе тиристоров, включаемых в цепь питания двигателя. Они обеспечивают постепенное нарастание напряжения на статоре за счёт фазового управления подачей тока. Такой способ позволяет точно контролировать начальный момент и ограничивать пусковой ток до заданного уровня.

В большинстве случаев УПП реализуются на базе симметричных тиристорных ключей, включённых последовательно в каждую фазу питания двигателя. Управление этими ключами осуществляется методом фазоимпульсной модуляции: в процессе пуска угол открывания тиристоров последовательно увеличивается, обеспечивая тем самым постепенное нарастание действующего значения напряжения на обмотках двигателя.

Такая реализация позволяет существенно снизить пусковой ток, уменьшить динамические нагрузки на приводной механизм и избежать механических ударов при старте. Благодаря этому устройства плавного пуска находят широкое применение в промышленности, особенно в ответственных и инерционных установках, где требуется защита оборудования от перегрузок и продление его ресурса.

Основное преимущество УПП заключается в простоте конструкции и сравнительно низкой стоимости по сравнению с преобразователями частоты. Они не требуют сложного программирования или настройки параметров частотно-регулируемого профиля, что делает их удобными в эксплуатации и обслуживании. Кроме того, такие устройства легко интегрируются в существующие системы электроснабжения и могут применяться для модернизации старых установок без необходимости замены двигателя.

Однако УПП обладают и рядом ограничений. После завершения процесса пуска устройство, как правило, полностью шунтируется, и двигатель продолжает работать от полной синусоидальной сетевой частоты (обычно 50 Гц), что исключает возможность регулирования скорости во время нормальной работы. Таким образом, устройства плавного пуска предназначены исключительно для запуска двигателя, а не для управления его рабочим режимом.

При корректной настройке УПП обеспечивают стабильный и надёжный пуск, но в случае несоответствия параметров нагрузки и характеристик пускового профиля могут возникать резонансные колебания, чрезмерное проскальзывание и недостаточный крутящий момент. Особенно это критично для механизмов с высоким моментом сопротивления на валу. В подобных ситуациях пуск может оказаться неустойчивым или вовсе невозможным.

В таблице 1 в краткой форме представлены сравнительные характеристики наиболее распространённых способов пуска.

В настоящей работе основное внимание уделено устройствам плавного пуска, поскольку данный способ пуска требует реализации алгоритмически сложного управления и, основан на применении микроконтроллеров. Такие методы требуют точной настройки и отладки управляющих алгоритмов, что делает актуальным применение разработанной в работе модели контроллера.

Другие методы пуска асинхронных двигателей в данном исследовании не рассматриваются, поскольку они либо не требуют управляющей логики вовсе, либо предполагают использование слишком специализированных и громоздких алгоритмов, реализация и отладка которых в рамках данной работы нецелесообразна.

Таблица 1 – Сравнительная характеристика пусков

Способ пуска	Преимущества	Недостатки
Прямой пуск	Простота реализации. Низкая стоимость. Максимальный пусковой момент.	Очень высокий пусковой ток (5–8 номинальных). Механические удары.
Пуск с пониженным напряжением	Снижение пускового тока пропорционально квадрату поданного напряжения.	Существенное снижение пускового момента. Токовые скачки при переходе на номинал.
Пуск звезда-треугольник	Снижение пускового тока до 1/3 от прямого. Простота реализации	Пониженный пусковой момент. Резкий токовый скачок при переключении. Не подходит при нагрузке на валу
Преобразователь частоты	Плавный разгон. Регулирование скорости. Пусковой ток близок к номинальному.	Высокая стоимость. Сложность настройки. Возможные гармоники в сети
Устройство плавного пуска	Плавное нарастание напряжения. Уменьшение пускового тока в 2–3 раза.	Пониженный момент при пуске. Отсутствие регулирования скорости.

1.2 ПРИНЦИП РАБОТЫ УСТРОЙСТВА ПЛАВНОГО ПУСКА

Принцип действия устройства плавного пуска (УПП) асинхронного двигателя основан на управлении напряжением питания в процессе запуска. Типичная схема такой системы представляет собой тиристорный регулятор напряжения, включённый между источником питания и асинхронным двигателем. Структура подобной системы представлена на рисунке 2.

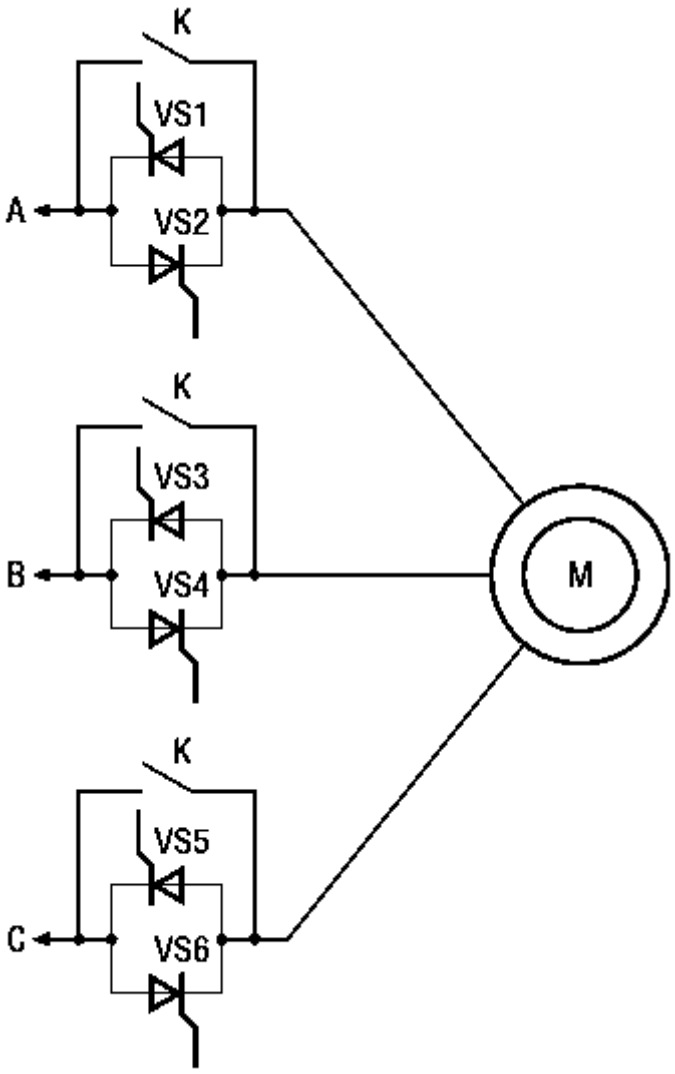


Рисунок 2 – Тиристорный регулятор напряжения

На вход регулятора подаётся трёхфазное переменное напряжение, которое затем поступает на силовые тиристоры (VS1–VS6), включённые в каждую фазу питающей линии. Управление этими тиристорами

осуществляется посредством управляющих импульсов, формируемых системой управления, как правило, реализованной на базе микроконтроллера. Основной задачей системы управления является генерация управляющих сигналов с определённой задержкой относительно момента перехода сетевого напряжения через ноль. Эта задержка характеризуется углом управления α, изменение которого позволяет регулировать среднее значение напряжения, подаваемого на статор двигателя [4].

На рисунке 3 представлен график напряжений для одной из фаз, поясняющий работу тиристорного регулятора.

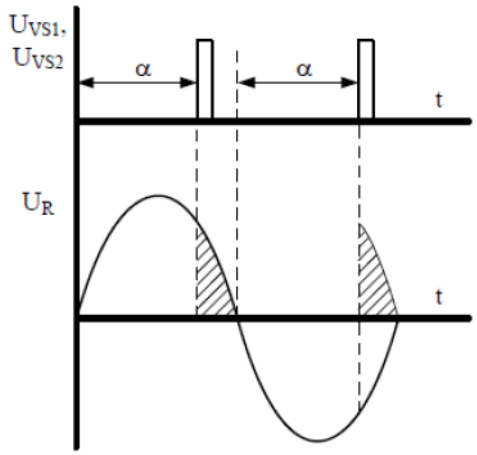


Рисунок 3 – Принцип работы тиристорного регулятора

Линия U_R отображает мгновенное значение напряжения фазы R. В моменты времени, определяемые сигналами U_{VS1} и U_{VS2}, происходят открытия тиристоров, что соответствует началу подачи напряжения на двигатель. Закрытие тиристора происходит в момент изменения полярности напряжения на его аноде и катоде, то есть при переходе напряжения U_R через ноль, при условии, что нагрузка не обладает значительной индуктивностью. Результирующее напряжение, приложенное к двигателю, определяется длительностью интервала, в течение которого тиристор остаётся открытым, и отображается на графике в виде заштрихованной области. Таким образом, величина результирующего напряжения зависит от значения угла управления

α: чем меньше угол, тем большую часть полуволны пропускает тиристор, и тем выше среднее значение напряжения на выходе преобразователя.

Процесс пуска асинхронного двигателя при использовании тиристорного регулятора реализуется посредством постепенного изменения угла управления α. На начальном этапе пуска значение α устанавливается близким к π, что соответствует минимальному напряжению на двигателе и, следовательно, сниженным пусковым токам и крутящему моменту. По мере разгона ротора угол α постепенно уменьшается, обеспечивая увеличение напряжения и, соответственно, электромагнитного момента. Когда угол достигает нуля, тиристоры открываются практически в начале каждого полупериода, и двигатель переходит в режим номинального питания. После завершения пуска тиристоры шунтируются байпасом (обходным контактором) К, благодаря чему в течение времени на тиристорах не рассеивается мощность, а значит, экономится электроэнергия.

При торможении двигателя процессы происходят в обратном порядке: после отключения контактора К угол управления α тиристорами максимален, напряжение на обмотках электродвигателя равно сетевому за вычетом падения напряжения на тиристорах. Затем угол управления α в течение времени уменьшается до минимального значения, которому соответствует напряжение отсечки, после чего угол проводимости тиристоров становится равным нулю и напряжение на обмотки не подаётся. На рисунке 4 приведены диаграммы тока одной из фаз двигателя при постепенном увеличении угла проводимости тиристоров.

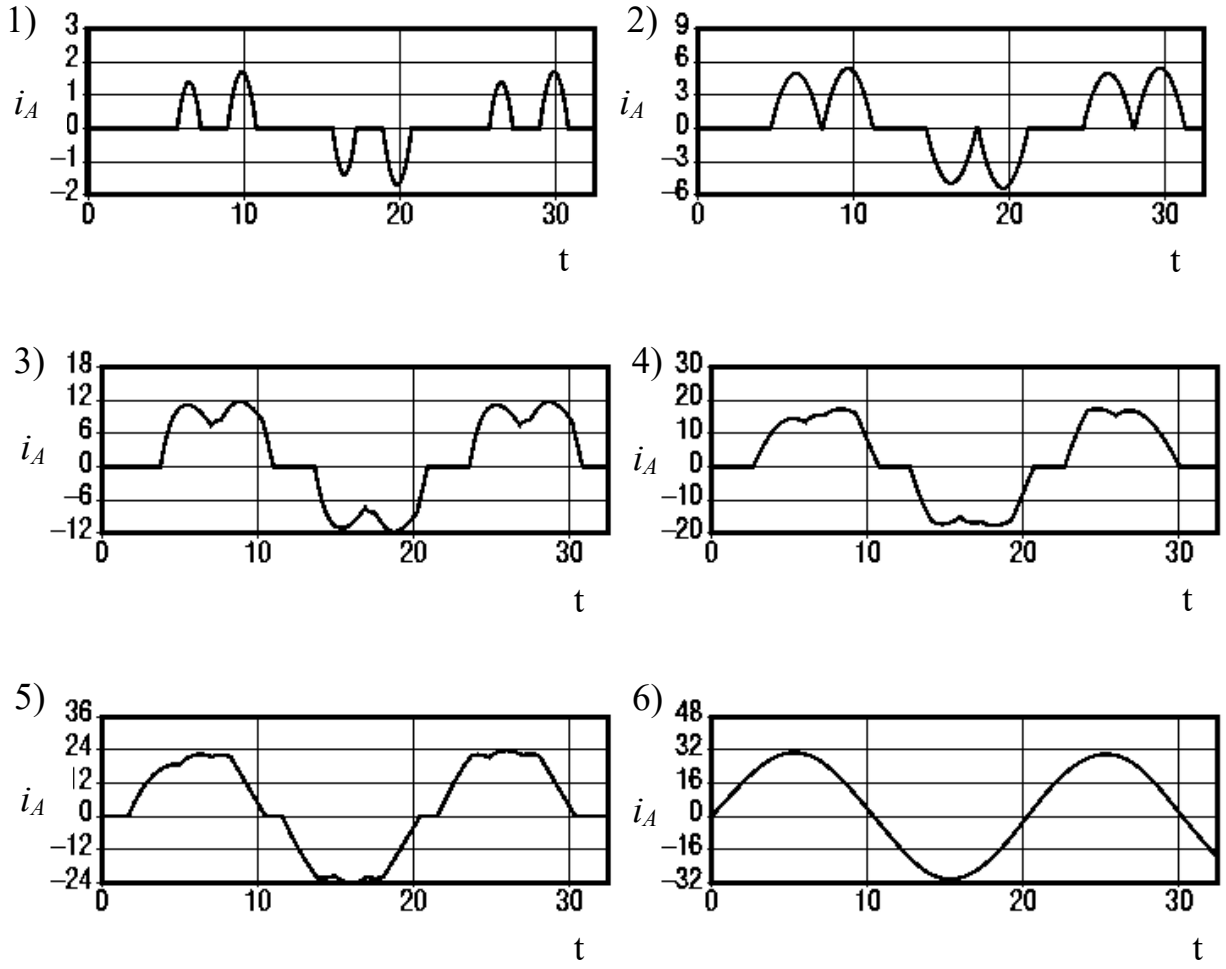


Рисунок 4 – Диаграмма тока в одной фазе двигателя при пуске

Управление тиристорами требует формирования точных временных импульсов, синхронизированных с фазами питающего напряжения, что предъявляет определённые требования к системе управления. Применение микроконтроллеров позволяет реализовать гибкие алгоритмы регулирования, адаптированные к различным условиям пуска. В таких алгоритмах может использоваться обратная связь по току, напряжению или скорости, а также различные схемы изменения α (линейные, ступенчатые, по заданной функции). Отладка этих алгоритмов требует высокой точности и воспроизводимости, что затруднительно при работе исключительно с физическим оборудованием.

С этой точки зрения применение программного моделирования, в частности моделирования микроконтроллера, управляющего УПП, предоставляет важное преимущество. Такая модель позволяет точно воспроизвести временные характеристики формирования управляющих сигналов, учесть особенности сетевого напряжения и параметров двигателя, а также исследовать реакцию системы на изменение алгоритма регулирования. Это особенно актуально в случае использования сложных или адаптивных алгоритмов управления, где ошибки в синхронизации или расчетах могут привести к значительным отклонениям в работе системы.

Таким образом, принцип работы УПП на базе тиристорного регулятора напряжения тесно связан не только с электротехническими особенностями преобразования энергии, но и с точной реализацией управляющих алгоритмов. Моделирование микроконтроллерной части управления в программной среде MATLAB позволяет выполнить предварительную отладку всех логических и временных зависимостей, что существенно повышает надёжность и сокращает сроки разработки готового устройства.

2 РАЗРАБОТКА МОДЕЛИ МИКРОКОНТРОЛЛЕРА В MATLAB

Современные микроконтроллеры (МК) широко используются в различных областях, от автоматизации процессов до встраиваемых систем. Однако, тестирование и отладка программного обеспечения для МК требуют наличия физического оборудования, что может быть дорогостоящим и неудобным. Одним из способов решения этой проблемы является использование программных симуляторов микроконтроллеров, позволяющих запускать и отлаживать код без необходимости в физическом устройстве. Рассмотрим некоторые из существующих решений.

QEMU (Quick Emulator) – это универсальный симулятор, который поддерживает множество архитектур, включая ARM, на которой основаны микроконтроллеры STM32. Он используется для симуляции не только микроконтроллеров, но и целых компьютерных систем, что делает его очень гибким инструментом. Возможна интеграция QEMU с MATLAB через дополнительный пакет Embedded Coder Interface to QEMU Emulator. Этот пакет позволяет развертывать скомпилированные приложения на эмуляторе QEMU, однако он не поддерживает симуляцию всей периферии микроконтроллера и требует настройки внешнего режима (External Mode) для взаимодействия с Simulink. Также он не поддерживает отладку исходного кода, написанного на языке С. Это добавляет сложности в процессе моделирования и отладки [5].

Другим инструментом является Proteus, система автоматизированного проектирования, которая позволяет моделировать не только микроконтроллеры, но и электронные схемы, в которые они встроены. Proteus поддерживает визуализацию работы схем и моделирование различных сценариев работы устройства, что позволяет проводить более точные симуляции. Однако, как и QEMU, Proteus не предоставляет полной симуляции всех периферийных устройств. Proteus также требуется уже скомпилированный файл прошивки для микроконтроллера, что ограничивает гибкость в процессе отладки [6].

Таким образом, несмотря на наличие различных средств симуляции микроконтроллеров, многие из них обладают рядом ограничений, которые усложняют процесс тестирования и отладки. Наиболее существенные проблемы связаны с ограниченной поддержкой периферийных устройств, отсутствием интеграции с современными средствами моделирования и невозможностью прямой отладки исходного кода микроконтроллера в привычной среде. Это создает значительные трудности для разработчиков, особенно при работе с комплексными системами, где взаимодействие с периферией играет ключевую роль.

В этих условиях возникает потребность в более гибком и интегрированном решении, которое позволяло бы не только запускать и проверять код микроконтроллера, но и моделировать поведение периферийных устройств в едином программном окружении. Особенно перспективным в этом контексте выглядит использование среды MATLAB – мощного инструмента для математического моделирования и анализа, широко применяемого в задачах управления и моделирования электротехнических систем.

С учётом вышеуказанных ограничений было принято решение о разработке собственного симулятора микроконтроллера, ориентированного на выполнение управляющих программ непосредственно в MATLAB. Предложенное в данной работе решение позволяет моделировать как выполнение кода МК, так и работу основных периферийных устройств, что существенно упрощает процесс отладки и ускоряет цикл разработки [7].

Основным преимуществом разработанного симулятора является его способность интегрировать программу на языке C с моделированием периферийных устройств. В отличие от существующих решений, таких как QEMU или Proteus, данное решение позволяет осуществлять отладку исходного кода, написанного на языке C, непосредственно в MATLAB, а также использовать пользовательскую реализацию периферии. Это открывает возможности для упрощенной симуляции периферийных устройств, которые

не оказывают критического влияния на функционирование программного обеспечения микроконтроллера.

Для реализации модели МК в ходе исследования были использованы следующие методы и инструменты:

- MATLAB/Simulink для создания и интеграции модели МК.
- для отладки программы МК в Simulink и компилятор Microsoft Visual C++ (MSVC) для компиляции программы МК для MATLAB.
- стандартные библиотеки и драйверы для STM32, адаптированные для MATLAB.

Программы для большинства микроконтроллеров разрабатываются на языке C. В MATLAB имеется блок S-Function, который позволяет интегрировать функции, написанные как на языке C/C++, так и на MATLAB. Это решение и было выбрано для портирования программы микроконтроллера в среду MATLAB. Важно отметить, что для компиляции исходного кода S-Function используется компилятор MSVC, который является сторонним. Однако его интеграция с MATLAB позволяет использовать все возможности среды для разработки и отладки программного обеспечения микроконтроллеров.

Таким образом, предложенная модель МК обеспечивает интегрированную среду для моделирования работы микроконтроллеров с возможностью отладки на уровне исходного кода и симуляции периферийных устройств, что делает процесс разработки более эффективным и удобным для инженеров и разработчиков.

Основная сложность моделирования программ МК в MATLAB заключается в различиях в принципах работы программ на микроконтроллере и в среде симуляции MATLAB. Программы, выполняющиеся на микроконтроллерах, обычно представляют собой бесконечный цикл. В каждой итерации этого цикла программа МК считывает данные с портов ввода-вывода, обрабатывает их и формирует выходные сигналы на тех же портах. В отличие от этого, S-Function в MATLAB представляет собой

функцию, которая вызывается на каждом шаге симуляции. Эта функция принимает входные данные, выполняет необходимые расчеты и формирует выходные сигналы. В процессе выполнения MATLAB ожидает завершения функции, и только после этого формируются выходные данные и происходит переход к следующему шагу симуляции. Таким образом, если S-Function включает бесконечный цикл, то симуляция просто зависнет, поскольку MATLAB будет ожидать завершения выполнения, которое никогда не наступит.

Кроме того, микроконтроллеры имеют доступ к периферийным устройствам, которые реализованы на аппаратном уровне. Программы на МК могут взаимодействовать с такими устройствами, как таймеры, АЦП, UART и другие, через их регистры и прерывания. В MATLAB же отсутствуют аппаратные устройства и регистры, что приводит к возникновению проблем при моделировании. Например, если программа МК в MATLAB ожидает выставления флага переполнения у таймера или флага приема байта по UART, то симуляция зависнет, так как этот флаг аппаратно не выставится.

В результате предложена структурная схема для реализации модели микроконтроллера, которая приведена рисунке 5. Схема отображает основные компоненты модели: S-Function, оболочка программы МК, симулятор периферии и формирование выходных сигналов.

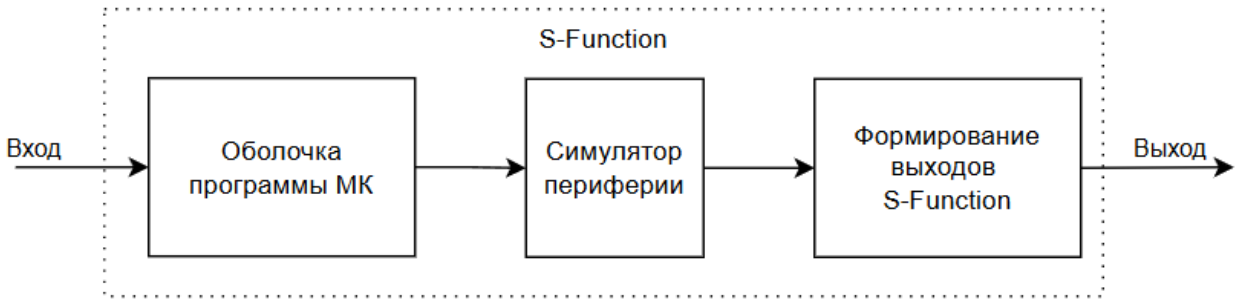


Рисунок 5 – Структурная схема модели микроконтроллера в MATLAB

Блок S-Function реализует интерфейс между моделью Simulink и моделью микроконтроллера. Он принимает входные сигналы из модели и передаёт их в оболочку.

Оболочка программы микроконтроллера отвечает за выполнение пользовательского кода. Получив входные данные от блока S-Function, она считывает принятые данные и инициирует запуск программы микроконтроллера.

После завершения исполнения пользовательского кода запускается симулятор периферийных устройств. Эти симуляторы реализованы в виде отдельных модулей на языке C (.c/.h файлов), которые компилируются вместе с остальной частью оболочки и программы. Их задача заключается в имитации поведения периферийных модулей, таких как таймеры, АЦП, UART и др. Симуляторы реагируют на обращения со стороны пользовательского кода и изменяют своё состояние в соответствии с внутренней логикой и текущими значениями регистров.

На финальном этапе формируются выходные сигналы, полученные как в результате выполнения пользовательской программы, так и в результате симуляции периферии. Эти сигналы возвращаются в модель Simulink и могут быть использованы для управления системой, визуализации внутренних процессов программы, анализа или дальнейшей обработки.

Таким образом, для того чтобы симулировать работу МК в MATLAB, необходимо решить две задачи:

- контролировать выполнение программы МК, чтобы она могла завершить свою работу в определенный момент и перейти к следующему шагу симуляции;
- реализовать симулятор периферии, который будет имитировать работу периферийных устройств, чтобы программа могла взаимодействовать с ними.

2.1 УПРАВЛЕНИЕ ВЫПОЛНЕНИЕМ ПРОГРАММЫ МИКРОКОНТРОЛЛЕРА

Для управления выполнением программы микроконтроллера в среде MATLAB была реализована оболочка программы МК, основанная на использовании многопоточности и механизма S-Function. Функциональная схема алгоритма приведена на рисунке 6.

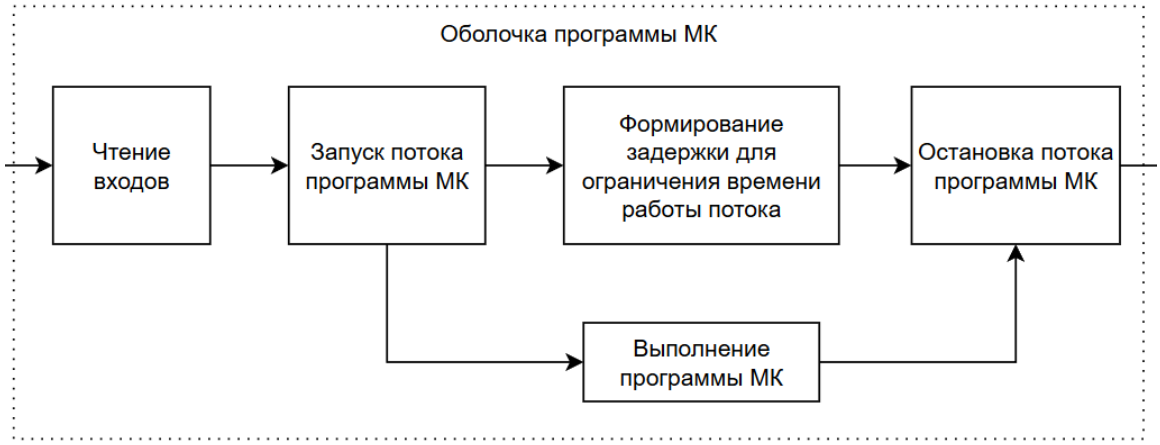


Рисунок 6 – Функциональная схема оболочки программы МК

При инициализации создается поток, запускающий код МК (главную функцию main).

```
/**
  * @brief      Главная функция приложения МК.
  * @details    Функция с которой начинается выполнение кода МК. Выход из данной
функции происходит только в конце симуляции @ref mdlTerminate
  */
extern int main(void);                  // extern while from main.c
/**
  * @brief      Поток приложения МК.
  * @details    Поток, который запускает и выполняет код МК (@ref main).
  */
unsigned __stdcall MCU_App_Thread(void) {
    main();     // run MCU code
    return 0;   // end thread
    // note: this return will reached only at the end of simulation, when all whiles
will be skipped due to @ref sim_while

/**
  * @brief      Инициализация симуляции МК.
  * @details    Пользовательский код, который создает поток для приложения МК
                и настраивает симулятор МК для симуляции.
  */
void SIM_Initialize_Simulation(void)
{
    // инициализация потока, который будет выполнять код МК
    hmcu.hMCUThread = (HANDLE)CreateThread(NULL, 0, MCU_App_Thread, 0,
CREATE_SUSPENDED, &hmcu.idMCUThread);
}
```

На каждом шаге симуляции оболочка инициирует запуск потока, в котором выполняется код микроконтроллера. Чтобы гарантировать завершение выполнения шага и избежать бесконечных зависаний, в поток встроена логика, ограничивающая время работы программы МК в рамках одного шага.

```
    ResumeThread(hmcu.hMCUThread);
    for (int i = DEKSTOP_CYCLES_FOR_MCU_APP; i > 0; i--) { }
    SuspendThread(hmcu.hMCUThread);
```

Это обеспечивает корректное взаимодействие непрерывного исполнения микроконтроллера с дискретным шагом симуляции.

При реализации данной схемы особое внимание уделяется корректному завершению работы потока программы в конце симуляции. Поскольку большинство микроконтроллерных приложений организовано в виде бесконечного цикла (while(1)), требуется обеспечить выход из этого цикла и из функции MCU_App_Thread по внешнему признаку завершения моделирования. Для этого используется переопределённый макрос while, в котором дополнительно проверяется флаг окончания симуляции. Когда моделирование завершено, все циклы немедленно пропускаются, а поток достигает конца функции main, завершая свою работу корректным образом.

```
  /**
    * @brief      While statement for emulate MCU code in Simulink.
    * @param      _expression_ - expression for while.
    * @details    Данный while необходим, чтобы в конце симуляции, завершить поток МК:
    *             При выставлении флага окончания симуляции, все while будут
    *             пропускаться и поток сможет дойти до конца функции main и завершиться
    */
#define sim_while(_expression_)     while((_expression_)&&(hmcu.fMCU_Stop == 0))
```

Описанная архитектура обеспечивает эффективное управление выполнением программы микроконтроллера, позволяя интегрировать непрерывный код микроконтроллера в пошаговый механизм симуляции Simulink и корректно завершать работу при остановке моделирования. Для реализации этой концепции в среде MATLAB была разработана программная оболочка, состоящая из нескольких ключевых модулей. Каждый из них

отвечает за отдельный аспект взаимодействия модели микроконтроллера с Simulink. Далее будет рассмотрено назначение и структура каждого из четырёх основных файлов, формирующих данную оболочку.

Файл “MCU.c” представляет собой основной исходный файл реализации S-Function в составе среды моделирования Simulink. Он отвечает за интеграцию модели микроконтроллера в цикл дискретной симуляции, выполняемой MATLAB/Simulink, и обеспечивает вызов основных процедур моделирования, описанных в отдельных модулях оболочки.

Функция mdlUpdate вызывается на каждом шаге моделирования и инициирует переход модели микроконтроллера в новое состояние, вызывая функцию MCU_Step_Simulation. Выходные данные S-Function формируются в функции mdlOutputs, где вызывается процедура SIM_writeOutputs, обеспечивающая передачу результатов симуляции во внешние выходные порты модели.

Функции mdlInitializeSizes и mdlInitializeSampleTimes определяют структуру входных и выходных портов, число состояний, временные параметры, а также настраивают размеры рабочих векторов. Начальная инициализация симуляции осуществляется в функции mdlStart, где вызывается SIM_Initialize_Simulation. По завершении моделирования вызывается функция mdlTerminate, в которой выполняется завершение симуляции и освобождение занятых ресурсов

Таким образом, файл “MCU.c” реализует S-Function. Он обеспечивает запуск симуляции, вызов функций модели на каждом временном шаге, а также завершение моделирования, действуя как интерфейс между Simulink и пользовательской оболочкой микроконтроллера.

Файл “mcu_wrapper.c” реализует функции взаимодействия между моделью Simulink и симулируемой программой микроконтроллера. Он выполняет роль программной оболочки, обеспечивая запуск, приостановку, выполнение и завершение потока моделируемого кода, а также обмен входными и выходными сигналами через S-Function.

В этом файле определён глобальный объект hmcu – дескриптор управления симуляцией, содержащий параметры симуляции и системное время. Если включена многопоточность, создаётся отдельный поток, выполняющий функцию main() пользователя – основную программу микроконтроллера, иначе пользователь сам определяет какие функции программы вызывать на каждом шаге симуляции.

Основная логика симуляции реализуется в функции MCU_Step_Simulation, вызываемой на каждом шаге моделирования. В ней выполняется имитация тактов системного времени, считывание входов, симуляция периферийных устройств, выполнение пользовательского кода, а затем сохранение выходных значений. Чтение входных данных из модели Simulink реализовано в MCU_readInputs, а передача выходных сигналов в буфер – в MCU_writeOutputs. Конечная запись выходных данных в блок S-Function выполняется функцией SIM_writeOutputs.

Функции SIM_Initialize_Simulation и SIM_deInitialize_Simulation отвечают за начальную и конечную инициализацию симуляции соответственно: настройку периферийной среды, запуск пользовательского кода и освобождение всех ресурсов по завершении моделирования.

Таким образом, “mcu_wrapper.c” реализует программную оболочку микроконтроллера и управляет выполнением его кода внутри среды моделирования. Он отвечает за запуск, приостановку и симуляцию пользовательской программы, а также за обмен данными с Simulink через порты ввода-вывода, позволяя тестировать встроенное ПО без физического микроконтроллера.

Файл “mcu_wrapper_conf.h” содержит основные конфигурационные параметры и определения, управляющие поведением оболочки микроконтроллера в процессе симуляции. Он включает набор #define-макросов, задающих структуру входов и выходов S-Function, размеры портов и массивов дискретных состояний, параметры тактирования и опции завершения симуляции. Также в файле определены структуры и типы,

необходимые для управления потоком выполнения программы микроконтроллера, в том числе переменные для счёта тактов и времени моделирования. Дополнительно реализованы макросы, позволяющие безопасно использовать циклы while в условиях симуляции – они учитывают флаг завершения моделирования и обеспечивают корректное поведение программы при остановке.

Таким образом, “mcu_wrapper_conf.h” позволяет настроить оболочку для микроконтроллерного кода под требования среды Simulink и управляет общими параметрами виртуального исполнения.

Файл “run_mex.bat” представляет собой сценарий Windows-командной строки, предназначенный для автоматизации процесса компиляции исходного кода микроконтроллерного приложения в формат MEX-функции, совместимый с MATLAB/Simulink. Он формирует команду компиляции с использованием mex, задавая все необходимые параметры: директивы препроцессора, пути к пользовательским заголовочным файлам и библиотекам, а также перечни исходных файлов – как из основного проекта пользователя, так и из вспомогательной оболочки для запуска этого проекта в MATLAB, включая модифицированные для симуляции версии библиотек HAL и CMSIS. Поддерживается режим отладки, который активируется передачей аргумента debug и добавляет символы отладки в результат компиляции.

Вся организация выполнения модели МК нацелена на обеспечение корректного взаимодействия с моделью Simulink и обеспечения полной управляемости со стороны симулятора. Такой подход позволяет реализовать комплексную симуляцию, в которой код микроконтроллера, написанный на языке С и предназначенный для выполнения на реальном устройстве, может быть отлажен и протестирован в виртуальной среде MATLAB без модификации основной логики программы.

В результате, оболочка микроконтроллера состоит из четырёх файлов, листинг которых приведен в приложении А.

2.2 МОДЕЛИРОВАНИЕ ПЕРИФЕРИЙНЫХ УСТРОЙСТВ МИКРОКОНТРОЛЛЕРА

Симуляция периферийных устройств представляет собой второй важный аспект моделирования, требующий детальной проработки. Микроконтроллеры взаимодействуют с аппаратными компонентами через регистры и прерывания, что требует эмуляции этих взаимодействий в MATLAB. Функциональная схема симулятора периферии приведена на рисунке 7. Необходимо реализовать три аспекта: симуляцию системных часов, симуляцию необходимой периферии и вызов соответствующих прерываний, формирование буфера выходных сигналов.

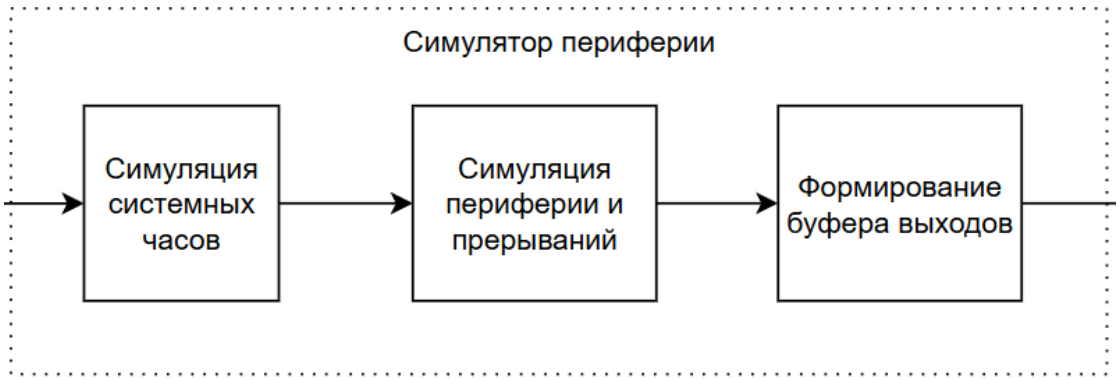


Рисунок 7 – Функциональная схема симулятора периферии

С симуляцией системных часов всё просто. Модель имеет параметр частота тактирования моделируемого микроконтроллера, и через неё легко рассчитать тики системных часов.

Отсутствие аппаратной периферии вынуждает писать модели для используемой периферии вручную. Можно максимально подробно прописать логику работы периферии, чтобы она полностью эмулировала реальную периферию МК. Или же можно написать упрощенную или абстрактную модель, которая не будет имитировать всех процессов в периферии, но её будет достаточно для симуляции программы.

Такой подход предоставляет гибкость в разработке: можно выбрать, какие части периферии симулировать детально, а какие оставить

упрощенными или вовсе исключить из модели. Это позволяет оптимизировать симуляцию в зависимости от конкретных потребностей тестирования и уменьшить вычислительные ресурсы, требуемые для симуляции.

Также все регистры микроконтроллера привязаны к определенным адресам памяти, которые недоступны в MATLAB. Поэтому необходимо выделить специальную область оперативной памяти для хранения регистров периферийных устройств. Также нужно переопределить макросы в библиотеках для работы с МК, чтобы обращение к регистрам периферии и происходило через новые адреса в выделенной памяти.

В данной работе рассматриваются программа устройства плавного пуска, в которой активно используются три периферийных устройства: GPIO, таймеры и UART.

Реализация GPIO не требует сложной логики, так как ее работа сводится к прямой передаче сигналов на порты ввода/вывода. Или же передачи отладочной информации в буфер выходных сигналов S-Function. Для просмотра отладочной информации в Simulink модели.

Таймеры, используемые для отсчета временных задержек и формирования широтно-импульсной модуляции (ШИМ), требуют высокой точности симуляции. Это обусловлено их ключевой ролью в управлении двигателем, где точность их работы напрямую влияет на эффективность работы системы.

Поэтому в модели симулятора микроконтроллера для MATLAB таймеры реализованы на уровне имитации поведения реальных периферийных устройств STM32. В основе лежит идея прямой эмуляции регистров и логики работы встроенных таймеров, характерной для микроконтроллеров STM32, включая поддержку различных режимов работы и взаимодействие с другими подсистемами (GPIO, прерывания и др.).

Каждый таймер представлен в симуляторе структурой, соответствующей TIM_TypeDef, как в CMSIS-библиотеках. Эмулятор

работает с этими регистрами напрямую: каждый шаг симуляции вызывает функцию TIM_Simulation, которая выполняет следующие ключевые действия:

- проверка переполнения таймера (Overflow_Check) – реализует поведение логики обновления счётчика и вызова прерываний, если достигнут предел (ARR) или значение счётчика выходит за границы (вверх/вниз);
- определение режима работы – симулятор анализирует содержимое регистра SMCR и выбирает поведение таймера: обычный счёт или счёт по внешнему триггеру с учетом синхронизации;
- обновление счётчика (CNT) – учитывает направление счёта (CR1.DIR), частоту (делитель PSC) и величину шага симуляции tx_step;
- симуляция выходных каналов – для каждого из 4 каналов реализованы отдельные функции, интерпретирующие содержимое регистров CCMRx, CCRn и CCER и формирующие значения выходов OCnREF.

Симулятор поддерживает почти все базовые режимы работы каналов OCx таймера:

- PWM Mode 1 и PWM Mode 2 – основные режимы генерации ШИМ, учитывающий сравнение CNT и CCRn;
- Toggle Mode – переключение логического состояния при совпадении счётчика и значения CCR;
- Active / Inactive Level Mode – установка выхода в 1 или 0 при срабатывании сравнения;
- Forced Output Modes – форсированная установка выхода вне зависимости от счётчика.

Каждое изменение OCxREF при необходимости приводит к изменению состояния соответствующих выводов GPIO – если включён альтернативный режим вывода и активирован соответствующий канал через CCER. Реализация выхода PWM/OC на пин GPIO осуществляется через проверку режима пина и состояния полярности канала. В зависимости от этого значение OCxREF

записывается в ODR соответствующего виртуального порта, симулируя фактическое состояние на ножке микроконтроллера.

В текущей реализации симулятора поддерживается:

- Обычный счётчик (вверх/вниз);
- Slave-режим по внешнему триггеру (Trigger Mode) – с возможностью запуска по синхронизирующему сигналу от другого таймера;
- Прерывания по переполнению (вызов обработчика);
- Выходы OC1–OC4 в режимах PWM и Toggle;
- Управление направлением и частотой счёта через DIR, PSC, ARR;
- Работа preload (ARPE) и маскировки обновлений (UDIS).

Таким образом, симуляция таймеров обеспечивает реалистичное поведение встроенных таймеров STM32, позволяя моделировать ШИМ, управление периферией по событиям, синхронизацию между таймерами и генерацию сигналов на выходы, что критично для задач управления электродвигателями.

Интерфейс UART в данной системе используется для приема параметров ШИМ по протоколу Modbus. В процессе симуляции работа с этим интерфейсом была упрощена до считывания входных данных через S-Function и записи их напрямую в регистры Modbus. Такой подход не только обеспечивает корректную работу программы, но и позволяет передавать параметры для управления непосредственно из MATLAB. Это исключает необходимость моделировать всю сложную логику работы интерфейса и протокола Modbus, что снижает нагрузку на симулятор. Реализация модуля UART была, по сути, перенесена в GPIO, что позволило упростить взаимодействие и минимизировать требования к симулятору. Такой подход оправдан, поскольку точная симуляция низкоуровневого UART не критична для функциональности управления двигателем в рамках данной задачи.

Модуль для симуляции входов/выходов микроконтроллера и модуль, реализующий симуляцию таймеров, представлен в приложении Б.

2.3 РАЗРАБОТАННАЯ МОДЕЛЬ МИКРОКОНТРОЛЛЕРА В MATLAB

Итоговая модель микроконтроллера в MATLAB имеет функциональную схему, приведенную на рисунке 8.

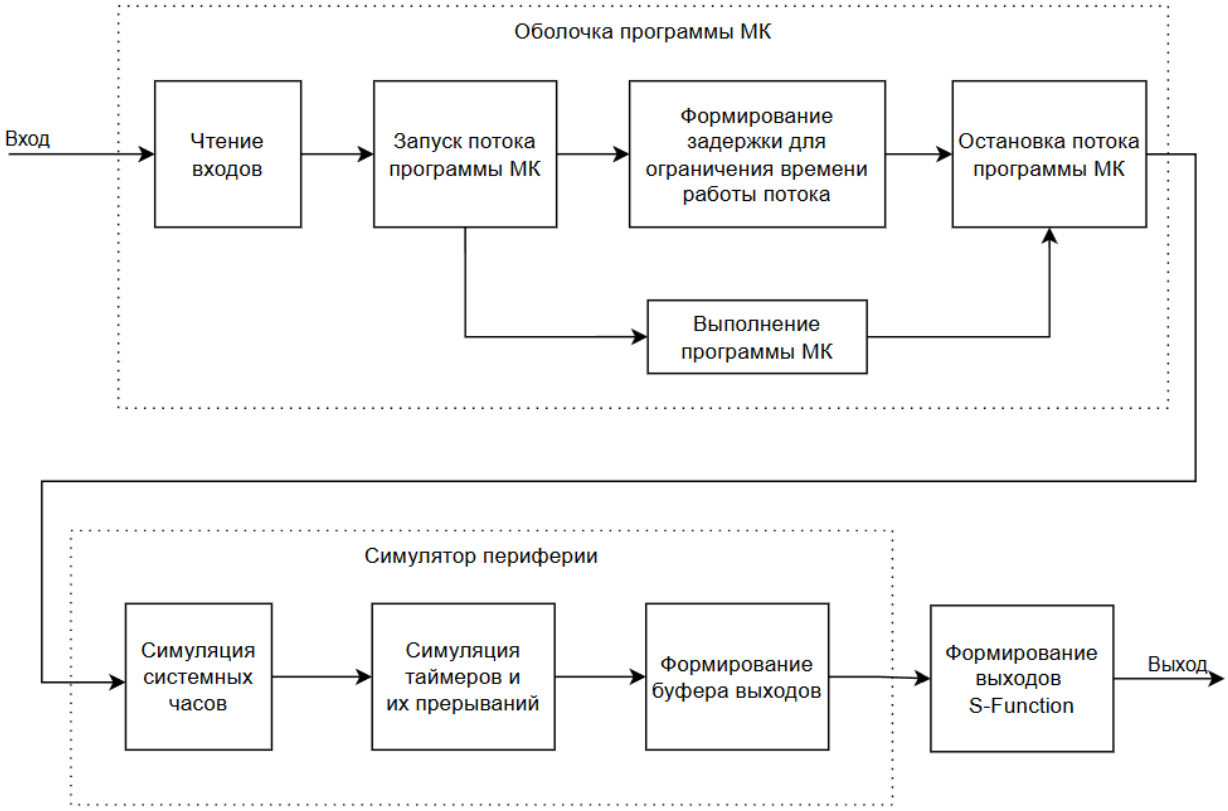


Рисунок 8 – Функциональная схема модели микроконтроллера в MATLAB

Был спроектирован подблок, приведенный на рисунке 9. Он включает в себя S-Function, которая принимает входной вектор сигналов, и несколько выходов: для симуляции GPIO и разной отладочной информации.

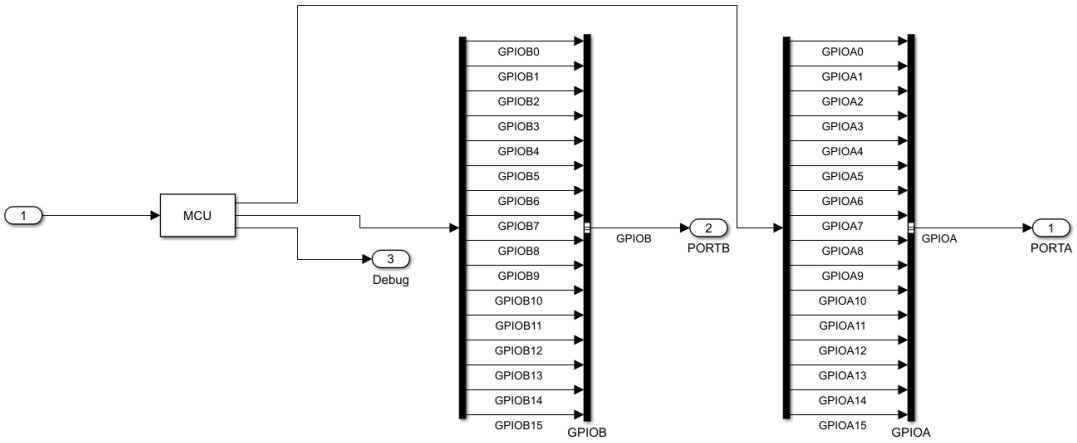


Рисунок 9 – Блок, имитирующий микроконтроллер в MATLAB

Разработанная модель микроконтроллера для среды MATLAB предназначена, прежде всего, для инженерных и научных задач, связанных с отладкой микроконтроллерного программного обеспечения в условиях виртуального моделирования. Она позволяет выполнять С-код, изначально написанный для микроконтроллера STM32, непосредственно в MATLAB, с полной или частичной эмуляцией периферийных устройств и возможностью интеграции с моделью внешней электроники в среде Simulink. Такой подход обеспечивает совместную отладку программной и аппаратной частей разрабатываемого устройства на ранних этапах проектирования, без необходимости физического оборудования.

Модель реализована в виде набора исходных файлов и библиотек, организованных по следующим основным папкам:

- Code – исходный пользовательский код для МК;
- MCU_Wrapper – оболочка, реализующая механизм запуска программы и её интеграцию в Simulink посредством S-Function;
- MCU_STM32xxx_Matlab – модифицированные драйверы STM32 и вспомогательные модули для эмуляции периферии и поддержки компиляции в среде MSVC.

Для эффективного использования симулятора пользователю требуются базовые навыки в следующих областях:

- знание языка программирования C, включая структуру типового проекта для STM32;
- понимание архитектуры STM32 и основных принципов работы встроенной периферии (таймеры, GPIO, UART, ADC и др.);
- владение средой MATLAB/Simulink на уровне создания простых моделей и работы с блоками S-Function;
- способность ориентироваться в структуре проекта и понимать принципы компиляции кода вне IDE (использование скриптов).

Для того чтобы интегрировать код микроконтроллера в MATLAB, используется механизм компиляции с помощью скрипта run_mex.bat. Этот файл содержит все необходимые параметры компиляции: список путей к заголовочным файлам, список исходников. Настройка run_mex.bat требует понимания структуры проекта и знания, какие именно модули следует компилировать.

Настройка параметров симуляции осуществляется в конфигурационном файле mcu_wrapper_conf.h, где задаются ключевые параметры модели: частота тактирования микроконтроллера, размерность векторов входов-выходов и ограничения по времени выполнения потока. Сборка кода выполняется запуском run_mex.bat, после чего, при успешной компиляции, создаётся MEX-файл, который автоматически подключается к модели в Simulink.

При портировании проектов могут возникать ошибки при компиляции или запуске модели Simulink, связанные с различиями в компиляторах, отсутствием поддержки weak-функций, попытками доступа по некорректным адресам, отсутствием в MSVC нужных макро-определений. Такие ошибки устраняются через модификацию библиотек микроконтроллера, создание dummy-функций, либо временное исключение фрагментов кода, не имеющих значения для текущей симуляции.

В результате, созданная модель симулятора представляет собой инструмент для тестирования и пошаговой отладки микроконтроллерных приложений, обеспечивая гибкую интеграцию с моделями внешней среды.

Это позволяет значительно сократить цикл отладки, снизить зависимость от оборудования и повысить повторяемость экспериментов.

Но у модели есть и недостаток: отсутствие учета временных затрат на выполнение инструкций управляющей программы. В реальном микроконтроллере выполнение кода сопровождается затратой времени на исполнение каждой машинной инструкции, определяемой частотой тактирования, архитектурой процессора и особенностями используемой периферии. Эти затраты напрямую влияют на динамику работы системы, особенно в контексте быстропротекающих переходных процессов, конкуренции прерываний, а также при использовании программных задержек.

В отличие от этого, в имитационной модели, реализованной с использованием механизма S-Function в среде MATLAB/Simulink, выполнение управляющего кода происходит вне привязки к реальной временной шкале. На каждом дискретном шаге моделирования осуществляется запуск кода (или его части в рамках логики симуляции) на определенное количество тактов, при этом предполагается, что выполнение происходит мгновенно относительно длительности шага симуляции. Таким образом, модель не учитывает, сколько времени фактически потребовалось бы микроконтроллеру на выполнение соответствующего участка программы.

Это допущение обеспечивает простоту реализации и повышает универсальность модели, однако накладывает ограничения на её применимость при анализе процессов, чувствительных к временным задержкам, таких как гонки прерываний, перегрузка по вычислительной нагрузке, или корректность выполнения критических временных участков кода. Модель адекватно отражает логическую структуру управляющего алгоритма и взаимодействие с периферийными интерфейсами, но не предназначена для оценки реальных временных характеристик исполнения кода на микроконтроллере.

3 РАЗРАБОТКА МИКРОКОНТРОЛЛЕРНЫХ ПРИЛОЖЕНИЙ В MATLAB

Разработка микроконтроллерных приложений в среде MATLAB на основе предложенного симулятора позволяет производить поэтапную отладку и тестирование пользовательской программы в контексте моделируемого оборудования. Такой подход позволяет на ранних стадиях убедиться в корректной работе кода, оценить реакцию системы на внешние воздействия, а также выявить и устранить возможные ошибки до тестирования программы на реальном микроконтроллере.

Благодаря полной интеграции симулятора с MATLAB становится возможным отслеживание внутренних состояний микроконтроллера, визуализация результатов и отладка без участия физического оборудования.

Кроме того, симулятор может использоваться для проверки уже готовых алгоритмов. Это особенно важно в тех случаях, когда в ходе эксплуатации системы проявляются трудноуловимые ошибки, не проявившиеся при тестировании в реальных условиях. Моделирование таких ситуаций в среде MATLAB позволяет детально воспроизводить различные ситуации и анализировать поведение алгоритма, внутренние состояния программы и выявить причины некорректной работы.

Таким образом, модель микроконтроллера в среде MATLAB служит как средством поэтапной разработки, так и инструментом последующего анализа.

3.1 РАЗРАБОТКА SIMULINK МОДЕЛИ УСТРОЙСТВА ПЛАВНОГО ПУСКА

Для построения модели устройства плавного пуска была использована среда MATLAB Simulink с пакетом Simscape. Simscape ориентирован на математическое и физическое блочное моделирование в конкретных, но довольно широких областях науки и техники – электричестве, электронике, механике, электромагнетизме, пневматике, гидравлике и термодинамике. Моделирование происходит на уровне фундаментальных физических процессов. Библиотека пакета Simscape находится в библиотеке пакета блочного имитационного моделирования Simulink [8].

Простейшая структурная схема устройства плавного пуска приведена на рисунке 10. На схеме: ТМ – тиристорный модуль; АД – асинхронный электродвигатель; БК – блок компараторов, для определения перехода через ноль; МК – микроконтроллер, содержащий алгоритм управления УПП; ПУ – пульт управления.

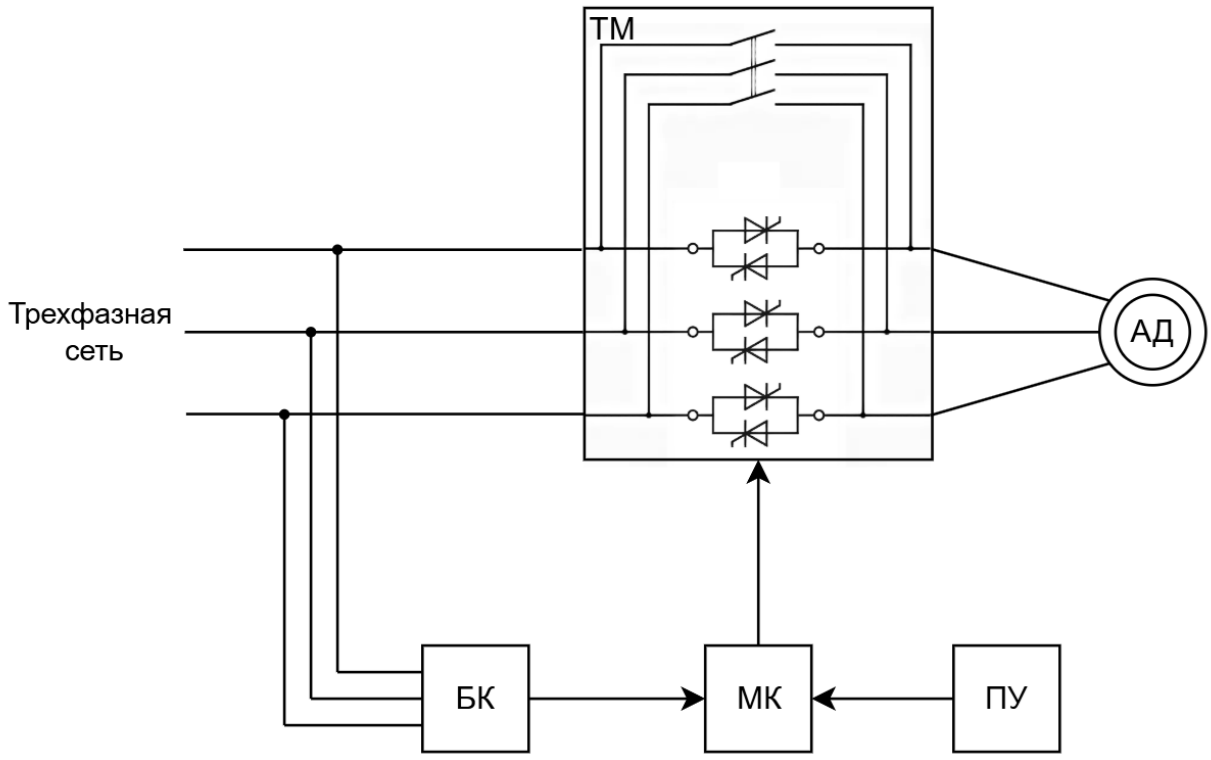


Рисунок 10 – Структурная схема УПП

Для реализации тиристорного модуля был создан подблок Thyristors. Его схема в Simulink представлена на рисунке 11. Блок обеспечивает интерфейс между электрической частью модели и управляющей логикой. К его входам относятся стандартные силовые контакты среды Simscape (A, B, C – входные фазы, a, b, c – выходы на нагрузку), а также два управляющих сигнала:

- g – отвечающий за подачу управляющего импульса на тиристоры;
- Shunt – позволяет шунтировать тиристор, подключив нагрузку напрямую к источнику питания, минуя управление по углу открытия.

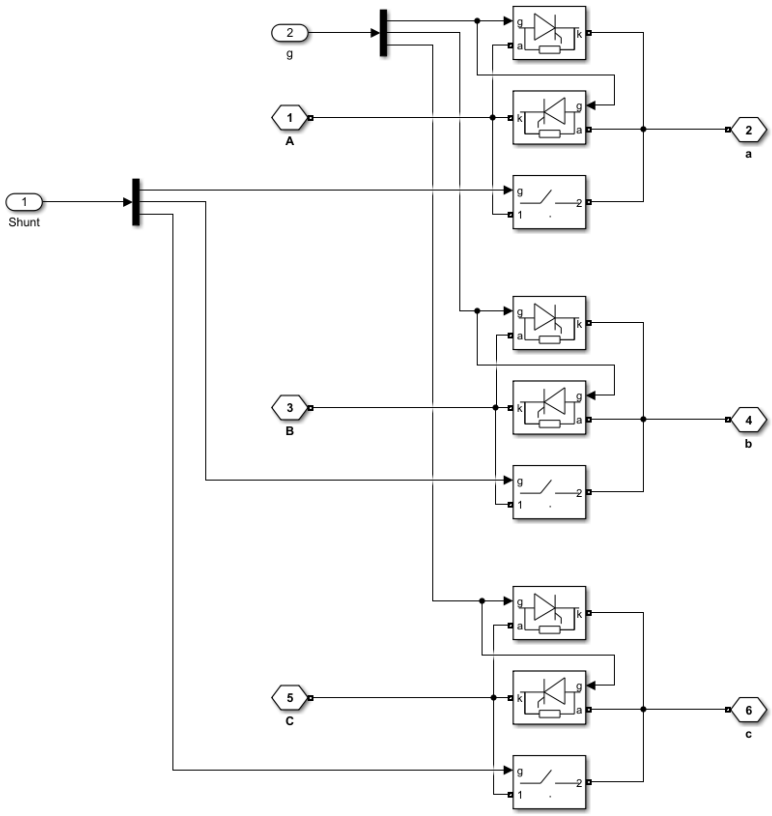


Рисунок 11 – Модель тиристорного модуля

Для проверки работы УПП и отладки алгоритма управления была собрана предварительная схема системы, включающая микроконтроллер, тиристорный модуль и нагрузку. Эта схема (рисунок 12) использовалась для проверки работы основных функций: формирования управляющих сигналов, фиксации переходов через ноль и задания угла открытия тиристоров.

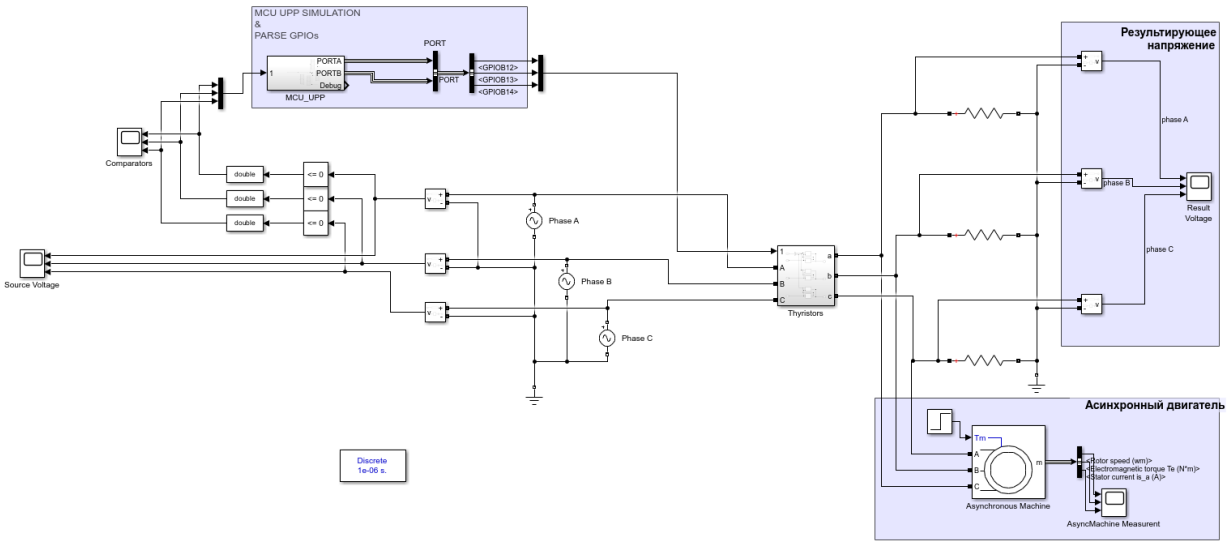


Рисунок 12 – Схема для тестирования алгоритма УПП

3.2 РАЗРАБОТКА ПРОГРАММЫ УСТРОЙСТВА ПЛАВНОГО ПУСКА

На следующем этапе работы началась реализация управляющей программы (УПП) в среде MATLAB с использованием симулятора микроконтроллера началась с создания базового шаблона проекта.

На начальном этапе разработки не ставилась задача реализовать полноценную функциональность – важно было подготовить минимально работоспособный код, запускающийся на микроконтроллере, и удостовериться, что интеграция с моделью в MATLAB выполняется корректно. Программа включала только базовые элементы: инициализацию системы и таймеров, бесконечный цикл.

После подготовки минимального шаблона код был перенесён в симуляционную среду MATLAB. Именно с этого момента начался основной этап разработки и пошаговой отладки, в процессе которого проект постепенно дополнялся необходимой функциональностью.

Блок схема алгоритма плавного пуска приведена на рисунке 13. Он состоит из нескольких этапов: определение перехода напряжения через ноль, открытие тиристора в соответствии с заданным углом открытия и плавное изменение этого угла во время пуска или остановки.

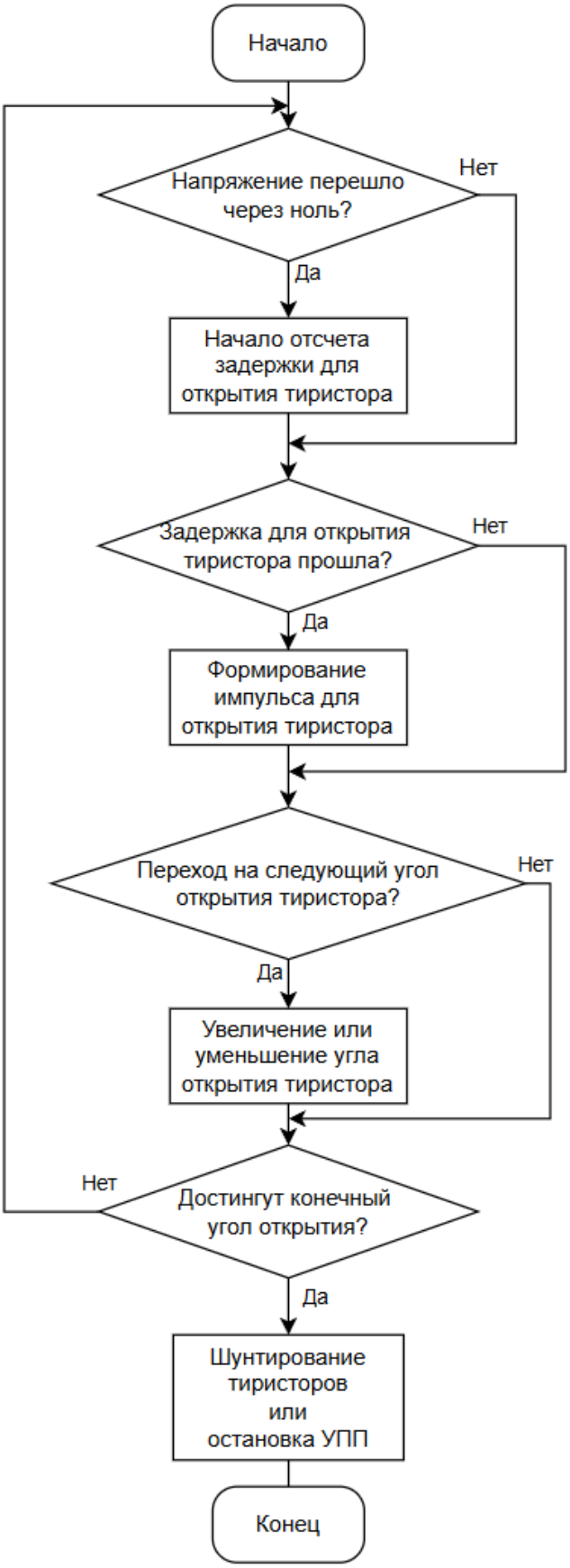


Рисунок 13 – Блок схема алгоритма плавного пуска

Сперва было решено реализовать логику определения перехода через ноль сетевого напряжения (zero-cross detection).

В функции zero_cross_update() реализована логика обновления флага ZeroCrossDetected. При использовании аппаратного метода обнаружения перезода через ноль zero_cross_update_EXTI() вызывается из обработчика прерывания EXTI и устанавливает соответствующий флаг, что обеспечивает быструю и надёжную фиксацию перехода через ноль.

```
/**
 * @brief Обновление флага перехода через ноль
 * @param zc Указатель на структуру ZeroCrossDetector_t
 * @details Просто переносим флаг EXTI с аппаратного детектора.
 */
void zero_cross_update(ZeroCrossDetector_t *zc)
{
    // Используем флаг аппаратного прерывания EXTI для установки флага
    zc->f.ZeroCrossDetected = zc->f.EXTIZeroCrossDetected;
}

/**
 * @brief Обработчик прерывания EXTI для аппаратного детектора перехода через ноль
 * @param zc Указатель на структуру ZeroCrossDetector_t
 * @details Вызывается в EXTI_IRQHandler и устанавливает флаг аппаратного перехода
через ноль.
 */
void zero_cross_update_EXTI(ZeroCrossDetector_t *zc)
{
    zc->f.EXTIZeroCrossDetected = 1;
}
```

Флаг обновляется по сигналу прерывания EXTI. Функция is_zero_cross() служит для проверки и сброса флага перехода через ноль, позволяя алгоритму реагировать на событие.

```
/**
 * @brief Проверка наличия перехода через ноль и сброс флагов
 * @param zc Указатель на структуру ZeroCrossDetector_t
 * @return int 1 - переход через ноль обнаружен, 0 - нет
 * @details Если переход обнаружен, сбрасываем флаги и возвращаем 1,
 * иначе возвращаем 0.
 */
int is_zero_cross(ZeroCrossDetector_t *zc)
{
    if(zc->f.ZeroCrossDetected)
    {
        // Сброс флагов после обнаружения
        zc->f.ZeroCrossDetected = 0;
        zc->f.EXTIZeroCrossDetected = 0;
        return 1;
    }
    return 0;
}
```

Модель для тестирования алгоритма определения перехода через ноль приведена на рис. 14. При определении перехода напряжения через ноль, программа переключает состояние ножки на противоположное. И как видно по результатам моделирования на рис. 15-16 алгоритм работает корректно.

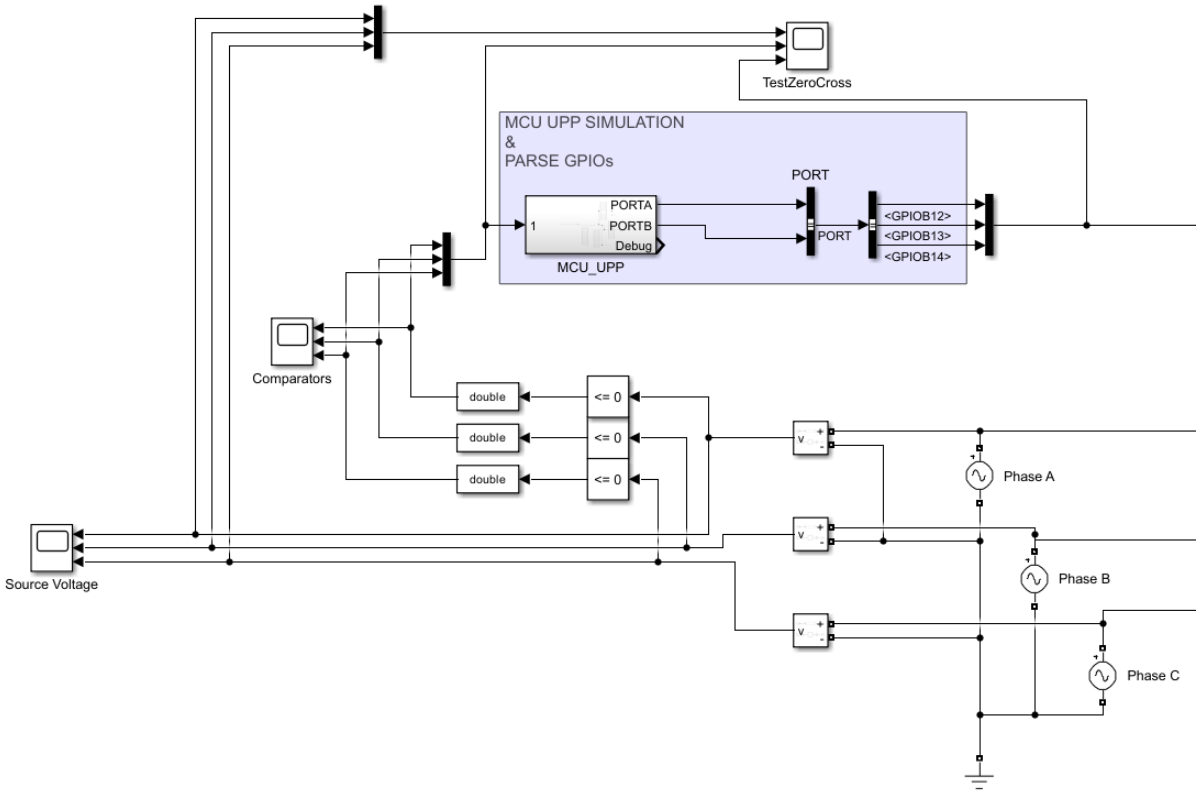


Рисунок 14 – Модель для тестирования алгоритма определения перехода через ноль

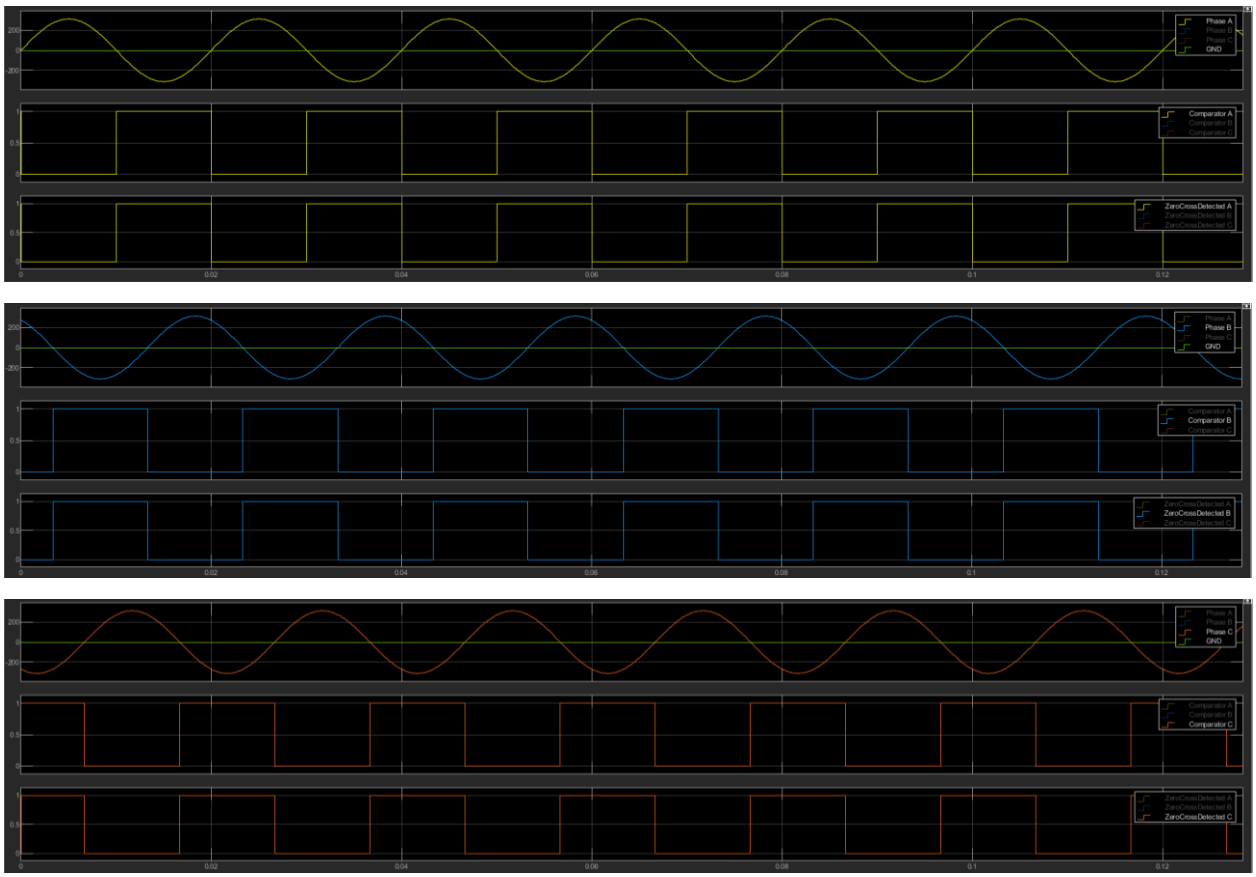


Рисунок 15 – Результаты моделирования алгоритма определения перехода через ноль для каждой фазы

На рисунке 14 на меньшем масштабе видно, что чтобы алгоритму среагировать на переход напряжения через ноль, модели необходим один шаг симуляции.

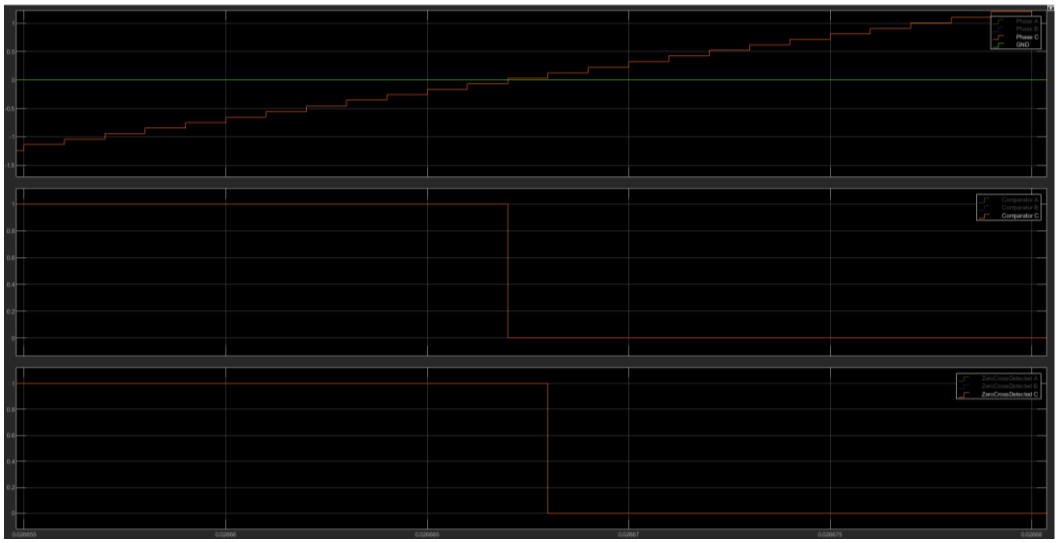


Рисунок 16 – Время определения перехода через ноль

После успешной отладки алгоритма определения перехода через ноль в проект была добавлена логика управления тиристорами, включающая несколько ключевых функций: включение тиристора, расчет угла его открытия и плавное изменение этого угла во времени.

Функция включения тиристора реализована в tiristor_control().

```
/**
 * @brief Управление состоянием тиристора (включение/выключение) по флагам и времени открытия
 * @param ctrl Указатель на структуру управления тиристором
 */
void tiristor_control(TiristorControl_t *ctrl)
{
    if(ctrl->f.EnableTiristor)  // Если разрешено включить тиристор
    {
        if(ctrl->f.TiristorIsEnable == 0)  // Если тиристор еще выключен
        {
            tiristor_enable(ctrl);           // Включить тиристор
            ctrl->enable_start_tick = HAL_GetTick();  // Запомнить время включения для отсчёта длительности открытия
        }
        else
        {
            // Если время с момента включения превысило заданное время открытия
            if(HAL_GetTick() - ctrl->enable_start_tick > ctrl->open_time)
            {
                tiristor_disable(ctrl);      // Выключить тиристор
                ctrl->f.EnableTiristor = 0;  // Снять разрешение на включение, чтобы не включался снова без команды
            }
        }
    }
    else  // Если тиристор не должен быть включен
    {
        if(ctrl->f.TiristorIsEnable)   // Если тиристор включен
            tiristor_disable(ctrl);    // Выключить тиристор
    }
}
```

Она контролирует состояние тиристора, включая его активацию и деактивацию на основе флага EnableTiristor. При первом включении соответствующий GPIO-пин устанавливается в активное состояние, подавая управляющий сигнал на затвор тиристора. По истечении заданного времени тиристор выключается.

Функция tiristor_angle_control() определяет момент открытия тиристора в зависимости от заданного угла открытия.

```
/**
 * @brief Контроль угла открытия тиристора с проверкой таймера и флага готовности
 * @param ctrl Указатель на структуру управления тиристором
```

```
 */
void tiristor_angle_control(TiristorControl_t *ctrl)
{
    tiristor_angle_update(&ctrl->angle);  // Обновляем задержку угла открытия

    if(ctrl->angle.delay_us != 0)  // Если задержка не нулевая
    {
        // Проверяем, прошла ли задержка с момента старта отсчёта таймера
        if (((uint16_t)TIMER->CNT - ctrl->angle.start_delay_tick) > ctrl-
>angle.delay_us)
        {
                ctrl->f.EnableTiristor = 1;  // Разрешаем включение тиристора
        }
    }
}
```

В данном контексте параметр delay_us представляет собой задержку во времени от момента перехода напряжения через ноль до момента подачи управляющего импульса на тиристор – то есть фактически соответствует углу открытия тиристора. Если разница между текущим значением таймера и временем начала задержки (start_delay_tick), зафиксированным в момент перехода через ноль, превышает рассчитанную задержку delay_us, и тиристор готов к включению (флаг TiristorReady установлен), активируется флаг EnableTiristor, который запускает процесс включения тиристора.

Функция tiristor_start_angle_delay() начинает отсчет угла открытия, фиксируя текущее значение счетчика таймера в start_delay_tick, что позволяет синхронизировать открытие тиристора с переходом через ноль, и устанавливает готовность тиристора TiristorReady к открытию.

Управление углом открытия реализовано в функции tiristor_angle_update().

```
/**
 * @brief Обновление значения задержки угла открытия тиристора в соответствии с
направлением и шагом
 * @param angle Указатель на структуру управления углом тиристора
 */
void tiristor_angle_update(TiristorAngleControl_t *angle)
{
    uint32_t current_time_ms = HAL_GetTick();  // Текущее время в миллисекундах

    // Проверяем, прошло ли нужное время с последнего обновления
    if ((current_time_ms - angle->last_update_ms) >= angle->Init->sample_time_ms)
    {
        angle->last_update_ms = current_time_ms;  // Обновляем время последнего
изменения задержки
```

```
        // Изменяем задержку в зависимости от направления (разгон или торможение)
        if(angle->Init->direction)
            angle->delay_us += angle->Init->delay_step_us;  // Увеличиваем задержку
(увеличиваем угол)
        else
            angle->delay_us -= angle->Init->delay_step_us;  // Уменьшаем задержку
(уменьшаем угол)

        // Ограничиваем задержку в пределах минимального и максимального значения
        if (angle->delay_us < angle->Init->delay_min_us)
        {
            angle->delay_us = angle->Init->delay_min_us;
        }
        else if (angle->delay_us > angle->Init->delay_max_us)
        {
            angle->delay_us = angle->Init->delay_max_us;
        }
    }
}
```

С заданной периодичностью происходит изменение задержки открытия delay_us с фиксированным шагом delay_step_us в зависимости от направления изменения (direction). При этом задержка delay_us ограничена минимальным (delay_min_us) и максимальным (delay_max_us) значениями, что гарантирует корректность управления и предотвращает выход за допустимые временные рамки. Таким образом, плавное изменение delay_us обеспечивает регулировку угла открытия тиристора, что напрямую влияет на мощность, подаваемую на нагрузку.

Таким образом, реализация данного модуля обеспечила гибкое и точное управление временем открытия тиристоров согласно требованиям алгоритма. Тестирование функций в среде MATLAB позволили своевременно выявить и устранить ошибки. Например, изначально алгоритм формировал сигнал управления тиристором несколько раз, в результате чего, при переходе через ноль тиристор не закрывался (рисунок 17).

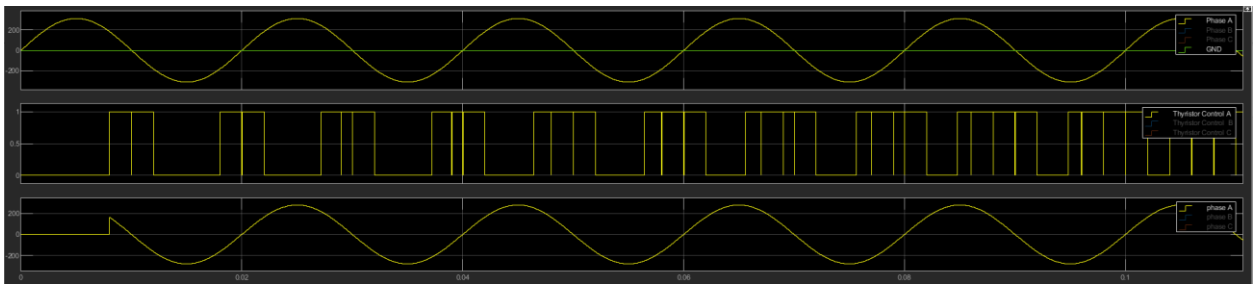


Рисунок 17 – Демонстрация некорректного алгоритма управления тиристорами

Поэтому был добавлен флаг TiristorReady, который не дает алгоритму сформировать новый импульс тиристора пока не будет определен переход напряжения через ноль. А график корректной работы тиристоров (тиристоры открываются только один раз за полупериод) приведен на рисунке 18.

```
if(ctrl->f.TiristorReady)  // Если тиристор готов к включению
{
    ctrl->f.EnableTiristor = 1;  // Разрешаем включение тиристора
    ctrl->f.TiristorReady = 0;   // Снимаем флаг готовности, чтобы не включать повторно
}
```

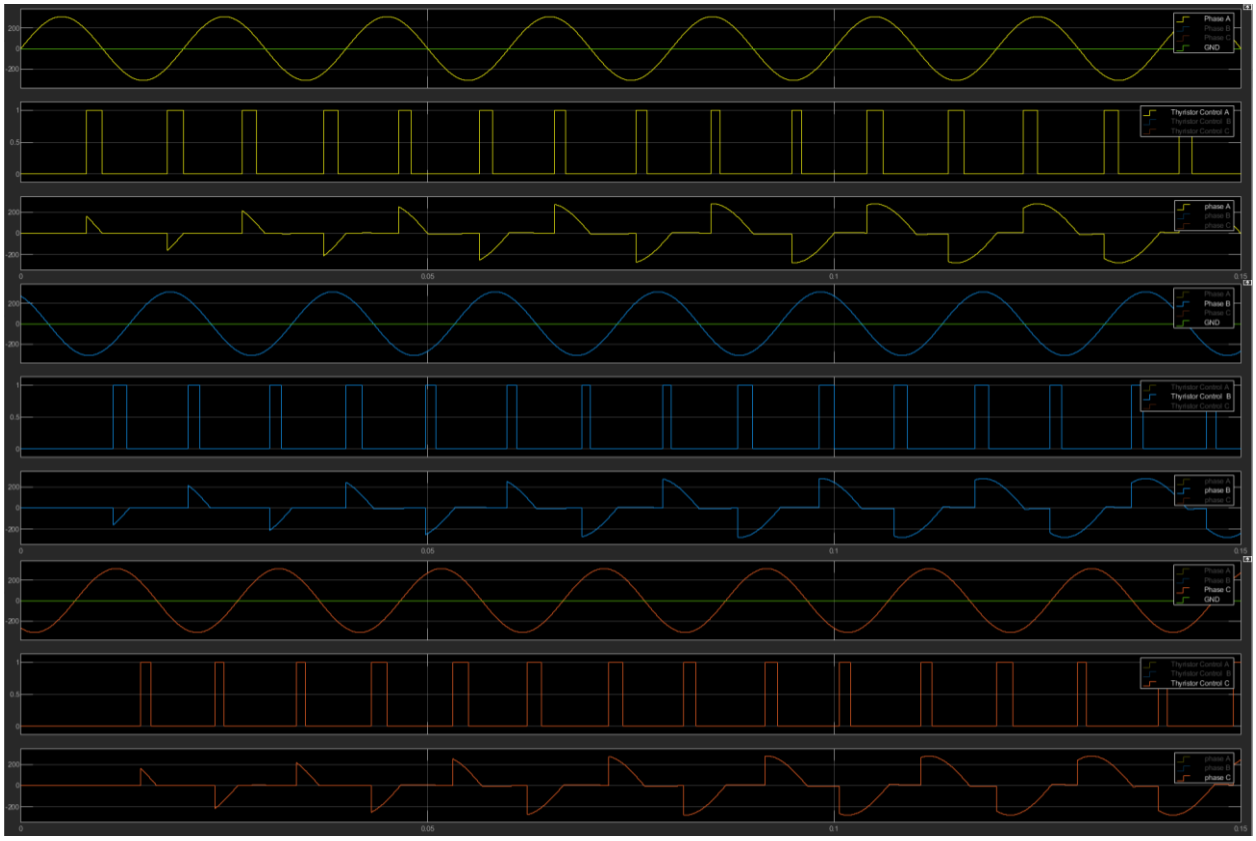


Рисунок 18 – Результаты моделирования исправленного алгоритма определения перехода через ноль

Также была другая проблема, приведенная на рисунке 19 – тиристор иногда открывался раньше, чем положено.

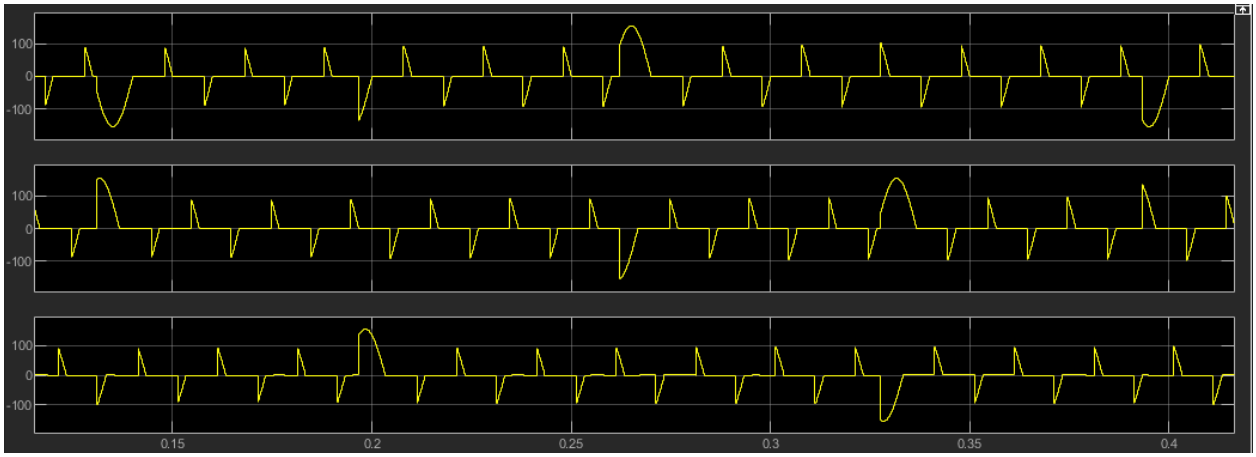


Рисунок 19 – Демонстрация некорректного алгоритма управления тиристорами

Для выявления причин этой ошибки были использованы возможности вывода отладочной информации с МК в виде временной диаграммы, которая приведена на рисунке 20. На ней представлено формирование управляющей задержки включения тиристора. Сигнал TM_CNT – start_tick (синий) отражает разность между текущим значением счётчика таймера и моментом перехода сетевого напряжения через ноль, который в программе рассчитывается так.

```
        // Проверяем, прошла ли задержка с момента старта отсчёта таймера
        if ((uint16_t)TIMER->CNT - ctrl->angle.start_delay_tick) > ctrl->angle.delay_us)
```

Горизонтальная линия angle.delay_us (жёлтый) соответствует рассчитанному значению задержки включения тиристора. Пересечение синего сигнала с жёлтым уровнем указывает на момент, когда выполняется условие включения тиристора.

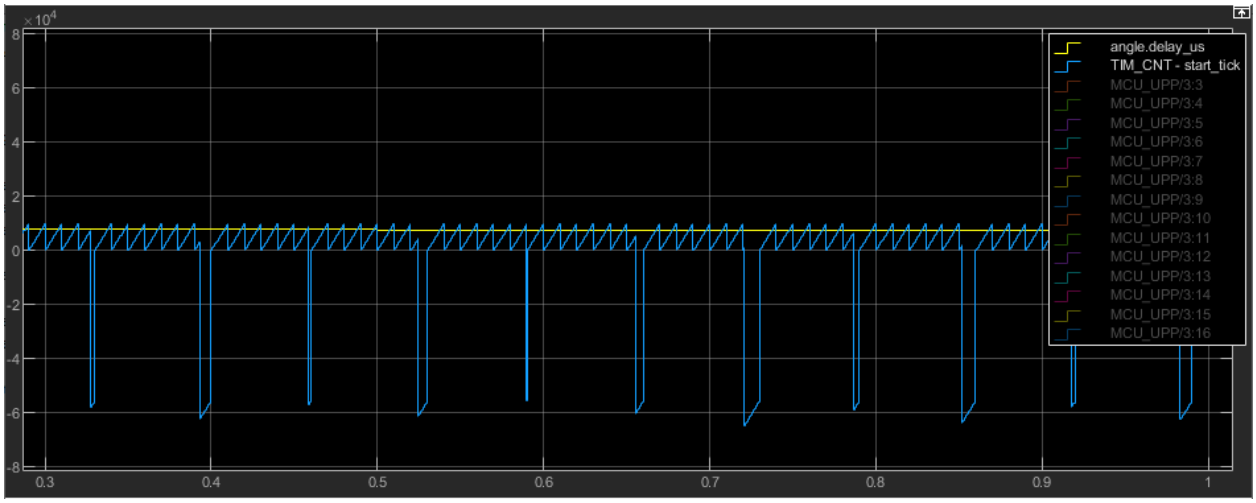


Рисунок 20 – Некорректная работа алгоритма расчета угла включения тиристора

Как видно, разность между TIM_CNT и start_tick иногда рассчитывалась некорректно. А из-за особенностей беззнакового типа данных, сравнение angle.delay_us и отрицательной разности TM_CNT – start_tick возвращало истину. И поэтому тиристор открывался раньше заданного угла. Поэтому было добавлено дополнительное приведение результата разности к типу данных uint16_t. График корректной работы алгоритма приведен на рисунке 21.

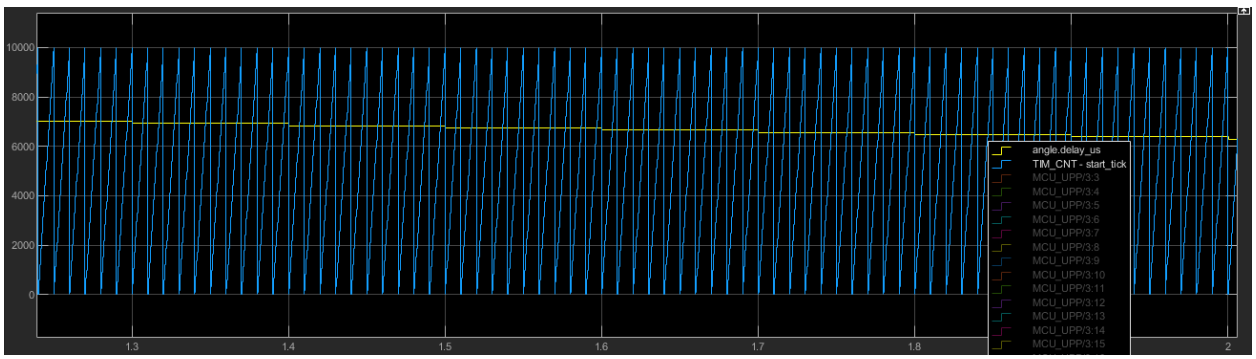


Рисунок 21 – Корректная работа алгоритма расчета угла включения тиристора

После исправления ошибок, функция tiristor_angle_control приняла следующий итоговый вид.

```
/**
 * @brief Контроль угла открытия тиристора с проверкой таймера и флага готовности
 * @param ctrl Указатель на структуру управления тиристором
 */
void tiristor_angle_control(TiristorControl_t *ctrl)
{
    tiristor_angle_update(&ctrl->angle);  // Обновляем задержку угла открытия

    if(ctrl->angle.delay_us != 0)  // Если задержка не нулевая
    {
        // Проверяем, прошла ли задержка с момента старта отсчёта таймера
        if ((uint16_t)((uint16_t)TIMER->CNT - ctrl->angle.start_delay_tick) > ctrl-
>angle.delay_us)
        {
            if(ctrl->f.TiristorReady)  // Если тиристор готов к включению
            {
                ctrl->f.EnableTiristor = 1;  // Разрешаем включение тиристора
                ctrl->f.TiristorReady = 0;   // Снимаем флаг готовности, чтобы не
включать повторно сразу
            }
        }
    }
}
```

Теперь завершающий этап разработки системы управления УПП: разработка полной логики работы программы и всех её режимов, а также добавление возможности внешнего управления системой.

Главной функцией системы является функция upp_main(), которая реализует основной цикл управления УПП. Эта функция последовательно выполняет проверку условий и управляет режимами работы, исходя из флагов состояния и текущих параметров системы. Блок-схема алгоритма управления состоянием УПП приведена на рисунке 22.

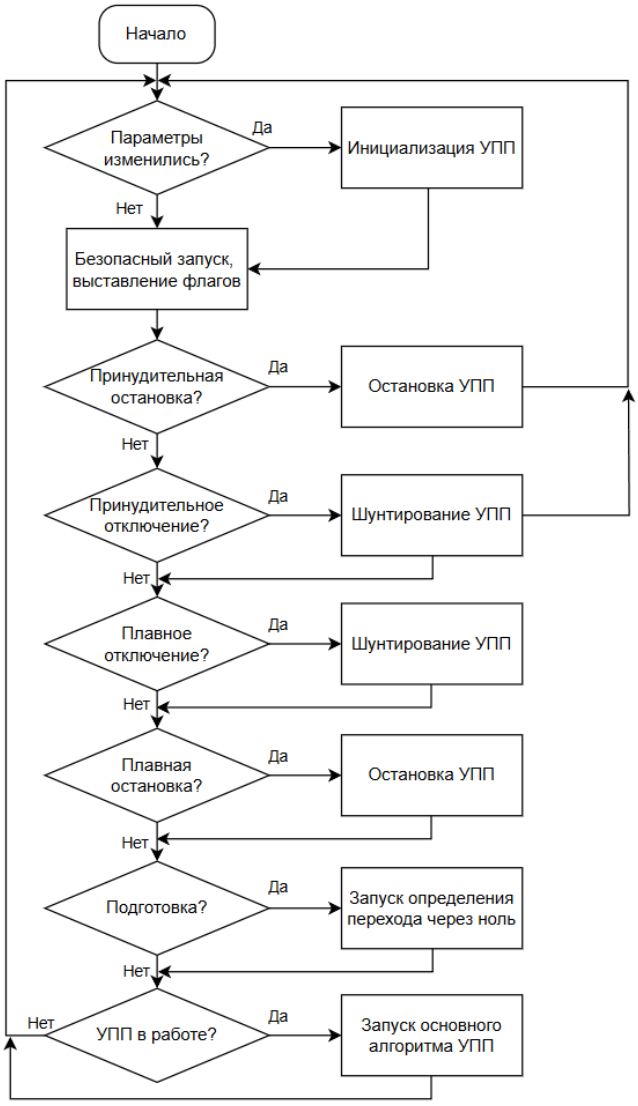


Рисунок 22 – Блок схема управления состоянием УПП

В начале работы функции происходит проверка необходимости инициализации углового параметра задержки открытия тиристоров. Для этого вызывается функция GetAngleInit(), которая анализирует изменения основных управляющих параметров – частоты сети, длительности разгона и коэффициентов скважности угла открытия (min_duty и max_duty). Если происходит изменение входных параметров, функция рассчитывает новые значения задержек задержки открытия тиристоров (delay_min_us и delay_max_us).

После инициализации угловых параметров происходит вызов функции upp_safe_go(), которая контролирует безопасный запуск или торможение УПП. Эта функция анализирует изменения флага GoSafe, который служит сигналом на плавный переход между режимами работы: при выставлении GoSafe в “1” значения происходит безопасный пуск с постепенным уменьшением задержки открытия тиристоров (т.е. увеличение мощности). А при выставлении в “0” – инициируется процесс торможения с увеличением задержки (уменьшение мощности).

В процессе безопасного запуска или торможения все три фазы (A, B и C) проходят подготовку и инициализацию состояния тиристоров). Это сбрасывает предыдущие параметры задержек и флаги, подготавливая систему к новому циклу работы.

Для обеспечения надежного и контролируемого отключения нагрузки на двигатель предусмотрены отдельные флаги и механизмы управления отключением и остановкой УПП.

Когда алгоритм достигает крайнего угла открытия тиристоров: при пуске – минимальный угол открытия, при остановке – максимальный, то выставляется соответствующий флаг шунтирования GoDisconnect или остановки GoStop.

Если выставлен флаг GoDisconnect, то происходит шунтирование тиристоров. Функция disconnect_upp() контролирует состояние каждой фазы, и когда тиристор на фазе открыт шунтирует её. Когда все три фазы отключены устанавливается флаг Disconnected, который сигнализирует о полной остановке подачи питания.

Флаг GoStop служит для окончательной остановки УПП, когда после торможения происходит полное отключение питания, что важно для предотвращения скачков тока и обеспечении плавного завершения работы двигателя. Функция connect_upp() выключает все тиристоры и отключает шунтирование, если оно было включено.

Также предусмотрены флаги ForceStop и ForceDisconnect, которые помимо соответствующего функционала, останавливают логику работы всей остальной программы УПП и блокируют управление ею.

Для контроля и управления фазами используются функции upp_phase_routine() и upp_phase_control().

В функции upp_phase_routine() для каждой фазы обновляется информация о переходе через ноль сетевого напряжения – ключевой момент, с которого начинается отсчет времени до открытия тиристоров. После фиксации перехода через ноль для каждой фазы вызывается функция tiristor_start_angle_delay(), которая фиксирует текущее значение счетчика таймера как начальную точку отсчета задержки открытия тиристора.

Если тиристор в данный момент открыт (TiristorIsEnable), он отключается, чтобы обеспечить корректное управление углом открытия на новом полупериоде.

Функция upp_phase_control() отвечает за непосредственное управление тиристором для каждой фазы. В ней происходит расчет задержки открытия в микросекундах, исходя из текущих параметров, и включение тиристора в точное время, что гарантирует реализацию заданного угла управления.

Ключевым моментом является функция GetAngleInit(), где происходит адаптивный пересчет задержек открытия тиристоров при изменении входных параметров.

```
if(update)
{
	// Расчёт длительности полупериода в микросекундах (с учётом вычета резерва на открытие тиристора)
	uint32_t half_period_us = (500000.0f / Upp.sine_freq) - 1000;

	// Расчёт максимальной и минимальной задержки (в мкс) по процентам скважности
	angle->delay_max_us = (uint32_t)(Upp.max_duty * half_period_us);
	angle->delay_min_us = (uint32_t)(Upp.min_duty * half_period_us);

	// Проверка, помещаются ли значения задержек в 16-битный таймер
	if((angle->delay_max_us > 0xFFFF) || (angle->delay_min_us > 0xFFFF))
	{
		// Если нет – увеличиваем прескалер в 10 раз (точность 10 мкс)
		angle->delay_max_us /= 10;
		angle->delay_min_us /= 10;
		TIMER->PSC = 719;
```

```
            if((angle->delay_max_us > 0xFFFF) || (angle->delay_min_us > 0xFFFF))
            {
                  // Если всё ещё не помещается – ещё в 10 раз (точность 0.1 мс)
                  angle->delay_max_us /= 10;
                  angle->delay_min_us /= 10;
                  TIMER->PSC = 7299;

                  if ((angle->delay_max_us > 0xFFFF) || (angle->delay_min_us >
0xFFFF))
                  {
                        // Если даже при этом переполнение – аварийная остановка
                        Upp.ForceStop = 1;
                  }
            }
      }
      else
      {
            // Задержки помещаются – устанавливаем стандартный прескалер (1 мкс)
            TIMER->PSC = 71;
      }

      // Перевод длительности разгона/торможения из секунд в миллисекунды
      float duration_ms = Duration_old * 1000.0f;
      uint32_t steps = duration_ms / angle->sample_time_ms;
      if (steps == 0) steps = 1;

      // Вычисление шага изменения задержки на каждом шаге
      if (angle->delay_max_us > angle->delay_min_us)
            angle->delay_step_us = (angle->delay_max_us - angle->delay_min_us) /
steps;
      else
            angle->delay_step_us = 0;
}
```

Задержки вычисляются как произведение коэффициентов скважности угла открытия (min_duty и max_duty) на полупериод сетевого напряжения, выраженный в микросекундах. Например, при частоте 50 Гц полупериод составляет 10 мс (10 000 мкс), и задержка открытия тиристора с коэффициентом 0,1 будет примерно 1 мс.

Чтобы избежать переполнения 16-битного таймера, при необходимости осуществляется изменение предделителя таймера (PSC), что позволяет измерять более длинные временные интервалы с пониженной точностью. Это важно для работы с низкочастотными сетями.

Кроме того, вычисляется количество шагов изменения угла задержки, исходя из длительности разгона (Duration), выраженной в миллисекундах, и частоты обновления (sample_time_ms). Таким образом, задается плавный и равномерный переход от минимальной задержки открытия (максимальная

подача мощности) к максимальной (минимальная подача мощности или полное отключение).

Параметр delay_step_us определяет шаг изменения задержки за один цикл обновления, что непосредственно влияет на скорость разгона или торможения.

Функция upp_init() выполняет начальную настройку системы: задаются начальные параметры, назначаются порты GPIO для управления тиристорами и отключением фаз, инициализируется таймер, а также настраивается модуль детектирования перехода через ноль. В частности, каждому из трех фазовых контроллеров присваивается ссылка на общие параметры угла открытия, что обеспечивает согласованное управление фазами.

Итоговая программа управления УПП приведена в приложении В.

Также была целиком переработана модель УПП в MATLAB. Она была структурирована и упрощена. Итоговая модель для демонстрации работы УПП приведена на рис. 23.

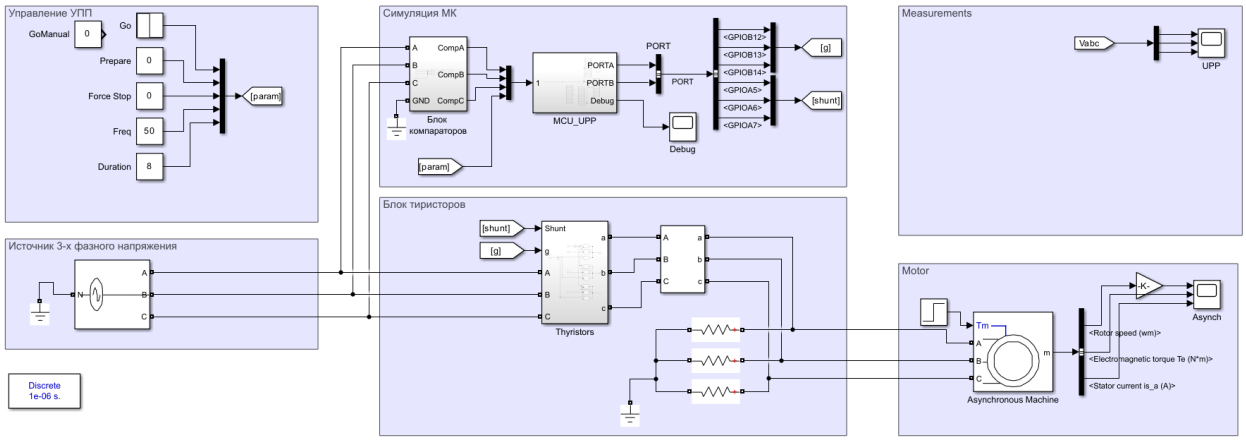


Рисунок 23 – Модель для симуляции УПП

4 ИССЛЕДОВАНИЕ МОДЕЛИ МИКРОКОНТРОЛЛЕРА В MATLAB

Разработка модели микроконтроллера в среде MATLAB/Simulink была направлена на создание инструмента, способного заменить физическое оборудование на этапе тестирования и отладки программного обеспечения встроенных систем. Однако наличие самой модели ещё не гарантирует её практическую применимость. Необходимо удостовериться, что симулятор воспроизводит поведение реального микроконтроллера с достаточной точностью как в логике исполнения программы, так и во взаимодействии с внешней моделью аппаратной части, реализованной в Simulink.

Для оценки качества и корректности работы разработанной модели была проведена серия экспериментов, направленных на сравнение результатов симуляции с поведением реального микроконтроллера при выполнении идентичного управляющего алгоритма.

Кроме того, чтобы продемонстрировать интеграцию модели микроконтроллера с физической моделью электрооборудования, рассматриваются результаты моделирования процессов пуска и останова асинхронного двигателя в Simulink как с использованием устройства плавного пуска, так и при прямом пуске. Сопоставление полученных характеристик (токов, напряжений, крутящего момента) позволяет судить о степени влияния управляющего алгоритма на поведение всей системы и подтверждает возможность использования разработанного инструмента в инженерной практике.И

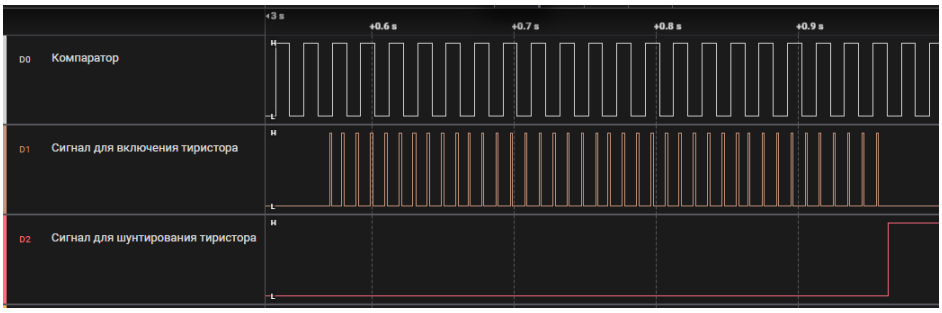
4.1 СРАВНЕНИЕ МОДЕЛИ С РЕАЛЬНЫМ МИКРОКОНТРОЛЛЕРОМ

Для оценки корректности разработанной модели микроконтроллера в среде MATLAB/Simulink было проведено экспериментальное сравнение результатов её работы с поведением реального микроконтроллера при выполнении идентичного управляющего алгоритма.

Использовался реальный STM32F103. На него подавался сигнал, идентичный сигналу с компаратора (меандр 50 Гц). С помощью логического анализатора снимались следующие сигналы: меандр с компаратора, сигнал для включения тиристоров, сигнал для шунтирования тиристора.

В рамках первого эксперимента рассматривалась ситуация с коротким временем пуска, равным 0,5 секунды, для оценки формирования сигналов для открытия тиристоров в зависимости от угла. Как видно на рисунке 24, сигналы совпадают по времени и форме, что подтверждает корректность модели.

а)



б)

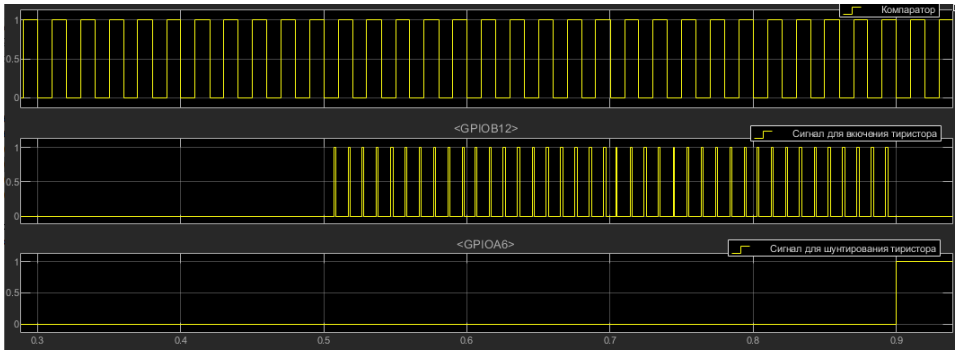


Рисунок 24 – Сравнение модели и реального микроконтроллера

а) осциллограмма с реального микроконтроллера

б) диаграмма симуляции в MATLAB

Дополнительно была проведена проверка корректности модели при увеличенном времени пуска, а именно с плавным разгоном двигателя в течение 8 секунд. Анализ временных диаграмм, полученных как с реального устройства, так и из результатов моделирования, показал , что с момента первого включения тиристоров до завершения пуска (шунтирования тиристоров) прошло одинаковое время – 7,89 секунды. Это свидетельствует о том, что при использовании реальных алгоритмов и физических параметров, модель адекватно отражает динамическое поведение микроконтроллерной системы даже в условиях протяжённых переходных процессов. Результаты этого сравнения приведены на рисунке 25.

а)

б)

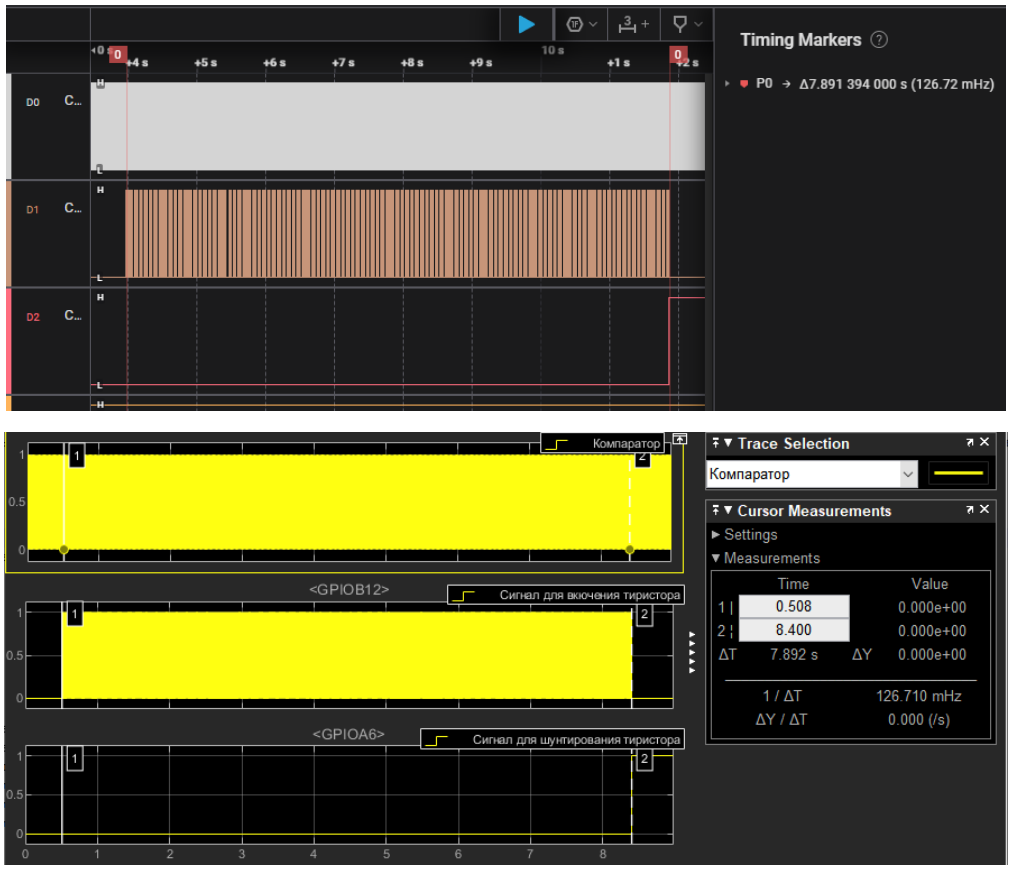


Рисунок 25 – Демонстрация одинакового времени пуска

а) осциллограмма с реального микроконтроллера

б) диаграмма симуляции в MATLAB

4.2 МОДЕЛИРОВАНИЕ УСТРОЙСТВА ПЛАВНОГО ПУСКА

Для начала рассмотрим режим прямого пуска асинхронного двигателя. Общая картина напряжений на двигателе, его частоты вращения и токов приведена на рисунке 26.

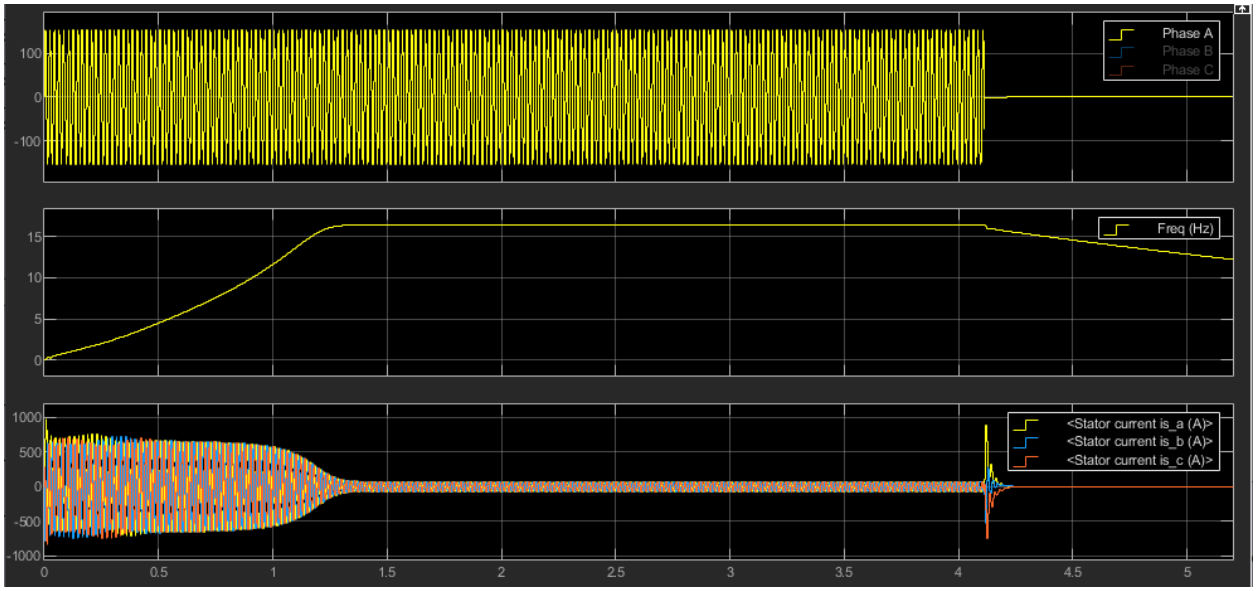


Рисунок 26 – Результаты моделирования прямого пуска асинхронного двигателя

На рисунке 27 отображены фрагменты токов статора в моменты запуска и остановки в увеличенном масштабе. В процессе пуска наблюдаются значительные пусковые токи: $i_{s\_a\_max}$ = 975 А, $i_{s\_b\_max}$ = 775 А, $i_{s\_c\_max}$ = 830 А. При остановке двигателя максимальные токи достигают соответственно: $i_{s\_a\_max}$ = 880 А, $i_{s\_b\_max}$ = 527 А, $i_{s\_c\_max}$ = 740 А.

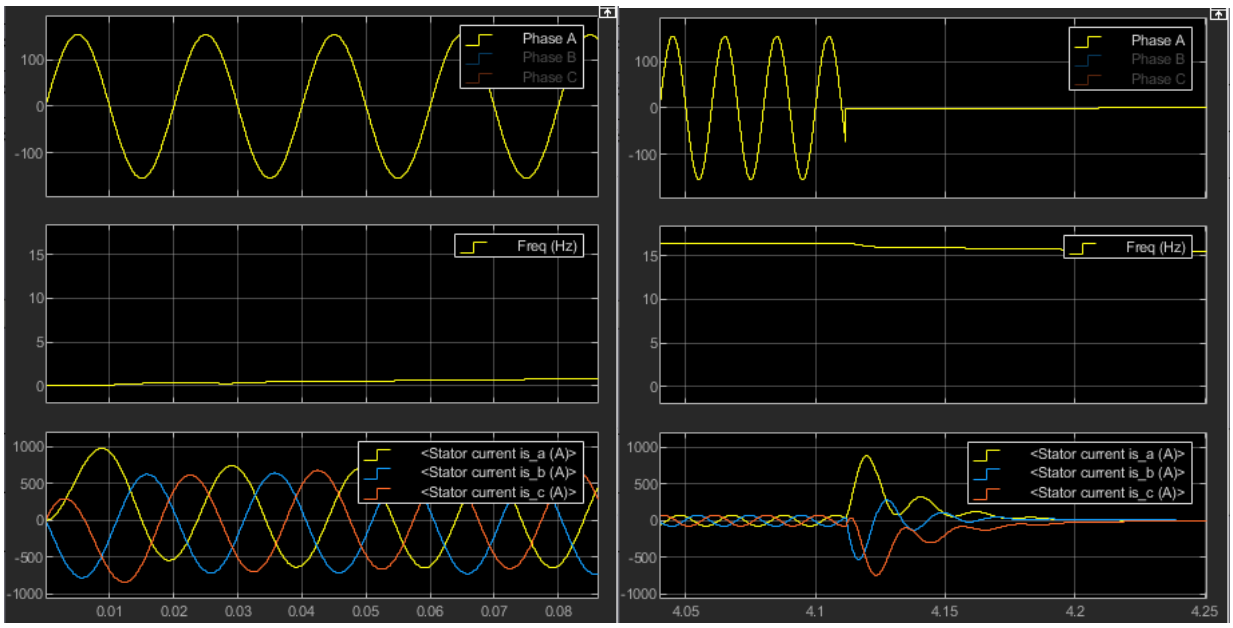


Рисунок 27 – Токи статора асинхронного двигателя в моменты запуска и остановки в режиме прямого пуска

На рисунке 28 приведен график крутящего момента двигателя в режиме прямого пуска. Видно, что крутящий момент *Te* при подключении к сети резко возрастает и сильно колеблется от -500 Н·м до 1000 Н·м. А при остановке, наблюдается значительный выброс до -1100 Н·м.

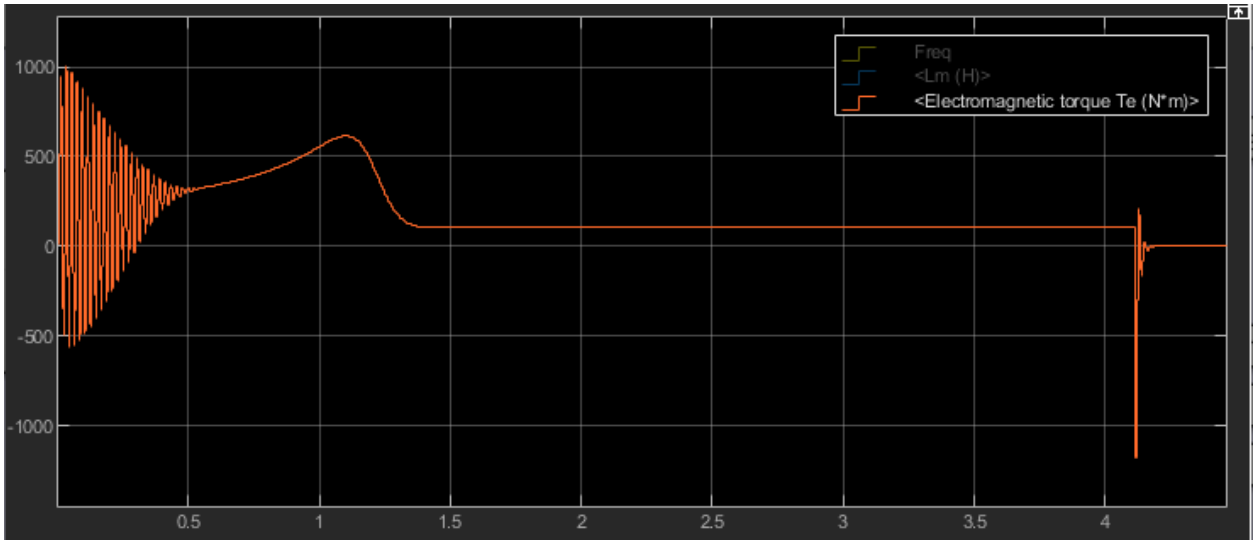


Рисунок 28 – Крутящий момент двигателя при прямом пуске

Далее рассмотрен пуск с использованием устройства плавного пуска (УПП). В качестве управляющего сигнала использовался управляющий бит Go (в программе GoSafe), обеспечивающий плавное нарастание и спад напряжения на выходе УПП. Общая динамика системы в этом режиме показана на рисунке 29.

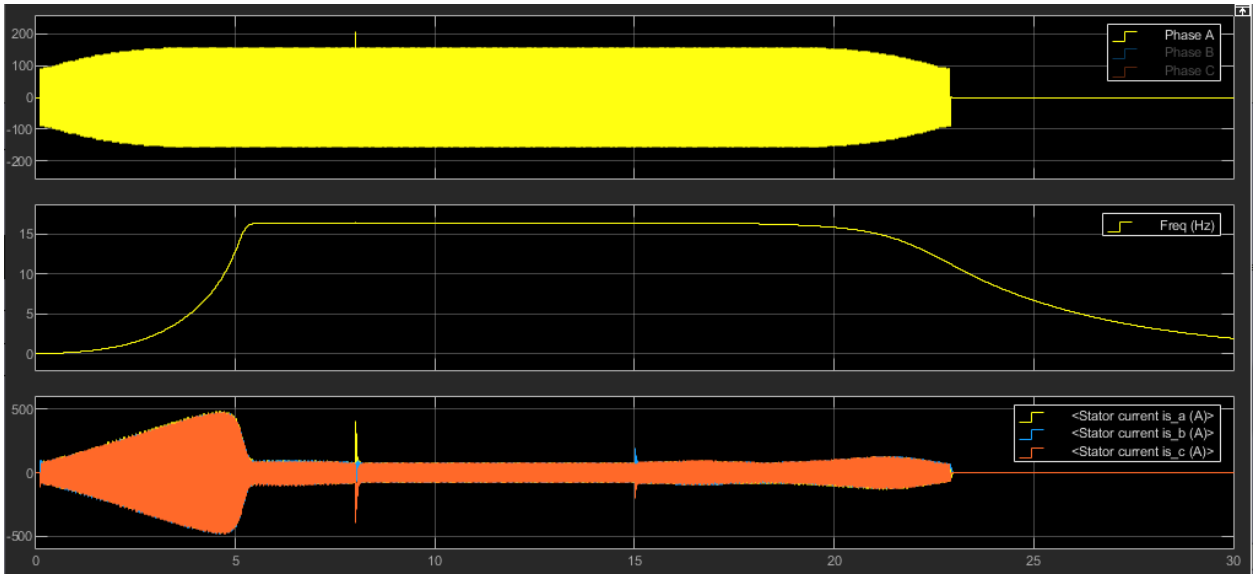


Рисунок 29 – Результаты моделирования пуска асинхронного двигателя через УПП

На рисунке 30 приведены моменты запуска и остановки.

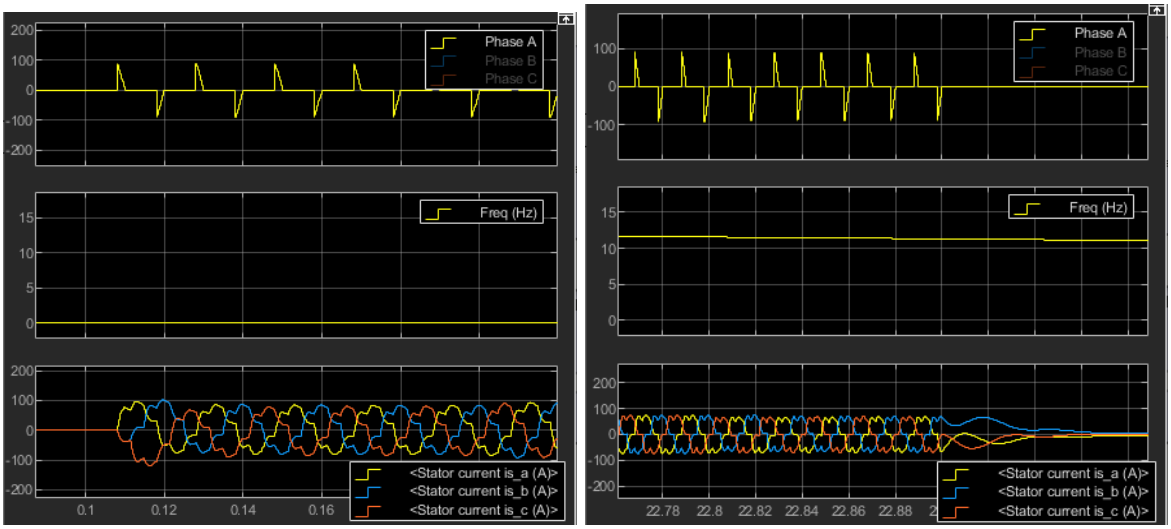


Рисунок 30 – Токи статора асинхронного двигателя в моменты запуска и остановки в режиме плавного пуска

Видно, что при плавном пуске ток статора плавно нарастает и спадает, не превышая 500 А. Но при шунтировании тиристорного блока, ток довольно резко подскакивает, хотя все равно не превышает 500 А.

На рисунке 31 приведен крутящий момент двигателя при запуске и остановке. Видно, что почти что пропали колебания крутящего момента при пуске двигателя. Но при подключении и отключении УПП также появляются выбросы, максимальный выбор был при отключении, и он составил -800 Н·м.

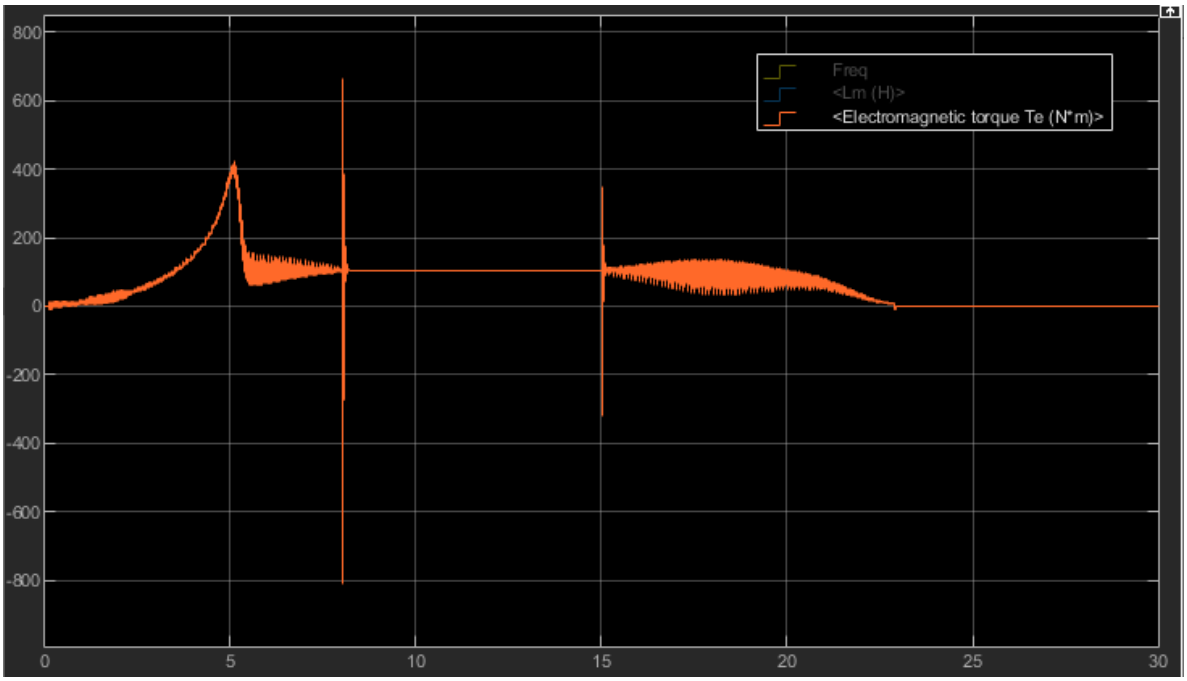


Рисунок 31 – Результаты моделирования прямого пуска асинхронного двигателя

Для снижения данных выбросов было решено попробовать изменить логику шунтирования тиристорного блока: вместо пофазного подключения, все фазы стали подключаться одновременно. Это позволило значительно снизить переходные явления при шунтировании. Максимальные значения выбросов тока уменьшились с 500 А до 270 А, а выбросы крутящего момента – с 800 Н·м до 400 Н·м. Диаграммы плавного пуска при новом алгоритме приведены на рисунках 32-33.

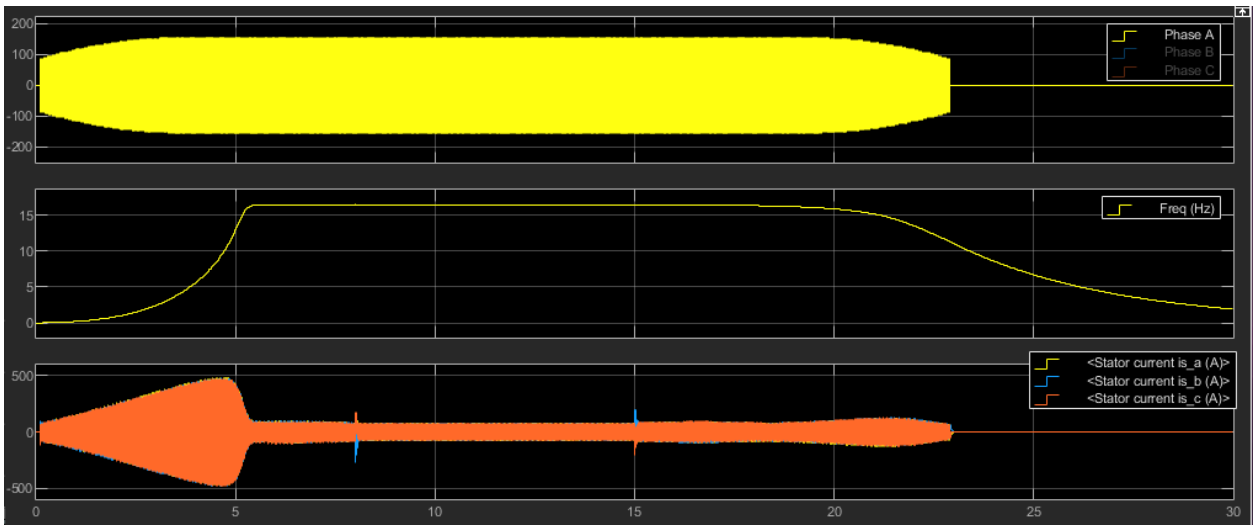


Рисунок 32 – Результаты моделирования прямого пуска асинхронного двигателя

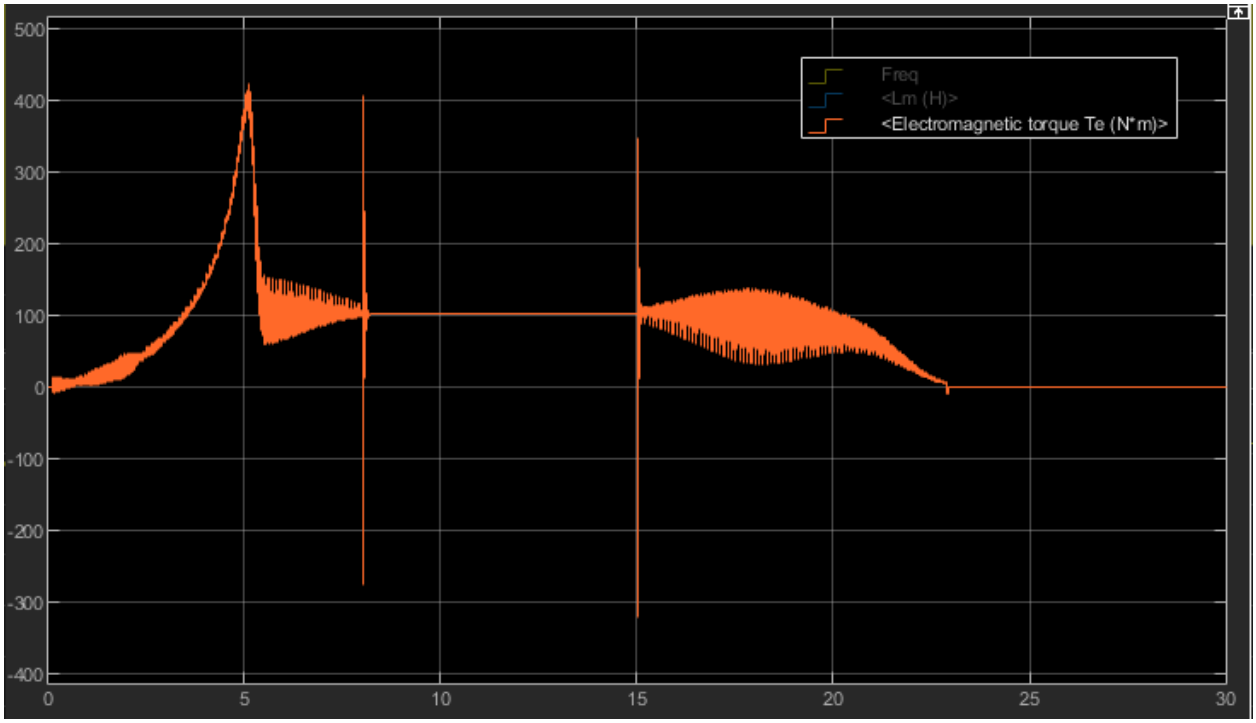


Рисунок 33 – Результаты моделирования прямого пуска асинхронного двигателя

Видно, что при подключении и отключении устройства плавного пуска токи статора значительно снизились. Амплитуда выбросов крутящего момента также уменьшилась, но не столь значительно.

4.3 ПРЕИМУЩЕСТВА ИСПОЛЬЗОВАНИЯ МОДЕЛИ МИКРОКОНТРОЛЛЕРА

Разработка модели микроконтроллера в среде MATLAB/Simulink была направлена не только на возможность отладки встроенного программного обеспечения без использования физического оборудования, но и на упрощение и унификацию процессов моделирования и тестирования.

До внедрения предложенного модуля моделирование управляющих систем в MATLAB и реализация этих же систем на микроконтроллере фактически выполнялись раздельно. Как правило, алгоритм сначала отлаживался в Simulink на уровне функциональной схемы или модели управления, после чего он переносится в среду программирования микроконтроллера – часто с необходимостью переписывания логики и адаптации под архитектуру платформы.

Такой подход сопровождался рядом ограничений:

- возможность расхождения поведения моделей, особенно при сложной логике или ошибках при переносе кода;
- невозможность воспроизведения ошибок реального исполнения на стадии моделирования, так как поведение микроконтроллера эмулировалось лишь приближённо и без учёта архитектурных особенностей.

Предложенная в данной работе модель позволяет непосредственно интегрировать реальный C-код, написанный для STM32, в среду Simulink. Таким образом, MATLAB использует для симуляции тот же исходный код, который в дальнейшем загружается на физическое устройство. Это исключает ошибки, возникающие при повторной реализации логики, и обеспечивает полную идентичность поведения модели и конечного устройства при условии корректной настройки периферии.

Внедрение разработанного модуля несколько увеличивает сложность моделирования, т.к. для его использования требуется базовое понимание компиляции C-кода, настройка MSVC-компилятора и конфигурация

структуры проекта. Однако это компенсируется значительным ростом точности моделирования и сокращением общего времени отладки. Разработчик получает возможность пошаговой отладки, мониторинга внутренних переменных, формирования временных диаграмм и удобной интеграции с физическими моделями в Simulink.

Таким образом, предложенный подход устраняет разрыв между симуляцией и программированием микроконтроллера. Это делает возможным более тесную интеграцию аппаратного и программного обеспечения ещё на стадии проектирования, что особенно важно при разработке сложных или критически важных систем управления.

ЗАКЛЮЧЕНИЕ

Целью данной работы являлась разработка симулятора микроконтроллера STM32 в среде MATLAB для пошаговой отладки и тестирования встроенных программных алгоритмов без необходимости использования физического оборудования. Такой подход позволяет ускорить цикл разработки, повысить воспроизводимость результатов и упростить отладку программ в условиях, приближенных к реальным.

В рамках работы была создана архитектура симулятора, способного исполнять пользовательский код на языке C с эмуляцией ключевых периферийных устройств (таймеры, GPIO, UART) и интеграцией с системой моделирования Simulink. Это позволяет тестировать управляющие алгоритмы в контексте работы с реальной электроникой и силовой частью.

В качестве демонстрационного примера возможностей разработанного симулятора был реализован проект управления тиристорным устройством плавного пуска асинхронного двигателя. Этот пример включал построение физической модели в среде Simulink, реализацию алгоритма управления на языке C и его тестирование с помощью разработанного симулятора. Проведено моделирование переходных режимов пуска и остановки, подтверждающее работоспособность всей системы.

Таким образом, в работе был разработан универсальный инструмент для отладки микроконтроллерных приложений в MATLAB/Simulink, позволяющий проводить моделирование как логики программы, так и её взаимодействия с внешними системами.

Разработанная модель ориентирована на пользователей, обладающих опытом работы с микроконтроллерами STM32, знанием языка программирования C, а также пониманием структуры проектов, создаваемых на микроконтроллерах. Дополнительно требуется базовое владение MATLAB и Simulink, поскольку симулятор интегрируется в процесс блочного моделирования через S-Function.

В то же время, структура инструмента позволяет адаптировать его под разные проекты без необходимости глубокого изменения внутренней архитектуры, что делает его удобным как для научных исследований, так и для инженерной практики. Например, в модели возможности для последующего расширения функциональности симулятора, включая поддержку дополнительных периферийных устройств и моделирование более сложных архитектур микроконтроллеров.

Предложенный подход позволяет воспроизводимо моделировать и отлаживать встроенное программное обеспечение в контексте сложных систем управления, еще до перехода к аппаратной реализации, что особенно актуально в образовательных, исследовательских и опытно-конструкторских задачах.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Проскуряков В. С., Соболев С. В. Асинхронный двигатель: учебное пособие. – Екатеринбург: Уральский государственный технический университет – УПИ, 2008. – 30 с.

2. Поминов, А. Д. Режимы управления асинхронного двигателя для робототехнических систем / А. Д. Поминов, Ю. Е. Лившиц // VII Международная научно-техническая интернет-конференция "Информационные технологии в образовании, науке и производстве", 16-17 ноября 2019 года, Минск, Беларусь [Электронный ресурс] / Белорусский национальный технический университет; сост. Е. В. Кондратёнок. – Минск : БНТУ, 2019. – С. 400-403

3. Кленэ Д. Устройства плавного пуска и преобразователи частоты. – М.: Шнейдер Электрик, 2011. – 27 с. – (Техническая коллекция Schneider Electric. Вып. 38).

4. Ситников А. Тиристорное устройство плавного пуска асинхронного электродвигателя // Современная электроника. – 2008. – № 9. – С. 50–53.

5. QEMU Emulator User Documentation [Электронный ресурс]. URL: https://www.qemu.org/docs/master/user/main.html (дата обращения 17.03.2015).

6. Филатов, М. Проектирование схем электрических принципиальных с использованием микроконтроллеров в программной среде Proteus 8.1 / М. Филатов // Компоненты и технологии. – 2015. – № 7(168). – С. 101-110. – EDN TZBZLX.

7. Анодина-Андриевская, Е. М. Моделирование таймеров микроконтроллера STM32 в MATLAB с возможностью отладки / Е. М. Анодина-Андриевская, А. В. Разваляев // Математические методы и модели в высокотехнологичном производстве: Сборник тезисов докладов IV Международного форума. В 2-х частях, Санкт-Петербург, 06 ноября 2024 года. – Санкт-Петербург: Санкт-Петербургский государственный университет аэрокосмического приборостроения, 2024. – С. 306-311. – EDN JHMMLR.

8. Дьяконов, В. MATLAB 8.0 (R2012b) - схемотехническое моделирование в Simscape и SimElectronics / В. Дьяконов // Компоненты и технологии. – 2014. – № 4(153). – С. 174-184. – EDN SCQMVF.

ПРИЛОЖЕНИЕ А

Листинг оболочки микроконтроллерной программы

Файл MCU.c:

```
/**
*****************************************************************************
* @file MCU.c
* @brief Исходный код S-Function.
*****************************************************************************
@details
Данный файл содержит функции S-Function, который вызывает MATLAB.
*****************************************************************************
@note
Описание функций по большей части сгенерировано MATLAB'ом, поэтому на английском
*****************************************************************************/

/**
  * @addtogroup             WRAPPER_SFUNC    S-Function funtions
  * @ingroup         MCU_WRAPPER
  * @brief                  Дефайны и функции блока S-Function
  * @details         Здесь собраны функции, с которыми непосредственно работает S-
Function
  *    @note                Описание функций по большей части сгенерировано MATLAB'ом,
поэтому на английском
  * @{
  */

#define S_FUNCTION_NAME      MCU
#define S_FUNCTION_LEVEL     2

#include "mcu_wrapper_conf.h"

#define MDL_UPDATE ///< для подключения mdlUpdate()
  /**
   *   @brief        Update S-Function at every step of simulation
   *   @param        S - pointer to S-Function (library struct from "simstruc.h")
   *   @details      Abstract:
   *    This function is called once for every major integration time step.
   *    Discrete states are typically updated here, but this function is useful
   *    for performing any tasks that should only take place once per
   *    integration step.
   */
static void mdlUpdate(SimStruct* S, int_T tid)
{
       // get time of simulation
       time_T TIME = ssGetT(S);

       //---------------SIMULATE MCU---------------
       MCU_Step_Simulation(S, TIME); // SIMULATE MCU
       //------------------------------------------
}//end mdlUpdate

/**
 *     @brief        Writting outputs of S-Function
 *     @param        S - pointer to S-Function (library struct from "simstruc.h")
 *     @details      Abstract:
 *    In this function, you compute the outputs of your S-function
 *    block. Generally outputs are placed in the output vector(s),
 *    ssGetOutputPortSignal.
 */
static void mdlOutputs(SimStruct* S, int_T tid)
```

```
{
	SIM_writeOutputs(S);
}//end mdlOutputs

#define MDL_CHECK_PARAMETERS   /* Change to #undef to remove function */
#if defined(MDL_CHECK_PARAMETERS) && defined(MATLAB_MEX_FILE)
static void mdlCheckParameters(SimStruct* S)
{
	int i;

	// Проверяем и принимаем параметры и разрешаем или запрещаем их менять
	// в процессе моделирования
	for (i = 0; i < 1; i++)
	{
		// Input parameter must be scalar or vector of type double
		if (!mxIsDouble(ssGetSFcnParam(S, i)) || mxIsComplex(ssGetSFcnParam(S,
i)) ||
			mxIsEmpty(ssGetSFcnParam(S, i)))
		{
			ssSetErrorStatus(S, "Input parameter must be of type double");
			return;
		}
		// Параметр м.б. только скаляром, вектором или матрицей
		if (mxGetNumberOfDimensions(ssGetSFcnParam(S, i)) > 2)
		{
			ssSetErrorStatus(S, "Параметр м.б. только скаляром, вектором или
матрицей");
			return;
		}
		//         sim_dt = mxGetPr(ssGetSFcnParam(S,0))[0];
			// Parameter not tunable
		//		ssSetSFcnParamTunable(S, i, SS_PRM_NOT_TUNABLE);
			// Parameter tunable (we must create a corresponding run-
time parameter)
		ssSetSFcnParamTunable(S, i, SS_PRM_TUNABLE);
		// Parameter tunable only during simulation
//		ssSetSFcnParamTunable(S, i, SS_PRM_SIM_ONLY_TUNABLE);

	}//for (i=0; i<NPARAMS; i++)

}//end mdlCheckParameters
#endif //MDL_CHECK_PARAMETERS
static void mdlInitializeSizes(SimStruct* S)
{
	ssSetNumSFcnParams(S, 1);
	// Кол-во ожидаемых и фактических параметров должно совпадать
	if (ssGetNumSFcnParams(S) == ssGetSFcnParamsCount(S))
	{
		// Проверяем и принимаем параметры
		mdlCheckParameters(S);
	}
	else
	{
		return;// Parameter mismatch will be reported by Simulink
	}

	// set up discrete states
	ssSetNumContStates(S, 0);                       // number of continuous states
	ssSetNumDiscStates(S, DISC_STATES_WIDTH);       // number of discrete states

	// set up input port
	if (!ssSetNumInputPorts(S, 1)) return;
	for (int i = 0; i < IN_PORT_NUMB; i++)
		ssSetInputPortWidth(S, i, IN_PORT_WIDTH);
```

```
	ssSetInputPortDirectFeedThrough(S, 0, 0);
	ssSetInputPortRequiredContiguous(S, 0, 1); // direct input signal access

	// set up output port
	if (!ssSetNumOutputPorts(S, OUT_PORT_NUMB)) return;
	for (int i = 0; i < OUT_PORT_NUMB; i++)
		ssSetOutputPortWidth(S, i, OUT_PORT_WIDTH);


	ssSetNumSampleTimes(S, 1);


	ssSetNumRWork(S, 5); // number of real work vector elements
	ssSetNumIWork(S, 5); // number of integer work vector elements
	ssSetNumPWork(S, 0); // number of pointer work vector elements
	ssSetNumModes(S, 0); // number of mode work vector elements
	ssSetNumNonsampledZCs(S, 0);	// number of nonsampled zero crossings


	ssSetRuntimeThreadSafetyCompliance(S, RUNTIME_THREAD_SAFETY_COMPLIANCE_TRUE);
	/* Take care when specifying exception free code - see sfuntmpl.doc */
	ssSetOptions(S, SS_OPTION_EXCEPTION_FREE_CODE);
}

#define MDL_START  /* Change to #undef to remove function */
#if defined(MDL_START)
/**
 *	@brief		Initialize S-Function at start of simulation
 *	@param		S - pointer to S-Function (library struct from "simstruc.h")
 *	@details	Abstract:
 *	   This function is called once at start of model execution. If you
 *	   have states that should be initialized once, this is the place
 *	   to do it.
 */
static void mdlStart(SimStruct* S)
{
	SIM_Initialize_Simulation();
}
#endif // MDL_START

/**
 *	@brief		Initialize Sample Time of Simulation
 *	@param		S - pointer to S-Function (library struct from "simstruc.h")
 *	@details	Abstract:
 *	   This function is used to specify the sample time(s) for your
 *	   S-function. You must register the same number of sample times as
 *	   specified in ssSetNumSampleTimes.
 */
static void mdlInitializeSampleTimes(SimStruct* S)
{
	// Шаг дискретизации
	hmcu.SIM_Sample_Time = mxGetPr(ssGetSFcnParam(S, NPARAMS - 1))[0];

	// Register one pair for each sample time
	ssSetSampleTime(S, 0, hmcu.SIM_Sample_Time);
	ssSetOffsetTime(S, 0, 0.0);
}

/**
 *	@brief		Terminate S-Function at the end of simulation
 *	@param		S - pointer to S-Function (library struct from "simstruc.h")
 *	@details	Abstract:
 *	   In this function, you should perform any actions that are necessary
 *	   at the termination of a simulation.  For example, if memory was
```

```
 *    allocated in mdlStart, this is the place to free it.
 */
static void mdlTerminate(SimStruct* S)
{
       hmcu.fMCU_Stop = 1;
#ifdef RUN_APP_MAIN_FUNC_THREAD
       ResumeThread(hmcu.hMCUThread);
       WaitForSingleObject(hmcu.hMCUThread, 10000);
#endif
       SIM_deInitialize_Simulation();
       mexUnlock();
}

/** WRAPPER_SFUNC
       * @}
       */

#ifdef MATLAB_MEX_FILE    /* Is this file being compiled as a
                                                  MEX-file? */
#include "simulink.c"     /* MEX-file interface mechanism */
#else
#include "cg_sfun.h"      /* Code generation registration
                                                  function */
#endif
```

Файл mcu_wrapper.c:

```
/**
******************************************************************************
* @file mcu_wrapper.c
* @brief Исходный код оболочки МК.
******************************************************************************
@details
Данный файл содержит функции для симуляции МК в Simulink (S-Function).
******************************************************************************/
#include "mcu_wrapper_conf.h"

/**
    * @addtogroup    WRAPPER_CONF
    * @{
    */

SIM__MCUHandleTypeDef hmcu;         ///< Хендл для управления потоком программы МК

/** MCU_WRAPPER
    * @}
    */
    //-------------------------------------------------------------//
    //-----------------CONTROLLER SIMULATE FUNCTIONS---------------//
    /* THREAD FOR MCU APP */
#ifdef RUN_APP_MAIN_FUNC_THREAD
/**
  * @brief      Главная функция приложения МК.
  * @details    Функция с которой начинается выполнение кода МК. Выход из данной
функции происходит только в конце симуляции @ref mdlTerminate
  */
extern int main(void);                  // extern while from main.c
/**
  * @brief      Поток приложения МК.
  * @details    Поток, который запускает и выполняет код МК (@ref main).
  */
unsigned __stdcall MCU_App_Thread(void) {
    main();     // run MCU code
    return 0;   // end thread
    // note: this return will reached only at the end of simulation, when all whiles
will be skipped due to @ref sim_while
}
#endif //RUN_APP_MAIN_FUNC_THREAD
/* SIMULATE MCU FOR ONE SIMULATION STEP */
/**
  * @brief      Симуляция МК на один такт симуляции.
 *     @param        S    - указатель на структуру S-Function из "simstruc.h"
  * @param      time - текущее время симуляции.
  * @details    Запускает поток, который выполняет код МК и управляет ходом потока:
  *             Если прошел таймаут, поток прерывается, симулируется периферия
  *             и на следующем шаге поток возобнавляется.
  *
  *             Вызывается из mdlUpdate()
  */
void MCU_Step_Simulation(SimStruct* S, time_T time)
{
    hmcu.SystemClockDouble += hmcu.SystemClock_step; // emulate core clock
    hmcu.SystemClock = hmcu.SystemClockDouble;
    hmcu.SimTime = time;

    MCU_readInputs(S); // считывание портов

    MCU_Periph_Simulation(); // simulate peripheral
```

```
    extern void upp_main(void);
    upp_main();
#ifdef RUN_APP_MAIN_FUNC_THREAD
    ResumeThread(hmcu.hMCUThread);
    for (int i = DEKSTOP_CYCLES_FOR_MCU_APP; i > 0; i--)
    {
    }
    SuspendThread(hmcu.hMCUThread);
#endif //RUN_APP_MAIN_FUNC_THREAD

    MCU_writeOutputs(S); // запись портов (по факту запись в буфер. запись в порты в
mdlOutputs)
}

/* SIMULATE MCU PERIPHERAL */
/**
  * @brief      Симуляция периферии МК
  * @details    Пользовательский код, который симулирует работу периферии МК.
  */
void MCU_Periph_Simulation(void)
{
    uwTick = hmcu.SystemClock / (MCU_CORE_CLOCK / 1000);

    Simulate_TIMs();
}

/* READ INPUTS S-FUNCTION TO MCU REGS */
/**
  * @brief      Считывание входов S-Function в порты ввода-вывода.
 *     @param        S - указатель на структуру S-Function из "simstruc.h"
  * @details    Пользовательский код, который записывает порты ввода-вывода из входов
S-Function.
  */
void MCU_readInputs(SimStruct* S)
{
    /* Get S-Function inputs */
    real_T* IN = ssGetInputPortRealSignal(S, 0);

    ReadToSFunc(IN);
}

/* WRITE OUTPUTS BUFFER S-FUNCTION FROM MCU REGS*/
/**
  * @brief      Запись портов ввода-вывода в буфер выхода S-Function
 *     @param        S - указатель на структуру S-Function из "simstruc.h"
  * @details    Пользовательский код, который записывает буфер выходов S-Function из
портов ввода-вывода.
  */
void MCU_writeOutputs(SimStruct* S)
{
    /* Get S-Function descrete array */
    real_T* Out_Buff = ssGetDiscStates(S);

    Simulate_GPIO_BSRR();
    WriteFromSFunc(Out_Buff);
}
//-----------------CONTROLLER SIMULATE FUNCTIONS---------------//
//-------------------------------------------------------------//

//-------------------------------------------------------------//
//----------------------SIMULINK FUNCTIONS---------------------//
/* WRITE OUTPUTS OF S-BLOCK */
/**
  * @brief      Формирование выходов S-Function.
```

```
 *     @param        S - указатель на структуру S-Function из "simstruc.h"
  * @details    Пользовательский код, который записывает выходы S-Function из буфера.
  */
void SIM_writeOutputs(SimStruct* S)
{
    real_T* GPIO;
    real_T* Out_Buff = ssGetDiscStates(S);

    //-------------WRITTING GPIOS---------------
    for (int j = 0; j < PORT_NUMB; j++)
    {
        GPIO = ssGetOutputPortRealSignal(S, j);
        for (int i = 0; i < PORT_WIDTH; i++)
        {
            GPIO[i] = Out_Buff[j * PORT_WIDTH + i];
            Out_Buff[j * PORT_WIDTH + i] = 0;
        }
    }
    //------------------------------------------
}
/* MCU WRAPPER DEINITIALIZATION */
/**
  * @brief      Инициализация симуляции МК.
  * @details    Пользовательский код, который создает поток для приложения МК
                и настраивает симулятор МК для симуляции.
  */
void SIM_Initialize_Simulation(void)
{
#ifdef RUN_APP_MAIN_FUNC_THREAD
    // инициализация потока, который будет выполнять код МК
    hmcu.hMCUThread = (HANDLE)CreateThread(NULL, 0, MCU_App_Thread, 0,
CREATE_SUSPENDED, &hmcu.idMCUThread);
#endif //RUN_APP_MAIN_FUNC_THREAD

    /* user initialization */
    Initialize_Periph_Sim();

    extern int main_init(void);
    main_init();
    /* wrapper initialization */
    hmcu.SystemClock_step = MCU_CORE_CLOCK * hmcu.SIM_Sample_Time;   // set system
clock step
}
/* MCU WRAPPER DEINITIALIZATION */
/**
  * @brief      Деинициализация симуляции МК.
  * @details    Пользовательский код, который будет очищать все структуры после
окончания симуляции.
  */
void SIM_deInitialize_Simulation(void)
{
//#ifdef DEINITIALIZE_AFTER_SIM
#include "upp.h"
    memset(&Upp, 0, sizeof(Upp));
    // simulate structures of peripheral deinitialization
    deInitialize_Periph_Sim();
    // mcu peripheral memory deinitialization
    deInitialize_MCU();
//#endif
}
//-------------------------------------------------------------//
```

Файл mcu_wrapper_conf.h:

```
/**
******************************************************************************
* @dir ../MCU_Wrapper
* @brief <b> Папка с исходным кодом оболочки МК. </b>
* @details
В этой папке содержаться оболочка(англ. wrapper) для запуска и контроля
эмуляции микроконтроллеров в MATLAB (любого МК, не только STM).
Оболочка представляет собой S-Function - блок в Simulink, который работает
по скомпилированому коду. Компиляция происходит с помощью MSVC-компилятора.
******************************************************************************/

/**
******************************************************************************
* @file mcu_wrapper_conf.h
* @brief Заголовочный файл для оболочки МК.
******************************************************************************
@details
Главный заголовочный файл для матлаба. Включает дейфайны для S-Function,
объявляет базовые функции для симуляции МК и подключает базовые библиотеки:
- для симуляции                "stm32fxxx_matlab_conf.h"
- для S-Function           "simstruc.h"
- для потоков              <process.h>
******************************************************************************/
#ifndef _WRAPPER_CONF_H_
#define _WRAPPER_CONF_H_

// Includes
#include "stm32f1xx_matlab_conf.h"  // For stm simulate functions
#include "simstruc.h"               // For S-Function variables
#include <process.h>                // For threads


/**
    * @defgroup      MCU_WRAPPER     MCU Wrapper
    * @brief         Всякое для оболочки МК
    */

    /**
        * @addtogroup      WRAPPER_CONF    Wrapper Configuration
        * @ingroup         MCU_WRAPPER
        * @brief           Параметры конфигурации для оболочки МК
        * @details         Здесь дефайнами задается параметры оболочки, которые
определяют как она будет работать
        * @{
        */

        // Parametrs of MCU simulator
//#define RUN_APP_MAIN_FUNC_THREAD                    ///< Enable using thread for MCU
main() func
#define DEKSTOP_CYCLES_FOR_MCU_APP      0xFF      ///< number of for() cycles after
which MCU thread would be suspended
#define MCU_CORE_CLOCK                  72000000


//#define DEINITIALIZE_AFTER_SIM                    ///< Enable deinitializing
structures at simulation ends


#define PORT_WIDTH                      16          ///< width of one port
#define PORT_NUMB                       3           ///< amount of ports
// Parameters of S_Function
```

```
#define NPARAMS                 1                           ///< number of input
parametrs (only Ts)
#define IN_PORT_WIDTH           (9)                         ///< width of input ports
#define IN_PORT_NUMB            1                           ///< number of input ports
#define OUT_PORT_WIDTH          PORT_WIDTH                  ///< width of output ports
#define OUT_PORT_NUMB           PORT_NUMB                   ///< number of output ports
#define DISC_STATES_WIDTH       PORT_WIDTH*PORT_NUMB        ///< width of discrete
states array

/** WRAPPER_CONF
  * @}
  */


  /**
    * @addtogroup    MCU_WRAPPER
    * @{
    */

    // Fixed parameters(?) of S_Function
#define NPARAMS                 1                               ///< number of input
parametrs (only Ts)
#define DISC_STATES_WIDTH       OUT_PORT_WIDTH*OUT_PORT_NUMB    ///< width of discrete
states array (outbup buffer)
/**
  * @brief      Define for creating thread in suspended state.
  * @details    Define from WinBase.h. We dont wanna include "Windows.h" or smth like
this, because of HAL there are a lot of redefine errors.
  */
#define CREATE_SUSPENDED                0x00000004
typedef void* HANDLE; ///< MCU handle typedef

/**
  * @brief  MCU handle Structure definition.
  * @note     Prefixes: h - handle, s - settings, f - flag
  */
typedef struct {
    // MCU Thread
    HANDLE          hMCUThread;                 ///< Хендл для потока МК
    uint32_t        idMCUThread;                ///< id потока МК (unused)
    // Flags
    unsigned        fMCU_Stop : 1;              ///< флаг для выхода из потока
программы МК
    double          SIM_Sample_Time;            ///< sample time of simulation

    double          SystemClockDouble;          ///< Счетчик в формате double для
точной симуляции системных тиков С промежуточными значений
    uint64_t        SystemClock;                ///< Счетчик тактов для симуляции
системных тиков (в целочисленном формате)
    double          SystemClock_step;           ///< Шаг тиков для их симуляции, в
формате double
    double          SimTime;
}SIM__MCUHandleTypeDef;
extern SIM__MCUHandleTypeDef hmcu;              // extern для видимости переменной во
всех файлах

//-------------------------------------------------------------//
//------------------  SIMULINK WHILE DEFINES  -----------------//
#ifdef RUN_APP_MAIN_FUNC_THREAD
/* DEFINE TO WHILE WITH SIMULINK WHILE */
/**
  * @brief      Redefine C while statement with sim_while() macro.
  * @param      _expression_ - expression for while.
```

```
  * @details    Это while который будет использоваться в симулинке @ref sim_while для
подробностей.
  */
#define while(_expression_)                          sim_while(_expression_)
#endif
  /* SIMULINK WHILE */
  /**
    * @brief      While statement for emulate MCU code in Simulink.
    * @param      _expression_ - expression for while.
    * @details    Данный while необходим, чтобы в конце симуляции, завершить поток МК:
    *             При выставлении флага окончания симуляции, все while будут
пропускаться
    *             и поток сможет дойти до конца функции main и завершить себя.
    */
#define sim_while(_expression_)
while((_expression_)&&(hmcu.fMCU_Stop == 0))

    /* DEFAULT WHILE */
    /**
      * @brief      Default/Native C while statement.
      * @param      _expression_ - expression for while.
      * @details    Данный while - аналог обычного while, без дополнительного
функционала.
      */
#define native_while(_expression_)                   for(; (_expression_); )
      /*************************************************************/

    //------------------  SIMULINK WHILE DEFINES  -----------------//
    //-------------------------------------------------------------//



    //-------------------------------------------------------------//
    //---------------- SIMULATE FUNCTIONS PROTOTYPES  -------------//
    /* Step simulation */
void MCU_Step_Simulation(SimStruct* S, time_T time);

/* MCU peripheral simulation */
void MCU_Periph_Simulation(void);

/* Initialize MCU simulation */
void SIM_Initialize_Simulation(void);

/* Deinitialize MCU simulation */
void SIM_deInitialize_Simulation(void);

/* Read inputs S-function  */
void MCU_readInputs(SimStruct* S);

/* Write pre-outputs S-function (out_buff states) */
void MCU_writeOutputs(SimStruct* S);

/* Write outputs of block of S-Function*/
void SIM_writeOutput(SimStruct* S);
//---------------- SIMULATE FUNCTIONS PROTOTYPES  -------------//
//-------------------------------------------------------------//

/** MCU_WRAPPER
    * @}
    */
#endif // _WRAPPER_CONF_H_
```

Файл run_mex.bat:

```
@echo off

set defines=-D"STM32F103xB" -D"USE_HAL_DRIVER"^
 -D"MATLAB"^
 -D"__sizeof_ptr=8"
:: ------------------------USERS PATHS AND CODE--------------------------
set includes_USER= -I"..\mcu_project\upp\Core\Inc" -I"..\mcu_project\upp\Core\upp"

set code_USER=.\App_Wrapper\main.c^
 .\App_Wrapper\app_io.c^
 ..\mcu_project\upp\Core\Src\gpio.c^
 ..\mcu_project\upp\Core\Src\adc.c^
 ..\mcu_project\upp\Core\Src\tim.c^
 ..\mcu_project\upp\Core\Src\stm32f1xx_hal_msp.c^
 ..\mcu_project\upp\Core\Src\stm32f1xx_it.c^
 ..\mcu_project\upp\Core\Src\system_stm32f1xx.c^
 ..\mcu_project\upp\Core\upp\upp.c^
 ..\mcu_project\upp\Core\upp\zero_cross.c^
 ..\mcu_project\upp\Core\upp\adc_filter.c^
 ..\mcu_project\upp\Core\upp\tiristor.c
::-----------------------------------------------------------------------

:: ----------------------MCU LIBRARIES & SIMULATOR----------------------
:: -----MCU LIBRARIES STUFF----
:: заголовочные файлы
set includes_MCU= -I".\MCU_STM32F1xx_Matlab"^
 -I".\MCU_STM32F1xx_Matlab\Drivers\STM32F1xx_SIMULINK"^
 -I".\MCU_STM32F1xx_Matlab\Drivers\CMSIS"^
 -I".\MCU_STM32F1xx_Matlab\Drivers\CMSIS\Device\STM32F1xx"^
 -I".\MCU_STM32F1xx_Matlab\Drivers\STM32F1xx_HAL_Driver\Inc"^
 -I".\MCU_STM32F1xx_Matlab\Drivers\STM32F1xx_HAL_Driver\Inc\Legacy"

:: код библиотек МК, переделанный для матлаб
set code_MCU=
.\MCU_STM32F1xx_Matlab\Drivers\STM32F1xx_HAL_Driver\Src\stm32f1xx_hal_rcc.c^
 .\MCU_STM32F1xx_Matlab\Drivers\STM32F1xx_HAL_Driver\Src\stm32f1xx_hal_gpio.c^
 .\MCU_STM32F1xx_Matlab\Drivers\STM32F1xx_HAL_Driver\Src\stm32f1xx_hal_pwr.c^
 .\MCU_STM32F1xx_Matlab\Drivers\STM32F1xx_HAL_Driver\Src\stm32f1xx_hal_cortex.c^
 .\MCU_STM32F1xx_Matlab\Drivers\STM32F1xx_HAL_Driver\Src\stm32f1xx_hal.c^
 .\MCU_STM32F1xx_Matlab\Drivers\STM32F1xx_HAL_Driver\Src\stm32f1xx_hal_adc.c^
 .\MCU_STM32F1xx_Matlab\Drivers\STM32F1xx_HAL_Driver\Src\stm32f1xx_hal_adc_ex.c^
 .\MCU_STM32F1xx_Matlab\Drivers\STM32F1xx_HAL_Driver\Src\stm32f1xx_hal_tim.c^
 .\MCU_STM32F1xx_Matlab\Drivers\STM32F1xx_HAL_Driver\Src\stm32f1xx_hal_tim_ex.c^
 .\MCU_STM32F1xx_Matlab\Drivers\STM32F1xx_HAL_Driver\Src\stm32f1xx_hal_dma.c^
 .\MCU_STM32F1xx_Matlab\Drivers\STM32F1xx_HAL_Driver\Src\stm32f1xx_hal_exti.c

::
.\MCU_STM32F1xx_Matlab\Drivers\STM32F1xx_HAL_Driver\Src\stm32f1xx_hal_flash_ramfunc.c^
:: .\MCU_STM32F1xx_Matlab\Drivers\STM32F1xx_HAL_Driver\Src\stm32f1xx_hal_flash.c^
:: .\MCU_STM32F1xx_Matlab\Drivers\STM32F1xx_HAL_Driver\Src\stm32f1xx_hal_flash_ex.c^
:: .\MCU_STM32F1xx_Matlab\Drivers\STM32F1xx_HAL_Driver\Src\stm32f1xx_hal_rcc_ex.c^

:: --------MCU SIMULATOR-------
:: код, которая будет симулировать перефирию МК в симулинке
set code_MCU_Sim= .\MCU_STM32F1xx_Matlab\stm32f1xx_matlab_conf.c^
 .\MCU_STM32F1xx_Matlab\Drivers\STM32F1xx_SIMULINK\stm32f1xx_matlab_gpio.c^
 .\MCU_STM32F1xx_Matlab\Drivers\STM32F1xx_SIMULINK\stm32f1xx_matlab_tim.c^
 .\MCU_STM32F1xx_Matlab\Drivers\STM32F1xx_SIMULINK\stm32f1xx_periph_registers.c
::-----------------------------------------------------------------------


:: ------------------------WRAPPER PATHS AND CODE-------------------------
```

```
:: оболочка, которая будет моделировать работу МК в симулинке
set includes_WRAPPER= -I".\MCU_Wrapper"
set code_WRAPPER= .\MCU_Wrapper\MCU.c^
 .\MCU_Wrapper\mcu_wrapper.c
::-------------------------------------------------------------------------


:: ---------------------SET PARAMS FOR MEX COMPILING----------------------
:: --------ALL INCLUDES--------
set includes= %includes_USER% %includes_MCU% %includes_WRAPPER%
set codes= %code_WRAPPER% %code_USER% %code_MCU% %code_MCU_Sim%
:: -------OUTPUT FOLDER--------
set output= -outdir "."

:: если нужен дебаг, до запускаем run mex с припиской debug
IF [%1]==[debug] (set debug= -g)
::-------------------------------------------------------------------------


::------START COMPILING-------
echo Compiling...
mex %output% %defines% %includes% %codes% %debug%
echo %DATE% %TIME%
```

ПРИЛОЖЕНИЕ Б

Листинг программы симуляции периферии МК

Файл app_io.c:

```
#include "stm32f1xx_matlab_gpio.h"
#include "upp.h"


#define detect_front(_in_numb_, _var_, _val_) {             \
if ((in[_in_numb_] > 0.5) && (prev_in[_in_numb_] <= 0.5))   \
{                                                           \
    _var_ = _val_;                                          \
} }

#define detect_rise(_in_numb_, _var_, _val_) {              \
if ((in[_in_numb_] < 0.5) && (prev_in[_in_numb_] >= 0.5))   \
{                                                           \
    _var_ = _val_;                                          \
} }



void WriteFromSFunc(real_T* disc)
{

    for (int i = 0; i < PORT_WIDTH; i++)
    {
        if (GPIOA->ODR & (1 << i))
        {
            disc[i] = 1;
        }

        if (GPIOB->ODR & (1 << i))
        {
            disc[PORT_WIDTH + i] = 1;
        }
    }
    disc[2 * PORT_WIDTH + 0] = phase_A.ctrl.angle.delay_us;
    disc[2 * PORT_WIDTH + 1] = (uint16_t)((uint16_t)TIMER->CNT -
phase_A.ctrl.angle.start_delay_tick);
    disc[2 * PORT_WIDTH + 2] = phase_A.ctrl.angle.start_delay_tick;
    disc[2 * PORT_WIDTH + 3] = TIMER->CNT;
}



void ReadToSFunc(real_T* in)
{
    static real_T prev_in[IN_PORT_WIDTH];

    detect_front(0, phase_A.zc_detector.f.EXTIZeroCrossDetected, 1);
    detect_rise(0, phase_A.zc_detector.f.EXTIZeroCrossDetected, 1);

    detect_front(1, phase_B.zc_detector.f.EXTIZeroCrossDetected, 1);
    detect_rise(1, phase_B.zc_detector.f.EXTIZeroCrossDetected, 1);

    detect_front(2, phase_C.zc_detector.f.EXTIZeroCrossDetected, 1);
    detect_rise(2, phase_C.zc_detector.f.EXTIZeroCrossDetected, 1);

    detect_front(3, Upp.GoSafe, 1);
    detect_rise(3, Upp.GoSafe, 0);
```

```
    detect_front(4, Upp.Prepare, 1);
    detect_rise(4, Upp.Prepare, 0);

    detect_front(5, Upp.ForceStop, 1);
    detect_rise(5, Upp.ForceStop, 0);

    detect_front(6, Upp.ForceDisconnect, 1);
    detect_rise(6, Upp.ForceDisconnect, 0);


    Upp.sine_freq = in[7];
    Upp.Duration = in[8];


    for (int i = 0; i < IN_PORT_WIDTH; i++)
    {
        prev_in[i] = in[i];
    }
}
```

Файл stm32f1xx_matlab_tim.c:

```
/**
******************************************************************************
* @file stm32f4xx_matlab_tim.c
* @brief Исходный код симулятора таймеров.
******************************************************************************
@details
Данный файл содержит функции для симуляции таймеров STM32F407xx.
******************************************************************************/
#include "stm32f4xx_matlab_tim.h"



struct SlaveChannels Slave_Channels; ///< структура для связи и
синхронизации таймеров


//----------------------TIMER BASE FUNCTIONS----------------------//
/**
  * @brief    Симуляция таймера на один такт симуляции.
  * @param    TIMx - таймер, каналы которого надо записать.
  * @param    TIMS - структура таймера для симуляции.
  * @details  Это базовая функция для симуляции таймера: она вызывается
каждый шаг симуляции
  *           и вызывает все другие функции, необходимые для симуляции:
  *           - Overflow_Check()
  *           - Slave_Mode_Check_Source()
  *           - TIMx_Count()
  *           - Channels_Simulation()
  */
void TIM_Simulation(TIM_TypeDef *TIMx, struct TIM_Sim *TIMS)
{
    Overflow_Check(TIMx, TIMS);



    // Выбор режима работы таймера
    switch (TIMx->SMCR & TIM_SMCR_SMS) // TIMER MODE
    {
        // обычный счет
        case(TIM_SLAVEMODE_DISABLE):// NORMAL MODE counting
            TIMx_Count(TIMx, TIMS);
            Channels_Simulation(TIMx, TIMS); // CaptureCompare and PWM
channels simulation
            break;


        // включение слейв таймера по ивенту
        case(TIM_SLAVEMODE_TRIGGER): // SLAVE MODE: TRIGGER MODE
            Slave_Mode_Check_Source(TIMx, TIMS);
            TIMx_Count(TIMx, TIMS);
            Channels_Simulation(TIMx, TIMS); // CaptureCompare and PWM
channels simulation
            break;
    }

}
/**
  * @brief    Симуляция счетчика таймера на один такт симуляции.
  * @param    TIMx - таймер, каналы которого надо записать.
  * @param    TIMS - структура таймера для симуляции.
```

```
  * @details  Данная функция проверяет направление таймера и увеличивает или
уменьшает
  *           значение счетчика на то число, на которое оно бы увеличилось
за шаг симуляции.
  * @note     Для счетчика используется double формат, т.к. кол-во счетов за
шаг симуляции
              может быть дробным. После в конце функции double счетчик
записывает с округлением
              в регистр таймера CNT.
  */
void TIMx_Count(TIM_TypeDef* TIMx, struct TIM_Sim* TIMS)
{
    if ((TIMx->CR1 & TIM_CR1_DIR) && TIMx->CR1) // up COUNTER and COUNTER
ENABLE
        TIMS->tx_cnt -= TIMS->tx_step / TIMx->PSC;
    else if (((TIMx->CR1 & TIM_CR1_DIR) == 0) && TIMx->CR1) // down COUNTER
and COUNTER ENABLE
        TIMS->tx_cnt += TIMS->tx_step / TIMx->PSC;
    TIMx->CNT = (uint32_t)TIMS->tx_cnt;
}

/**
  * @brief    Проверка на переполнение и дальнейшая его обработка.
  * @param    TIMx - таймер, каналы которого надо записать.
  * @param    TIMS - структура таймера для симуляции.
  * @details  Данная функция проверяет когда таймер переполниться и если
надо,
              вызывает соответствующее прерывание:
              - call_IRQHandller()
  */
void Overflow_Check(TIM_TypeDef* TIMx, struct TIM_Sim* TIMS)
{
    // Переполнение таймера: сброс таймера и вызов прерывания
    if ((TIMx->CR1 & TIM_CR1_UDIS) == 0) // UPDATE enable
    {
        if ((TIMx->CR1 & TIM_CR1_ARPE) == 0)  TIMS->RELOAD = TIMx->ARR; //
PRELOAD disable - update ARR every itteration
        if (TIMS->tx_cnt > TIMS->RELOAD || TIMS->tx_cnt < 0) // OVERFLOW
        {
            TIMS->RELOAD = TIMx->ARR; // RELOAD ARR

            if (TIMS->tx_cnt > TIMx->ARR) // reset COUNTER
                TIMS->tx_cnt = 0;
            else if (TIMS->tx_cnt < 0)
                TIMS->tx_cnt = TIMx->ARR;

            if(TIMx->DIER & TIM_DIER_UIE) // if update interrupt enable
                call_IRQHandller(TIMx); // call HANDLER
        }
    }
}
//-----------------------------------------------------------------//


//----------------------------CHANNELS-----------------------------//
/**
  * @brief    Симуляция каналов таймера.
  * @param    TIMx - таймер, каналы которого надо записать.
  * @param    TIMS - структура таймера для симуляции.
  * @details  Данная функция симулирует работу всех каналов таймера.
  *           Она вызывает функции:
  *           - CC_PWM_Ch1_Simulation()
  *           - CC_PWM_Ch2_Simulation()
```

```
  *           - CC_PWM_Ch3_Simulation()
  *           - CC_PWM_Ch4_Simulation()
  *           - Write_OC_to_GPIO()
  *           - Write_OC_to_TRGO()
  */
void Channels_Simulation(TIM_TypeDef* TIMx, struct TIM_Sim* TIMS)
{
    CC_PWM_Ch1_Simulation(TIMx, TIMS);
    CC_PWM_Ch2_Simulation(TIMx, TIMS);
    CC_PWM_Ch3_Simulation(TIMx, TIMS);
    CC_PWM_Ch4_Simulation(TIMx, TIMS);

    Write_OC_to_GPIO(TIMx, TIMS);

    Write_OC_to_TRGO(TIMx, TIMS);
}
//-----------------CAPTURE COPMARE & PWM FUNCTIONS-----------------//
/**
  * @brief    Выбор режима первого канала и его симуляция.
  * @param    TIMx - таймер, каналы которого надо записать.
  * @param    TIMS - структура таймера для симуляции.
  * @details  Данная функция по регистрам таймера проверяет как настроен
              первый канал и соответствующе симулирует его работу.
  */
void CC_PWM_Ch1_Simulation(TIM_TypeDef *TIMx, struct TIM_Sim *TIMS)
{ // определяет режим канала
switch (TIMx->CCMR1 & TIM_CCMR1_OC1M)
{
    case (TIM_OCMODE_ACTIVE): // ACTIVE mode
        if (abs(TIMx->CNT - TIMx->CCR1) < 2*TIMS->tx_step)
            TIMS->Channels.OC1REF = 1;
        break;

    case (TIM_OCMODE_INACTIVE): // INACTIVE mode
        if (abs(TIMx->CNT - TIMx->CCR1) < 2*TIMS->tx_step)
            TIMS->Channels.OC1REF = 0;
        break;

    case (TIM_OCMODE_TOGGLE): // TOOGLE mode
        if (abs(TIMx->CNT - TIMx->CCR1) < 2*TIMS->tx_step)
            TIMS->Channels.OC1REF = ~TIMS->Channels.OC1REF;
        break;

    case (TIM_OCMODE_PWM1): // PWM MODE 1 mode
        if (TIMx->CNT < TIMx->CCR1)
            TIMS->Channels.OC1REF = 1;
        else
            TIMS->Channels.OC1REF = 0;
        break;

    case (TIM_OCMODE_PWM2): // PWM MODE 2 mode
        if (TIMx->CNT < TIMx->CCR1)
            TIMS->Channels.OC1REF = 0;
        else
            TIMS->Channels.OC1REF = 1;
        break;

    case (TIM_OCMODE_FORCED_ACTIVE): // FORCED ACTIVE mode
        TIMS->Channels.OC1REF = 1; break;

    case (TIM_OCMODE_FORCED_INACTIVE): // FORCED INACTIVE mode
        TIMS->Channels.OC1REF = 0; break;
```

```
}
}
/**
  * @brief    Выбор режима второго канала и его симуляция.
  * @param    TIMx - таймер, каналы которого надо записать.
  * @param    TIMS - структура таймера для симуляции.
  * @details  Данная функция по регистрам таймера проверяет как настроен
              второй канал и соответствующе симулирует его работу.
  */
void CC_PWM_Ch2_Simulation(TIM_TypeDef *TIMx, struct TIM_Sim *TIMS)
{ // определяет режим канала
switch (TIMx->CCMR1 & TIM_CCMR1_OC2M)
{
    case ((TIM_OCMODE_ACTIVE) << (TIM_OCMODE_SECOND_SHIFT)): // ACTIVE mode
        if (abs(TIMx->CNT - TIMx->CCR2) < 2*TIMS->tx_step)
            TIMS->Channels.OC2REF = 1;
        break;

    case ((TIM_OCMODE_INACTIVE) << (TIM_OCMODE_SECOND_SHIFT)): // INACTIVE
mode
        if (abs(TIMx->CNT - TIMx->CCR2) < 2*TIMS->tx_step)
            TIMS->Channels.OC2REF = 0;
        break;

    case ((TIM_OCMODE_TOGGLE) << (TIM_OCMODE_SECOND_SHIFT)): // Toogle mode
        if (abs(TIMx->CNT - TIMx->CCR2) < 2*TIMS->tx_step)
            TIMS->Channels.OC2REF = ~TIMS->Channels.OC2REF;
        break;

    case ((TIM_OCMODE_PWM1) << (TIM_OCMODE_SECOND_SHIFT)): // PWM mode 1
mode
        if (TIMx->CNT < TIMx->CCR2)
            TIMS->Channels.OC2REF = 1;
        else
            TIMS->Channels.OC2REF = 0;
        break;

    case ((TIM_OCMODE_PWM2) << (TIM_OCMODE_SECOND_SHIFT)): // PWM mode 2
mode
        if (TIMx->CNT < TIMx->CCR2)
            TIMS->Channels.OC2REF = 0;
        else
            TIMS->Channels.OC2REF = 1;
        break;

    case ((TIM_OCMODE_FORCED_ACTIVE) << (TIM_OCMODE_SECOND_SHIFT)): //
FORCED ACTIVE mode
        TIMS->Channels.OC2REF = 1; break;

    case ((TIM_OCMODE_FORCED_INACTIVE) << (TIM_OCMODE_SECOND_SHIFT)): //
FORCED INACTIVE mode
        TIMS->Channels.OC2REF = 0; break;

}
}
/**
  * @brief    Выбор режима третьего канала и его симуляция.
  * @param    TIMx - таймер, каналы которого надо записать.
  * @param    TIMS - структура таймера для симуляции.
  * @details  Данная функция по регистрам таймера проверяет как настроен
              третий канал и соответствующе симулирует его работу.
  */
void CC_PWM_Ch3_Simulation(TIM_TypeDef *TIMx, struct TIM_Sim *TIMS)
```

```
{ // определяет режим канала
switch (TIMx->CCMR2 & TIM_CCMR1_OC1M)
{
    case (TIM_OCMODE_ACTIVE): // ACTIVE mode
        if (abs(TIMx->CNT - TIMx->CCR3) < 2*TIMS->tx_step)
            TIMS->Channels.OC3REF = 1;
        break;

    case (TIM_OCMODE_INACTIVE): // INACTIVE mode
        if (abs(TIMx->CNT - TIMx->CCR3) < 2*TIMS->tx_step)
            TIMS->Channels.OC3REF = 0;
        break;

    case (TIM_OCMODE_TOGGLE): // Toogle mode
        if (abs(TIMx->CNT - TIMx->CCR3) < 2*TIMS->tx_step)
            TIMS->Channels.OC3REF = ~TIMS->Channels.OC3REF;
        break;

    case (TIM_OCMODE_PWM1): // PWM mode 1 mode
        if (TIMx->CNT < TIMx->CCR3)
            TIMS->Channels.OC3REF = 1;
        else
            TIMS->Channels.OC3REF = 0;
        break;

    case (TIM_OCMODE_PWM2): // PWM mode 2 mode
        if (TIMx->CNT < TIMx->CCR3)
            TIMS->Channels.OC3REF = 0;
        else
            TIMS->Channels.OC3REF = 1;
        break;

    case (TIM_OCMODE_FORCED_ACTIVE): // FORCED ACTIVE mode
        TIMS->Channels.OC3REF = 1; break;

    case (TIM_OCMODE_FORCED_INACTIVE): // FORCED INACTIVE mode
        TIMS->Channels.OC3REF = 0; break;

}
}
/**
  * @brief    Выбор режима четвертого канала и его симуляция.
  * @param    TIMx – таймер, каналы которого надо записать.
  * @param    TIMS – структура таймера для симуляции.
  * @details  Данная функция по регистрам таймера проверяет как настроен
              четвертый канал и соответствующе симулирует его работу.
  */
void CC_PWM_Ch4_Simulation(TIM_TypeDef *TIMx, struct TIM_Sim *TIMS)
{ // определяет режим канала
switch (TIMx->CCMR2 & TIM_CCMR1_OC2M)
{
    case ((TIM_OCMODE_ACTIVE) << (TIM_OCMODE_SECOND_SHIFT)): // ACTIVE mode
        if (abs(TIMx->CNT - TIMx->CCR4) < 2*TIMS->tx_step)
            TIMS->Channels.OC4REF = 1;
        break;

    case ((TIM_OCMODE_INACTIVE) << (TIM_OCMODE_SECOND_SHIFT)): // INACTIVE
mode
        if (abs(TIMx->CNT - TIMx->CCR4) < 2*TIMS->tx_step)
            TIMS->Channels.OC4REF = 0;
        break;

    case ((TIM_OCMODE_TOGGLE) << (TIM_OCMODE_SECOND_SHIFT)): // Toogle mode
```

```
        if (abs(TIMx->CNT - TIMx->CCR4) < 2*TIMS->tx_step)
            TIMS->Channels.OC4REF = ~TIMS->Channels.OC4REF;
        break;

    case ((TIM_OCMODE_PWM1) << (TIM_OCMODE_SECOND_SHIFT)): // PWM mode 1
mode
        if (TIMx->CNT < TIMx->CCR4)
            TIMS->Channels.OC4REF = 1;
        else
            TIMS->Channels.OC4REF = 0;
        break;

    case ((TIM_OCMODE_PWM2) << (TIM_OCMODE_SECOND_SHIFT)): // PWM mode 2
mode
        if (TIMx->CNT < TIMx->CCR4)
            TIMS->Channels.OC4REF = 0;
        else
            TIMS->Channels.OC4REF = 1;
        break;

    case ((TIM_OCMODE_FORCED_ACTIVE) << (TIM_OCMODE_SECOND_SHIFT)): //
FORCED ACTIVE mode
        TIMS->Channels.OC4REF = 1; break;

    case ((TIM_OCMODE_FORCED_INACTIVE) << (TIM_OCMODE_SECOND_SHIFT)): //
FORCED INACTIVE mode
        TIMS->Channels.OC4REF = 0; break;

}
}

/**
  * @brief    Запись каналов таймера в порты GPIO.
  * @param    TIMx - таймер, каналы которого надо записать.
  * @param    TIMS - структура того же таймера для симуляции.
  * @details  Данная функция записывает каналы OC в порты GPIO, определенные
в TIMS.
  *           Запись происходит только если пин настроен на альтернативную
функцию.
  */
void Write_OC_to_GPIO(TIM_TypeDef *TIMx, struct TIM_Sim *TIMS)
{
    // write gpio pin if need
    if (Check_OC1_GPIO_Output(TIMS)) // check OC OUTPUT 4 enable (GPIO AF
MODE)
    {
        uint32_t temp2 = ~(uint32_t)(1 << (TIMS->Channels.OC1_PIN_SHIFT));
        if (TIMx->CCER & TIM_CCER_CC1P) // POLARITY check
        {   // low POLARITY
            if (TIMS->Channels.OC1REF)
                TIMS->Channels.OC1_GPIOx->ODR &= ~(uint32_t)(1 << (TIMS-
>Channels.OC1_PIN_SHIFT));
            else
                TIMS->Channels.OC1_GPIOx->ODR |= 1 << (TIMS-
>Channels.OC1_PIN_SHIFT);
        }
        else
        {   // high POLARITY
            if (TIMS->Channels.OC1REF)
                TIMS->Channels.OC1_GPIOx->ODR |= 1 << (TIMS-
>Channels.OC1_PIN_SHIFT);
            else
```

```
                TIMS->Channels.OC1_GPIOx->ODR &= ~(uint32_t)(1 << (TIMS-
>Channels.OC1_PIN_SHIFT));
        }
    }
    if (Check_OC2_GPIO_Output(TIMS)) // check OC OUTPUT 4 enable (GPIO AF
MODE)
    {
        if (TIMx->CCER & TIM_CCER_CC2P) // POLARITY check
        {   // low POLARITY
            if (TIMS->Channels.OC2REF)
                TIMS->Channels.OC2_GPIOx->ODR &= ~(uint32_t)(1 << (TIMS-
>Channels.OC2_PIN_SHIFT));
            else
                TIMS->Channels.OC2_GPIOx->ODR |= 1 << (TIMS-
>Channels.OC2_PIN_SHIFT);
        }
        else
        {   // high POLARITY
            if (TIMS->Channels.OC2REF)
                TIMS->Channels.OC2_GPIOx->ODR |= 1 << (TIMS-
>Channels.OC2_PIN_SHIFT);
            else
                TIMS->Channels.OC2_GPIOx->ODR &= ~(uint32_t)(1 << (TIMS-
>Channels.OC2_PIN_SHIFT));
        }
    }
    if (Check_OC3_GPIO_Output(TIMS)) // check OC OUTPUT 4 enable (GPIO AF
MODE)
    {
        if (TIMx->CCER & TIM_CCER_CC3P) // POLARITY check
        {   // low POLARITY
            if (TIMS->Channels.OC3REF)
                TIMS->Channels.OC3_GPIOx->ODR &= ~(uint32_t)(1 << (TIMS-
>Channels.OC3_PIN_SHIFT));
            else
                TIMS->Channels.OC3_GPIOx->ODR |= 1 << (TIMS-
>Channels.OC3_PIN_SHIFT);
        }
        else
        {   // high POLARITY
            if (TIMS->Channels.OC3REF)
                TIMS->Channels.OC3_GPIOx->ODR |= 1 << (TIMS-
>Channels.OC3_PIN_SHIFT);
            else
                TIMS->Channels.OC3_GPIOx->ODR &= ~(uint32_t)(1 << (TIMS-
>Channels.OC3_PIN_SHIFT));
        }
    }
    if (Check_OC4_GPIO_Output(TIMS)) // check OC CHANNEL 4 enable (GPIO AF
MODE)
    {
        if (TIMx->CCER & TIM_CCER_CC4P) // POLARITY check
        {   // low POLARITY
            if (TIMS->Channels.OC4REF)
                TIMS->Channels.OC4_GPIOx->ODR &= ~(uint32_t)(1 << (TIMS-
>Channels.OC4_PIN_SHIFT));
            else
                TIMS->Channels.OC4_GPIOx->ODR |= (1) << (TIMS-
>Channels.OC4_PIN_SHIFT);
        }
        else
        {   // high POLARITY
            if (TIMS->Channels.OC4REF)
```

```
                TIMS->Channels.OC4_GPIOx->ODR |= (1) << (TIMS-
>Channels.OC4_PIN_SHIFT);
            else
                TIMS->Channels.OC4_GPIOx->ODR &= ~(uint32_t)(1 << (TIMS-
>Channels.OC4_PIN_SHIFT));
        }
    }
}
/** Запись результата compare в глабальную структуру с TRIGGER OUTPUT */
/**
  * @brief    Запись каналов таймера в глабальную структуру с TRIGGER
OUTPUT.
  * @param    TIMx - таймер, каналы которого надо записать.
  * @param    TIMS - структура того же таймера для симуляции.
  * @details  Данная функция считывает каналы OC и записывает их в внешний
канал триггера TRGO.
  */
void Write_OC_to_TRGO(TIM_TypeDef* TIMx, struct TIM_Sim* TIMS)
{
    // write trigger output from OCxREF pin if need
    unsigned temp_trgo;
    if ((TIMx->CR2 & TIM_CR2_MMS) == (0b100 << TIM_CR2_MMS_Pos))
    {
        temp_trgo = TIMS->Channels.OC1REF;
    }
    else if ((TIMx->CR2 & TIM_CR2_MMS) == (0b101 << TIM_CR2_MMS_Pos))
    {
        temp_trgo = TIMS->Channels.OC2REF;
    }
    else if ((TIMx->CR2 & TIM_CR2_MMS) == (0b110 << TIM_CR2_MMS_Pos))
    {
        temp_trgo = TIMS->Channels.OC3REF;
    }
    else if ((TIMx->CR2 & TIM_CR2_MMS) == (0b111 << TIM_CR2_MMS_Pos))
    {
        temp_trgo = TIMS->Channels.OC4REF;
    }
    // select TIMx TRGO
    if (TIMx == TIM1)
        Slave_Channels.TIM1_TRGO = temp_trgo;
    else if (TIMx == TIM2)
        Slave_Channels.TIM2_TRGO = temp_trgo;
    else if (TIMx == TIM3)
        Slave_Channels.TIM3_TRGO = temp_trgo;
    else if (TIMx == TIM4)
        Slave_Channels.TIM4_TRGO = temp_trgo;
    else if (TIMx == TIM5)
        Slave_Channels.TIM5_TRGO = temp_trgo;
    else if (TIMx == TIM6)
        Slave_Channels.TIM6_TRGO = temp_trgo;
    else if (TIMx == TIM7)
        Slave_Channels.TIM7_TRGO = temp_trgo;
    else if (TIMx == TIM8)
        Slave_Channels.TIM8_TRGO = temp_trgo;
    temp_trgo = 0;
}
//-----------------------------------------------------------------------//

//--------------------MISC (temporary) FUNCTIONS--------------------//
/** Определение источника для запуска таймера в SLAVE MODE */
/**
  * @brief    Определение источника для запуска таймера в SLAVE MODE.
  * @param    TIMx - таймер, который надо включить.
```

```
  * @param    TIMx - таймер, прерываний которого надо вызвать.
  * @details  Данная функция проверяет какой триггер выбран для запуска
таймера,
  *           после записывает значение канала триггера в бит включения
таймера.
  *           Таким образом, при лог.1 в канале триггера - таймер
включиться.
  */
void Slave_Mode_Check_Source(TIM_TypeDef* TIMx)
{
    if (TIMx == TIM2)
    {
        if ((TIMx->SMCR & TIM_SMCR_TS) == TIM_TS_ITR0)
            TIMx->CR1 |= (Slave_Channels.TIM1_TRGO << TIM_CR1_CEN_Pos);
        else if ((TIMx->SMCR & TIM_SMCR_TS) == TIM_TS_ITR1)
            TIMx->CR1 |= (Slave_Channels.TIM1_TRGO << TIM_CR1_CEN_Pos);
        else if ((TIMx->SMCR & TIM_SMCR_TS) == TIM_TS_ITR2)
            TIMx->CR1 |= (Slave_Channels.TIM1_TRGO << TIM_CR1_CEN_Pos);
        else if ((TIMx->SMCR & TIM_SMCR_TS) == TIM_TS_ITR3)
            TIMx->CR1 |= (Slave_Channels.TIM8_TRGO << TIM_CR1_CEN_Pos);
    }
    else if (TIMx == TIM3)
    {
        if ((TIMx->SMCR & TIM_SMCR_TS) == TIM_TS_ITR0)
            TIMx->CR1 |= (Slave_Channels.TIM8_TRGO << TIM_CR1_CEN_Pos);
        else if ((TIMx->SMCR & TIM_SMCR_TS) == TIM_TS_ITR1)
            TIMx->CR1 |= (Slave_Channels.TIM2_TRGO << TIM_CR1_CEN_Pos);
        else if ((TIMx->SMCR & TIM_SMCR_TS) == TIM_TS_ITR2)
            TIMx->CR1 |= (Slave_Channels.TIM2_TRGO << TIM_CR1_CEN_Pos);
        else if ((TIMx->SMCR & TIM_SMCR_TS) == TIM_TS_ITR3)
            TIMx->CR1 |= (Slave_Channels.TIM3_TRGO << TIM_CR1_CEN_Pos);
    }
    else if (TIMx == TIM4)
    {
        if ((TIMx->SMCR & TIM_SMCR_TS) == TIM_TS_ITR0)
            TIMx->CR1 |= (Slave_Channels.TIM3_TRGO << TIM_CR1_CEN_Pos);
        else if ((TIMx->SMCR & TIM_SMCR_TS) == TIM_TS_ITR1)
            TIMx->CR1 |= (Slave_Channels.TIM5_TRGO << TIM_CR1_CEN_Pos);
        else if ((TIMx->SMCR & TIM_SMCR_TS) == TIM_TS_ITR2)
            TIMx->CR1 |= (Slave_Channels.TIM3_TRGO << TIM_CR1_CEN_Pos);
        else if ((TIMx->SMCR & TIM_SMCR_TS) == TIM_TS_ITR3)
            TIMx->CR1 |= (Slave_Channels.TIM4_TRGO << TIM_CR1_CEN_Pos);
    }
    else if (TIMx == TIM5)
    {
        if ((TIMx->SMCR & TIM_SMCR_TS) == TIM_TS_ITR0)
            TIMx->CR1 |= (Slave_Channels.TIM4_TRGO << TIM_CR1_CEN_Pos);
        else if ((TIMx->SMCR & TIM_SMCR_TS) == TIM_TS_ITR1)
            TIMx->CR1 |= (Slave_Channels.TIM4_TRGO << TIM_CR1_CEN_Pos);
        else if ((TIMx->SMCR & TIM_SMCR_TS) == TIM_TS_ITR2)
            TIMx->CR1 |= (Slave_Channels.TIM7_TRGO << TIM_CR1_CEN_Pos);
        else if ((TIMx->SMCR & TIM_SMCR_TS) == TIM_TS_ITR3)
            TIMx->CR1 |= (Slave_Channels.TIM7_TRGO << TIM_CR1_CEN_Pos);
    }
}
//-----------------------------------------------------------------------//

//------------------------SIMULINK FUNCTIONS-----------------------------//
/** Симулирование выбранных через дефайн таймеров */
/**
  * @brief    Симуляция выбранных таймеров.
  * @details  Таймеры для симуляции выбираются через дефайны в
stm32f4xx_matlab_conf.h.
```

```
                Функция вызывается на каждом шаге симуляции.
  *
  * @attention  Добавить это в функцию MCU_Periph_Simulation()
  */
void Simulate_TIMs(void)
{
#ifdef USE_TIM1
    TIM_Simulation(TIM1, &tim1s);
#endif
#ifdef USE_TIM2
    TIM_Simulation(TIM2, &tim2s);
#endif
#ifdef USE_TIM3
    TIM_Simulation(TIM3, &tim3s);
#endif
#ifdef USE_TIM4
    TIM_Simulation(TIM4, &tim4s);
#endif
#ifdef USE_TIM5
    TIM_Simulation(TIM5, &tim5s);
#endif
#ifdef USE_TIM6
    TIM_Simulation(TIM6, &tim6s);
#endif
#ifdef USE_TIM7
    TIM_Simulation(TIM7, &tim7s);
#endif
#ifdef USE_TIM8
    TIM_Simulation(TIM8, &tim8s);
#endif
#ifdef USE_TIM9
    TIM_Simulation(TIM9, &tim9s);
#endif
#ifdef USE_TIM10
    TIM_Simulation(TIM10, &tim10s);
#endif
#ifdef USE_TIM11
    TIM_Simulation(TIM11, &tim11s);
#endif
#ifdef USE_TIM12
    TIM_Simulation(TIM12, &tim12s);
#endif
#ifdef USE_TIM13
    TIM_Simulation(TIM13, &tim13s);
#endif
#ifdef USE_TIM14
    TIM_Simulation(TIM14, &tim14s);
#endif
}
/**
  * @brief    Деинициализирование выбранных таймеров.
  * @details  Таймеры для деинициализизации выбираются через дефайны в
stm32f4xx_matlab_conf.h.
                Функция вызывается в конце симуляции.
  */
void TIM_SIM_DEINIT(void)
{
#ifdef USE_TIM1
    memset(&tim1s, 0, sizeof(tim1s));
#endif
#ifdef USE_TIM2
    memset(&tim2s, 0, sizeof(tim2s));
#endif
```

```
#ifdef USE_TIM3
    memset(&tim3s, 0, sizeof(tim3s));
#endif
#ifdef USE_TIM4
    memset(&tim4s, 0, sizeof(tim4s));
#endif
#ifdef USE_TIM5
    memset(&tim5s, 0, sizeof(tim5s));
#endif
#ifdef USE_TIM6
    memset(&tim6s, 0, sizeof(tim6s));
#endif
#ifdef USE_TIM7
    memset(&tim7s, 0, sizeof(tim7s));
#endif
#ifdef USE_TIM8
    memset(&tim8s, 0, sizeof(tim8s));
#endif
#ifdef USE_TIM9
    memset(&tim9s, 0, sizeof(tim9s));
#endif
#ifdef USE_TIM10
    memset(&tim10s, 0, sizeof(tim10s));
#endif
#ifdef USE_TIM11
    memset(&tim11s, 0, sizeof(tim11s));
#endif
#ifdef USE_TIM12
    memset(&tim12s, 0, sizeof(tim12s));
#endif
#ifdef USE_TIM13
    memset(&tim13s, 0, sizeof(tim13s));
#endif
#ifdef USE_TIM14
    memset(&tim14s, 0, sizeof(tim14s));
#endif
}
//-----------------------------------------------------------------------//

//------------------TIM'S HANDLERS (BETA) FUNCTIONS-----------------//
// Определение обработчиков, которые не используются
// Т.к. в MSVC нет понятия weak function, необходимо объявить все колбеки
// И если какой-то колбек не используется, его надо определить
#ifndef USE_TIM1_UP_TIM10_HANDLER
void TIM1_UP_TIM10_IRQHandler(void) {}
#endif
#ifndef USE_TIM2_HANDLER
void TIM2_IRQHandler(void) {}
#endif
#ifndef USE_TIM3_HANDLER
void TIM3_IRQHandler(void) {}
#endif
#ifndef USE_TIM4_HANDLER
void TIM4_IRQHandler(void) {}
#endif

/**
  * @brief    Вызов прерывания таймера TIMx.
  * @param    TIMx - таймер, прерываний которого надо вызвать.
  * @details  Данная функция симулирует аппаратный вызов прерывания
              таймера по какому-либо событию.
  */
void call_IRQHandller(TIM_TypeDef* TIMx)
```

```
{ // calling HANDLER
    if ((TIMx == TIM1) || (TIMx == TIM10))
        TIM1_UP_TIM10_IRQHandler();
    else if (TIMx == TIM2)
        TIM2_IRQHandler();
    else if (TIMx == TIM3)
        TIM3_IRQHandler();
    else if (TIMx == TIM4)
        TIM4_IRQHandler();
}
//------------------------------------------------------------------------//
```

Файл stm32f1xx_matlab_tim.h:

```
/**
******************************************************************************
* @file stm32f1xx_matlab_tim.h
* @brief Заголовочный файл для симулятора таймеров.
******************************************************************************
@details
Данный файл содержит объявления всякого для симуляции таймеров STM32F407xx.
******************************************************************************/
#ifndef _MATLAB_TIM_H_
#define _MATLAB_TIM_H_

#include "stm32f4xx_hal.h"
#include "stm32f4xx_it.h"
#include "mcu_wrapper_conf.h"

/**
    * @addtogroup   TIM_SIMULATOR   TIM Simulator
    * @ingroup      MAIN_SIMULATOR
    * @brief        Симулятор для таймеров
    * @details      Дефайны и функции для симуляции таймеров.
    * @{
    */

//////////////////////////////---DEFINES---//////////////////////////////
/**
  * @brief Дефайн для сдвига между первой и второй половиной CCMRx регистров
  */
#define TIM_OCMODE_SECOND_SHIFT                 (TIM_CCMR1_OC2M_Pos -
TIM_CCMR1_OC1M_Pos)

/**
  * @brief    Дефайн для проверки выводить ли канал таймера на GPIO
  * @details  Данный дефайн проверяет, настроен ли пин GPIO на
альтернативную функцию. Если да - то таймер выводиться на этот пин
  */
#define Check_OCx_GPIO_Output(_tims_, _OCx_GPIOx_, _OCx_PIN_SHIFT_)
(_tims_->Channels._OCx_GPIOx_->MODER & (0b11<<(2*_tims_-
>Channels._OCx_PIN_SHIFT_))) == (0b10<<(2*_tims_->Channels._OCx_PIN_SHIFT_))
/**
  * @brief Дефайн для проверки выводить ли канал 1 на GPIO (настроен ли GPIO
на альтернативную функцию)
  */
#define Check_OC1_GPIO_Output(_tims_)
Check_OCx_GPIO_Output(_tims_, OC1_GPIOx, OC1_PIN_SHIFT)
/**
  * @brief Дефайн для проверки выводить ли канал 2 на GPIO (настроен ли GPIO
на альтернативную функцию)
  */
#define Check_OC2_GPIO_Output(_tims_)
Check_OCx_GPIO_Output(_tims_, OC2_GPIOx, OC2_PIN_SHIFT)
/**
  * @brief Дефайн для проверки выводить ли канал 3 на GPIO (настроен ли GPIO
на альтернативную функцию)
  */
#define Check_OC3_GPIO_Output(_tims_)
Check_OCx_GPIO_Output(_tims_, OC3_GPIOx, OC3_PIN_SHIFT)
/**
  * @brief Дефайн для проверки выводить ли канал 4 на GPIO (настроен ли GPIO
на альтернативную функцию)
  */
```

```
#define Check_OC4_GPIO_Output(_tims_)
Check_OCx_GPIO_Output(_tims_, OC4_GPIOx, OC4_PIN_SHIFT)

////////////////////////////////////////////////////////////////////////



////////////////////////////---STRUCTURES---//////////////////////////
/**
  * @brief Структура для управления Слейв Таймерами
  */
struct SlaveChannels
{
    unsigned TIM1_TRGO : 1;      ///< Синган синхронизации таймера 1
    unsigned TIM2_TRGO : 1;      ///< Синган синхронизации таймера 2
    unsigned TIM3_TRGO : 1;      ///< Синган синхронизации таймера 3
    unsigned TIM4_TRGO : 1;      ///< Синган синхронизации таймера 4
    unsigned TIM5_TRGO : 1;      ///< Синган синхронизации таймера 5
    unsigned TIM6_TRGO : 1;      ///< Синган синхронизации таймера 6
    unsigned TIM7_TRGO : 1;      ///< Синган синхронизации таймера 7
    unsigned TIM8_TRGO : 1;      ///< Синган синхронизации таймера 8

};

/**
  * @brief Структура для моделирования каналов таймера
  */
struct Channels_Sim
{
    // Каналы таймера
    unsigned OC1REF:1;           ///< Первый канал
    unsigned OC2REF:1;           ///< Второй канал
    unsigned OC3REF:1;           ///< Третий канал
    unsigned OC4REF:1;           ///< Четвертый канал

    // связанные с каналами GPIO порты и пины
    GPIO_TypeDef *OC1_GPIOx;     ///< Порт первого канала
    uint32_t OC1_PIN_SHIFT;      ///< Пин первого канала

    GPIO_TypeDef *OC2_GPIOx;     ///< Порт второго канала
    uint32_t OC2_PIN_SHIFT;      ///< Пин второго канала

    GPIO_TypeDef *OC3_GPIOx;     ///< Порт третьего канала
    uint32_t OC3_PIN_SHIFT;      ///< Пин третьего канала

    GPIO_TypeDef *OC4_GPIOx;     ///< Порт четвертого канала
    uint32_t OC4_PIN_SHIFT;      ///< Пин четвертого канала

};

/**
  * @brief Структура для моделирования таймера
  */
struct TIM_Sim
{
    double tx_cnt;                  ///< Счетчик таймера (double, т.к. кол-
во тактов за шаг симуляции может быть дробным)
    double tx_step;                 ///< Шаг счета за один шаг симуляции
(double, т.к. кол-во тактов за шаг симуляции может быть дробным)
    int RELOAD;                     ///< Буфер для периода таймера (для
реализации функции PRELOAD)
    struct Channels_Sim Channels;   ///< Структура для симуляции каналов
};
```

```
////////////////////////////////////////////////////////////////////////////



////////////////////////////---FUNCTIONS---////////////////////////////

//----------------------TIMER BASE FUNCTIONS-----------------------//
/* Базовая функция для симуляции таймера: она вызывается каждый шаг
симуляции */
void TIM_Simulation(TIM_TypeDef *TIMx, struct TIM_Sim *TIMS);
/* Счет таймера за один такт */
void TIMx_Count(TIM_TypeDef* TIMx, struct TIM_Sim* TIMS);
/* Проверка на переполнение и дальнейшая его обработка */
void Overflow_Check(TIM_TypeDef* TIMx, struct TIM_Sim* TIMS);
/* Вызов прерывания */
void call_IRQHandller(TIM_TypeDef *TIMx);
//----------------------------------------------------------------//


//------------------------CHANNELS FUNCTIONS----------------------//
/* Симуляция каналов таймера */
void Channels_Simulation(TIM_TypeDef *TIMx, struct TIM_Sim *TIMS);
/*---------------- - CAPTURE COPMARE & PWM FUNCTIONS-----------------*/
/* Выбор режима CaptureCompare или PWM и симуляция для каждого канала */
void CC_PWM_Ch1_Simulation(TIM_TypeDef* TIMx, struct TIM_Sim* TIMS);
void CC_PWM_Ch2_Simulation(TIM_TypeDef* TIMx, struct TIM_Sim* TIMS);
void CC_PWM_Ch3_Simulation(TIM_TypeDef* TIMx, struct TIM_Sim* TIMS);
void CC_PWM_Ch4_Simulation(TIM_TypeDef* TIMx, struct TIM_Sim* TIMS);
/* Запись каналов таймера в порты GPIO */
void Write_OC_to_GPIO(TIM_TypeDef* TIMx, struct TIM_Sim* TIMS);
/* Запись результата compare в глабальную структуру с TRIGGER OUTPUT */
void Write_OC_to_TRGO(TIM_TypeDef* TIMx, struct TIM_Sim* TIMS);
//----------------------------------------------------------------//


//--------------------MISC (temporary) FUNCTIONS------------------//
/* Определение источника для запуска таймера в SLAVE MODE */
void Slave_Mode_Check_Source(TIM_TypeDef* TIMx);
//----------------------------------------------------------------//


//------------------------SIMULINK FUNCTIONS----------------------//
// Симулирование выбранных таймеров
void Simulate_TIMs(void);
// Деинициализирование выбранных таймеров (вызывается в конце симуляции)
void TIM_SIM_DEINIT(void);
//----------------------------------------------------------------//
/** TIM_SIMULATOR
  * @}
  */
#endif // _MATLAB_TIM_H_
```

ПРИЛОЖЕНИЕ В

Листинг программы управления УПП

Файл upp.c:

```
#include "upp.h"

Phase_t phase_A; /*< Фаза управления тиристорами A */
Phase_t phase_B; /*< Фаза управления тиристорами B */
Phase_t phase_C; /*< Фаза управления тиристорами C */
UPP_Control_t Upp; /*< Структура управления УПП */

/**
 * @brief Главная функция управления УПП
 *
 * @details Выполняет основную логику управления пускателем:
 * инициализация углов, безопасный запуск,
 * проверка флагов остановки/отключения,
 * управление фазами и тиристорами.
 */void upp_main(void)
{
	// Проверяем необходимость обновления параметров угла управления тиристорами
	if(GetAngleInit(&Upp.angleInit))
	{
		// Если параметры изменились, сбрасываем углы для всех фаз
		tiristor_angle_reset(&phase_A.ctrl);
		tiristor_angle_reset(&phase_B.ctrl);
		tiristor_angle_reset(&phase_C.ctrl);
	}

	// Выполняем безопасный запуск (обработка изменения направления и стартового
состояния)
	upp_safe_go();

	// Если установлен флаг принудительной остановки, выключаем питание УПП и
подключаем выход
	if(Upp.ForceStop)
	{
		Upp.Go = 0;		// Останавливаем работу
		connect_upp();		// Подключаем УПП (прямое питание)
		return;			// Выход из функции, дальнейшая логика не
выполняется
	}

	// Если установлен флаг принудительного отключения, выставляем готовность
тиристоров и отключаем УПП
	if(Upp.ForceDisconnect)
	{
		phase_A.ctrl.f.TiristorReady = 1;
		phase_B.ctrl.f.TiristorReady = 1;
		phase_C.ctrl.f.TiristorReady = 1;
		Upp.Go = 0;			// Останавливаем работу
		disconnect_upp();		// Отключаем УПП (снимаем питание с выхода)
		return;
	}

	// Если установлен флаг плавного отключения УПП, готовим тиристоры и отключаем
УПП
	if(Upp.GoDisconnect)
	{
		phase_A.ctrl.f.TiristorReady = 1;
		phase_B.ctrl.f.TiristorReady = 1;
```

```
		phase_C.ctrl.f.TiristorReady = 1;
		Upp.Go = 0;
		disconnect_upp();
	}

	// Если установлен флаг остановки, останавливаем работу и подключаем УПП (прямое
питание)
	if(Upp.GoStop)
	{
		Upp.Go = 0;
		connect_upp();
	}

	// Если в режиме подготовки (запуска)
	if(Upp.Prepare)
	{
		// Если УПП в состоянии отключения, подключаем его (готовим к работе)
		if(Upp.Disconnected)
		{
			connect_upp();
		}

		// Обрабатываем каждую фазу (детектирование нуля, управление углом
тиристора)
		upp_phase_routine(&phase_A);
		upp_phase_routine(&phase_B);
		upp_phase_routine(&phase_C);
	}

	// Если работа разрешена (флаг Go)
	if(Upp.Go)
	{
		// Если всё ещё в подготовке, проверяем готовность тиристоров
		if(Upp.Prepare)
		{
			// Если все тиристоры готовы – снимаем флаг подготовки и продолжаем
работу
			if(phase_A.ctrl.f.TiristorReady && phase_B.ctrl.f.TiristorReady &&
phase_C.ctrl.f.TiristorReady)
			{
				Upp.Prepare = 0;
			}
			else
			{
				// Если хоть один тиристор не готов – выходим, не продолжая
управление
				return;
			}
		}

		// Если во время работы произошло отключение УПП – ставим флаг
принудительной остановки
		if(Upp.Disconnected)
		{
			Upp.ForceStop = 1;
			return;
		}

		// Проверяем условие достижения минимального угла (минимальная задержка)
во время запуска (direction == 0)
		// Это значит, что тиристоры открыты максимально рано – можно перейти на
прямое питание двигателя
		if(	(phase_A.ctrl.angle.delay_us == phase_A.ctrl.angle.Init-
>delay_min_us) &&
```

```
                        (phase_B.ctrl.angle.delay_us == phase_B.ctrl.angle.Init-
>delay_min_us) &&
                        (phase_C.ctrl.angle.delay_us == phase_C.ctrl.angle.Init-
>delay_min_us) && (Upp.angleInit.direction == 0))
            {
                  Upp.GoDisconnect = 1;  // Флаг для отключения УПП и подачи питания
напрямую
            }
            else
            {
                  Upp.GoDisconnect = 0;
            }

            // Проверяем условие достижения максимального угла (максимальная
задержка) во время торможения (direction == 1)
            // Это значит, что тиристоры максимально закрыты – нужно остановить
питание двигателя
            if(    (phase_A.ctrl.angle.delay_us == phase_A.ctrl.angle.Init-
>delay_max_us) &&
                        (phase_B.ctrl.angle.delay_us == phase_B.ctrl.angle.Init-
>delay_max_us) &&
                        (phase_C.ctrl.angle.delay_us == phase_C.ctrl.angle.Init-
>delay_max_us) && (Upp.angleInit.direction == 1))
            {
                  Upp.GoStop = 1;  // Флаг для остановки УПП и отключения питания
            }
            else
            {
                  Upp.GoStop = 0;
            }

            // Продолжаем обработку фаз – обновляем состояние и проверяем условия
управления тиристорами
            upp_phase_routine(&phase_A);
            upp_phase_routine(&phase_B);
            upp_phase_routine(&phase_C);

            // Управляем тиристорами каждой фазы с помощью функций контроля угла и
самого тиристора
            upp_phase_control(&phase_A);
            upp_phase_control(&phase_B);
            upp_phase_control(&phase_C);
      }
      else
      {
            // Если флаг Go не установлен, сбрасываем углы управления тиристорами для
всех фаз
            tiristor_angle_reset(&phase_A.ctrl);
            tiristor_angle_reset(&phase_B.ctrl);
            tiristor_angle_reset(&phase_C.ctrl);
      }
}


/**
 * @brief Функция безопасного запуска УПП
 *
 * @details Следит за изменениями флага GoSafe и запускает или останавливает пускатель,
 * сбрасывая угол задержки тиристоров в зависимости от направления.
 */
void upp_safe_go(void)
{
      static int prev_gosafe;  // Статическая переменная для хранения предыдущего
значения флага GoSafe
```

```
	// Если текущее значение GoSafe больше предыдущего – это сигнал о старте в
режиме запуска (направление 0)
	if(Upp.GoSafe > prev_gosafe)
	{
		Upp.angleInit.direction = 0;  // Устанавливаем направление пуска (разгон)
		Upp.Prepare = 1;              // Включаем режим подготовки
		Upp.Go = 1;                   // Включаем основной флаг запуска работы
УПП

		// Сбрасываем углы управления тиристорами для всех фаз – начинаем с
начального состояния
		tiristor_angle_reset(&phase_A.ctrl);
		tiristor_angle_reset(&phase_B.ctrl);
		tiristor_angle_reset(&phase_C.ctrl);
	}
	// Если текущее значение GoSafe меньше предыдущего – это сигнал о старте в
режиме торможения (направление 1)
	else if (Upp.GoSafe < prev_gosafe)
	{
		Upp.angleInit.direction = 1;  // Устанавливаем направление торможения
		Upp.Prepare = 1;              // Включаем режим подготовки
		Upp.Go = 1;                   // Включаем основной флаг запуска работы
УПП

		// Сбрасываем углы управления тиристорами для всех фаз – начинаем с
начального состояния
		tiristor_angle_reset(&phase_A.ctrl);
		tiristor_angle_reset(&phase_B.ctrl);
		tiristor_angle_reset(&phase_C.ctrl);
	}

	// Обновляем сохранённое предыдущее значение GoSafe для отслеживания изменений в
следующем вызове
	prev_gosafe = Upp.GoSafe;
}



/**
 * @brief Отключение питания УПП (разрыв всех фаз)
 *
 * @details Если тиристор готов, вызывает макросы отключения фаз,
 * после чего выставляет соответствующие флаги состояния.
 */
void disconnect_upp(void)
{
	// Если тиристоры фазы А открыты, подключаем фазу напрямую
	if(phase_A.ctrl.f.TiristorReady)
	{
		disconnect_phase(&phase_A);
	}

	// Аналогично для фазы В
	if(phase_B.ctrl.f.TiristorReady)
	{
		disconnect_phase(&phase_B);
	}

	// Аналогично для фазы С
	if(phase_C.ctrl.f.TiristorReady)
	{
		disconnect_phase(&phase_C);
	}
```

```
	// Если УПП на всех трех фазах отключены
	if(phase_A.disconnect.Disconnected && phase_B.disconnect.Disconnected &&
phase_C.disconnect.Disconnected)
	{
		Upp.Disconnected = 1;	// Устанавливаем флаг, что УПП полностью
отключена
		Upp.GoDisconnect = 0;	// Сбрасываем флаг запроса на отключение
		Upp.Go = 0;		// Прекращаем работу УПП
	}

}

/**
 * @brief Подключение питания УПП (соединение всех фаз)
 *
 * @details Вызывает отключение тиристоров и макросы подключения фаз,
 * сбрасывает флаг отключения.
 */
void connect_upp(void)
{
	// Отключаем управление тиристорами для всех фаз
	tiristor_disable(&phase_A.ctrl);
	tiristor_disable(&phase_B.ctrl);
	tiristor_disable(&phase_C.ctrl);

	// Подключаем УПП к каждой фазе)
	connect_phase(&phase_A);
	connect_phase(&phase_B);
	connect_phase(&phase_C);

	// Сбрасываем флаг, указывающий на то, что УПП было отключено
	Upp.Disconnected = 0;
}

/**
 * @brief Управление одной фазой УПП
 * @param phase Указатель на структуру фазы Phase_t
 *
 * @details Контролирует угол и включает/отключает тиристор для данной фазы.
 */
void upp_phase_control(Phase_t *phase)
{
	tiristor_angle_control(&phase->ctrl);
	tiristor_control(&phase->ctrl);
}

/**
 * @brief Обработка фазы при каждом нулевом переходе синусоиды
 * @param phase Указатель на структуру фазы Phase_t
 *
 * @details Обновляет состояние детектора нулевого перехода,
 * запускает задержку угла тиристора,
 * отключает тиристор, если он был включен.
 */
void upp_phase_routine(Phase_t *phase)
{
	// Обновляем детектор нулевого перехода по текущему состоянию входного сигнала
	zero_cross_update(&phase->zc_detector);

	// Если обнаружен нулевой переход (синусоида пересекла 0)
	if(is_zero_cross(&phase->zc_detector))
	{
		// Запускаем отсчёт задержки до открытия тиристора (по углу)
```

```
                tiristor_start_angle_delay(&phase->ctrl);

                // Если тиристор был включён в предыдущем полупериоде – отключаем его
                if (phase->ctrl.f.TiristorIsEnable)
                        tiristor_disable(&phase->ctrl);
        }
}


/**
 * @brief Расчёт параметров угла запуска тиристора
 * @param angle Указатель на структуру AngleInit_t для записи параметров
 * @return int 1, если произошли изменения параметров, иначе 0
 *
 * @details Проверяет изменения в параметрах управления и при необходимости
 * пересчитывает максимальные и минимальные задержки, шаг изменения угла,
 * а также изменяет прескалер таймера.
 */
int GetAngleInit(AngleInit_t *angle)
{
        static float sine_freq_old = 0;
        static float Duration_old = 0;
        static float max_duty_old = 0, min_duty_old = 0;  // Задаются в процентах
        int update = 0;

        // Проверка, изменились ли параметры: частота, скважности
        if(     (Upp.sine_freq != sine_freq_old) &&
                        (Upp.max_duty != max_duty_old) &&
                        (Upp.min_duty != min_duty_old) )
        {
                update = 1;
                min_duty_old = Upp.min_duty;
                max_duty_old = Upp.max_duty;
                sine_freq_old = Upp.sine_freq;
        }

        // Проверка, изменились ли длительность
        if(Upp.Duration != Duration_old)
        {
                update = 1;
                Duration_old = Upp.Duration;
        }

        if(update)
        {
                // Расчёт длительности полупериода в микросекундах (с учётом вычета
резерва на открытие тиристора)
                uint32_t half_period_us = (500000.0f / Upp.sine_freq) - 1000;

                // Расчёт максимальной и минимальной задержки (в мкс) по процентам
скважности
                angle->delay_max_us = (uint32_t)(Upp.max_duty * half_period_us);
                angle->delay_min_us = (uint32_t)(Upp.min_duty * half_period_us);

                // Проверка, помещаются ли значения задержек в 16-битный таймер
                if((angle->delay_max_us > 0xFFFF) || (angle->delay_min_us > 0xFFFF))
                {
                        // Если нет – увеличиваем прескалер в 10 раз (точность 10 мкс)
                        angle->delay_max_us /= 10;
                        angle->delay_min_us /= 10;
                        TIMER->PSC = 719;

                        if((angle->delay_max_us > 0xFFFF) || (angle->delay_min_us >
0xFFFF))
```

```
                        {
                                // Если всё ещё не помещается – ещё в 10 раз (точность 0.1
мс)
                                angle->delay_max_us /= 10;
                                angle->delay_min_us /= 10;
                                TIMER->PSC = 7299;

                                if ((angle->delay_max_us > 0xFFFF) || (angle->delay_min_us >
0xFFFF))
                                {
                                        // Если даже при этом переполнение – аварийная
остановка
                                        Upp.ForceStop = 1;
                                }
                        }
                }
                else
                {
                        // Задержки помещаются – устанавливаем стандартный прескалер (1
мкс)
                        TIMER->PSC = 71;
                }

                // Перевод длительности разгона/торможения из секунд в миллисекунды
                float duration_ms = Duration_old * 1000.0f;
                uint32_t steps = duration_ms / angle->sample_time_ms;
                if (steps == 0) steps = 1;

                // Вычисление шага изменения задержки на каждом шаге
                if (angle->delay_max_us > angle->delay_min_us)
                        angle->delay_step_us = (angle->delay_max_us - angle->delay_min_us)
/ steps;
                else
                        angle->delay_step_us = 0;
        }

        return update;
}

/**
 * @brief Инициализация УПП и связанных структур
 *
 * @details Настраивает параметры управления, GPIO для фаз,
 * инициализирует тиристоры, запускает таймер и настраивает детектор нулевого перехода.
 */
void upp_init(void)
{
        Upp.max_duty = 0.9;
        Upp.min_duty = 0.1;
        Upp.angleInit.sample_time_ms = 100;

        phase_A.disconnect.gpiox = GPIOA;
        phase_A.disconnect.gpio_pin = GPIO_PIN_5;

        phase_B.disconnect.gpiox = GPIOA;
        phase_B.disconnect.gpio_pin = GPIO_PIN_6;

        phase_C.disconnect.gpiox = GPIOA;
        phase_C.disconnect.gpio_pin = GPIO_PIN_7;

        phase_A.ctrl.angle.Init = &Upp.angleInit;
        phase_B.ctrl.angle.Init = &Upp.angleInit;
        phase_C.ctrl.angle.Init = &Upp.angleInit;
```

```
        tiristor_init(&phase_A.ctrl, GPIOB, GPIO_PIN_12);
        tiristor_init(&phase_B.ctrl, GPIOB, GPIO_PIN_13);
        tiristor_init(&phase_C.ctrl, GPIOB, GPIO_PIN_14);

        TIMER->CR1 |= TIM_CR1_CEN;

        tiristor_angle_reset(&phase_A.ctrl);
        tiristor_angle_reset(&phase_B.ctrl);
        tiristor_angle_reset(&phase_C.ctrl);
        //Upp.GoSafe = 1;
}
```

Файл upp.h:

```
#ifndef __UPP_H
#define __UPP_H

#include "main.h"
#include "adc.h"
#include "tim.h"
#include "usart.h"
#include "gpio.h"
#include "zero_cross.h"
#include "tiristor.h"

/**
 * @brief Определение используемого ADC
 */
#define hadc                                            hadc1

/**
 * @brief Начальный уровень отсчёта для определения нуля в ADC (обычно середина
диапазона)
 */
#define ADC_INITIAL_ZERO_LEVEL    2048

/**
 * @brief Макрос для разрыва (отключения) фазы – устанавливает GPIO в высокий уровень и
флаг Disconnected
 * @param _ph_ Указатель на структуру фазы Phase_t
 */
#define disconnect_phase(_ph_)      { (_ph_)->disconnect.gpiox->ODR |= (_ph_)-
>disconnect.gpio_pin; (_ph_)->disconnect.Disconnected = 1;}

/**
 * @brief Макрос для подключения фазы – сбрасывает GPIO в низкий уровень и флаг
Disconnected
 * @param _ph_ Указатель на структуру фазы Phase_t
 */
#define connect_phase(_ph_)             { (_ph_)->disconnect.gpiox->ODR &= ~(_ph_)-
>disconnect.gpio_pin; (_ph_)->disconnect.Disconnected = 0;}

/**
 * @struct Phase_t
 * @brief Структура, описывающая одну фазу с состоянием тиристора и детектом нуля
 */
typedef struct
{
    struct
```

```
    {
        unsigned Disconnected:1;         /**< Флаг разрыва фазы */
        GPIO_TypeDef *gpiox;             /**< Порт GPIO для разрыва */
        uint32_t gpio_pin;               /**< Пин GPIO для разрыва */
    }disconnect;

    ZeroCrossDetector_t zc_detector;     /**< Детектор пересечения нуля */
    TiristorControl_t ctrl;              /**< Управление тиристором */
} Phase_t;


/**
 * @struct UPP_Control_t
 * @brief Основная структура управления устройством плавного пуска
 */
typedef struct
{
    unsigned GoSafe:1;            /**< Флаг безопасного запуска */
    unsigned Go:1;                /**< Флаг запуска */
    unsigned GoStop:1;            /**< Флаг остановки */
    unsigned Prepare:1;           /**< Флаг подготовки */
    unsigned Disconnected:1;      /**< Флаг разрыва */
    unsigned GoDisconnect:1;      /**< Флаг отключения */
    unsigned ForceStop:1;         /**< Флаг форсированной остановки */
    unsigned ForceDisconnect:1;   /**< Флаг форсированного отключения */
    unsigned PreGoDone:1;         /**< Флаг завершения подготовки */

    float Duration;               /**< Время нарастания и спада напряжение через УПП */
    float sine_freq;              /**< Частота сети */
            float max_duty;                    /**< Максимальная скважность угла
открытия */
    float min_duty;               /**< Минимальная скважность угла открытия */

    AngleInit_t angleInit;        /**< Настройки угла открытия тиристора */
} UPP_Control_t;


extern Phase_t phase_A;
extern Phase_t phase_B;
extern Phase_t phase_C;
extern UPP_Control_t Upp;

/** Основной цикл работы устройства плавного пуска */
void upp_main(void);

/** Выполнение безопасного запуска устройства */
void upp_safe_go(void);

/** Отключение устройства плавного пуска (разрыв фаз) */
void disconnect_upp(void);

/** Подключение устройства плавного пуска (восстановление фаз) */
void connect_upp(void);

/** Выполнение обработки одной фазы в цикле */
void upp_phase_routine(Phase_t *phase);

/** Управление фазой с контролем тиристора и нуля */
void upp_phase_control(Phase_t *phase);

/** Получение настроек угла открытия тиристора */
int GetAngleInit(AngleInit_t *angle);

/** Инициализация устройства плавного пуска */
```

```
void upp_init(void);

#endif //__UPP_H
```

Файл tiristor.c:

```
#include "tiristor.h"

/**
 * @brief Управление состоянием тиристора (включение/выключение) по флагам и времени открытия
 * @param ctrl Указатель на структуру управления тиристором
 */
void tiristor_control(TiristorControl_t *ctrl)
{
    if(ctrl->f.EnableTiristor)  // Если разрешено включить тиристор
    {
        if(ctrl->f.TiristorIsEnable == 0)  // Если тиристор еще выключен
        {
            tiristor_enable(ctrl);           // Включить тиристор
            ctrl->enable_start_tick = HAL_GetTick();  // Запомнить время включения для отсчёта длительности открытия
        }
        else
        {
            // Если время с момента включения превысило заданное время открытия
            if(HAL_GetTick() - ctrl->enable_start_tick > ctrl->open_time)
            {
                tiristor_disable(ctrl);      // Выключить тиристор
                ctrl->f.EnableTiristor = 0;  // Снять разрешение на включение, чтобы не включался снова без команды
            }
        }
    }
    else  // Если тиристор не должен быть включен
    {
        if(ctrl->f.TiristorIsEnable)   // Если тиристор включен
            tiristor_disable(ctrl);    // Выключить тиристор
    }
}

/**
 * @brief Обновление значения задержки угла открытия тиристора в соответствии с направлением и шагом
 * @param angle Указатель на структуру управления углом тиристора
 */
void tiristor_angle_update(TiristorAngleControl_t *angle)
{
    uint32_t current_time_ms = HAL_GetTick();  // Текущее время в миллисекундах

    // Проверяем, прошло ли нужное время с последнего обновления
    if ((current_time_ms - angle->last_update_ms) >= angle->Init->sample_time_ms)
    {
        angle->last_update_ms = current_time_ms;  // Обновляем время последнего изменения задержки

        // Изменяем задержку в зависимости от направления (разгон или торможение)
        if(angle->Init->direction)
            angle->delay_us += angle->Init->delay_step_us;  // Увеличиваем задержку (увеличиваем угол)
```

```
        else
            angle->delay_us -= angle->Init->delay_step_us;  // Уменьшаем задержку
(уменьшаем угол)

        // Ограничиваем задержку в пределах минимального и максимального значения
        if (angle->delay_us < angle->Init->delay_min_us)
        {
            angle->delay_us = angle->Init->delay_min_us;
        }
        else if (angle->delay_us > angle->Init->delay_max_us)
        {
            angle->delay_us = angle->Init->delay_max_us;
        }
    }
}

/**
 * @brief Контроль угла открытия тиристора с проверкой таймера и флага готовности
 * @param ctrl Указатель на структуру управления тиристором
 */
void tiristor_angle_control(TiristorControl_t *ctrl)
{
    tiristor_angle_update(&ctrl->angle);  // Обновляем задержку угла открытия

    if(ctrl->angle.delay_us != 0)  // Если задержка не нулевая
    {
        // Проверяем, прошла ли задержка с момента старта отсчёта таймера
        if ((uint16_t)((uint16_t)TIMER->CNT - ctrl->angle.start_delay_tick) > ctrl-
>angle.delay_us)
        {
            if(ctrl->f.TiristorReady)  // Если тиристор готов к включению
            {
                ctrl->f.EnableTiristor = 1;  // Разрешаем включение тиристора
                ctrl->f.TiristorReady = 0;   // Снимаем флаг готовности, чтобы не
включать повторно сразу
            }
        }
    }
}

/**
 * @brief Запуск отсчёта задержки для открытия тиристора
 * @param ctrl Указатель на структуру управления тиристором
 */
void tiristor_start_angle_delay(TiristorControl_t *ctrl)
{
    ctrl->f.TiristorReady = 1;            // Устанавливаем флаг готовности тиристора к
включению
    ctrl->angle.start_delay_tick = TIMER->CNT;  // Запоминаем текущее значение счётчика
таймера
}

/**
 * @brief Включение тиристора путём установки GPIO в состояние открытия
 * @param ctrl Указатель на структуру управления тиристором
 */
void tiristor_enable(TiristorControl_t *ctrl)
{
    // Открываем тиристор, установив соответствующий пин в высокое состояние
    ctrl->gpiox->ODR |= ctrl->gpio_pin;
    ctrl->f.TiristorIsEnable = 1;  // Устанавливаем флаг, что тиристор включен
}

/**
```

```
 * @brief Выключение тиристора путём установки GPIO в состояние закрытия
 * @param ctrl Указатель на структуру управления тиристором
 */
void tiristor_disable(TiristorControl_t *ctrl)
{
    // Закрываем тиристор, сбросив соответствующий пин в низкое состояние
    ctrl->gpiox->ODR &= ~ctrl->gpio_pin;
    ctrl->f.TiristorIsEnable = 0;  // Снимаем флаг включения тиристора
}

/**
 * @brief Сброс значения задержки угла открытия тиристора к начальному в зависимости от направления
 * @param ctrl Указатель на структуру управления тиристором
 */
void tiristor_angle_reset(TiristorControl_t *ctrl)
{
    // В зависимости от направления устанавливаем задержку на минимальное или максимальное значение
    if(ctrl->angle.Init->direction)
        ctrl->angle.delay_us = ctrl->angle.Init->delay_min_us;
    else
        ctrl->angle.delay_us = ctrl->angle.Init->delay_max_us;
}

/**
 * @brief Инициализация структуры управления тиристором, установка GPIO и сброс угла открытия
 * @param ctrl Указатель на структуру управления тиристором
 * @param gpiox Указатель на порт GPIO
 * @param gpio_pin Номер пина GPIO
 */
void tiristor_init(TiristorControl_t *ctrl, GPIO_TypeDef *gpiox, uint32_t gpio_pin)
{
    ctrl->gpiox = gpiox;          // Сохраняем порт GPIO
    ctrl->gpio_pin = gpio_pin;    // Сохраняем номер пина GPIO
    tiristor_angle_reset(ctrl);   // Сбрасываем угол открытия тиристора на начальное значение
}
```

Файл tiristor.h:

```
#ifndef __TIRISTORS_H
#define __TIRISTORS_H

#include "main.h"
#include "tim.h"


#define GPIO_TIRISTOR_OPEN      GPIO_PIN_SET    /**< Состояние GPIO для открытия тиристора */
#define GPIO_TIRISTOR_CLOSE     GPIO_PIN_RESET  /**< Состояние GPIO для закрытия тиристора */

#define TIMER   TIM2            /**< Таймер, используемый для управления тиристором */

/**
 * @brief Флаги состояния управления тиристором
 */
typedef struct
```

```
{
    unsigned EnableTiristor:1;    /**< Флаг разрешения управления тиристором */
    unsigned TiristorIsEnable:1;  /**< Флаг, указывающий, что тиристор включен */
    unsigned TiristorReady:1;     /**< Флаг готовности тиристора к работе */
} TiristorControlFlags;

/**
 * @brief Параметры инициализации угла открытия тиристора
 */
typedef struct
{
    uint32_t delay_min_us;       /**< Минимальная задержка (микросекунды),
соответствует максимальному открытию тиристора */
    uint32_t delay_max_us;       /**< Начальная задержка (микросекунды), соответствует
практически закрытому тиристору */
    uint32_t delay_step_us;      /**< Шаг уменьшения задержки (микросекунды) */
    uint32_t sample_time_ms;     /**< Интервал времени между шагами регулировки
(миллисекунды) */
    unsigned direction;          /**< Направление регулировки: разгон (увеличение
открытого угла) или торможение */
} AngleInit_t;

/**
 * @brief Структура управления углом открытия тиристора
 */
typedef struct
{
    AngleInit_t *Init;           /**< Указатель на структуру параметров инициализации
угла */
    uint32_t last_update_ms;     /**< Время последнего обновления (миллисекунды) */
    uint32_t delay_us;           /**< Текущая задержка (микросекунды) */
    uint16_t start_delay_tick;  /**< Значение таймера при старте задержки */
} TiristorAngleControl_t;

typedef struct TiristorControl_t TiristorControl_t;

/**
 * @brief Основная структура управления тиристором
 */
struct TiristorControl_t
{
    TiristorControlFlags f;          /**< Флаги состояния тиристора */
    TiristorAngleControl_t angle;    /**< Управление углом открытия */
    GPIO_TypeDef *gpiox;             /**< Порт GPIO для управления тиристором */
    uint32_t gpio_pin;               /**< Номер пина GPIO */
    uint32_t open_time;              /**< Время открытия тиристора */
    uint32_t enable_start_tick;     /**< Время включения тиристора по таймеру */

};

/**
 * @brief Управление состоянием тиристора (включение/выключение)
 * @param ctrl Указатель на структуру управления тиристором
 */
void tiristor_control(TiristorControl_t *ctrl);

/**
 * @brief Обновление угла открытия тиристора согласно параметрам
 * @param angle Указатель на структуру управления углом тиристора
 */
void tiristor_angle_update(TiristorAngleControl_t *angle);

/**
 * @brief Контроль угла открытия тиристора, включая обновление состояния
```

```
 * @param ctrl Указатель на структуру управления тиристором
 */
void tiristor_angle_control(TiristorControl_t *ctrl);

/**
 * @brief Запуск задержки открытия тиристора
 * @param ctrl Указатель на структуру управления тиристором
 */
void tiristor_start_angle_delay(TiristorControl_t* ctrl);

/**
 * @brief Сброс угла открытия тиристора к начальному состоянию
 * @param ctrl Указатель на структуру управления тиристором
 */
void tiristor_angle_reset(TiristorControl_t *ctrl);

/**
 * @brief Включение тиристора
 * @param ctrl Указатель на структуру управления тиристором
 */
void tiristor_enable(TiristorControl_t *ctrl);

/**
 * @brief Выключение тиристора
 * @param ctrl Указатель на структуру управления тиристором
 */
void tiristor_disable(TiristorControl_t *ctrl);

/**
 * @brief Инициализация структуры управления тиристором
 * @param ctrl Указатель на структуру управления тиристором
 * @param gpiox Указатель на GPIO порт
 * @param gpio_pin Номер GPIO пина
 */
void tiristor_init(TiristorControl_t *ctrl, GPIO_TypeDef *gpiox, uint32_t gpio_pin);

#endif //__TIRISTORS_H
```

Файл zero_cross.c:

```
#include "zero_cross.h"


/**
 * @brief Инициализация структуры детектора перехода через ноль
 * @param zc Указатель на структуру ZeroCrossDetector_t
 * @param zeroLevel Значение уровня АЦП, соответствующее нулю (mid-scale)
 */
void zero_cross_Init(ZeroCrossDetector_t *zc, uint16_t zeroLevel)
{
    // Сбрасываем флаг обнаружения перехода через ноль
    zc->f.ZeroCrossDetected = 0;
}

/**
 * @brief Обновление флага перехода через ноль
 * @param zc Указатель на структуру ZeroCrossDetector_t
 * @details Просто переносим флаг EXTI с аппаратного детектора.
 */
void zero_cross_update(ZeroCrossDetector_t *zc)
```

```
{
    // Используем флаг аппаратного прерывания EXTI для установки флага перехода через
ноль
    zc->f.ZeroCrossDetected = zc->f.EXTIZeroCrossDetected;
}

/**
 * @brief Проверка наличия перехода через ноль и сброс флагов
 * @param zc Указатель на структуру ZeroCrossDetector_t
 * @return int 1 – переход через ноль обнаружен, 0 – нет
 * @details Если переход обнаружен, сбрасываем флаги и возвращаем 1,
 * иначе возвращаем 0.
 */
int is_zero_cross(ZeroCrossDetector_t *zc)
{
    if(zc->f.ZeroCrossDetected)
    {
        // Сброс флагов после обнаружения
        zc->f.ZeroCrossDetected = 0;
        zc->f.EXTIZeroCrossDetected = 0;
        return 1;
    }
    return 0;
}

/**
 * @brief Обработчик прерывания EXTI для аппаратного детектора перехода через ноль
 * @param zc Указатель на структуру ZeroCrossDetector_t
 * @details Устанавливает флаг аппаратного перехода через ноль.
 */
void zero_cross_update_EXTI(ZeroCrossDetector_t *zc)
{
    zc->f.EXTIZeroCrossDetected = 1;
}
```

Файл zero_cross.h:

```
#ifndef __ZERO_CROSS_H
#define __ZERO_CROSS_H

#include "main.h"
#include "adc_filter.h"

/**
 * @brief Флаги состояния детектора нуля
 */
typedef struct
{
    unsigned WaitForZeroCrossDetected : 1;  /**< Ожидание обнаружения перехода через
ноль */
    unsigned EXTIZeroCrossDetected : 1;     /**< Флаг обнаружения нуля аппаратным
прерыванием EXTI */
    unsigned ZeroCrossDetected : 1;         /**< Флаг обнаружения перехода через ноль
*/
} ZeroCrossFlags;

/**
 * @brief Структура детектора перехода через ноль
 */
typedef struct
```

```
{
    ZeroCrossFlags f;          /**< Флаги состояния детектора */
} ZeroCrossDetector_t;

/**  Инициализация структуры детектора перехода через ноль */
void zero_cross_Init(ZeroCrossDetector_t *zc, uint16_t zeroLevel);
/** Обновление состояния детектора перехода через ноль */
void zero_cross_update(ZeroCrossDetector_t *zc);
/** Проверка, был ли обнаружен переход через ноль */
int is_zero_cross(ZeroCrossDetector_t *zc);
/** Обновление состояния детектора перехода через ноль при аппаратном прерывании EXTI
*/
void zero_cross_update_EXTI(ZeroCrossDetector_t *zc);

#endif //__ZERO_CROSS_H
```