Федеральное государственное бюджетное учреждение науки
«Институт автоматики и электрометрии»
Сибирского отделения Российской академии наук

На правах рукописи

Розов Андрей Сергеевич

РАЗРАБОТКА ЯЗЫКОВЫХ И ИНСТРУМЕНТАЛЬНЫХ СРЕДСТВ ПРОЦЕСС-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ ДЛЯ ОТКРЫТЫХ МИКРОКОНТРОЛЛЕРНЫХ ПЛАТФОРМ

Специальность: 05.13.18 “математическое моделирование, численные методы и комплексы программ”

Диссертация на соискание ученой степени
кандидата технических наук

Научный руководитель:
д-р техн. наук, доцент.
Зюбин Владимир Евгеньевич

Новосибирск, 2019

СОДЕРЖАНИЕ

ОБЩАЯ ХАРАКТЕРИСТИКА РАБОТЫ

Актуальность

Снижение стоимости микропроцессоров и тенденция к совмещению нескольких устройств в одной микросхеме привели к активному развитию встраиваемых систем (ВС). Замещение специализированных цифровых схем универсальными микроконтроллерами позволяет значительно удешевить разработку и производство ВС. Затраты на создание программного обеспечения (ПО) встраиваемых систем при этом многократно превышают расходы на проектирование и производство аппаратной составляющей. Этот эффект значительно усиливается активным развитием микроконтроллерных платформ с открытой архитектурой (МПОА), таких как Arduino [1] и Itead Maple [2]. Использование готовых аппаратных решений и широкий выбор совместимых периферийных устройств делают возможным быстрое прототипирование аппаратной составляющей встраиваемых систем. При этом исключается необходимость в дорогостоящих и затратных по времени операциях по изготовлению специализированных печатных плат. Таким образом, стоимость результирующей системы определяется в первую очередь стоимостью разработки ПО микроконтроллера.

Трудоемкость разработки ПО в значительной мере зависит от выбранных методик и языков программирования и их соответствия решаемой задаче. В настоящее время для программирования микроконтроллеров преимущественно используются языки Си/Си++. Понятийный аппарат этих языков изначально ориентирован на вычислительные задачи и обработку сложно структурированных данных. Разработка ПО ВС при помощи объектно-ориентированных технологий приводит к быстрому росту сложности программ по мере увеличении числа компонентов внешней среды. Обеспечение надежности работы системы в этом случае требует значительных временных затрат, что увеличивает стоимость разработки и сопровождения ПО.

Встраиваемые системы обладают многими свойствами алгоритмов управления в промышленной автоматизации – сложное поведение системы, одновременное взаимодействие с множеством внешних для системы устройств. Для описания алгоритмов управления разработаны специализированные методы, средства и языки программирования. Методики событийного программирования на основе конечных автоматов, такие как switch-технология [3], иерархические автоматы [4] , язык QuickStep [5] позволяют эффективно описывать взаимодействие системы с внешней средой. Описание системы в виде набора взаимодействующих автоматов позволяет обеспечить логический параллелизм ПО ВС. Этот подход используется в технологии процесс-ориентированного программирования [6], [7], [8].

Использование наработок в области промышленной автоматизации может снизить трудоемкость разработки и отладки ПО ВС. Программирование микроконтроллеров предполагает активную работу с аппаратными прерываниями и минимизацию накладных расходов. Современные методы и языки описания алгоритмов управления ориентированы на аппаратные платформы промышленной автоматизации, существенно отличающиеся от микроконтроллерных платформ. Эффективное применение таких подходов в области встраиваемых систем требует их доработки с учетом специфики программирования микроконтроллеров.

Таким образом, задача разработки методик и языков программирования, учитывающих особенности алгоритмов управления и специфику встраиваемых систем, является актуальной.

Объект исследования – программное обеспечение встраиваемых систем на базе микроконтроллерных платформ с открытой архитектурой.

Предмет исследования – понятийные, языковые и инструментальные средства спецификации процесс-ориентированного программного обеспечения открытых микроконтроллерных платформ.

Цель работы – разработка понятийных, языковых и инструментальных средств процесс-ориентированного программирования открытых микроконтроллерных платформ во встраиваемых системах.

Задачи:

1. Проанализировать специфику программирования микроконтроллеров и существующие подходы к описанию управляющих алгоритмов;
2. Сформулировать требования к разрабатываемому языку;
3. Разработать и формализовать понятийный аппарат языка;
4. Определить синтаксис специализированного языка программирования;
5. Разработать эквивалентное представление программ на языке Си;
6. Разработать и реализовать транслятор языка в язык Си;
7. Разработать и реализовать интегрированную среду разработки для языка;
8. Исследовать свойства полученных средств на задачах разработки встраиваемых систем.

Основная гипотеза:

Встраиваемые системы на микроконтроллерных платформах имеют ряд общих свойств с алгоритмами управления в промышленной автоматизации и могут эффективно описываться с применением процесс-ориентированных технологий.

На базе процесс-ориентированного подхода и с учетом специфики программирования микроконтроллеров могут быть созданы понятийные,

языковые и инструментальные средства, позволяющие существенно снизить затраты на разработку ПО встраиваемых систем.

Научная новизна:

1. Предложена математическая модель ПО ВС в виде набора гиперпроцессов с различными источниками активации и определены механизмы взаимодействия гиперпроцессов;
2. Разработан синтаксис процесс-ориентированного языка программирования IndustrialC для микроконтроллерных платформ;
3. Разработаны и формализованы правила трансляции IndustrialC-программ в язык C, задающее трансляционную семантику языка;
4. Выявлены свойства разработанных средств при исползовании в разработке ПО микроконтроллеров во встраиваемых системах.

Теоретическая и практическая значимость:

Разработанный метод описания ПО МПОА снижает временные затраты на разработку и сопровождение управляющих систем на базе микроконтроллеров и повышает надежность создаваемых систем. Использование специализированного языка программирования IndustrialC повышает читаемость кода программ на базе микроконтроллеров и снижает требования к квалификации разработчика.

Результаты работы внедрены в учебный процесс ФИТ НГУ и позволили повысить качество подготовки. Разработанные методы и средства использовалась в инициативных проектах при создании системы метеосервера, системы автоматизации установки термовакуумного напыления, ПО адаптера электронного блока весоизмерительной системы, при выполнении хоздоговоров СОРБИ-16, СОРБИ-17, что подтверждается актами о внедрении.

Методология и методы исследования:

Исследование осуществлялось с использованием общенаучных теоретико-эмпирических методов и специальных методов из областей автоматного программирования, теории управления, математического моделирования и итеративных методов разработки.

Анализ предметной области выполнялся с использованием содержательного и логико-исторического подходов. При разработке модели понятийного аппарата и синтаксиса языка, а также при создании программных средств использовался формальный подход. Исследование свойств полученных средств разработки на практических задачах выполнялось с использованием эмпирического подхода в виде экспериментов с системами управления физическими процессами и численного эксперимента с программной имитацией объектов управления.

Область исследования:

Содержание диссертации соответствует паспорту специальности 05.13.18 «Математическое моделирование, численные методы и комплексы программ» (технические науки) по следующим областям исследований: п.1 «Разработка новых математических методов моделирования объектов и явлений», п.3 «Разработка, обоснование и тестирование эффективных вычислительных методов с применением современных компьютерных технологий», п. 4 «Реализация эффективных численных методов и алгоритмов в виде комплексов проблемно-ориентированных программ для проведения вычислительного эксперимента» и п. 8 «Разработка систем компьютерного и имитационного моделирования».

Положения, выносимые на защиту:

1. Математическая модель ПО ВС в виде набора гиперпроцессов с различными источниками активации;
2. Грамматика специализированного языка программирования IndustrialC;
3. Трансляционная семантика языка IndustrialC;

4. Результаты апробации предложенных средств, показавшие их эффективность при разработке встраиваемых систем на базе МПОА.

Апробация работы:

Результаты работы докладывались на: Международной научной студенческой конференции "Студент и научно-технический прогресс" (Новосибирск, 2013, 2014, 2015), XIII Всероссийской конференции молодых ученых по математическому моделированию и информационным технологиям (Новосибирск, 2012), X Международной IEEE-Сибирской конференции по управлению и связи SIBCON-2013 (Красноярск, 2013) и международной конференции "International Conference on Advanced Technology & Sciences" ICAT'15 (Анталия, 2015).

Публикации

По теме диссертации опубликовано 20 печатных работ, из них шесть статей в журналах, рекомендованных ВАК и два свидетельства официальной регистрации программ.

Структура и объем диссертации

Диссертация состоит из введения, пяти глав и заключения. Объем работы 127 страниц, количество рисунков 12.

В первой главе приводится обзор существующих подходов к созданию ПО встраиваемых систем, анализируются особенности систем управления и специфика микроконтроллерных платформ, формулируются требования к разрабатываемым средствам. Во второй главе описывается математическая модель ПО встраиваемых систем, задающая понятийный аппарат для программирования микроконтроллеров с использованием процесс-ориентированной технологии. В третьей главе описывается грамматика специализированного языка IndustrialC и его трансляционная семантика, задающая эквивалентное представление программ на языке Си. В четвертой

главе описываются разработанные транслятор и среды разработки IndustrialC. В пятой главе приводятся результаты практической апробации разработанных понятийного аппарата, языка и инструментальных средств в проектах по созданию ВС МПОА.

Глава 1. Исследование предметной области

Эффективная разработка встраиваемых систем требует использования методик, одновременно учитывающих особенности управляющих алгоритмов и специфику программирования микроконтроллеров.

1.1. Особенности алгоритмов управления

Системы управления, в отличие от классических вычислительных систем, обладают свойством открытости – наличием внешней среды и необходимостью постоянного взаимодействия с ней. В роли внешней среды выступают объект управления и оператор системы. Система управления открыта, то есть микроконтроллер формирует управляющие воздействия по событиям, возникающим во внешней среде. Взаимодействие с внешней средой осуществляется посредством датчиков и исполнительных устройств. В силу независимости событий, возникающих во внешней физической среде, в микроконтроллере требуется параллельная обработка событий. Поскольку в микроконтроллерах отсутствуют достаточные средства физически параллельной обработки событий это требование обеспечивается средствами логического параллелизма, реализуемого программно, за счет разделения процессорного времени. Что в свою очередь предполагает наличие специальных механизмов для разрешения противоречий, возникающих при работе с разделяемыми ресурсами. Дополнительно к этому при формировании управляющих воздействий необходимо учитывать динамические характеристики внешней среды. Это обуславливает наличие в микроконтроллерах программно-реализуемых средств работы с временными интервалами.

Поскольку алгоритмы управления влияют на физические процессы во внешней среде, от их работы зависит безопасность обслуживающего персонала и сохранность дорогостоящего оборудования. В связи с этим к системам управления предъявляются требования по надежности и

устойчивости: система должна сохранять корректное поведение на протяжении длительного времени работы в нормальных условиях и предусматривать реакцию на возникновение нештатных ситуаций.

Разработка системы управления во многих случаях ведется параллельно разработке объекта управления. При этом отладка алгоритма управления на целевом объекте оказывается невозможна или нежелательна в связи с риском повреждения оборудования. Разработка физических имитаторов объектов управления требует больших денежных затрат. Затраты на отладку существенно сокращаются при использовании программных моделей объектов управления и средств автоматической верификации алгоритмов управления. Необходимость обеспечения верифицируемости алгоритма накладывает ограничения на средства его описания. Понятийный аппарат используемых методик и языков программирования должен соответствовать требованиям существующих средств и алгоритмов автоматической верификации.

В задачах автоматизации часто возникает необходимость корректировки поведения управляющей системы или расширения ее функциональности в процессе эксплуатации. Расширяемость системы и порог вхождения использованных при разработке методов и языковых средств могут существенно повлиять на стоимость доработки. Эти показатели составляют расширяемость системы и определяются адекватностью используемых методик и языков решаемой задаче и размером области их специализации. Использование специализированных методов и языков программирования позволяет уменьшить размер исходного кода программ и повысить его сопровождаемость.

Таким образом, алгоритмы управления в задачах автоматизации должны обладать следующими свойствами: открытость, событийность, параллелизм, синхронизм, надежность, устойчивость и сопровождаемость [6]. Эти свойства определяют требования к методикам и языкам программирования, используемым для описания алгоритмов управления. Эти требования

справедливы также и при разработке встраиваемых систем на базе микроконтроллеров.

1.2. Специфика программирования микроконтроллеров

Выбор средств разработки встраиваемых систем требует также учета специфики микроконтроллерных платформ. Основное отличие микроконтроллеров от промышленных ПК и ПЛК – существенная ограниченность вычислительных ресурсов. В связи с ограничениями по энергопотреблению и габаритам, микроконтроллерные платформы не имеют активного охлаждения и работают на относительно низких тактовых частотах (десятки МГц). Объемы встроенной в процессор памяти также относительно малы – десятки КБ ОЗУ и сотни КБ ПЗУ. Использование внешней памяти повышает стоимость устройства, увеличивает его габариты и энергопотребление.

Появление на рынке дешевых 32-битных микроконтроллеров с ARM-архитектурой расширило возможности для реализации сложных алгоритмов. Типичный пример – микроконтроллеры серии STM32F103 с тактовыми частотами до 72 МГц, объемами флэш-памяти до 1 МБ и ОЗУ до 96 КБ. Распространению этого класса микроконтроллеров препятствует сравнительно высокий порог вхождения (относительно PIC или AVR) и отсутствие моделей, поддерживающих работу с периферийными устройствами, рассчитанными на 5 В логические уровни. Преимущество ARM-микроконтроллеров по объемам памяти обесценивается тем, что 32-битная RISC архитектура предполагает значительное увеличение размеров исполняемого кода программ, в сравнении с 8-битными реализациями.

Недостаток вычислительных мощностей компенсируется наличием большого количества встроенных в микроконтроллер периферийных устройств – таймеров/счетчиков, генераторов ШИМ, АЦП, аппаратной поддержки низкоуровневых протоколов связи, таких как UART, SPI, I2C и так далее. Взаимодействие со встроенной периферией и внешними устройствами

преимущественно осуществляется с помощью аппаратных прерываний. При этом, в отличие от ПЛК, где для работы с прерываниями используются абстрагирующие программные прослойки, при программировании микроконтроллеров разработчик описывает процедуры обработки прерываний непосредственно.

Таким образом, основные особенности программирования микроконтроллеров следующие: ограниченность вычислительных ресурсов, непосредственное взаимодействие со встроенной периферией без использования операционной системы и активное использование аппаратных прерываний.

1.3. Методики программирования микроконтроллеров

За время развития вычислительной техники было создано большое количество моделей и подходов к описанию реактивных критических к надежности параллельных систем. Большинство таких подходов основываются либо на событийных моделях исполнения [9], либо на моделях потоков данных [10,11]. В отдельный класс выделяются так называемые синхронные языки программирования [12-18] в основе которых лежит гипотеза идеального синхронизма [19]. Эти языки нацелены на системы жесткого реального времени, как правило имеют четко определенную формальную семантику и гарантируют выполнение требований к системам за счет введения значительных ограничений на формат их описания. Эти ограничения, в совокупности с математически-ориентированным синтаксисом, далеким от реально используемых языков программирования, не позволили синхронным языкам получить широкое распространение на практике.

Степень соответствия понятийного аппарата и языковых средств особенностям управляющих алгоритмов и специфике программирования микроконтроллеров – основной критерий эффективности использования языков и методов программирования микроконтроллеров. Далее

рассматриваются наиболее распространенные подходы к разработке встраиваемых микроконтроллерных систем.

1.3.1 Объектно-ориентированное программирование

Наиболее распространенные языки программирования микроконтроллеров на данный момент – языки Си и Си++. Разработка ведется на основе как процедурного, так и объектно-ориентированного программирования (ООП) [20]. Эти концепции разрабатывались с целью облегчения переиспользования программного кода, что было достигнуто ценой прозрачности потока управления программы. Поток управления в Си/Си++ постоянно перемещается между частями программы (процедурами/методами), и его отслеживание требует использования пошагового отладчика. В случае наличия в программе множества параллельно исполняющихся процессов, отслеживание потока управления такими методами становится практически невозможным.

ООП обеспечивает программиста мощным понятийным аппаратом для описания сложных иерархий объектов. Принципы, лежащие в основе ООП – абстракция, инкапсуляция, композиция, полиморфизм типов – явно ориентированы на описание структуры представления данных в виде иерархии классов и объектов, что затрудняет описание систем со сложным поведением.

Попытки реализации управляющих систем на языках Си/Си++ приводят к образованию так называемого спагетти-кода. По мере развития и расширения системы сопровождаемость кода быстро снижается, что приводит к трудно диагностируемым и исправляемым ошибкам в программе.

Причина этих проблем – в несоответствии объектно-ориентированных принципов особенностям управляющих алгоритмов. Языки Си/Си++ ориентированы на разработку программ с относительно простым поведением, и их использование в области встраиваемых систем неоправданно трудоемко.

1.3.2. Языки стандарта МЭК61131-3

Наиболее распространенная платформа промышленной автоматизации – ПЛК в сочетании с языками стандарта МЭК 61131-3. Известны попытки адаптации этих языков для программирования микроконтроллерных платформ [21]. Стандарт включает набор языков, описывающих управляющее ПО на различных уровнях с использованием различных понятийных аппаратов.

Языки стандарта имеют общие элементы – типы данных, переменные и функции (процедуры), облегчающие их совместное использование в одной системе. Понятийный аппарат языка ST – императивное процедурное программирование – в применении к системам управления имеет те же недостатки, что и аппарат языков Си/Си++. Язык IL описывает программу на уровне элементарных инструкций и не имеет механизмов обеспечения событийности и параллелизма. Язык LD предназначен для описания систем, ранее реализованных на релейной логике. Язык FBD основан на концепции потоков данных, предполагает логический параллелизм и допускает использование средств синхронизации потоков управления. В языке отсутствуют средства обеспечения событийности алгоритма управления. Наибольший интерес представляет язык SFC. Понятийный аппарат языка основан на сетях Петри и предоставляет возможность организации событийности, параллелизма и синхронизации. Слабая структурированность программ снижает сопровождаемость и делает язык неудобным для описания сложных алгоритмов с множеством параллельных процессов [22]. Подробное описание недостатков языков этого стандарта приведено в работе [23]. Применение языков МЭК 61131-3 для создания встраиваемых систем дополнительно осложнено отсутствием поддержки работы с аппаратными прерываниями и ограниченным набором поддерживаемых микроконтроллеров.

1.3.3. MATLAB/Simulink

При разработке кибер-физических систем таких как [24] применяется пакет MATLAB/Simulink [25]. Управляющая система при этом описывается одновременно с моделью объекта управления на графическом языке программирования потоков данных (dataflow). Simulink предоставляет возможность генерации исполняемого кода, в том числе для микроконтроллера, непосредственно из модели. Подход эффективен при разработке систем непрерывного управления, однако dataflow-парадигма, как и в случае языка FBD из состава МЭК 61131-3, сильно затрудняет описание поведенческих алгоритмов.

По этой причине на практике дискретная часть поведения системы описывается машинами состояний на текстовом языке Matlab. При этом синтаксис языка не имеет встроенных конструкций для описания машин состояний, что затрудняет чтение и модификацию кода. Необходимость описания системы на двух языках – текстовом и графическом также негативно сказывается на сопровождаемости программ.

1.3.4. Операционные системы реального времени

Операционные системы реального времени (RTOS) предоставляют механизмы для обеспечения параллелизма и синхронизации. Такие системы ориентированы на облегчение структурирования программы, планировку разделения времени, контроль доступа к памяти и организацию контекста исполнения процессов [26]. Значительный вклад в сокращение времени разработки на основе RTOS вносит наличие готовых библиотек «драйверов» для наиболее часто используемых периферийных устройств [27].

В основе большинства RTOS лежит планировщик, обеспечивающий вытесняющую многозадачность [28] и равномерное распределение времени между несколькими независимыми вычислительными задачами. Эта стратегия показывает хорошие результаты при постоянной загрузке процессора. Однако большинство встраиваемых систем характеризуется

краткими промежутками активности с длительными периодами бездействия. В случае автономного питания от аккумулятора и наличия требования энергосбережения разработчики вынуждены отказываться от использования RTOS из-за отсутствия механизмов управления режимом сна [29].

Построение ПО микроконтроллеров на базе RTOS увеличивает риск возникновения гонок, повышает требования к объемам используемой памяти и затраты на синхронизацию по сравнению с кооперативным вариантом организации многозадачности [30, 31, 32].

Использование RTOS и ООП также затрудняет создание устойчивого ПО: необходимость обрабатывать события, поступающие от внутренней и внешней периферии микроконтроллера, вынуждает разработчиков решать возникающую при этом проблему параллелизма средствами «сложных низкоуровневых примитивов» [33], что, как следствие, приводит к большому количеству плохо отслеживаемых и отлаживаемых ошибок в таких программах [33, 34].

По этим причинам поиск альтернативных подходов к организации программ на микроконтроллерах постоянно привлекает внимание исследователей. Основное направление активности – разработка специализированных языков программирования с возможностями статического анализа кода и обнаружением семантических ошибок на этапе компиляции программ.

1.3.5. Комбинированный подход

В TinyOS [35] предпринята попытка устранить недостатки RTOS-подхода введением специализированного языка nesC [36]. Язык предоставляет высокоуровневые конструкции для обеспечения вытесняющий и, частично, кооперативной многозадачности. Анализ кода, производимый компилятором nesC позволяет диагностировать места вероятного возникновения гонок. Синтаксис языка ориентирован на разработку так называемой «умной пыли» (smart dust) [37] – распределенных систем, состоящих из большого количества

функционально простых микроконтроллерных модулей – и слабо подходит для спецификации встраиваемых систем широкого класса. По результатам практического использования языка разработчики обнаружили большое количество участков кода, организованных в виде конечных автоматов. Отсутствие встроенных средств организации программы в виде конечных автоматов было отмечено ими в качестве недостатка подхода [38].

1.3.6. Событийное программирование на основе конечных автоматов

Сложное поведение алгоритмов управления наиболее успешно описывается моделями на основе конечных автоматов. Конечный автомат предполагает наличие внешней среды, представленной входом и выходом автомата. В каждом состоянии автомат сопоставляет входным значениям набор действий – реакций. Таким образом, модель автомата обладает свойством событийности. Автомат может не иметь конечного состояния и работать циклически неопределенное время. Благодаря этим свойствам, конечные автоматы широко применяются для моделирования программных и аппаратных дискретных систем. Большинство алгоритмов автоматической верификации требуют, чтобы система была представлена в виде конечного автомата.

Основной недостаток этой модели – отсутствие поддержки параллелизма. Попытка описать систему с множеством параллельных процессов приводит к комбинаторному взрыву сложности автомата. Даже с небольшим количеством параллельных процессов, модель становится сложной для восприятия, поддержания и расширения. Модель конечного автомата не предусматривает возможности хранения и обработки данных. Предшествующие входные значения могут быть сохранены только в виде состояний. Это дополнительно увеличивает число состояний и увеличивает сложность автомата. В классических моделях автоматов не определены сложные операции с данными. Реагируя на входное значение, автомат может

только изменить свое состояние и выдать некоторое значение на выход. В ряде подходов эти недостатки были исправлены за счет расширения классических моделей конечного автомата операциями работы с данными и временными интервалами и внедрения механизмов эффективного описания параллельных процессов.

Для обеспечения поддержки параллелизма было предложено несколько модификаций модели автомата. Пример такого расширения – платформа Quantum Platform (QP) [39,40], имеющая в том числе реализацию для Arduino. QP представляет систему в виде множества активных объектов (актеров). Поведение актеров описывается иерархическим конечным автоматом (HSM) [4]. Платформа реализует планировщик задач и обеспечивает раздельные контексты процессов. Таким образом, QP фактически представляет собой операционную систему реального времени. Платформа QP может быть использована непосредственно как фреймворк при разработке на языках Си/Си++, но ориентирована на использование совместно с инструментом моделирования QP Modeler (QM). Разработка в QP ведется на графическом языке с последующей кодогенерацией из диаграмм состояний (statecharts).

Формализм иерархических конечных автоматов, используемый в QP, разрабатывался для борьбы с комбинаторным ростом числа состояний и уменьшает избыточность автомата за счет переиспользования кода состояний. HSM затрудняет описание параллелизма, поскольку алгоритм управления задается одним автоматом.

1.3.7. Процесс-ориентированное программирование

Другой способ решения проблем комбинаторного взрыва сложности и облегчения восприятия системы – представление системы в виде набора отдельных параллельно исполняющихся автоматов [7]. В методике процесс-ориентированного программирования [6] эта идея используется для описания алгоритмов управления.

В основе процесс-ориентированного программирования лежит понятие гиперпроцесса – набора взаимодействующих процессов. Каждый процесс реализуется конечным автоматом. Модель автомата расширена операциями работы с данными и межпроцессного взаимодействия. На практике процессы образуют иерархию: процессы высокого уровня координируют работу низкоуровневых процессов. Несмотря на сходство с иерархическим конечным автоматом, такое представление обеспечивает более высокую читаемость и сопровождаемость системы за счет разделения на множество небольших автоматов.

Методика процесс-ориентированного программирования реализована в специализированном языке Reflex [41], ориентированном на разработку систем промышленной автоматизации на базе ПЛК. Этот язык показал высокую эффективность при разработке алгоритмов управления в промышленных задачах [42-45].

Модель гиперпроцесса обеспечивает событийность, логический параллелизм и синхронизм алгоритмов управления, удовлетворяя всем требованиям задач автоматизации. Специфика программирования микроконтроллеров дополнительно требует поддержки работы с аппаратными прерываниями. Существующие реализации языка Reflex и модели гиперпроцесса предполагают абстракцию обработки прерываний в отдельном программном слое. Таким образом, средствами Reflex реализуется только высокоуровневая часть алгоритма управления. Обработка прерываний реализуется средствами языка Си. Разработка и поддержание такой системы требуют работы с двумя языками программирования, увеличивая требования к квалификации разработчика. Эффективная методика программирования микроконтроллеров должна предполагать реализацию всей программы единым набором средств.

1.4. Требования к разрабатываемым средствам

На основании проведенного анализа специфики программирования микроконтроллеров и существующих подходов к разработке управляющих алгоритмов были сформулированы требования к разрабатываемым средствам:

1. возможность представления системы в виде набора независимо исполняющихся взаимодействующих автоматов;
2. встроенная поддержка работы с временными интервалами;
3. встроенная поддержка работы с прерываниями;
4. минимизация накладных расходов на обеспечение работы и взаимодействия автоматов;
5. реализация в специализированном языке программирования с трансляцией в язык Си.

Основные выводы главы

Микроконтроллерные платформы имеют относительно ограниченный объем вычислительных ресурсов — низкие тактовые частоты, малый объем ОЗУ и ПЗУ, доступных для работы и хранения программ. Для снижения энергопотребления многие микроконтроллеры предоставляют возможность перехода в спящий режим, выход из которого осуществляется по прерыванию.

Ограниченность общих вычислительных ресурсов микроконтроллеров компенсируется наличием большого количества интегрированных периферийных устройств, таких как таймеры, АЦП, программируемые входы выходы и аппаратные интерфейсы передачи данных. Работа с этими устройствами предполагает использование аппаратных прерываний.

Эти особенности программирования микроконтроллеров — ограниченность ресурсов, наличие спящего режима процессора и активная работа с внутренней периферией через аппаратные прерывания — должны учитываться при выборе подходов к разработке систем на микроконтроллерных платформах.

Управляющие алгоритмы имеют ряд свойств, отличающих их от так называемых трансформационных или вычислительных систем.

Необходимость постоянного взаимодействия с внешней средой, представленной объектом управления, обуславливает открытость управляющего ПО. Система управления отслеживает события внешней среды, поступающие с датчиков и реагирует на них управляющими воздействиями. В силу независимости событий, возникающих во внешней физической среде, требуется параллельная обработка событий, что в условиях одного физического процессора, реализуется средствами многозадачности. Для учета динамических свойств внешней среды, управляющее ПО должно также реализовать работу с временными интервалами. Системы управления должны работать неопределенно долгое время без вмешательства оператора, при этом сбой в работе системы зачастую может привести к катастрофическим последствиям. В связи с этим к управляющему ПО предъявляются повышенные требования по надежности и устойчивости.

Таким образом, алгоритмы управления должны обладать следующими свойствами: открытость, событийность, параллелизм, синхронизм, надежность и устойчивость. Эти свойства определяют требования к методикам и языкам программирования, используемым для описания алгоритмов управления и справедливы также и при разработке встраиваемых систем на базе микроконтроллеров.

Были рассмотрены существующие подходы к разработке ПО встраиваемых микроконтроллерных систем:

1. Использование процесс-ориентированных языков общего назначения;
2. Использование языков стандарта МЭК 61131-3;
3. Автоматическая генерация Си-кода из среды Matlab/Simulink;
4. Применение операционных систем реального времени (RTOS);
5. Комбинированный подход с реализацией операционной системы реального времени в специализированном языке программирования
6. Различные подходы на основе автоматного программирования;
7. Процесс-ориентированный подход.

Разработка ПО со сложным поведением в объектно-ориентированном стиле приводит к запутанному, плохо читаемому коду и высоким накладным расходам. Операционные системы реального времени используют модели вытесняющей многозадачности, приводящие к возникновению гонок. Специализированные методы подходы на основе конечных автоматов демонстрируют наибольшую эффективность при описании систем со сложным поведением. Наиболее ярко эти методы представлены в работах Д. Харела (D. Harel), Ф. Вагнера (F. Wagner) и А. А. Шалыто.

Попытки описать весь алгоритм управления одним автоматом приводят к комбинаторному взрыву сложности при описании систем со множеством параллельных процессов. Модель иерархического автомата позволяет устранить повторяющиеся реакции на события за счет использования вложенных состояний. Другой способ решения проблем комбинаторного взрыва сложности и облегчения восприятия системы – представление системы в виде набора отдельных параллельно исполняющихся автоматов – гиперпроцесса. В методике процесс-ориентированного программирования, реализованной в специализированном языке программирования Reflex, эта идея используется для описания алгоритмов управления промышленной автоматизации на базе ПЛК.

Понятийный аппарат процесс-ориентированного программирования обеспечивает событийность, логический параллелизм и синхронизм алгоритмов управления, удовлетворяя всем требованиям управляющих систем. Специфика программирования микроконтроллеров дополнительно требует поддержки работы с аппаратными прерываниями. Существующие реализации языка Reflex и модели гиперпроцесса предполагают абстракцию обработки прерываний в отдельном программном слое. Таким образом, средствами Reflex реализуется только высокоуровневая часть алгоритма управления. Обработка прерываний реализуется средствами языка Си. Разработка и поддержание такой системы требуют работы с двумя языками программирования, увеличивая требования к квалификации разработчика.

Эффективная методика программирования микроконтроллеров должна предполагать реализацию всей программы единым набором средств.

На основании проведенного анализа специфики программирования микроконтроллеров и существующих подходов к разработке управляющих алгоритмов были сформулированы требования к разрабатываемым средствам.

Глава 2. Математическая модель ПО встраиваемых систем

В результате анализа существующих подходов и моделей, применяемых в промышленной автоматизации и при создании встраиваемых систем, за основу разрабатываемого понятийного аппарата была выбрана модель гиперпроцесса, лежащая в основе процесс-ориентированного подхода и языка Reflex. Эта модель алгоритма управления предполагает описание управляющего ПО в виде набора независимых синхронно исполняющихся процессов. Процесс представлен машиной состояний, расширенной арифметическими операциями, условными операторами, операциями для межпроцессного взаимодействия и работы с временными интервалами.

Формально гиперпроцесс H описывается как совокупность множества процессов P, начального процесса p_1 и периода активации T_H:

$$H = (P,\ p_1,\ T_H).$$

Процесс $p_i \in P$ представлен модифицированной моделью конечного автомата (машиной состояний) и задается следующим образом:

$$p_i = (F_i, F_i^p, f_i^1, f_i^{cur}, T_i), \text{ где:}$$

$p_i \in P$, $i = 1, 2, \dots, M$,

F_i – упорядоченный набор функций-состояний,

F_i^p – множество пассивных функций-состояний, $F_i^p \subset Fi$,

f_i^1 – начальная функция-состояние,

f_i^{cur} – текущая функция-состояние,

T_i – дискретное время нахождения процесса в состоянии.

Функция-состояние процесса описывается упорядоченными множествами событий и реакций:

$$F_{ji} = (X_{ji},\ Y_{ji}), \text{ где:}$$

$F_{ji} \in F_i$,

X_{ji} – множество событий,

Y_{ji} – множество реакций.

Соответствие между событиями и реакциями задается их порядком в множествах X_{ji} и Y_{ji}. В качестве событий рассматривается произвольная суперпозиция фактов – преимущественно формулы над внутренними переменными и значениями входных регистров, составленные отношениями частичного порядка и операциями логики высказываний, типичными для С-подобных языков. Помимо этого, определены события нахождения процессов в пассивных или активных состояниях и событие тайм-аута процесса, обеспечивающее механизм работы с дискретными временными интервалами (кратными T_H):

$$passive(i) \equiv f_i^{cur} \in F_i^p,$$

$$active(i) \equiv \neg\, passive(i)$$

$$timeout(i) \equiv T_i > T_{ji}^{timeout}$$

Реакции определяются как суперпозиции операций над переменными, входными/выходными значениями регистров. Отдельно выделены реакции смены состояния и межпроцессного взаимодействия – процесс может перейти в другое состояние, запустить или остановить другой процесс, или остановиться сам.

Для формального описания условия выполнения реакций можно ввести событие активации гиперпроцесса x_h, возникающее регулярно с периодом активации гиперпроцесса T_H. В таком случае, реакция y_{ji}^k, сопоставленная в функции-состоянии f_{ji} событию x_{ji}^k выполнится при следующем условии:

$$(x_h \wedge f_i^{cur} = f_{ji} \wedge x_{ji}^k)$$

Исполнение гиперпроцесса происходит регулярно с заданным периодом активации T_H, при этом во время каждой активации гиперпроцесса для каждого процесса исполняется набор реакций в его текущей функции-состоянии, для которых выполняются связанные с ними события. Исполнение гиперпроцесса предполагает атомарность функций состояний, а также конечное и небольшое (относительно T_H) время исполнения всех реакций внутри отдельно взятой функции-состояния, что соответствует кооперативной форме многозадачности. Для обеспечения генерации событий тайм-аута при

каждой активации гиперпроцесса дополнительно исполняется реакция y_{ts}, состоящая в увеличении счетчиков времени T_i для всех процессов:

$$y_{ts} \equiv T_1{:=}T_1{+}1, \ldots T_M{:=}T_M{+}1.$$

Таким образом, в модели неявно присутствует служба времени, осуществляющая активацию гиперпроцесса и обеспечение работы тайм-аут-инструкций.

2.1. Абстрактная модель ПО ВС МПОА

На основе модели гиперпроцесса разработана модель ПО встраиваемых систем, учитывающая требования, обусловленные спецификой программирования микроконтроллерных встраиваемых систем. Основной недостаток модели гиперпроцесса в применении к микроконтроллерным системам заключается в ее строгой синхронности на уровне функций-состояний процессов. Это свойство в значительной мере обуславливает высокую эффективность процесс-ориентированного похода, однако совершенно не допускает описание аппаратных прерываний, которые по определению исполняются асинхронно, вплоть до уровня инструкций процессора. Причем эта асинхронность является необходимым условием, поскольку прерывания используются для обработки событий, критических к времени реакции.

В связи с этим предложено расширить модель алгоритма управления до совокупности асинхронно исполняющихся компонентов при сохранении синхронного исполнения внутри каждого компонента. В существующих исследованиях этот подход к моделированию называется «глобально асинхронные локально синхронные системы» (GALS) [46-49] и применяется для описания поведения распределенных систем управления, состоящих из множества отдельных вычислительных устройств с независимым тактированием.

Формально, ПО микроконтроллера в предложенной модели представляется в виде совокупности множеств:

(H, V, C, R, T), где:

H – множество модифицированных гиперпроцессов, $H = \{h_0,\ldots, h_n\}$

$V = \{v_1,\ldots,v_{nv}\}$ и $C = \{c_1,\ldots,c_{nc}\}$ – множества глобальных переменных и констант программы, типов $t_1^v,\ldots,t_{nv}^v$ и $t_1^c,\ldots,t_{nc}^c$ соответственно.

$R =\{r_1,\ldots,r_{nr}\}$ – множество используемых в программе входных и выходных регистров.

T – счетчик системного времени.

Гиперпроцесс $h_i \in H$ определяется парой источник активации – множество процессов:

$$h_i \equiv (a_i, P_i).$$

Таким образом, модель алгоритма управления, используемая в процесс-ориентированном программировании, была расширена преставлением программы в виде множества гиперпроцессов с различными источниками активации и дополнена переменными, константами и регистрами ввода/вывода, представляющими внешнюю для программы среду. Понятие источника активации является абстракцией службы времени в оригинальной модели гиперпроцесса. Источник активации произвольным образом генерирует событие x_h^i и может быть остановлен.

Эти изменения также потребовали изменения внутренней структуры гиперпроцесса и составляющих его процессов. Процесс $p_j^i \in P_i$ представляется множеством функций-состояний S_l^i, выделенными начальным состоянием $s_j^{i,1}$ текущим состоянием cs_j^i и временем t_j^i последней смены состояния:

$$p_j^i \equiv (S_l^i, s_j^{i,1}, cs_j^i, t_j^i).$$

Для всех процессов в программе также определено общее состояние останова s_{stop}. k-ое состояние j-ого процесса i-ого гиперпроцесса обозначается $s_k^{i,j}$. Состояние s процесса представлено множеством событий X, множеством реакций Y и бинарным отношением $\rightarrow \in X \times Y$, задающим соответствие реакций событиям:

$$s \equiv (X, Y, \rightarrow).$$

Поскольку период активации T_H в гиперпроцессе был замещен источником активации, гиперпроцесс уже не имеет четко заданного периода тактирования и может активироваться нерегулярно. По этой причине временные характеристики процессов не могут измеряться дискретно в количестве тактов гиперпроцесса, и время t_p становится (условно) непрерывным. Это также означает, что вместо службы времени, отвечающей за тактирование, в программе присутствует счетчик времени T, общий для всех гиперпроцессов. Событие тайм-аута переопределяется следующим образом:

$$timeout(p_j^i, t_{timeout}) \equiv (T - t_j^i \geq t_{timeout}).$$

В оригинальной модели гиперпроцесса событие активации гиперпроцесса является необходимым условием для исполнения всех реакций. В предложенной модиификации не гарантируется регулярная и периодичекая активация гиперпроцессов, и событие тайм-аута может не быть обнаружено в случае если оно тактируется тем же источником активации, что и остальной код процесса. В связи с этим активация гиперпроцесса не должна входить в условия исполнения реакции тайм-аута – для исполнения реакции должно быть достаточно того, что гиперпроцесс h_i активен, процесс p_j^i находится в соответствующем состоянии $s_k^{i,j}$ и выполнено условие $timeout$ $(p_j^i, s_k^{i,j}, t_{timeout})$.

На практике это означает необходимость наличия в системе хотя бы одного источника активации, который работает регулярно с достаточно небольшим периодом и не может быть остановлен. Обработка событий тайм-аутов при этом производится в служебном гиперпроцессе, активируемым этим источником.

С введением счетчика системного времени T изменяются также определения реакций смены состояния, запуска и остановки процессов и сброса тайм-аута:

Операция/Событие	**Описание**
$start\_p(p_j^i) \equiv (cs_j^i := s_j^{i,1};\ t_j^i := T)$	Запуск процесса

$stop\_p(p_j^i) \equiv (cs_j^i := s_{stop})$	Остановка процесса
$set\_state(p_j^i, s_j^{i,k}) \equiv (cs_j^i := s_j^{i,k}; t_j^i := T)$	Смена состояния процесса
$reset(p_j^i) \equiv (t_j^i := T)$	Сброс таймаута процесса
$timeout(p_j^i, t_{timeout}) \equiv (T - t_j^i \geq t_{timeout})$	Тайм-аут
$inactive(p_j^i) \equiv (cs_j^i = s_{stop})$	Проверка неактивности процесса
$active(p_j^i) \equiv \neg (cs_j^i = s_{stop})$	Проверка активности процесса

Взаимодействие между отдельными гиперпроцессами определяется аналогично межпроцессному взаимодействию и представлено операциями:

- Запуска гиперпроцесса $start\_h(a_i)$,
- Остановки гиперпроцесса $stop\_h(a_i)$,
- Проверки активности гиперпроцесса $active\_h(a_i)$.

При этом все эти операции осуществляются над источниками активации.

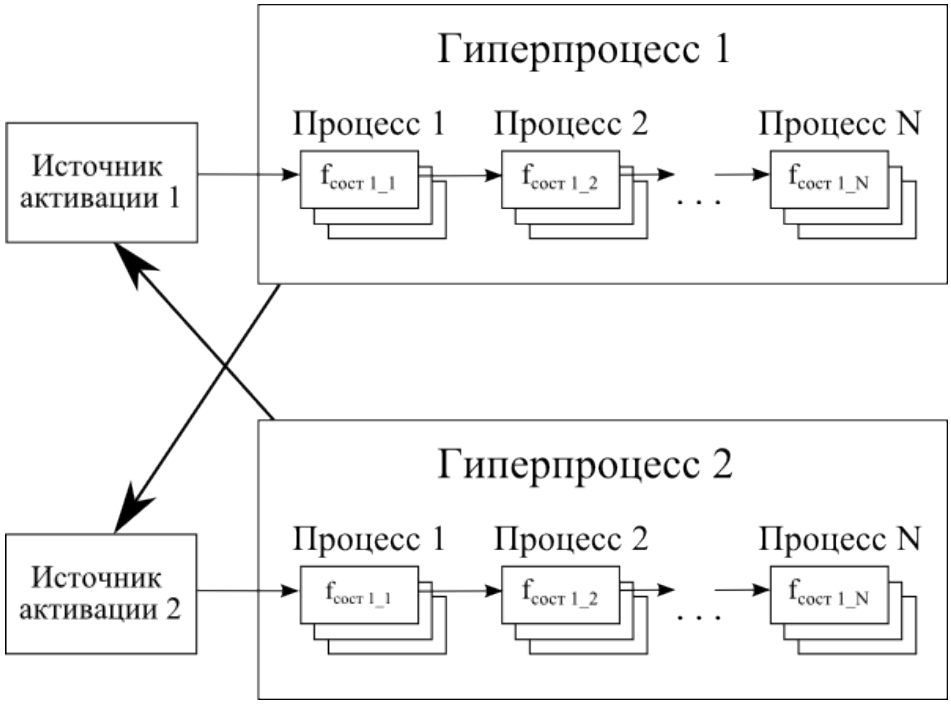


Рис. 1. Управление источниками активации гиперпроцессов

Возможность запуска и остановки гипрепроцессов также играет важную роль в задании точек синхронизации между гиперпроцессами. В то время как

полностью исключить асинхронное исполнение кода невозможно, в ряде случаев возникает необходимость обеспечения атомарности операций над разделяемыми данными. В некоторых ГАЛС-моделях, основанных на взаимодействующих последовательных процессах [50], эта проблема решается организацией очередей передачи данных. Этот подход хорошо подходит при теоретическом рассмотрении сложных распределенных систем, однако оказывается слишком ресурсоемким в случае встраиваемых систем на МПОА. Помимо этого, возникают вопросы задания длины очереди, а также корректности поведения системы в случаях ее переполнения.

Альтернативный подход, являющийся вырожденным случаем предыдущего, заключается в использовании механизмов организации критических секций вокруг операций над небольшими участками разделяемых данных. В случае систем на 8-битных архитектурах это особенно актуально, поскольку неконтролируемое общее использование даже 2-байтовых переменных отдельными гиперпроцессами может привести к гонкам, влекущим повреждение данных и неопределенное поведение программы. В разработанной модели такой механизм выражен атрибутом *atomic* отдельных (как правило вычислительных) реакций в функциях-состояниях процессов. Наличие этого атрибута в общем случае означает, что на время исполнения реакции приостанавливается работа источников активации всех гиперпроцессов, кроме того, в котором эта реакция исполняется. Это эквивалентно тому, что исполнение реакции предваряется набором операций останова гиперпроцесса *stop_h*(h_i), и завершается набором операций запуска *start_h*(h_i) для всех гиперпроцессов h_i, потенциально способных вмешаться в исполнение кода критической секции.

2.2. Модель ПО ВС на микроконтроллерах

Рассмотрим описанную модель алгоритма управления в приложении к построению систем на микроконтроллерах. Введение понятия источника активации гиперпроцесса позволяет описывать прерывания

микроконтроллера в рамках этой модели. Источником активации гиперпроцесса может выступать таймер, основной цикл программы, или аппаратное прерывание. Гиперпроцессы, источником активации для которых служат прерывания, описываются внутри функций-обработчиков прерываний (Interrupt Service Routine). Выделяется фоновый гиперпроцесс h_0, источником активации для которого служит основной цикл программы.

2.2.1. Взаимодействие между гиперпроцессами

Для каждого источника активации нобходимо определить условия генерации активирующего события, а также операции запуска и останова гиперпроцесса. В случае гиперпроцессов, активируемых прерываниями, событие активации генерируется внешним для процессора периферийным устройством, а операции запуска/останова гиперпроцесса определяются как разрешение/запрет соответствующего прерывания путем установки бит в регистрах микроконтроллера. Для фонового гиперпроцесса событие активации генерируется безусловно, операции остановки и запуска гиперпроцесса, в зависимости от возможностей микроконтроллера, могут быть запрещены, либо определены как вход процессора в спящий режим и, соответственно, выход из него.

2.2.2. Служба времени и тайм-ауты прерываний

Для обеспечения отслеживания системного времени T в системе присутствует служебный процесс, активируемым прерыванием переполнения одного из таймеров микроконтроллера. Частота срабатывания этого прерывания задается достаточной для отслеживания временных интервалов с точностью до милли- или микросекунды, в звыисимости от требований и возмодностей аппаратной платформы. Это же прерывание может использоваться для отслеживания тайм-аутов гиперпроцессов. В случае, если операция останова фонового гиперпроцесса не определена, и он всегда активен, обработка тайм-аутов гиперпроцессов прерываний осуществляется в

фоновом гиперпроцессе. С точки зрения реализации это означает, что для каждого процесса из гиперпроцессов прерываний создается его сокращенная копия, содержащая для кажого состояния процесса только операции обработки тайм-аутов и всегда находящаяся в одном состоянии с самим процессом.

2.2.3. Синхронизация доступа к данным

На практике разрешение вложенных прерываний нецелесообразно и может приводить к переполнению стека. На большинстве микроконтроллерных платформ прерывания могут прерывать только исполнение основного цикла программы. Исполнение кода процедур обработки прерываний, таким образом, оказывается атомарным. Соответственно, атомарным будет и исполнение кода состояний процессов, описанных в гиперпоцессах прерываний. Исключение составляет обработка тайм-аутов, выносимая в фоновый гиперпроцесс.

Организация критических секций на микроконтроллерах может осуществляться механизмами временного запрета все прерываний. Во многих микроконтроллерных архитектурах приисутствует воможность маскирования прерываний, при это операции установки маски реализуются одной-двумя инструкциями процессора, что делает накладные расходы на организацию критических секций пренебрежимыми.

2.2.4. Исполнение модели ПО на микроконтроллере

Для демонстрации динамических свойств разработанной модели, определим ее основные составляющие с точки зрения порядка их исполнения на микроконтроллере. Активация гиперпроцесса заключается в последовательном исполнении текущих функций-состояний входящих в него процессов:

$$exec(h) \equiv (exec(f_1^{cur}); \ldots; exec(f_N^{cur})).$$

Оператором *exec*() здесь обозначено исполнение кода на процессоре, точка с запятой используется для обозначения последовательной композиции. При начальной активации гиперпроцесса выделенный начальный процесс p_1 находится в начальной функции-состоянии f_1^1, остальные процессы находятся в состоянии останова f_i^{stop}. Исполнение алгоритма управления представляется как циклическая активация фонового гиперпоцесса и асинхронная активация остальных гиперпроцессов по возникновению соответствующих прерываний:

$$exec(H) \equiv exec(h_{bkg})^* \ || \ Int_1 \rightarrow exec(h_{Int1}) || \ldots || \ Int_M \rightarrow exec(h_{IntM}) ..$$

На каждом цикле активации (такте) гиперпроцесса последовательно исполняются текущие функции-состояния всех активных процессов этого гиперпроцесса (Рис. 2). Во время исполнения функции-состояния проверяются все обрабатываемые ей события и выполняются реакции, определенные для этих событий.

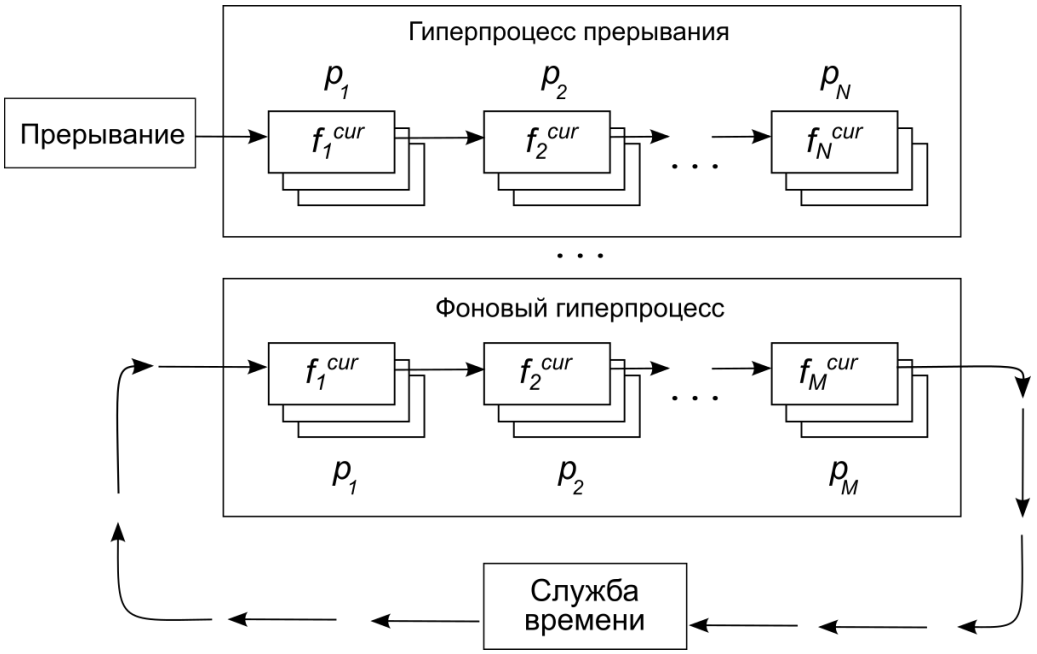


Рис. 2. Активация гиперпроцессов прерываний и фонового гиперпроцесса

2.3. Пример спецификации ПО микроконтроллера

Для примера рассмотрим упрощенную систему управления микроволновой печью. Система имеет два входных сигнала – кнопку и датчик открытия двери, и два выходных сигнала для управления нагревателем и для выдачи звукового сигнала. По нажатию на кнопку счетчик времени готовки

выставляется на 1 секунду и запускается нагреватель. Каждое последующее нажатие на кнопку увеличивает время готовки на одну секунду. По истечении заданного времени нагреватель отключается, и выдается звуковой сигнал. Открытие двери в процессе готовки отключает нагрев и сбрасывает время готовки в ноль.

Рассмотрим вариант описания этого алгоритма управления при помощи разработанного формализма. Система состоит из двух гиперпроцессов – фонового гиперпроцесса h_0 и гиперпроцесса h_1, активируемого внешним прерыванием по сигналу с кнопки. Алгоритм описывается четырьмя процессами:

Процесс	**Описание**	**Гиперпроцесс**	**Состояния**
p_1^0	Инициализация	h_0	$s_1^{0,1}$ – начальное состояние
p_2^0	Готовка	h_0	$s_2^{0,1}$ – начало готовки $s_2^{0,2}$ – готовка $s_2^{0,3}$ – звуковой сигнал
p_3^0	Контроль двери	h_0	$s_3^{0,1}$ – контроль двери
p_1^1	Контроль кнопки	h_1 – внешнее прерывание по сигналу с кнопки	$s_1^{1,1}$ – нажатие $s_1^{1,2}$ – контроль дребезга нажатия $s_1^{1,3}$ – нажато $s_1^{1,4}$ – контроль дребезга отпускания

В начале работы системы активен только процесс p_1^0, который осуществляет инициализацию встроенной аппаратной периферии и запускает процессы, участвующие в штатной работе программы, после чего останавливается. Процесс состоит из одного начального состояния $s_1^{0,1}$, представленного набором безусловных реакций:

→ heater := OFF	Инициализация нагревателя
→ buzzer := OFF	Инициализация звукового
→ cooking_time := 0	излучателя
→ *set_h1_back*	Инициализация времени готовки
	Установка внешнего прерывания на
→ *start_p*(p_3^0)	срабатывание по заднему фронту
→ *start_p*(p_1^1)	Запуск процесса контроля двери
→ *start_h*(h_1)	Запуск процесса контроля кнопки
→ *stop_p*(p_1^0)	Запуск гиперпроцесса прерывания h_1
	Остановка процесса

Для упрощения записи событие активации x_h здесь было опущено. Соответственно, запись $\rightarrow y$ эквивалентна $x_h \rightarrow y$, а запись $x \rightarrow y$ эквивалентна $x_h \wedge x \rightarrow y$, где y – произвольная реакция, x – произвольное событие, кроме события тайм-аута.

Процесс готовки p_2^0 в норме не активен, по запуску включает нагреватель и переходит в состояние готовки, в котором отслеживает заданное время. По истечении времени готовки процесс отключает нагреватель, включает звуковой излучатель и переходит в состояние выдачи звукового сигнала. По истечении времени звукового сигнала (1 с), процесс отключает звуковой излучатель, обнуляет счетчик времени готовки и останавливается.

$s_2^{0,1}$ – начало готовки

→ heater := ON

→ *set_state*($s_2^{0,2}$)

$s_2^{0,2}$ – готовка

timeout(cooking_time) → heater := OFF

timeout(cooking_time) → buzzer := ON

timeout(cooking_time) → cooking_time := 0

timeout(cooking_time) → *set_state*($s_2^{0,3}$)

$s_2^{0,3}$ – звуковой сигнал

timeout(1000) → buzzer := OFF

timeout(1000) → *stop_p*(p_2^0)

Процесс контроля двери состоит из одного состояния, в котором отслеживает входной сигнал с датчика двери, и по событию открытия двери останавливает процесс готовки, отключает нагреватель и звуковой излучатель и обнуляет время готовки.

$s_3^{0,1}$ – контроль двери

door_open → heater := OFF

door_open → buzzer := OFF

door_open → *stop_p* (p_2^0)

door_open → cooking_time := 0

Процесс контроля кнопки p_1^1 активируется в функции-обработчике внешнего прерывания микроконтроллера, изначально настроенного на задний фронт сигнала. В отпущенном состоянии сигнал с кнопки находится в высоком логическом уровне. При активации в состоянии $s_1^{1,1}$, процесс увеличивает время готовки на одну секунду, настраивает внешнее прерывание на срабатывание по переднему фронту сигнала, запускает процесс готовки p_2^0, если он не активен, и переходит в состояние контроля дребезга нажатия. После задержки на дребезг, процесс фиксирует первый передний фронт сигнала в состоянии $s_1^{1,3}$ и переходит к контролю дребезга отпускания. После задержки на дребезг отпускания, процесс настраивает внешнее прерывание на срабатывание по заднему фронту и возвращается в начальное состояние.

$s_1^{1,1}$ – нажатие

→ cooking_time := cooking_time + 1000

→ *set_h1_front*

→ *set_state*($s_1^{1,2}$)

¬*active*(p_2^0) → *start_p* (p_2^0)

$s_1^{1,2}$ – контроль дребезга нажатия

timeout(10) → *set_state* ($s_1^{1,3}$)

$s_1^{1,3}$ – нажато

→ *set_state* ($s_1^{1,4}$)

$s_1^{1,4}$ – контроль дребезга отпускания

timeout(10) → *set_h1_back*

timeout(10) → *set_state* ($s_1^{1,1}$)

Следует отметить существенное отличие описания процесса, активируемого внешним прерыванием от процессов, исполняющихся регулярно в основном цикле программы. В процессе p_1^1 сам факт активации уже является значимым событием, причем смысл этого события может меняться перенастройкой источника активации – внешнего прерывания по мере исполнения процесса. Для сравнения рассмотрим описание процесса контроля кнопки в фоновом гиперпроцессе:

$s_4^{0,1}$ – начальное состояние

button = 1 → *set_state* ($s_4^{0,2}$)

button = 0 → *set_state* ($s_4^{0,4}$)

$s_4^{0,2}$ – кнопка отпущена

button = 0 → cooking_time := cooking_time + 1000

button = 0 ^ ¬*active*(p_2^0) → *start_p* (p_2^0)

button = 0 → *set_state* ($s_4^{0,3}$)

$s_4^{0,3}$ – контроль дребезга нажатия

timeout(10) → *set_state* ($s_4^{0,4}$)

$s_4^{0,4}$ – кнопка нажата

button = 1 → *set_state* ($s_4^{0,5}$)

$s_4^{0,5}$ – контроль дребезга отпускания

timeout(10) → *set_state* ($s_4^{0,2}$)

Описание процесса в прерывании получилось более кратким. В случае реализации в фоновом гиперпроцессе опрос кнопки происходит периодически, в то время как использование внешнего прерывания позволяет

избежать лишних активаций процесса. Также прерывание обеспечивает реакцию непосредственно после нажатия кнопки, по первому заднему фронту сигнала. Процесс, реализованный в основном цикле программы может пропустить первые изменения сигнала и теоретически может не отреагировать на нажатие до окончания дребезга. Незамедлительная реакция на событие не имеет большого значения в случае обработки нажатий на кнопку, но в ряде других приложений это может быть необходимым требованием.

Реакции в состояниях процессов, активируемых прерываниями, исполняются только в случае возникновения события, привязанного к этому прерыванию. Таким образом, отсутствует возможность задания безусловных реакций в таких процессах. При этом для обеспечения работы тайм-аутов, их обработка выносится в фоновый гиперпроцесс. Это приводит к интересному побочному эффекту: тайм-ауты могут использоваться в этих процессах для исполнения части реакций независимо от того, возникло прерывание или нет. Например, реакции, связанные с событием *timeout* (0) будут гарантированно исполняться независимо от возникновения прерываний. Этот эффект может быть использован, в частности, для инициализации данных или периферийных устройств, используемых процессами, активируемых прерываниями, что позволит сохранить целостность описания функциональности в одном процессе. В рассмотренном примере начальная настройка прерывания и даже запуск гиперпроцесса для процесса контроля кнопки могут быть вынесены из процесса инициализации в дополнительное начальное состояние процесса p_1^1:

$s_1^{1,1}$ –инициализация

timeout(0) → *set_h1_back*

timeout(0) → *start_h*(h_1)

timeout(0) → *set_state*($s_1^{1,2}$)

При этом описание процесса p_1^1 становится самодостаточным и может быть переиспользовано без необходимости внесения дополнительных изменений в другие процессы.

Таким образом, разработанный формализм позволяет специфицировать алгоритмы управления на микроконтроллерных платформах. Описанная в примере управляющая программа была реализована средствами языка C на платформе Seeeduino 2.21 на базе микроконтроллера ATmega168. Использованные для упрощения описания метаоперации «heater := ON/OFF», «buzzer := ON/OFF», «set_h1_back» и т. п. в программе представлены чтением/записью соответствующих битов в регистрах микроконтроллера. Подробное описание представления систем на базе разработанного формализма в программах на языке C рассматривается в Главе 3.

В зависимости от задачи, счетчик физического времени T может хранить значение времени в миллисекундах, или микросекундах. Диапазон измеряемых временных интервалов в случае миллисекундного счетчика – от 1 мс до почти 50 суток, в случае микросекундного – от 1 мкс до чуть более 1 часа, при условии что используется 32-битные значения. В ряде задач может одновременно требоваться возможность измерения очень малых и очень больших временных промежутков. При управлении физическими процессами может потребоваться измерение временных промежутков более 1 часа, в то время как протоколы высокоскоростной коммуникации с периферийными устройствами микроконтроллера могут предполагать реакцию на события с микросекундной точностью. В этом случае в программе может одновременно использоваться два счетчика времени. При этом время t_j^i нахождения процесса в состоянии также будет представлено двумя значениями – в миллисекундах и в микросекундах.

2.4. Структура состояния процесса

Как видно из описания состояний процессов, одному событию в системе может сопоставляться более одной реакции. Для упрощения записи такие реакции можно объединять в более сложные реакции, однако при этом возникает вопрос порядка вычисления событий и исполнения реакций.

Например, состояние $s_2^{0,2}$ процесса p_2^0 можно представить следующим образом введя оператор последовательного исполнения «;»:

timeout(cooking_time) → heater := OFF ;
buzzer := ON ;
cooking_time := 0 ;
set_state($s_2^{0,3}$)

При этом семантика такой записи неоднозначна и может отличаться при реализации программы. Существенная особенность разработанного формализма состоит в том, что порядок исполнения реакций имеет значение. Исполнение гиперпроцесса в языке Reflex предполагает, что значения всех переменных и входных/выходных сигналов изменяются только в начале/конце цикла активации. Внутри цикла активации эти значения фиксированы, что позволяет не учитывать порядок исполнения реакций. Реализация этого механизма требует дублирования всех переменных для хранения изначальных и измененных значений, что значительно увеличивает используемый программой объем памяти. На микроконтроллерных, где самый жесткие ограничения накладываются именно на доступные объемы ПЗУ и ОЗУ, эти накладные расходы оказываются существенными. По этой причине в разработанном формализме от этого механизма решено было отказаться. Как следствие, оказывается важным не только порядок исполнения реакций, но и последовательность во времени вычисления событий. Для примера рассмотрим состояние $s_2^{0,1}$ процесса p_2^0. Все реакции в этом состоянии привязаны к событию тайм-аута, зависящему от значения переменной cooking_time. При этом в одной из реакций значение этой переменной изменяется. В случае, если вычисление событий и исполнение реакций будут происходить в порядке, в котором они описаны, поведение программы будет некорректным.

Также в этой записи неявно задаются точки синхронизации между гиперпроцессами. Переменная cooking_time используется как в фоновом гиперпроцессе, так и в гиперпроцессе прерывания. Соответственно, для всех

операций с этой переменной внутри фонового гиперпроцесса необходимо обеспечить атомарность посредством критических секций. Это включает в себя проверку события *timeout*(cooking_time), так как обнаружение этого события предполагает чтение переменной cooking_time. Критические секции также необходимы для операций над состояниями процессов – запуска/остановки процессов и проверки тайм-аутов. Для расстановки критических секций необходимо однозначно задать порядок исполнения всех операций в состояниях процессов и учесть, что вычисление события также является операцией.

В связи с этим было предложено изменить структуру состояния процесса. Состояние представляется упорядоченным множеством операций. Операция может состоять из безусловной вычислительной реакции, управляющих реакций смены состояния, запуска/остановки процессов/гиперпроцессов, или условной операции – пары событие-операция. Вычисление события при этом также является операцией, исполняемой непосредственно перед проверкой события. Пара событие-реакция $x \rightarrow y$ представляется последовательностью действий:

- вычислить x;
- если x, выполнить y.

При этом реакция y может быть составлена из нескольких последовательных операций, в том числе условных. В случае, если в вычислении события x участвуют разделяемые между гиперпроцессами переменные, операция вычисления события x может быть помещена в критическую секцию, а результат вычисления может быть сохранен в неразделяемой памяти.

Отдельно выделяется операция обработки тайм-аута, поскольку проверка события тайм-аута и исполнение соответствующей реакции происходят независимо от активации гиперпроцесса и исполнения остальных операций в состоянии. Таким образом, состояние $s_k^{i,j} \in S_j^i$ процесса p_j^i

представляется упорядоченным множеством операций $sbody_k^{i,j}$, составляющих тело состояния, и выделенной операцией обработки тайм-аута $ts_k^{i,j}$:

$$s_k^{i,j} \equiv (sbody_k^{i,j}, ts_k^{i,j}),\ ts_k^{i,j} \in sbody_k^{i,j}\ .$$

Такое представление состояния процесса однозначно задает последовательность исполнения операций, и также обеспечивает терминологическую базу для последующего описания синтаксиса языка и эквивалентного представления программ на языке С.

Основные выводы главы

В ходе работы разработан формализм описания управляющих алгоритмов встраиваемых систем на микроконтроллерных платформах. За основу был взят формализм гиперпроцесса, используемый в процесс-ориентированном языке Reflex. Программа микроконтроллера описывается набором асинхронно исполняющихся гиперпроцессов с раздельными источникам активации. На микроконтроллерах в качестве источников активации выступают аппаратные прерывания и основной цикл программы. При этом процедуры обработки прерываний оформляются в виде гиперпроцессов.

В связи с асинхронным исполнением гиперпроцессов, изменены механизмы отслеживания времени. Вместо дискретного времени, измеряемого тактами гиперпроцесса, введен счетчик физического времени в миллисекундах, обновляемый одним из таймеров микроконтроллера. В некоторых задачах для обеспечения возможности более точного измерения временных интервалов, может быть введен дополнительный счетчик, хранящий время в микросекундах. Использование одновременно двух счетчиков позволяет одновременно работать с малыми временными интервалами (от 1 мкс) и со сравнительно большими (до почти 50 суток) временными интервалами.

Переопределены операции смены состояния, запуска и остановки процессов и сброса тайм-аута, а также события тайм-аута и проверки

активности процессов. Предложена схема независимого тактирования операций обработки тайм-аутов, обеспечивающая сохранение корректной работы тайм-аутов в гиперпроцессах с нерегулярной активацией.

Определены механизмы взаимодействия между гиперпроцессами посредством операций запуска/останова источников активации и механизмы синхронизации работы с разделяемой памятью при помощи критических секций.

В связи с отказом от ресурсоемкого механизма дублирования значений переменных и регистров ввода/вывода изменена структура состояний процессов. Состояние процесса описывается набором операций со строго заданной последовательностью, при этом вычисление событий также определяется как операция, исполняемая непосредственно перед реакциями на события. Строгое задание порядка исполнения операций необходимо для однозначного определения точек синхронизации между гиперпроцессами и оформления критических секций.

Применение разработанного формализма для описания ПО микроконтроллеров продемонстрировано на примере упрощенного алгоритма управления микроволновой печью.

Разработанный формализм отражает специфику программирования микроконтроллеров при сохранении принципов процесс-ориентированного программирования и выполняет следующие функции:

- Задает понятийный аппарат для разработанного языка IndustrialC
- Служит в качестве промежуточного представления между программой на IndustrialC и ее эквивалентным представлением в процессе трансляции.
- Дает терминологическую базу для дальнейшего определения эквивалентного представления на языке C в привязке к синтаксису языка.

Глава 3. Язык IndustrialC

В предыдущей главе был представлен разработанный формализм описания программ микроконтроллеров во встраиваемых системах и приведен пример спецификации упрощенной системы управления с его помощью. На основе представленного формализма был разработан специализированный язык программирования IndustrialC. В данной главе рассматривается синтаксис языка и его неформальная семантика, приводятся примеры реализации программ с его помощью.

В основе синтаксиса языка лежат языки C и Reflex. IndustralC включает большую часть синтаксических конструкций обоих языков, при этом специализированные конструкции разрабатывались в общем стиле языков C и C++.

3.1. Основные лексические элементы языка.

3.1.1. Идентификаторы

Идентификаторы представляются произвольными последовательностями латинских прописных и строчных букв, десятичных цифр и символа «_». Идентификаторы имеют произвольную длину, не могут начинаться с цифры и не должны совпадать с ключевыми словами языка. Формально правила построения идентификаторов задаются регулярным выражением:

```
[a-zA-Z_][a-zA-Z0-9_]*
```

Идентификаторы используются в языке для задания имен процессов, гиперпроцессов, состояний, функций, переменных, а также векторов, регистров и битов микроконтроллера. Идентификаторы переменных,

векторов, регистров и битов должны быть объявлены до их первого использования в программе.

3.1.2. Константы

Десятичные константы составляются из набора десятичных цифр и описываются следующим регулярным выражением:

```
[0-9]+
```

Шестнадцатиричные константы начинаются с последовательности символов «0x», за которой следует произвольная последовательность шестнадцатиричных цифр, включая строчные и прописные буквы a-f и A-F, и описываются регулярным выражением:

```
"0x"[a-fA-F0-9]+
```

Двоичные константы состоят из символьной последовательности «0b» за которой следует произвольный набор двоичных цифр 0 и 1, и описываются регулярным выражением:

```
"0b"[0-1]+
```

Введение возможности описания двоичных констант, не предусмотренных стандартом языка C обусловлено спецификой программирования микроконтроллеров и позволяет сделать более удобной работу с битовыми последовательностями, в том чиле при записи и чтении регистров микроконтроллера.

Для обеспечения работы типа bool поддерживаются логические константы `true` и `false`;

Строковые константы и строковые литералы определяются по правилам языка C и задаются соответственно регулярными выражениями:

```
'(\\.|[^\\'])+' и
\"(\\.|[^\\"])*\"
```

В программах на IndustrialC поддерживаются комментарии в стиле C и C++, а также стандартные директивы C-препроцессора #include, #define, #ifdef, #endif и т. п. Список ключевых слов языка представлен в приложении.

Примеры описанных элементов языка:

`Control_valve2` - идентификатор

`666` – десятеричная целочисленная константа

`0xffee79` – шестнадцатеричная целочисленная константа

`0b11010` – двоичная целочисленная константа

`0.05` – десятеричная дробная константа

`'a'` – строковый литерал

`"IndustrialC"` – строковая константа

3.2. Структура программы

Программа на IndustrialC представлена набором из следующих элементов:

- Объявлений и определений переменных (включая массивы и именованные константы);
- служебных объявлений векторов, регистров и битов
- определений гиперпроцессов
- определений процессов
- определений функций
- строчных вставок на языке C

3.2.1. Объявления переменных

IndustrialC поддерживает объявление и определение переменных в стиле языка Си. Возможно объявление или определение списка переменных через запятую, объявление массивов постоянной длины, инициализация переменных и массивов таким же образом, как в языке Си.

Допустимые форматы объявления переменных:

<тип> <идентификатор>;

<тип> <список идентификаторов через запятую>;

Аналогично, формат определения перерменных:

<тип> <идентификатор> = <инициализатор>;

<тип> <список пар идентификатор-инициализатор>;

Тип представляется набором следующих спецификаторов и квалификаторов:

Спецификатор/Квалификатор	**Описание**
`char`	Символ
`int`	целое
`short`	Короткое
`long`	Длинное целое
`float`	Дробное с плавающей запятой
`double`	Длинное дробное с плавающей запятой
`signed`	Знаковое
`unsigned`	Беззнаковое
`bool`	Логическое
`const`	Квалификатор именованной константы
`volatile`	Квалификатор, запрещающий оптимизацию

Тип `bool` семантически соответствует типу `_Bool` в современных стандартах языка С. Семантика остальных типов переменных полностью соответствует соответсвующей семантике типов языка С и зависит от конкретной реализации используемого С-компилятора и целевой платформы. Например, в случае использования компилятора avr-g++ для 8-битных микркоонтроллеров семейства AVR, типы `int` и `short` совпадают и обозначают 16-битное целое, а тип `double` имеет ту же семантику, что и тип `float`.

В качестве инициализаторов переменных могут выступать соответствующие по типу выражения. Инициализация массивов, в том числе многомерных производится таким же образом, как в языке С, при этом

размерность и тип инициализаитора должны соответствовать размерности и типу массива.

Примеры объявлений и определений переменных, именованных констант и массивов:

```
unsigned int a = 5, b, c = 1;
float heater_temp, heater_temp_filtered;
float sensor_values[100];
const char PortNames [3][6] = { "PortA", "PortB", "PortC",};
char some_text[]="Process-Oriented Programming";
```

Семантика переменных и массивов, объявленных вне определений функций и состояний процессов, соответствует глобальным переменным языка C. Переменные, объявленные внутри определений функций и состояний процессов и вложенных операций в их составе, имеют семантику локальных переменных языка C, хранятся на стеке и существуют только во время исполнения кода своей области видимости.

3.2.2. Объявления векторов, регистров и битов

Формат объявления векторов регистор и битов в языке IndustrialC:

```
vector <имя_вектора>;
register <имя_регистра>;
bit <имя_бита>;
```

Эти объявления используются для обозначения доступных программе ресурсов микроконтроллера. Идентификаторы в объявлениях должны совпадать с соответствующими идентификаторами векторов, регистров и битов, объявленных на языке C в заголовочных файлах для выбранной модели микроконтроллера.

Язык предполагает два варианта использования этих объявлений. В первом случае программист объявляет идентификаторы по мере необходимости в ходе разработки программы. Это повышает

документированность исходного кода и позволяет в любой момент ознакомиться со списком прерываний и регистров, а соответственно и периферийных устройств, задействованных текущей версией программы. Во втором случае программист может подключить поставляемый заголовочный файл на IndustrialC для используемой модели микроконтроллера, содержащий все доступные в данной модели идентификаторы.

Примеры:

```
register UCSR0A;
vector TIMER2_COMPA_vect;
vector USART_UDRE_vect;
bit MPCM0;
bit U2X0;
```

3.2.3. Определения гиперпроцессов

Формат объявления гиперпроцессов в языке IndustrialC:

```
hyperprocess <идентификатор>
{
    vector = <идентификатор>;
    register = <идентификатор>;
    bit = <идентификатор>;
}
```

Определения гиперпроцессов используются для задания источников активации гиперпроцессов прерываний. Фоновый гиперпроцесс `background` отдельно в программе не определяется. Идентификатор имени гиперпроцесса выбирается произвольным образом и должен отражать природу события, по которому возникает используемое прерывание. Идентификаторы вектора, регистра и бита должны быть заранее объявлены в программе или включенном заголовочном файле. Идентификатор вектора задает прерывание, которым будет активироваться гиперпроцесс, регистр и бит используются для включения/выключения прерывания операторами

`start hyperprocess` и `stop hyperprocess`. В реализации языка для микроконтоллеров семейства AVR предполагается что прерывание разрешено, когда бит в регистре выставлен.

Пример – определение гиперпроцесса внешнего прерывания по кнопке для системы управления микроволновой печью, описанной в Главе 2:

```
hyperprocess ButtonExtInt
{
    vector = INT0_vect;
    register = EIMSK;
    bit = INT0;
}
```

3.2.4. Определения процессов

Формат определения процесса в языке IndustrialC:

```
process <идентификатор> : <идентификатор>
{
    <тело_процесса>
}
```

Первый идентификатор задается произвольным и обозначает имя процесса, которое должно отражать суть выполняемой процессом функции. Второй идентификатор – имя гиперпроцесса, тактирующего определяемый процесс. Это может быть ключевое слово `background`, обозначающее фоновый гиперпроцесс, либо идентификатор гиперпроцесса прерывания, определенного в программе. Определение гиперпроцесса при этом может следовать после определения тактируемых им процессов. Имена процессов в программе не должны повторяться, в том числе между гипепроцессами.

Тело процесса – набор:

- Объявлений и определений переменных.
- Определений состояний.

Переменные, объявленные или определенные внутри определения процесса имеют семантику глобальных переменных языка C. Область видимости этих переменных ограничена телом процесса и вложенными в него областями видимости.

3.2.5. Определения состояний

Формат определения состояния в языке IndustrialC:

```
state <идентификатор>
{
    <тело_состояния>
}
```

Идентификатор задает имя состояния. Имена состояний не должны повторяться внутри одного процесса, но могут повторяться между процессами. Для каждого процесса должно быть определено начальное состояние с именем `FS_START`. При запуске процесса операцией start process, процесс начинает работу в состоянии `FS_START`.

Допустимо также определение состояния `FS_STOP`, код которого будет выполняться, когда процесс остановлен. Такое определение может быть использовано для деинициализации при его остановке извне, например, для приведения используемой процессом периферии в безопасное состояние.

Тело состояния состоит из набора:

- Объявлений и определений переменных
- Операций
- Не более одной операции обработки тайм-аута.

В теле состояния должны присутствовать хотя бы одно определение, объявление или операция.

Семантика переменных, объявленных или определенных внутри состояния процесса, соответствует семантике локальных переменных в функциях языка C. Область видимости таких переменных также ограничена телом состояния и вложенными в него областями видимости. Определения

переменных внутри блочных операций семантически содержат инструкции, заново устанавливающие значения этих переменных при каждом исполнении блока кода.

3.2.6. Операции языка C

Язык IndustrialC поддерживает большинство операций языка C:

Операция-выражение используется для осуществления вычислительных действий над переменными и регистрами:

```
<выражение>;
```

Блочная операция используется для организации кода и создает новую область видимости для объявляемых в ней переменных:

```
{ <набор операций и/или объявлений/определений переменных> }
```

Условные операторы `if`, `if/else` и `switch/case/default` имеют синтаксис и семантику, идентичные этим операциям в языке C:

```
if(выражение) <операция>
if(выражение) <операция> else <операция>
switch (<выражение>)
{
    case <константа>: <операция> break;
    ...
    default: <операция>
}
```

Язык поддерживает оператор цикла for, синтаксис и семантика оператора соответствуют языку C:

```
for(<выражение>;<выражение>;<выражение>) <операция>
for(<определение переменной>;
<выражение>;<выражение>) <операция>
```

Оператор `break` может использоваться как в операции `switch/case`, так внутри циклов наряду с оператором `continue`. Семантика эти операторов соответствует языку C.

В IndustrialC накладывается ограничение на условие цикла – цикл не может быть безусловным и количество итераций должно быть константой. Эти ограничения позволяют избежать блокирующих операций, недопустимых при кооперативной модели многозадачности.

Оператор `return` используется в определениях функций:

```
return;
return выражение;
```

3.2.7. Специализированные операции

Кроме описанных операций языка C, в IndustrialC используются также специализированные операции управления процессами и гиперпоцессами, оформления критических секций и работы с временными интервалами посредством тайм-аутов. Семантика операций смены состояния, управления процессам, задания критической секции и обработки/сброса тайм-аута формально задана в Главе 2. Семантика операций управления гиперпроцессами зависит от целевой платформы – формальное описание применительно к семейству микроконтроллеров AVR приводится в Главе 4. Все перечисленные операции заданы только для состояний процессов и не могут использоваться внутри функций.

Операция смены состояния переводит процесса в указанное состояние:

```
set state <идентификатор>;
```

Состояние с таким идентификатором должно быть объявлено внутри тела рассматриваемого процесса, при этом не обязательно до операции set state. С помощью этой инструкции процесс может изменить только свое собственное состояние. Важно отметить, что операция set state не прерывает исполнение кода текущего состояния. После исполнения этой операции продолжится исполнение следующих операций по коду. Процесс окажется в

новом состоянии только при следующей активации. Если при исполнении состояния последовательно выполнятся несколько операций `set state` или инструкций `stop process`, на следующем цикле активации процесс окажется в состоянии, заданном последней из них.

Операция запуска процесса позволяет процессу запустить другой процесс, в том числе между гипепроцессами:

```
start process <идентификатор>;
```

Процесс с таким именем должен быть объявлен в любом месте программы. Если на момент исполнения инструкции процесс уже был активен, он перейдет в начальное состояние FS_START. В случае процессов, активируемых прерываниями, запуск процесса не гарантирует его исполнение если активирующее его прерывание отключено, или не возникает привязанное к этому прерыванию событие.

Операция остановки процесса позволяет процессу остановить самого себя, или другой процесс.

```
stop process <идентификатор>;
stop process;
```

Операции запуска и остановки гиперпроцессов позволяют процессам управлять источниками активации:

```
start hyperprocess <идентификатор>;
stop hyperprocess <идентификатор>;
stop hyperprocess;
```

Для гиперпроцессов прерываний операции остановки и запуска включают и отключают соответствующее прерывание. Для фонового гиперпроцесса в текущих реализациях языка эти операции не определены, но могут определяться как переход процессора в спящий режим и выход из него.

Операция atomic используется для задания критических секций:

```
atomic <операция>
```

На время исполнения операции внутри блока `atomic`, в зависимости от реализации, запрещаются либо все прерывания, либо прерывания,

активирующие процессы, пересекающиеся с ней по данным. В случае отсутствия возможности вложенных прерываний, использование этой операции в фоновом гиперпроцессе не имеет смысла.

3.2.8. Тайм-ауты

Операция обработки тайм-аута используется для работы с временными интервалами и для избежания «зависания» процессов в одном состоянии:

```
timeout (<целочисленная константа>)
{ <тело тайм-аута> }
```

Операция сброса тайм-аута семантически эквивалентна переходу процесса в текущее состояние:

```
reset timeout;
```

Пример:

```
timeout(1000)
{
    stop hyperprocess Blink;
    OCR1A = 2000;
    start hyperprocess Blink;
    reset timeout;
}
```

Константа задает временной интервал срабатывания тайм-аута в миллисекундах. Тело тайм-аута состоит из операций и объявлений/определений переменных. Код в теле тайм-аута срабатывает, если процесс находится в текущем состоянии дольше заданного времени. Определение нескольких тайм-аутов в одном состоянии запрещено. Тело тайм-аута должно содержать хотя бы одну операцию. Обработка тайм-аутов выполняется независимо от активации процессов.

3.2.9. Выражения

Синтаксис и семантика выражений IndustrialC практически полностью соответствуют языку C. В качестве простого выражения может выступать: -

- константа
- идентификатор переменной
- (*<выражение>*)
- выражения проверки активности процессов
- вставки выражений на языке C

Проверка активности процессов осуществляется двумя выражениями:

<идентификатор процесса> `active`

<идентификатор процесса> `inactive`

Семантика этих выражений формально описана в Главе 2 и Главе 4. Выражения `active` и `passive` имеют логический тип `bool` и принимают значения `true` и `false` (соответственно не ноль и ноль согласно семантике логических выражений языка C). Выражение `active` принимает значение `true`, если процесс находится в активном состоянии, то есть любом определенном состоянии, кроме `FS_STOP`. Выражение `inactive` принимает значение `true`, если процесс находится в состоянии `FS_STOP`.

При составлении более сложных выражений применяются операторы присваивания, унарные и бинарные операторы, оператор преобразования типов и вызовы функций:

Операторы присваивания	`=, >>=, <<=, +=, -=, *=, /=, %=, &=, ^=, |=`
Унарные префиксные операторы	`++, --, -, ~, !`
Унарные постфиксные операторы	`++, --`
Бинарные операторы	`||, &&, |, ^, &, ==, !=, <, >, <=, >=, <<, >>, +, -, *, /, %`

Синтаксис преобразования типов:

(<*тип*>)<*выражение*>

Допустимые типы данных описаны в начале главы в описании объявлений переменных.

Синтаксис вызова функций:

<*идентификатор*>()

<*идентификатор*>(<*список выражений через запятую*>)

Семантика перечисленных операторов, кроме active и inactive соответствует семантике идентичных операторов языка С. Для обеспечения возможности семантического анализа кода и автоматической расстановки критических секций в языке отсутствуют операторы работы с указателями.

3.2.10. Вставки кода на С

Несмотря на то, что IndustrialC поддерживает значительную часть синтаксиса языка С, в ряде случаев возникает необходимость использования неподдерживаемых конструкций. Примером могут послужить операции, требующие работы с указателями или динамического выделения памяти.

В языке предусмотрена возможность вставлять код на языке Си двумя способами:

- В виде строк, начинающихся символом $ и заканчивающихся символом новой строки \n
- Внутри одной строки между двумя ограничителями $$ … $$

Вставки С-кода первым способом допускаются в программе, вне определений функций, процессов и гиперпроцессов, а также внутри блочных операций, наряду с операциями и объявлениями/определениями переменных. Использование второго варианта ограничено выражениями, при этом вставка на С рассматривается как простое выражение, наряду с константами, идентификаторами и. т. д. Внутри кода на Си можно использовать переменные, объявленные средствами IndustrialC. Для этого перед переменной ставиться знак $. Второй вариант вставки С-выражений также позволяет

использовать в IndustrialC-коде переменные, объявленные внутри C-вставок. Необходимость такой конструкции обусловлена тем, что в сгенерированном Си - коде переменные из industrialC будут называться иначе, чем в изначальной программе – к их именам добавятся префиксы областей видимости (например, процесса или состояния).

Таким образом, обеспечивается двусторонняя связь между участками программы, реализованными на двух разных языках. Символ «$» не входит в состав языка C или C-препроцессора, поэтому такое обозначение не вызывает конфликтов или неоднозначностей.

Примеры строчных вставок C-кода:

```
int a, b, c;
float d;
$int a, b, c;
$int d = $a + $b - c;
```

В последней строке переменные `a` и `b` – из IndustrialC-кода, а переменная `c` – из кода на C. Пример вставки C-кода в выражениях:

```
if(a == b || $$ 0==strcmp($str, "msg") $$) { … }
```

3.2.11. Определения функций

IndustrialC позволяет определять функции аналогично определениям на языке C:

<тип> <идентификатор> (<список аргументов>) <блочная операция>

При этом по сравнению с объявлениями переменных, список допустимых спецификаторов типов расширяется спецификатором функций `inline` и спецификатором типа `void`. Список аргументов может быть пустым, либо состоять из перечисленных через запятую пар вида <тип> <идентификатор>

Тело функции не может содержать специализированных операций управления процессами и гиперпроцессами, обработки и сброса тайм-аутов и выражений проверки активности процессов.

3.3. Пример спецификации ПО микроконтроллера

Для демонстрации использования описанных конструкций языка приведем пример спецификации упрощенной системы управления микроволновой печью, описанной в Главе 2:

```
vector INT0_vect;
register EIMSK; bit INT0;
register DDRB;
register PORTB; bit PINB4; bit PINB5;
register EICRA; bit ISC00; bit ISC01;
volatile unsigned long cooking_time;
hyperprocess ButtonExtInt {
  vector = INT0_vect;
  register = EIMSK;
  bit = INT0;
}
process ButtonControl : ButtonExtInt{
  state FS_START {
    cooking_time += 1000;
    if (Cooking inactive)
      start process Cooking;
    //set_h1_front
    EICRA = ISC00 | ISC01;
    set state DebounceDown;
  }
  state DebounceDown {
    timeout(10) {
      set state Down;
    }
  }
  state Down {
    set state DebounceUp;
  }
  state DebounceUp {
    timeout(10) {
    EICRA = ISC01;//set_h1_back
    set state FS_START;
    }
  }
}
process Init : background {
  state FS_START    {
  DDRB = 0xff;
  PORTB = 0;
  EICRA = ISC01;//set_h1_back
  cooking_time = 0;
  start process ButtonControl;
  start hyperprocess ButtonExtInt;
  stop process;
  }
}
process Cooking : background {
  state FS_START {
  PORTB |= 1<<PINB5;//heater=ON;
  set state TrackTime;
  }
  state TrackTime {
    timeout(cooking_time) {
      //heater=OFF;
      PORTB &= ~(1<<PINB5);
      PORTB |= 1<<PINB4;//buzzer=ON;
      set state Buzz;
    }
  }
  state Buzz {
    timeout(1000) {
      //buzzer=OFF;
      PORTB &= ~(1<<PINB4);
      cooking_time = 0;
      stop process;
    }
  }
}
```

Для краткости описания в этой реализации отсутствует процесс контроля датчика открытия двери. Сразу после запуска программы активен только процесс, описанный первым, и принадлежащий к гиперпроцессу

background. Этот процесс находится в начальном состоянии `FS_START`. Все остальные процессы изначально неактивны и находятся в состоянии останова `FS_STOP`. В приведенном примере первый описанный фоновый процесс – Init – осуществляет инициализацию периферийных устройств, запускает процесс ButtonControl обработки событий с кнопки и останавливается. Процесс ButtonControl работает постоянно и активируется внешним прерыванием, перенастраивая его на срабатывание по переднему или заднему фронту сигнала.

Из примера видно соответствие представления программы на IndustrialC разработанному формализму описания ПО микроконтроллеров. В описании этого алгоритма, представленном в Главе 2, использовались метаоперации включения/выключения нагревателя и звукового излучателя и переключения фронтов срабатывания прерывания. В реализации на IndustrialC эти операции заменены на взаимодействие с периферийными устройствами – портом ввода/вывода B и внешним прерыванием – через регистры микроконтроллера. Также стоит отметить, что переменная `cooking_time` используется в гиперпроцессе прерывания, и операции с ней в фоновом гиперпроцессе следует помещать в критические секции операцией `atomic`. Однако в приведенном примере эта переменная используется фоновым гиперпроцессом только в операции тайм-аута, атомарность которой обеспечивается встроенными средствами языка.

Основные выводы главы

В главе рассмотрен разработанный синтаксис специализированного языка программирования IndustrialC, описаны основные лексические элементы и конструкции языка их семантика в привязке к языку C, а также к разработанному формализму описания ПО МК. За основу взяты языки C и Reflex. При разработке синтаксиса специфических для IndustrialC конструкций использовался синтаксис из C-подобных языков, таких как C++.

Синтаксис IndustrialC практически полностью включает синтаксис C, за исключением конструкций работы с памятью через указатели. Семантика конструкций IndustrialC, заимствованных из C полностью сохранена.

Из языка Refex заимствованы конструкции описания процессов и состояний, операций тайм-аутов, а также операций и выражений для взаимодействия между процессами. При этом семантика этих конструкций существенно отличается от их синтаксических эквивалентов в Reflex.

Синтаксис языка расширен конструкциями, позволяющими описывать процессы, активируемые прерываниями, унифицированными операциями управления источниками активации и обеспечения синхронизации доступа к разделяемой памяти. Для полного сохранения гибкости язык предусматривает механизмы вставки кода на языке C с возможностью совместного использования общих данных между двумя языками.

Соответствие между синтаксисом языка и разработанным формализмом описания ПО МК продемонстрировано на примере описания на IndustrialC системы управления, ранее описанной в Главе 2.

Глава 4. Транслятор и среда разработки

В данной главе задаются правила трансляции программ на IndustrialC для микроконтроллеров семейства AVR в код на языке C, рассматриваются средства реализации транслятора и среды разработки и описываются решения, лежащие в основе разработанных инструментальных средств.

4.1. Эквивалентное представление программ на языке C

Семантика большинства используемых на данный момент языков программирования описывается неформально, оставляя множество неоднозначностей, приводящих к различному поведению программ, собранных разными компиляторами для разных платформ. В таких случаях семантика языка программирования фактически формально фиксируется только правилами анализа и кодогенерации конкретного компилятора, а языком формального описания служит язык программирования, на котором реализован транслятор. При этом исходные коды трансляторов сложны для восприятия, а в ряде случаев могут быть просто закрытыми.

Во избежание неоднозначностей, семантика языка должна быть задана формально, при этом такое задание должно быть достаточно кратким и простым в восприятии. По этой причине семантику языка IndustrialC было решено формализовать через правила трансляции в язык C, семантика которого задана формально, например в [51,52,53], а также известна широкому кругу программистов, в том числе в области встраиваемых систем. Такой вариант задания семантики языка программирования в литературе называется трансляционной семантикой ([54,55,56]) и является разновидностью денотационной семантики с использованием целевого языка программирования (в данном случае C) в качестве денотационного метаязыка [57].

Представленное описание правил трансляции рассматривает только корректные IndustrialC-программы и предполагает, что конфликты имен и области видимости переменных предварительно разрешаются на начальных этапах трансляции. Описанные правила трансляции корректны для текущей реализации транслятора, нацеленной на 8-битные микроконтроллеры семейства AVR. В случае 32-битных микроконтроллеров реализация описания процедур обработки прерываний и механизмов разрешения/запрета прерываний отличаются. Следует также отметить, что цель данной формализации – однозначное задание семантики языка в применении к определенному классу целевых платформ, а также описание общих принципов работы созданных средств разработки.

Обозначим C_{ic} и C_c соответственно множества конструкций языков IndustrialC и C, а $C = C_{ic} \cup C_c$ – смешанное множество конструкций обоих языков. Правила трансляции программ на IndustrialC в язык C задаются определением бинарного отношения $\longmapsto \in C \times C$ для которого обеспечено свойство

$$\neg(c_1 \longmapsto c_2) \text{ для } c_1, c_2 \in C_c,$$

то есть конструкции, состоящие исключительно из конструкций языка C далее не транслируются. Описание правил трансляции также использует терминологию, введенную для описания разработанного формализма спецификации ПО микроконтроллеров в Главе 2.

4.1.1. Общая структура эквивалентного представления на языке C.

Программа P_{ic} на языке IndustrialC транслируется в программу P_c на языке C согласно следующему правилу:

$P_{ic} \longmapsto$

dec_{ints}

dec_{cv}

enum states$\{s_{stop},\ s_1^{0,1}, s_2^{0,1}, \dots, s_{l_{0,1}}^{0,1}, \dots, s_1^{n,1}, s_2^{n,1}, \dots, s_{l_n m_n}^{n,m_n}\}$;

```
states cs_1^0, cs_2^0, ... , cs_{m_0}^0, ..., cs_1^n, cs_2^n, ... , cs_{m_n}^n ;
unsigned long T, t_1^0, t_2^0, ..., t_{m_0}^0, ..., t_1^n, t_2^n, ..., t_{m_n}^n;
void main () {
    init_ts();
    init_procs();
    for(;;) {
        T = get_time_ms();
        timeouts();
        exec_h0();
    }
}
ISR(vec1) { exec_h1(); }
ISR(vec2) { exec_h2(); }
    ...
ISR(vecn) { exec_hn(); }
```

dec_{ints} обозначает множество макроопределений, задающих параметры активаторов гиперпроцессов. dec_{cv} состоит из набора объявлений переменных и констант.

Служебная функция `init_procs()` обнуляет значения t_j^i для всех процессов, выставляет текущие состояния cs_j^i всех процессов в значение s_{stop}, затем устанавливает текущее состояние cs_1^0 первого объявленного фонового процесса p_1^0 в его начальное состояние $s_1^{0,1}$:

```
inline void init_procs(){
    int i, j;
    for(i = 0; i < n; i++){
        for(j = 0; j < m_n; j++) {
            cs_j^i = s_stop;
            t_j^i = 0;
        }
```

```
    }
}
```

Функция `timeouts()` составлена из операций обработки тайм-аутов текущих состояний процессов, активируемых прерываниями:

```
inline void timeouts() {
    timeout_p_1^1(); timeout_p_2^1(); ... timeout_p_{m_1}^1();
    timeout_p_1^2(); timeout_p_2^2(); ... timeout_p_{m_2}^2();
        ...
    timeout_p_1^n(); timeout_p_2^n(); ... timeout_p_{m_n}^n();
}
```

Функция $timeout\_p_j^i()$ фактически реализует копию процесса p_j^i, разделяющую переменные состояния cs_j^i и времени с последней смены состояния:

```
inline void timeout_p_j^i() {
    switch(cs_j^i) {
        case s_{k_1}^{i,j}: ts_{k_1}^{i,j}; break;
        case s_{k_2}^{i,j}: ts_{k_2}^{i,j}; break;
            ...
        case s_{k_l}^{i,j}: ts_{k_l}^{i,j}; break;
        default: break;
    }
}
```

Функция $exec\_h_i$ реализует исполнение гиперпроцесса h_i, заключающееся в последовательной активации всех входящих в него процессов:

```
inline void exec_h_i () {
    exec_p_1^i ();
    exec_p_2^i ();
        ...
```

```
    exec_p^i_{m_i}();
}
```

Активация процессов, представленная функциями $exec\_p_j^i()$ имеет различные реализации для процессов исполняемых в фоновом гиперпроцессе h_0 и процессов, исполняемых в процедурах обработки прерываний.

Процесс реализуется машиной состояний, в зависимости от текущего состояния процесса cs_j^i, исполняется код (набор операций) соответствующего состояния. В описываемом представлении на языке С этот механизм реализуется при помощи switch-case-конструкций:

```
inline void exec_p_j^0() {
    switch (cs_j^0) {
        case s_1^{0,j}: { sbody_1^{0,j} ;break;}
        case s_2^{0,j}: { sbody_2^{0,j} ;break;}
            ...
        case s_{l_{0,j}}^{0,j}: { sbody_{l_{0,j}}^{0,j} ;break;}
        default: break;
    }
}
```

Другой распространенный способ реализации машин состояний на языке С – использование табличного представления с указателями на функции. Такой подход позволяет увеличить быстродействие исполнения автомата, однако порождает более сложный для восприятия и описания результирующий код. Дополнительно, использование динамических вызовов функций увеличивает использование оперативной памяти микроконтроллера. В описываемых правилах трансляции участки кода оформляются в подпрограммы с директивой inline, статически подставляемые при компиляции и сборке в исполняемый файл. Существуют также варианты

реализации машин состояний с использованием объектно-ориентированных технологий [8].

Реализация машины состояний для процессов $p_j^i, i \neq 0$, активируемых прерываниями, отличается тем, что из набора операций исключены операции обработки тайм-аутов – они исполняются в основном цикле программы в отдельных копиях этих процессов, реализуемых функциями $timeout\_p_j^i()$:

```
inline void exec_p_j^i() {
    switch (cs_j^i) {
        case s_1^{i,j}: { sbody_1^{i,j}\ts_1^{i,j} ;break; }
        case s_2^{i,j}: { sbody_2^{i,j}\ts_2^{i,j} ;break; }
            ...
        case s_{l_{i,j}}^{i,j}: { sbody_{l_{i,j}}^{i,j}\ts_{l_{i,j}}^{i,j} ;break; }
        default: break;
    }
}
```

4.1.2. Служебные объявления.

Служебные объявления идентификаторов векторов прерываний, регистров микроконтроллера и битов в этих регистрах служат для указания транслятору, что эти идентификаторы допустимы для использования в программе в соответствующих ролях. В случае, если идентификатор вектора, регистра или бита используется без предварительного объявления, транслятор выдаст ошибку. Этот механизм служит двум различным целям, в зависимости от варианта использования. С одной стороны, он позволяет на любом этапе разработки программы отслеживать, какие вектора прерываний, регистры микроконтроллера и биты этих регистров на данный момент используются в программе. Это также позволяет транслятору выдавать предупреждения/напоминания в случае, если идентификатор был объявлен, но не используется в программе. С другой стороны, предоставляемые вместе с

транслятором заголовочные файлы содержат список объявлений для конкретной модели микроконтроллера, что позволяет избежать трудно отлавливаемых ошибок программиста, вызванных, например, неправильным написанием идентификатора вектора прерывания. Из практики известны случаи, когда компилятор avr-g++ успешно осуществлял сборку программы с ошибочным написанием идентификатора вектора прерывания и не выдавал при этом сообщений об ошибке, что в итоге проявлялось в виде некорректной работы программы.

Поскольку служебные объявления векторов, регистров и битов служат только для указания компилятору об их использовании, в результирующем коде программы на C они никак не проявляются:

`vector` *vec_id*`;` $\longmapsto \varepsilon$

`register` *reg_id*`;` $\longmapsto \varepsilon$

`bit` *bit_id*`;` $\longmapsto \varepsilon$, где ε обозначает пустую строку.

Объявление гиперпроцесса задает идентификатор вектора прерывания, используемый при описании функции-обработчика прерывания, а также идентификаторы регистра и бита, отвечающих за разрешение/запрет этого прерывания, которые используются в операциях запуска/остановки гиперпроцесса. В контексте описываемых правил трансляции описание гиперпроцесса транслируется в набор соответствующих макроопределений вектора, регистра и бита:

```
hyperprocess h_id_i {
    vector = vec_id_i;
    register = reg_id_i;
    bit = bit_id_i;
}
```

$\longmapsto$

```
#define vec_i vec_id_i;
#define reg_i reg_id_i;
#define bit_i bit_id_i;
```

Совокупность этих макроопределений по всей программе составляет элемент dec_{ints} в правиле трансляции программы.

4.1.3. Объявления переменных, констант и функций

Синтаксис объявления и определения переменных и констант языка IndustrialC является строгим подмножеством соответствующего синтаксиса языка C. Семантика объявлений, определений и типов данных для этих двух языков также в целом совпадает. Существенно различие существует между переменными, объявленными/определенными в программе/внутри процессов, и переменными, объявленными/определенными внутри состояний процессов и входящих в их состав операций.

Глобальные объявления и определения переменных в программе на IndustrialC, а также объявления и определения переменных, расположенные внутри определений процессов, но вне определений состояний транслируются в глобальные объявления и определения в результирующей программе и входят в состав элемента dec_{cv}. Таким же образом транслируются все определения констант, в том числе встречающиеся внутри состояний и операций в их составе. Следует отметить, что существенной разницы между переменными и константами при трансляции нет. Тип переменной или константы задается набором ключевых слов, допустимых в этом контексте синтаксисом языка, и транслируется без изменений. Различие между переменной и константой заключается исключительно в наличии ключевого слова «const» в этом списке. Объявления и определения массивов в данном описании приравниваются к определениям и объявлениям переменных.

Объявления и определения переменных, расположенные в исходном коде внутри состояний и входящих их состав операций, имеют семантику локальных переменных функций языка C, то есть хранятся на стеке. При трансляции такие объявления/определения переносятся в результирующий код наравне с операциями и в места, соответствующие их расположению относительно других операций в исходном коде. В случае процессов основного цикла программы, эти переменные оказываются внутри определения функции `main()`, а в случае процессов, активируемых

прерываниями, такие переменные попадают в процедуры обработки прерываний, также являющиеся функциями с точки зрения языка C.

Для разрешения конфликтов имен, часть идентификаторов в программе на IndustrialC – имена процессов, состояний, переменных и констант – заменяются на уникальные. В текущей реализации транслятора для этого используются префиксы областей видимости, однако в данном описании конкретный механизм замены не важен.

Определения функций транслируются без каких-либо изменений, причем идентификаторы имен функций также переносятся в результирующий код без добавления каких-либо префиксов. Это обусловлено тем, что функции определяются в глобальной области видимости, в отличие от переменных и констант, которые могут иметь сложные вложенные области видимости.

Пусть t – тип данных, составленный набором ключевых слов, v – имя (идентификатор) переменой в исходной программе и пусть $v \longmapsto v'$, аналогично, c – имя константы и $c \longmapsto c'$, *initializer* – инициализатор, то есть выражение, либо инициализатор массива. Для определений констант, а также объявлений и определений переменных, расположенных в исходном коде вне состояний:

$t\ v; \longmapsto t\ v'; \in dec_{cv}$

$t\ v = initializer; \longmapsto t\ v' = initializer; \in dec_{cv}$

$t\ c = initializer; \longmapsto t\ c' = initializer; \in dec_{cv}$

Для объявлений и определений переменных внутри состояний:

$t\ v; \longmapsto t\ v';$

$t\ v = initializer; \longmapsto t\ v' = initializer;$

4.1.4. Выражения

Выражения языка IndustrialC являются подмножеством выражений языка C за исключением предикатов проверки активности процессов, для которых определяются следующие правила трансляции:

p_j^i `active` $\mapsto$ `(`cs_j^i `!=` s_{stop}`)`

p_j^i `inactive` $\mapsto$ `(`cs_j^i `==` s_{stop}`)`

Все остальные выражения в исходном коде на IndustrialC транслируются в результирующий код на C без изменений, с учетом правила подстановки, описываемого далее.

4.1.5. Операции

Операции языка IndustrialC, аналогично выражениям, представлены подмножеством операций языка C, транслируемых без изменений, и специализированными операциями управления процессами и гиперпроцессами, для которых определяются следующие правила трансляции:

`set state` $s_k^{i,j}$`;` $\mapsto$ `{`cs_j^i `=` $s_k^{i,j}$`;` t_j^i `= 0;}`

`start process` p_q^i`;` $\mapsto$ `{`cs_q^i `=` $s_1^{i,q}$`;` t_q^i `= 0;}`

`stop process` p_q^i`;` $\mapsto$ `{`cs_q^i `=` s_{stop}`;}`

`stop process ;` $\mapsto$ `{`cs_j^i `=` s_{stop}`;}`

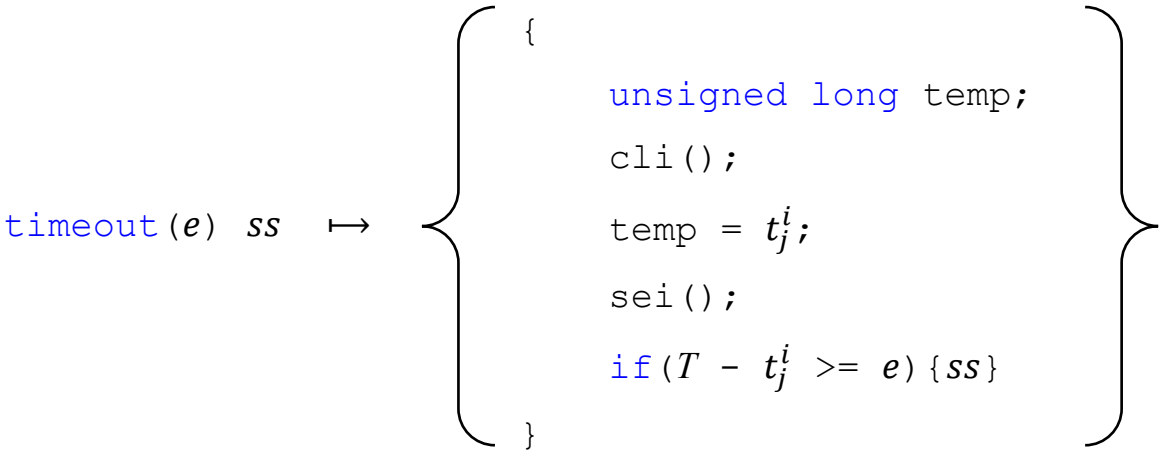
`start hyperprocess` h_id_q`;` $\mapsto$ `{`reg_q `|= (1<<`bit_q`);}`

`stop hyperprocess` h_id_q`;` $\mapsto$ `{`reg_q `&= ~(1<<`bit_q`);}`

`atomic` ss $\mapsto$ `{cli(); `ss `sei();}`

`timeout(`e`)` ss $\mapsto$ `if(`T `-` t_j^i `>=` e`){`ss`}`

По результатам семантического анализа, проводимого при трансляции, эти операции (кроме atomic) могут быть окружены оператором atomic для обеспечения синхронизации между гиперпроцессами. Для операции обработки тайм-аута синхронизация в этом случае специальным образом:



4.1.6. Правило подстановки

Поскольку отношение ⟼ задано на объединенном множестве С конструкций языков IndustrialC и С, которые, более того, изначально пересекаются, промежуточные результаты трансляции состоят из вложенных друг в друга конструкций обоих языков. По этой причине формальное задание правил трансляции требует уточнения механизма подстановки вложенных конструкций. Для этого воспользуемся терминологией, применяемой в системах переписывания термов [58].

Пусть c, c_1, $c_1' \in C$ – некоторые смешанные конструкции языков IndustrialC и С, и конструкция c имеет включение конструкции c_1. Обозначим $c[c_1]$ конструкцию c с выделенным включением c_1. $c[\ ]$ обозначает контекст – конструкцию c, в которой на месте выбранного включения c_1 находится пустая конструкция □. Правило подстановки вложенных конструкций заключается в том, что контекст при подстановке сохраняется. То есть, если $c_1 \longmapsto c_1'$, то $c[c_1] \longmapsto c[c_1']$. Порядок подстановки при этом задается в направлении от начала к концу кода программы.

Поясним это правило на примере: пусть $c \in C$, и c имеет вид:

$$\texttt{if (}\ p_j^i\ \texttt{active) stop process}\ p_j^i;$$

Эта конструкция содержит конструкции языка IndustrialC – выражение проверки активности процесса c_1, имеющее вид «p_j^i active» и операцию c_2

остановки процесса «stop process p_j^i;». Из ранее определенных правил трансляции известно, что $c_1 \longmapsto c_1' =$ «$(cs_j^i$!= $s_{stop})$» и $c_2 \longmapsto c_2' =$ «$\{cs_j^i = s_{stop}\}$».

Рассмотрим включение $c[c_1]$ конструкции c_1 в c. Контекст при этом имеет вид:

$$c[\,] = \text{« if (□) stop process } p_j^i\text{; »}$$

Используя правило подстановки, получаем новую конструкцию c':

$$c[c_1] \longmapsto c[c_1'] = c' = \text{« if } ((cs_j^i \; != \; s_{stop})) \text{ stop process } p_j^i\text{; ».}$$

Аналогично рассматриваем включение $c'[c_2]$, контекст в этом случае:

$$c'[\,] = \text{« if } ((cs_j^i \; != \; s_{stop})) \; \square \text{ ».}$$

После применения правила подстановки получаем конструкцию c'':

$$c'[c_2] \longmapsto \; c'[c_2'] = c'' = \text{« if } ((cs_j^i \; != \; s_{stop})) \; \{cs_j^i = s_{stop}\} \text{ ».}$$

Полученная конструкция $c'' \in C_c$ не содержит элементов языка IndustrialC, и, соответственно, для нее отношение $\longmapsto$ не выполняется. Эта конструкция без дальнейших изменений попадет в результирующий код С-программы.

4.1.7. Вставки С-кода

Синтаксис IndustrialC предполагает возможность размещения вставок на С в трех вариантах:

- В глобальном коде программы – вне определений процессов.
- Внутри блочных операций «{ }» - соответственно внутри кода состояний, функций и внутри операций, определяемых в состояниях и функций.
- Внутри выражений в любом месте исходной программы, предполагающем наличие выражений.

Во всех трех случаях код внутри вставок на С не анализируется и переносится в результирующий С-код без изменений. Исключение составляют включения IndustrialC-переменных и констант, обозначенные в С-вставках знаками «$». Для этих включений IndustrialC-идентификаторы заменяются на

соответствующие им идентификаторы в C-коде согласно определенному ранее правилу подстановки.

4.1.8. Платформенно-зависимые определения

Функции `init_ts()`, `sei()`, `cli()` и `get_time_ms()` реализуются отдельно для каждой модели микроконтроллера, их реализация размещается в соответствующих заголовочных файлах. Функция `get_time_ms()` возвращает текущее значение счетчика времени в миллисекундах. Функция `init_ts()` содержит платформенно-зависимый код, необходимый для инициализации периферии, обеспечивающей работу службы времени. Макроопределение `ISR()` также реализуется в зависимости от используемой платформы и обозначает процедуру обработки прерывания. Функция `cli()` используется в критических секциях для запрета выполнения всех прерываний, функция `sei()`, соответственно, разрешает исполнение прерываний в конце критической секции.

В случае микроконтроллеров семейства AVR, функции `sei()`, `cli()` и макроопределение `ISR()` предопределены. Функции `init_ts()` и `get_time_ms()` определяются в предоставляемых с транслятором заголовочных файлах и могут быть переопределены при необходимости. В случаях реализации службы времени с двойным счетчиком также может присутствовать функция `get_time_us()`, возвращающая текущее значение времени в микросекундах.

4.2. Транслятор IndustrialC-программ в язык C

К транслятору предъявлялись следующие требования:

- открытость и бесплатность используемых средств
- генерация результирующего кода на языке C
- статический анализ исходного кода
- выдача диагностических сообщений

- работа с текстовым представлением исходного кода
- поддержка разработанной грамматики IndustrialC

Исходя из этих требований были рассмотрены возможные варианты подходов к реализации транслятора industrialC в эквивалентный код на языке C:

1. Реализация лексического и синтаксического анализаторов и кодогенерации «вручную» на языках общего назначения;
2. Реализация языка в виде расширения языков Си/Си++, с использованием средства автоматизации процесса сборки Apache Ant [59] для статического анализа кода;
3. Использование средств фреймворка Xtext [60,61];
4. Использование парсер-генератора ANTLR [62];
5. Создание языка в среде MPS (Meta-Programming System) [63,64] на основе расширения mbeddr [65,66], содержащего реализацию языка Си в MPS.
6. Использование средств Flex/Bison [67,68] для реализации лексического и синтаксического анализаторов;

Реализация всего транслятора «вручную» на языках общего назначения C/C++ дает наибольшие возможности для выбора архитектуры и алгоритмов транслятора. При этом отсутствуют ограничения на класс транслируемых грамматик и статический анализ исходного кода. При этом такой подход предполагает значительные временные затраты и является наиболее трудоемким из рассмотренных.

Использование средств Apache Ant позволяет расширить синтаксис языков C/C++ специализированными конструкциями языка IndustrialC. При этом трансляция в язык C не осуществляется, а IndustrialC фактически становится фреймворком/надстройкой над языком C++. Такой подход наименее трудоемок, однако дает возможность только для динамического анализа программы. Реализация статических проверок при таком подходе оказывается невозможной.

Фреймворк разработки специализированных языков программирования Xtext предоставляет возможность генерации парсеров и дополнительно предоставляет готовую среду разработки. При этом Xtext предполагает генерацию результирующего целевого кода только на языке Java и не дает возможности трансляции в язык C, необходимой для дальнейшего использования полученного кода на микроконтроллерах.

Парсер-генератор ANTLR предоставляет значительно более широкие возможности по сравнению с Xtext, поддерживает большой класс грамматик LL(*) [69], и не ограничивает статический анализ исходного кода. При этом высокий порог вхождения и строгая ориентированность на разработку на языке Java обуславливают высокую трудоемкость использования этого подхода для разработки транслятора IndustrialC.

Версия языка C, реализованная в mbeddr, несовместима с программированием микроконтроллеров, что видно из описания проекта mbeddr for Arduino [70]. При этом специфика среды MPS, исключающая рассмотрение кода программ в виде текста может значительно затруднить разработку программ, а общая сложность системы и недостаток документации делают этот подход наиболее трудоемким из рассмотренных (Таблица 1).

Парсер-генератор GNU Bison и сопутствующий ему лексический анализатор Flex являются современной модификацией классических средств разработки трансляторов и компиляторов Lex/Yacc [71]. Средства имеют реализации на множестве распространенных языков программирования и прежде всего ориентированы на языки C/C++. Сложность разбираемых грамматик изначально ограничивается возможностью генерации только LALR(1)-парсеров, однако это ограничение легко обходится дополнительными средствами за счет гибкости используемого механизма привязки произвольных действий к правилам грамматики.

В результате анализа для реализации транслятора был выбран подход, предполагающий реализацию лексического и синтаксического анализаторов

средствами Flex/Bison и реализацию семантического анализатора и кодогенератора «вручную» на языке C++.

Подход	**Стат. анализ**	**Грамматики**	**Работа с тестовым представлением**	**Генерация С кода**	**IDE**	**Трудоемкость**
Реализация «вручную»	+	любые	+	+	–	Высокая
Расширение C++ с помощью Ant	-	-	+	+	–	Низкая
Xtext	+	LL(*)	+	-	+	Низкая
ANTLR	+	LL(*)	+	+	+	Высокая
MPS/mbeddr	+	любые	-	-	+	Высокая
Flex/Bison	+	LARL(1) IELR(1) GLR LR(1)	+	+	–	средняя

Транслятор IndustrialC был реализован на языке C++ с использованием лексического и синтаксического анализаторов, генерируемых при помощи средств GNU Flex/Bison. Транслятор оформлен в виде приложения командной строки, на вход транслятор принимает набор опций, включая имена входного и выходного файлов.

4.2.1. Поддержка директив препроцессора

Синтаксис IndustrialC не предполагает механизмов разбиения исходного кода программы на несколько файлов, при этом такая возможность фактически является обязательным условием при разработке сложных программ. Эта проблема была решена при помощи интеграции с C-препроцессором используемого компилятора.

В программах на IndustrialC могут использоваться все стандартные директивы C-препроцессора. Перед трансляцией программы выполняется обработка этих директив компилятором avr-g++ с опцией –E. При этом компиляция останавливается после прохода препроцессора. В результате этой процедуры получается один общий файл, не содержащий директив #include и #define, с расставленными в нем маркерами строк вида:

<номер> "<название файла>" <номер строки>

Эти строки распознаются транслятором IndustrialC на уровне лексического анализа и используются для отслеживания расположения конструкций языка в исходных файлах. Для каждого узла дерева синтаксического разбора сохраняется информация о названии исходного файла и номере строки, где находилась соответствующая конструкция языка. Эта информация в дальнейшем используется транслятором при выдаче диагностических сообщений.

4.2.2. Общая схема трансляции

Трансляция осуществляется в три этапа:

- Лексический и синтаксический анализ
- Семантический анализ
- Кодогенерация

На этапе лексического и синтаксического анализа происходит разбор программы средствами Flex/Bison в соответствии с правилами грамматики языка. В ходе разбора в оперативной памяти строится модель управляющей системы, соответствующая по структуре разработанному формализму, описанному в Главе 2. Синтаксический разбор производится снизу-вверх по глубине синтаксического дерева, то есть первыми разбираются терминальные символы и содержащие их нетерминальные символы. Общая структура программы разбирается в последнюю очередь. Построение модели осуществляется при помощи действий, представляющих собой участки C++ кода, ассоциированные с правилами грамматики языка. Лексический анализ

производится одновременно с синтаксическим по ходу разбора кода программы.

На этапе синтаксического разбора также осуществляется проверка соответствия областей видимости и для элементов модели собираются данные, впоследствии используемые при семантическом анализе и кодогенерации:

- данные о расположении конструкции в исходной программе – имя файла и номер строки;
- имя функции/процесса/состояния;
- будут ли операция или выражение размещены в процедуре обработки прерывания.

Для хранения этих данных и дерева областей видимости используется механизм контекста – глобальный динамически обновляемый объект, видимый из всех действий, ассоциированных с правилами трансляции, и хранящий данные об окружении разбираемого в данный момент участка кода. Также составляется список элементов модели, для которых необходимо провести семантические проверки.

4.2.3. Семантический анализ

На этапе семантического анализа проводится осуществление проверок из списка, составленного в ходе синтаксического разбора и преобразование модели перед кодогенерацией:

- проверка определения гиперпроцессов, процессов и состояний для операций смены состояния, управления процессами и гиперпроцессами и выражений проверки активности процессов;
- проверка использования переменных в процедурах обработки прерываний и выдача сообщений о необходимости добавления квалификатора volatile;

- проверка возможности использования переменных одновременно фоновым гиперпроцессом и гиперпроцессами прерываний и выдача сообщений о необходимости использования критических секций;

- проверка необходимости и автоматическая расстановка критических секций вокруг кода служебных операций смены состояния, управления процессами, проверки активности процессов, операций обработки и сброса тайм-аутов;

- выделение и перенос кода операций обработки тайм-аутов процессов, активируемых прерываниями в фоновый гиперпроцесса.

Критерии выявления необходимости использования критических секций для служебных операций зависят от целевой платформы и варианта реализации. В текущей реализации для микроконтроллеров семейства AVR учитывается, что:

- запрещены вложенные прерывания, то есть код процедур обработки прерываний атомарен и не требует критических секций;

- в 8-битной архитектуре операции с переменными размером 8 бит атомарны;

- переменные cs_j^i, хранящие текущие состояния процессов имеют 8-битное представление, что накладывает ограничение на максимальное количество состояний для процесса, включая - 256, включая состояние останова;

- переменные t_j^i имеют 32-битный тип unsigned long и асинхронный доступ к ним может привести к некорректному значению;

- операции обработки тайм-аутов процессов, активируемых прерываниями выполняются в основном цикле программы.

С учетом этих особенностей, используются следующие правила обеспечения синхронизации между гиперпроцессами:

- операции останова и проверки активности процессов используют только переменные cs_j^i, доступ к которым атомарен. Соответственно эти операции не требуют критических секций;

- служебные операции, исполняемые внутри процедур обработки прерываний не требуют критических секций.

Для операций, исполняемых в основном коде программы, включая проверки событий тайм-аутов и операции обработки тайм-аутов для процессов, активируемых прерываниями:

- для операций set state, находящейся в состоянии фонового процесса p_q требуется критическая секция, если над этим процессом осуществляются операции start process или stop process из кода, исполняемого в процедурах обработки прерываний. При этом недостаточно только обеспечить атомарность пары (cs_j^i, t_j^i) процесса. Рассмотрим ситуацию: во время исполнения кода состояния фонового p_q процесса происходит прерывание, в котором используется инструкция stop process p_q; После возврата из прерывания сразу же исполняется инструкция set state, состояние процесса изменяется, и он не останавливается. Таким образом, в этом случае инструкции смены состояния необходимо также предварять проверкой, что процесс все еще находится в состоянии, в котором эта инструкция описана;

- для операции start process действует то же правило, что и для set state, за исключением того, что доступ из процедуры обработки прерываний должен осуществляться не к процессу, в котором инструкция исполняется, а к процессу, над состоянием которого производится изменение;

- для выражения проверки тайм-аута и для операции сброса тайм-аута требуется критическая секция, если над этим процессом выполняются операции start process или stop process из процедур обработки прерываний.

Описанные расстановки критических секций и дополнительных проверок осуществляются над хранящимся в памяти представлением управляющей программы до начала генерации результирующего кода.

4.2.4. Кодогенерация

Генерация результирующей программы осуществляется в соответствии с описанными правилами трансляции после этапа семантического анализа. В случае, если на предыдущих этапах трансляции были выявлены критические ошибки, генерация C-программы не производится.

Во время кодогенерации транслятор расставляет в результирующей программе на C маркеры файлов и строк, которые распознаются препроцессором компилятора при сборке программы, и позволяют компилятору C корректно локализовать участки кода при выдаче своих диагностических сообщений. Это в частности позволяет эффективно обнаруживать ошибки во вставках кода на C, которые не анализируются транслятором IndustrialC. Кроме того, использование этого механизма позволило исключить анализ многих синтаксических ошибок при трансляции. Поскольку синтаксис и семантика многих операций в языках IndustrialC и C совпадают, при трансляции может осуществляться только проверка корректности специфических конструкций IndustrialC. Обнаружение остальных синтаксических и семантических ошибок, общих для обоих языков при этом осуществляется компилятором C. В список таких ошибок входят несоответствие типов или lvalue/rvalue в выражениях, некорректность инициализации массивов, вызов необъявленных или неопределенных функций и несоответствие количества аргументов при вызове функции.

4.3. Интегрированная среда разработки

Требования, предъявляемые к среде разработки:

- открытость и бесплатность используемых средств
- подсветка синтаксиса в исходном коде
- трансляция, сборка и загрузка программ из IDE
- отображение диагностических сообщений транслятора
- возможность организации исходных файлов в проекты

В качестве основы для среды разработки был выбран открытый редактор Notepad++ [72]. Редактор имеет встроенные возможности для произвольного задания подсветки синтаксиса и форматирования кода, а также функции для оформления набора исходных файлов в виде проектов. Функциональность редактора может быть значительно расширена за счет широкого выбора доступных плагинов и открытого интерфейса для разработки новых расширений.

В ходе работы был разработан плагин для Notepad++, осуществляющий следующие функции:

- сборка и загрузка программы из среды разработки
- меню настройки модели процессора, программатора, способа загрузки с опцией сохранения файла программы на С
- хранение последних установленных настроек в конфигурационном файле
- отображение окна выдачи диагностических сообщений
- автоматическое сохранение исходных файлов перед сборкой программы

Плагин оформляется в виде .dll-библиотеки и устанавливается путем размещения этого файла в соответствующей папке Notepad++.

В дополнение к плагину созданы xml-файл с правилами подсветки синтаксиса языка IndustrialC и набор заголовочных файлов, содержащих служебные объявления и реализации функций службы времени для наиболее часто используемых моделей микроконтроллеров.

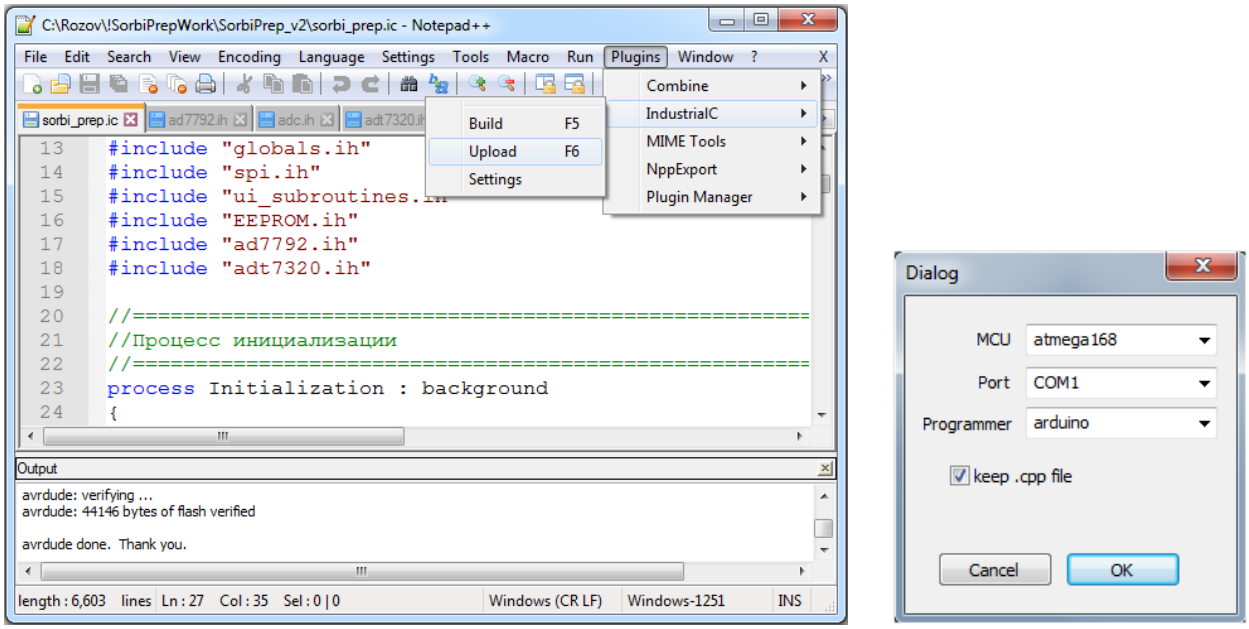


Рис. 3. Интегрированная среда разработки на базе Notepad++

Основные выводы главы

В главе представлены разработанные формальные правила трансляции программ на языке IndustrialC в эквивалентные программы на языке C, принципы работы транслятора языка и методы реализации транслятора и интегрированной среды разработки.

Правила трансляции задаются для 8-битных микроконтроллеров семейства AVR. По причине общей высокой зависимости программ микроконтроллера от аппаратной платформы, сохранение семантики конструкций языка в реализации для других семейств микроконтроллеров может потребовать изменения правил трансляции. При описании правил трансляции используются понятийный аппарат формализма описания ПО МК, заданный в Главе 2. Описанные правила используются в реализованном трансляторе языка, а также задают трансляционную семантику языка, которая в совокупности с формальными семантиками языка C может быть в дальнейшем использована для статической верификации программ микроконтроллеров.

Транслятор языка реализован с использованием генераторов лексического и синтаксического анализаторов Flex/Bison, что значительно

упрощает дальнейшие модификацию и развитие языка. Для обеспечения поддержки директив препроцессора разработан подход с предварительной обработкой программы С-препроцессором и анализом оставляемых им меток. Значительное пересечение по синтаксису и семантике с языком С позволило значительно сократить объем анализа кода, производимый транслятором. Транслятор обрабатывает только специфические для IndustrialC ошибки, при этом остальные ошибки обрабатываются С-компилятором в результирующем коде. Корректность выдачи диагностических сообщений С-компилятором обеспечивается путем расстановки транслятором меток препроцессора в С-коде. Разработанные алгоритмы и критерии семантического анализа кода обеспечивают возможность автоматической расстановки критических секций на этапе трансляции и выдачи предупреждающих сообщений.

Интегрированная среда разработки на базе редактора Notepad++ предоставляет удобный интерфейс для редактирования программ с подсветкой синтаксиса языка, возможностью организации файлов в проекты, трансляции, сборки и загрузки программ на контроллер непосредственно из редактора.

Глава 5. Практическая апробация

В данной главе описываются результаты применения полученных средств на практических задачах разработки встраиваемых систем и внедрения их в образовательный процесс. Разработанный формализм описания ПО микроконтроллер и его реализация на языке С использовались при разработке метеосервера Института автоматики и электрометрии СО РАН. Язык IndustrialC, транслятор и среда разработки были опробованы на задачах автоматизации установки термовакуумного напыления УВН-71 П3 и при разработке системы управления станции пробоподготовки SorbiPrep.

5.1. Апробация подхода к описанию ПО МК на языке С

Разработанный формализм описания ПО МК и реализация на языке С использовались при разработке метеосервера на базе открытой микроконтроллерной платформы. Многие научные и образовательные учреждения имеют собственные метеостанции и предоставляют возможность просмотра получаемых погодных данных в сети Интернет. Такие системы реализуются преимущественно с использованием ПК. В задаче предлагалось разработать систему предоставления погодных данных на микроконтроллерной платформе, используя данные о температуре и давлении с электронного табло ПК-Электроникс ОЗАК-280-Pt-485. Система периодически считывает информацию с табло через интерфейс RS-485, отрисовывает ее на баннере с логотипом Института автоматики и электрометрии СО РАН и предоставляет полученное изображение по запросу через Ethernet по протоколу HTTP. Также предусмотрена возможность связи с ПК для вывода отладочной информации и задания настроек метесервера. Система относится к классу так называемых систем интернета вещей (IoT) [73] – быстро развивающегося подмножества встраиваемых систем.

В качестве аппаратной платформы для метеосервера выбрана плата Seeeduino Mega v1.23 на базе микроконтроллера Atmel ATmega1280. Связь с

табло осуществляется через протокол UART с использованием микросхемы-преобразователя интерфейса MAX485. Ethernet-соединение обеспечивается модулем Wiznet Ethernet Shield на базе микросхемы W5100, работающей через шину SPI (рис. 2). Протоколы UART и SPI реализуются за счет встроенной аппаратной периферии микроконтроллера.

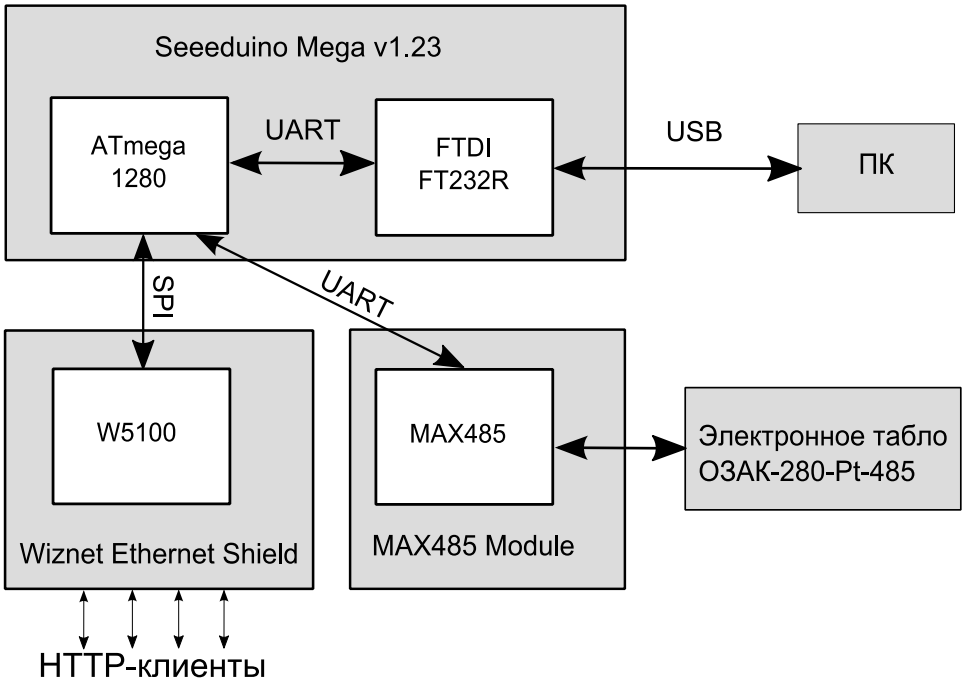


Рис. 4. Аппаратная архитектура метеосервера

ПО системы было описано в соответствии с разработанным формализмом ПО МК и реализовано на языке С способом, описанным в главе 4. На рис. 3 представлена архитектура управляющей программы.

Система описывается семью процессами, исполняющимися в четырех гиперпроцессах. Считывание данных с табло предполагает передачу команды, ожидание и прием ответа. Прием и передача сырых данных посредством встроенного модуля USART3 осуществляется двумя процессами – процесс приема реализован в фоновом гиперпроцессе, процесс передачи исполняется в процедуре обработки прерывания Data Register Empty контроллера USART3. Оба процесса помимо побайтной передачи пакетов данных реализуют алгоритм Byte Stuffing, предусмотренный протоколом связи электронного табло.

Процесс работы с RS-485 периодически считывает текущие значения температуры и давления с табло, используя процессы приема и передачи через USART3, и производит их отрисовку на хранящемся в памяти изображении.

Процесс работы с Ethernet ожидает и обрабатывает запросы клиентов: при первом запросе отсылается код на языке JavaScript, при исполнении этого кода на клиенте отправляется второй запрос, по которому пересылается изображение. Первый запрос необходим для окружения изображения HTML-кодом, выводящим значения температуры и давления при наведении мыши. На время отрисовки изображения процессом работы с RS-485 процесс работы с Ethernet останавливается.

Процесс работы с RS-485 и процесс работы с Ethernet используют процесс передачи отладочной информации на ПК, исполняющийся в процедуре обработки прерывания Data Register Empty аппаратного контроллера USART0. Следует отметить, что, хотя все эти процессы используют общие данные, это не влияет на поведение системы – все управляющее взаимодействие между процессами осуществляется командами запуска, останова процессов и проверки их состояния.

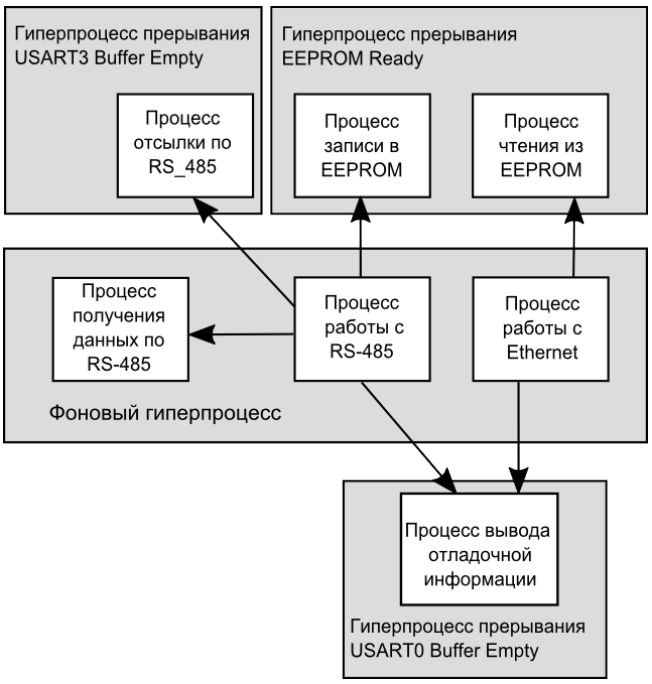


Рис. 5. Архитектура ПО метеосервера

Разработанная система продемонстрировала устойчивую работу в условиях стресс-тестирования Ethernet-запросами и на протяжении длительной эксплуатации в штатных условиях. Разработанный формализм обеспечивает однородное описание алгоритма и показал простоту расширяемости кода при последующем добавлении процессов работы с памятью EEPROM [15]. Разработанный метеосервер внедрен в Институте автоматики и электрометрии СО РАН и интегрирован с вебсайтом института (рис. 4).

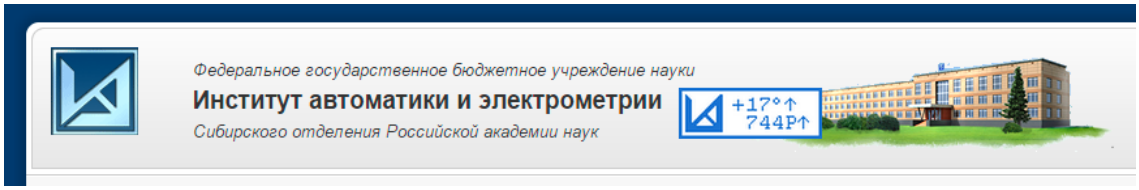


Рис. 6. Баннер с текущими температурой и давлением на сайте ИАиЭ СО РАН, формируемый созданным метеосервером

5.2. Автоматизация установки термовакуумного напыления

Язык IndustrialC и созданные инструментальные средства были опробованы на задаче автоматизации установки термовакуумного напыления УВН-71-П-3, используемой в Институте автоматики и электрометрии СО РАН для отработки технологий нанесения халькогенидных плёнок на оптические поверхности. Ранее работа на установке полностью осуществлялась в ручном режиме, что не позволяло обеспечить необходимую повторяемость результатов напыления (толщины пленки). Установка УВН-71П-3 позволяет наносить слои халькогенидов толщиной от 100 нм. Изображение установки и ее упрощенная схема представлены на рис 1.

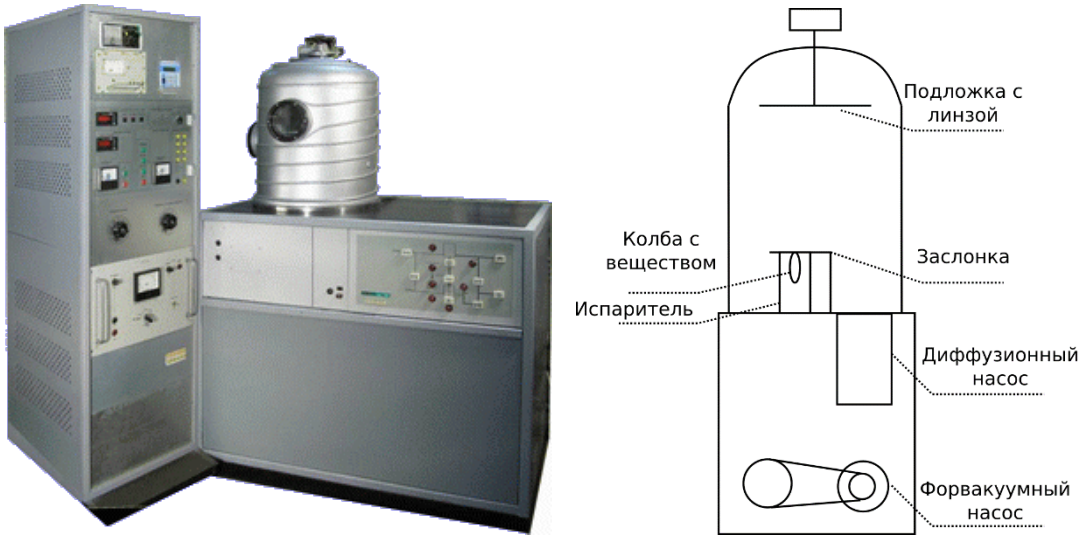


Рис. 7. Установка УВН-71-П-3 и ее упрощенная схема

Процесс напыления в ручном режиме состоит из следующих основных этапов:

1. Оптический элемент помещается на подложку с планетарным вращением, устанавливается трафарет напыления.
2. В испаритель помещается напыляемый материал, поток испаряемого вещества перекрывается заслонкой.
3. В камере с линзой с помощью форвакуумного насоса откачивается воздух.
4. Оператор вручную регулирует мощность, подаваемую на нагреватель, для установления и поддержания требуемой температуры. В зависимости от напыляемых материалов, температура варьируется от 1 °C до 1000 °C.
5. Оператор открывает заслонку, и пары напыляемого вещества оседают на поверхность оптического элемента по трафарету.
6. По завершении процесса напыления оператор закрывает заслонку.

Процесс напыления может занимать от 10 минут до нескольких часов, при этом допустимая абсолютная погрешность температуры составляет 5 градусов.

Система автоматизации представлена ПК оператора, центрального микроконтроллерного модуля, модуля термопары, модуля управления

мощностью и реле для управления заслонкой. Схема аппаратных составляющих системы представлена на рис 2. Центральный модуль выполнен на основе открытой микроконтроллерной платформы Seeeduino v2.21 ATmega168. Он отвечает за получение температуры с термопары, обмен данными с ПК по протоколу Modbus, управление мощностью испарителя и положением заслонки. Модуль управления заслонкой состоит из реле, управляемого сигналом 5 В. Модуль термопары включает преобразователь MAX6675 и обеспечивает компенсацию холодного спая и вывод 12-битного значения температуры через гальванически изолированный интерфейс по протоколу SPI. Модуль управления мощностью гальванически изолирует и преобразует входной ШИМ-сигнал с микроконтроллера в аналоговый сигнал для управления нагревателем испарителя установки.

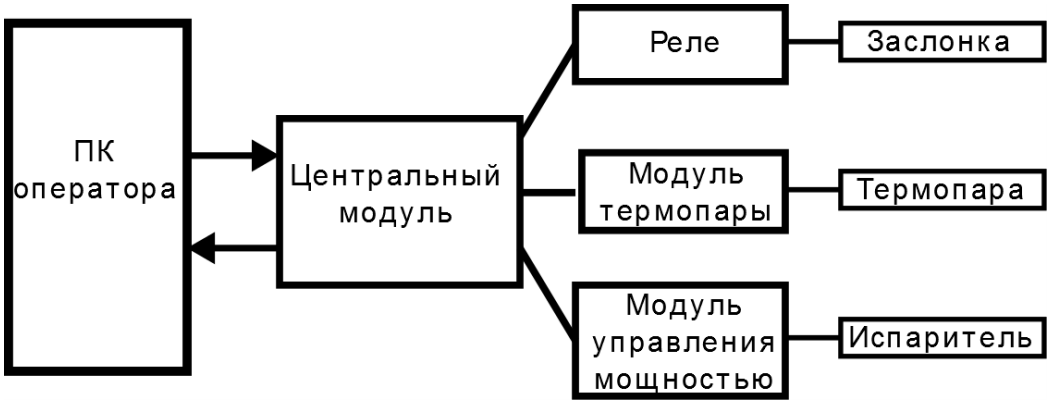


Рис. 8. Аппаратная архитектура системы автоматизации установки термовакуумного напыления

ПО микроконтроллера центрального модуля реализовано на разработанном языке IndustrialC с использованием созданных инструментальных средств и состоит из 4 гиперпроцессов и 7 процессов (Рис. 9).

Вырожденный процесс инициализации состоит из одного состояния, которое и исполняется один раз в начале программы. В нем осуществляются инициализация внутренних и внешних периферийных устройств, а также запуск процессов, участвующих в штатной работе системы.

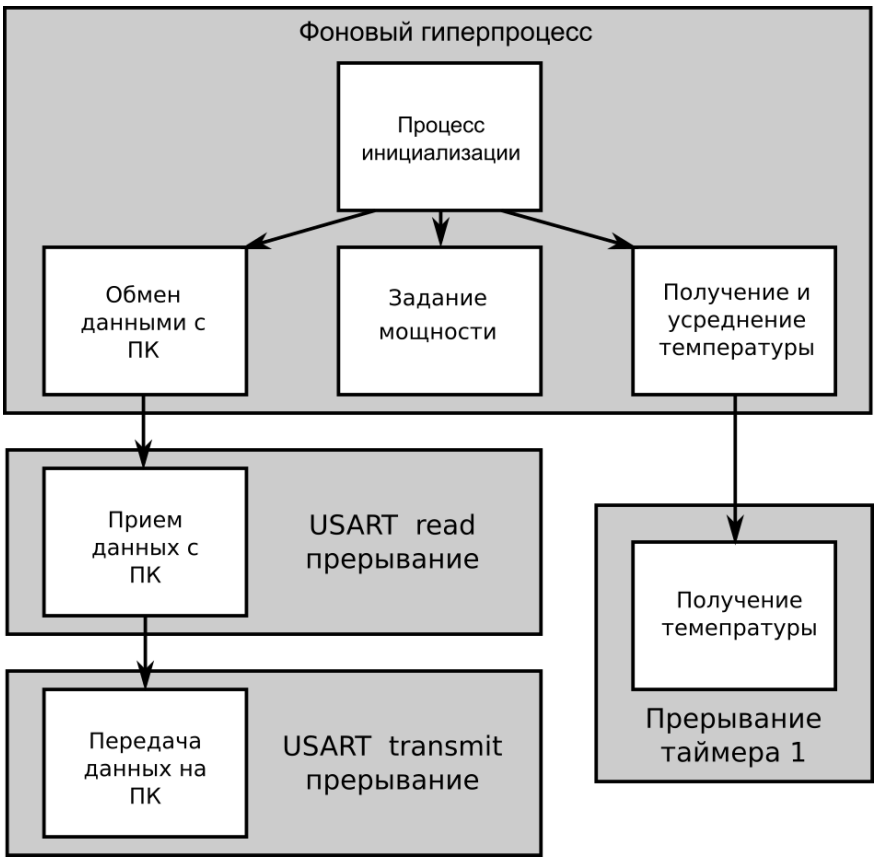


Рис. 9. Архитектура ПО микроконтроллера системы автоматизации установки термовакуумного напыления

Процесс обмена данными с ПК отвечает за обработку входящих запросов по протоколу Modbus, запуск процесса отправки данных и процесса приема данных. Во время обработки запросов происходит проверка соответствия данных протоколу Modbus, расширенному возможностью передачи чисел с плавающей точкой.

Получение температуры происходит через преобразователь MAX 6675 по протоколу SPI. Частота сигнала синхронизации равна 5.3 кГц. Процесс задания мощности состоит из трех состояний. В первом состоянии происходит проверка режима задания мощности, в зависимости от режима процесс переходит либо в состояние ручного задания мощности с интерфейса оператора, либо в режим автоматического регулирования температуры по алгоритму ПИД. В процессе работы система отслеживает и обрабатывает внештатные ситуации отключения термопары, потери связи с ПК и выхода из строя нагревателя.

5.3. Станция пробоподготовки SorbiPrep

Язык IndustrialC, транслятор языка и среда разработки были использованы при создании системы управления станцией пробоподготовки образцов SorbiPrep по заказу ЗАО «МЕТА». Станция применяется для дегазации образцов перед измерениями удельной площади поверхности и пористости в приборах линейки Sorbi.

Работа ПО станции пробоподготовки предполагает одновременное управление большим количеством процессов и обработку множества событий, включая события, критичные к времени реакции, независимую и непрерывную работу системы на протяжении длительного времени и необходимость отслеживания временных интервалов. При этом из соображений экономической целесообразности в качестве вычислительной платформы рассматривались 8-битные микроконтроллеры. Таким образом, разработка системы управления станции пробоподготовки SorbiPrep – типичная задача в области встраиваемых систем на микроконтроллерных платформах и полностью соответствует рассматриваемой области применения разработанных средств.

5.3.1. Основные параметры объекта управления и требования к ПО

Станция пробоподготовки представляет собой независимое устройство, применяемое для термотренировки/дегазации порошкообразных образцов для последующих измерений в приборах Sorbi. Станция предназначена для эксплуатации преимущественно в лабораторных условиях и используется совместно с Sorbi при контроле качества и определении характеристик материалов, используемых на промышленных предприятиях, а также при проведении научных исследований. Процесс дегазации предполагает постепенный нагрев образца и поддержание заданной температуры с попутной продувкой образца инертным газом. Предъявляются высокие требования к точности поддержания температуры на различных этапах процесса.

Станция имеет три независимых порта дегазации и пользовательский интерфейс, представленный текстовым дисплеем, клавиатурой, световыми и звуковым индикаторами. Каждый из трех портов дегазации включает измеритель, представленный К-термопарой и нагревательный элемент, рассчитанный на переменный ток с сетевым напряжением 220 В, 50 Гц. Мощность нагревателей – порядка 50 Вт. Порты также подключены к газовой магистрали с механическими игольчатыми клапанами, задающими поток инертного газа. Прибор не требует подключения к ПК или другим сторонним вычислительным средствам. При этом предусматривается UART-USB-канал для возможности вывода служебной информации на ПК при отладке и настройке станции.

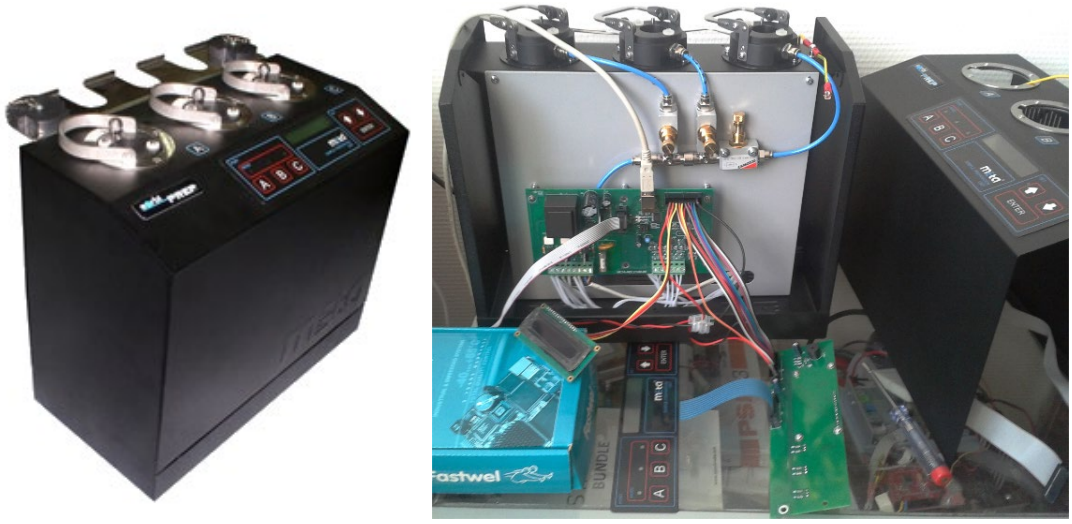


Рис. 10. Станция пробоподготовки SorbiPrep в собранном виде и в процессе отладки

В состав станции пробоподготовки входят:

- плата индикации;
- плата контроллера;
- 3 порта дегазации;

Плата индикации включает следующие устройства:

- Электромагнитный звуковой излучатель (зуммер);
- Символьный индикатор 16x2 символов с интерфейсом SPI;
- Три трехцветных светодиодных индикатора состояния портов;

- Клавиатура с 6 клавишами – три клавиши выбора порта дегазации, две клавиши выбора и изменения параметров и клавиша ввода.

На плате контроллера размещаются следующие устройства:

- Программируемый микроконтроллер ATmega128A;
- Трехканальный АЦП AD7792 с интерфейсом SPI;
- Датчик температуры холодного спая ADT7320 с интерфейсом SPI;
- Три оптоизолированный симисторных блока;
- Оптоизолированная схема генерации опорного сигнала сети;
- Интерфейсный преобразователь UART-USB FT232R;
- Стабилизированный блок питания 5 В.

Соответственно, определены следующие цифровые входные и выходные сигналы микроконтроллера.

Входные сигналы:

№	Наименование	Назначение
1	SPI_MISO	SPI, последовательный вывод на AD7792, ADT7320, символьный дисплей
2	MAINS_REF	Опорный сигнал сети 100 Гц
3	BUTTON_ A	Клавиша А клавиатуры
4	BUTTON_ B	Клавиша В клавиатуры
5	BUTTON_ C	Клавиша С клавиатуры
6	BUTTON_ UP	Клавиша «вверх» клавиатуры
7	BUTTON_DOWN	Клавиша «вниз» клавиатуры
8	BUTTON_ENTER	Клавиша «Enter» клавиатуры

Выходные сигналы:

№	Наименование	Назначение
1	BUZZER	Вывод ШИМ на звуковой излучатель
2	SPI_SCK	SPI, сигнал тактирования
3	SPI_MOSI	SPI, последовательный ввод с AD7792, ADT7320

4	AD7792_CS	SPI, сигнал выбора устройства AD7792
5	ADT7320_CS	SPI, сигнал выбора устройства ADT7320
6	DISPLAY_CS	SPI, сигнал выбора устройства символьного дисплея
7	HEATER_A	Вывод на блок управления нагревателем порта A
8	HEATER_B	Вывод на блок управления нагревателем порта B
9	HEATER_C	Вывод на блок управления нагревателем порта C
10	LED_A_RED	Красный светодиод порта A
11	LED_A_GRN	Зеленый светодиод порта A
12	LED_B_RED	Красный светодиод порта B
13	LED_B_GRN	Зеленый светодиод порта B
14	LED_C_RED	Красный светодиод порта C
15	LED_C_GRN	Зеленый светодиод порта C
16	USART_TX	Вывод через последовательный интерфейс UART

5.3.2. Требования к системе управления

Управляющая программа должна обеспечивать выполнение следующих функций:

1. Ввод с 6-клавишной клавиатуры и индикация на OLED-дисплее (16x2) параметров дегазации:

- названия порта дегазации
- температуры дегазации в диапазоне 40 ÷ 400 °C с шагом задания 5 °C отдельно для каждого порта дегазации;
- времени дегазации в диапазоне 10 ÷ 360 мин с шагом задания 5 мин отдельно для каждого порта дегазации;
- параметров регулирования – временная константа, пропорциональный, интегральный и дифференциальный коэффициенты – отдельно для каждого порта дегазации (в режиме настройки)

2. Запуск и остановка процесса дегазации по команде с клавиатуры.

3. По запуску процесса дегазации на порту осуществлять выход на заданную температуру, ее поддержание с точностью ±1°С и отслеживание времени дегазации с момента достижения 90% от заданной температуры.

4. Индикация текущего состояния портов дегазации на символьном дисплее в режиме просмотра параметров портов – времени с начала дегазации и текущей температуры.

5. Индикация состояния трех портов дегазации на светодиодных индикаторах – порт готов к работе/идет дегазация/аварийная ситуация.

6. Обнаружение аварийных ситуаций: обрыв измерителя температуры в порту, обрыв нагревателя, сопровождающееся звуковой и светодиодной индикацией порта, выдачей на индикатор диагностического сообщения с указанием порта и типа аварийной ситуации.

7. Автоматическая остановка процесса дегазации со звуковой индикацией по истечении заданного времени или в случае аварийной ситуации.

8. Сохранение в энергонезависимой памяти параметров регулирования и параметров дегазации для каждого порта после выключения станции.

5.3.3. Управляющее ПО станции пробоподготовки

Программа микроконтроллера реализована на разработанном языке IndustrialC с использованием созданной среды разработки на базе Notepad++. В соответствии с требованиями к системе, ПО выполняет следующие основные функции:

- Инициализация встроенных и внешних периферийных устройств в начале работы станции
- Обработка сигналов с клавиатуры
- Обеспечение функционирования пользовательского меню и меню настройки

- Выдача данных на средства индикации (символьный дисплей, светодиодные индикаторы и звуковой индикатор)
- Передача отладочной и настроечной информации на ПК
- Хранение и загрузка настроечных и технологических параметров в энергонезависимой памяти EEPROM
- Чтение и фильтрация значения температуры внутри станции
- Независимо для каждого из трех портов дегазации обеспечивается:
    - Чтение и фильтрация значения температуры в порту
    - Регулирование температуры
    - Отработка технологической программы
    - Управление нагревателем
    - Обнаружение неисправностей измерителя и нагревателя

Инициализация периферийных устройств осуществляется в фоновом процессе, который запускается первым в начале исполнения программы, запускает процессы, участвующие в штатной работе системы и затем останавливается. Процессом производится инициализация следующих внутренних и внешних периферийных устройств, используемых в программе:

Внутреннее периферийное устр-во	**Использование в системе**
Контроллер SPI	Взаимодействие с АЦП AD7792, датчиком температуры ADT7320 и символьным дисплеем
Контроллер USART	Передача отладочных и настроечных данных на ПК через интерфейс UART-USB
Контроллер внешнего прерывания	Тактирование процессов управления нагревателями по сигналу синхронизации сети

8-битный Таймер/Счетчик 0	Используется в реализации службы времени, обеспечивающей работу механизма тайм-аутов
8-битный Таймер/Счетчик 2	Вывод ШИМ-сигнала на электромагнитный звуковой индикатор
Контроллер EEPROM	Загрузка и сохранение настроечных и технологических параметров
Порт ввода/вывода A	Считывание состояния клавиатуры
Порт ввода/вывода B	Вывод управляющих сигналов на нагреватели. Также включает в себя используемый вывод ШИМ Таймера/Счетчика 2 и выводы аппаратного SPI-контроллера
Порт ввода/вывода C	Вывод сигналов на двухцветные светодиодные индикаторы
Порт ввода/вывода E	Включает в себя вывод, используемый внешним прерыванием для сигнала синхронизации сети
Порт ввода/вывода G	Вывод сигналов-селекторов шины SPI
Внешнее периферийное устройство	**Использование в системе**
ADT7320	Получение значения температуры внутри станции
AD7792	Получение значений напряжения термопар
Символьный индикатор	Отображение меню пользователя и настроечного меню

Процессы контроля клавиатуры выполняют следующие функции:

- программную обработку дребезга,
- контроль единичного нажатия,
- контроль двух степеней длительного зажатия для промотки,
- контроль одновременного длительного зажатия двух клавиш.

Меню пользователя и настроечное меню реализуются отдельными процессами, состояния в которых соответствуют страницам меню в режимах просмотра и редактирования параметров. Вывод текста на символьный индикатор реализуется подпрограммами, однако включение/выключение режима мигания всего экрана потребовало оформления отдельного процесса мигания, запускаемого на время редактирования параметров в меню. Для вывода одинарного, двойного и тройного звуковых сигналов реализован отдельный процесс, использующий Таймер/Счетчик 2 для аппаратной генерации прямоугольного сигнала.

Передача отладочной и настроечной информации на ПК реализуется двумя взаимодействующими процессами – процессом вывода строки и процессом вывода графиков. Процесс вывода строки активируется аппаратным прерыванием Data Register Empty встроенного USART-контроллера. После запуска процесс при каждой активации отправляет один байт из заданной строки. После отправки всей строки процесс останавливается и останавливает активирующий его гиперпроцесс. Такое поведение обусловлено особенностью данного аппаратного прерывания – оно генерируется постоянно, пока регистр данных USART пуст. Процесс вывода графиков формирует строки с отсчетами графиков уставки регулятора, температуры порта, температуры модели порта и составляющих регулятора. Для отправки данных на ПК процесс вывода графиков запускает процесс отправки строки и активирующий его гиперпроцесс, после чего дожидается остановки процесса отправки строки. Хотя эти два процесса пересекаются по данным и исполняются в разных гиперпроцессах, использование критических

секций в данном случае не требуется, поскольку одновременный доступ к данным исключен алгоритмом работы процесса вывода графиков.

Получение данных о температуре в порту включает в себя управление АЦП через интерфейс SPI и считывание текущего значения напряжения термопары, фильтрацию (усреднение) значения напряжения термопары и вычисление значения температуры в порту на основании отфильтрованных значений напряжения термопары с учетом калибровочных параметров и с компенсацией холодного спая на основе значения температуры внутри станции. Фильтрация получаемых с АЦП значений включает их усреднение, а также исключение из рассмотрения неправдоподобных значений – резких выбросов, вызванных шумом или нарушением целостности измерителя.

Технологическая программа включает этапы нагрева и поддержания температуры. На этапе нагрева производится линейная интерполяция уставки регулятора для обеспечения заданной скорости нагрева. На этапе поддержания температуры отслеживается время дегазации, заданное пользователем для конкретного типа образца.

Регулирование температуры осуществляется с использованием модифицированного ПИД-алгоритма, разработанного с учетом термодинамических особенностей портов дегазации станции. В дополнение к классической форме ПИД-регулятора добавлена базовая составляющая, обеспечивающая быстрый выход на заданную скорость нагрева. Для предотвращения перерегулирования при переходе между режимами нагрева и поддержания температуры, рост интегральной составляющей дополнительно ограничивается в начале нагрева и при достижении достаточной скорости роста температуры. Перерегулирование, обусловленное прогревом порта в режиме поддержания температуры, снижается за счет увеличения скорости уменьшения интегральной составляющей.

Управляющее воздействие регулятора U_i рассчитывается по формуле:

$$U_i = U_i^B + U_i^P + U_i^I + U_i^D .$$

Базовая составляющая U_i^B рассчитывается в зависимости от разности целевой температуры t^{deg} и температуры внутри станции t^{int}:

$$U_i^B = (t^{\deg} - t_i^{\text{int}}) \times K_{pow}.$$

Пропорциональная составляющая:

$U_i^P = e_i \times K_p$, где e_i – текущее рассогласование

Интегральная составляющая U_i^{Int} рассчитывается по формуле:

$$U_i^{Int} = \sum_{k=0}^{i} e_i^{inc} \times \Delta T, \quad e_i^{inc} = \begin{cases} 0, & T_i - T_{start} < T_{id} \\ 0, & \Delta t > \dfrac{dt}{K_{rate}} \\ e_i \times K_{dec}, & e_i < 0 \\ e_i & \textit{иначе} \end{cases}, \text{ где}$$

e_i – текущее рассогласование, $e_i = t^0_i - t_i$

t_i – текущая температура нагревателя

t^0_i – текущее значение уставки регулятора

$\Delta t = t_i - t_{i-1}$ – текущее приращение температуры

Принцип расчета по формуле проиллюстрирован на рис. 1.

Дифференциальная составляющая рассчитывается по формуле:

$$U_i^D = (e_i - e_{i-1}) \times K_d.$$

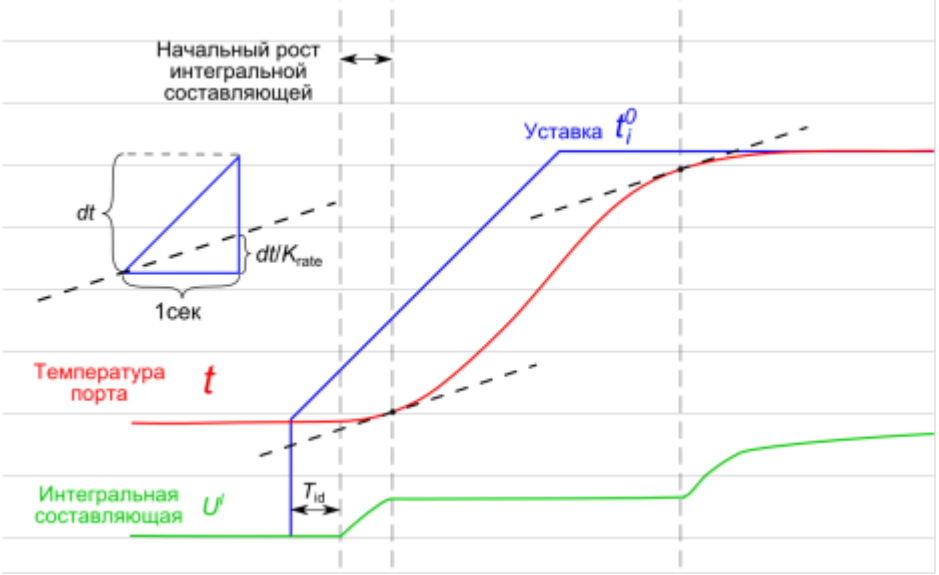


Рис. 11. Ограничение интегральной составляющей регулятора температуры в порту дегазации

Для управления нагревателями через симисторы используется алгоритм Брезенхема с накоплением ошибки, позволяющий задавать произвольные значения мощности нагревателя. Процессы управления нагревателями активируются аппаратным внешним прерыванием по восходящем фронту сигнала синхронизации сети (100 Гц), ширина импульса 760 мкс, центр импульса совпадает с моментом пересечения нуля синусоидальным сигналом сети 220 В. При каждой активации процесс либо не пропускает полуволну и увеличивает значение ошибки, либо пропускает полуволну путем подачи сигнала на симистор, в случае если значение ошибки превысило единицу. Использование аппаратного прерывания в данном случае оправдано необходимостью своевременной реакции на пересечение нуля напряжением сети. В случае, если управляющий сигнал будет подаваться с запозданием, открытие симистора может не произойти, что приведет к погрешности задания мощности нагревателя.

Программа обеспечивает обнаружение и обработку двух видов неисправностей в порту дегазации – обрыв измерителя (термопары) и обрыв, либо значительное изменение характеристик нагревателя. Контроль корректности работы измерителя осуществляется процессом фильтрации значений напряжения, а также процессом взаимодействия с внешним АЦП. Ситуация «обрыв нагревателя» устанавливается в случае отсутствия связи с АЦП или считывания неправдоподобных данных в течение определенного промежутка времени. Обнаружение неисправности нагревателя осуществляется по косвенным признакам. Для каждого порта в программе исполняется процесс, моделирующий его поведение. Эти процессы моделируют нагрев порта и охлаждение через окружающую среду по формуле:

$$dT = \frac{P \times dt}{C} - K \times (T - T_{\text{int}}) \times dt \text{, где}$$

dT - приращение температуры модели

P - текущая мощность нагревателя

dt - временной промежуток

C - теплоемкость модели

K - коэффициент теплоотдачи

T - текущая температура модели

T_{int} - текущая температура окружающей среды (внутри станции)

Задержка поступления тепла от нагревателя имитируется фильтрацией устанавливаемого значения мощности нагрева. Также учитывается нелинейность зависимости фактической мощности нагревателя от заданной, связанная с изменением сопротивления нагревательного элемента по мере роста температуры. Процессы, моделирующие порты дегазации используют те же текущие значения мощности, заданные регулятором, что и процессы управления реальными нагревателями. Внештатная ситуация «повреждение нагревателя» устанавливается в случае, если температура и направление изменения температуры реального порта дегазации значительно отклоняются от поведения модели в течение определенного времени.

В случае обнаружения внештатной ситуации в порту дегазации, все процессы управления этим портом останавливаются, нагреватель отключается, и на символьный дисплей выдается соответствующее неисправности диагностическое сообщение.

Следует отметить, что используемые программой внутренние периферийные устройства SPI и контроллер EEPROM предоставляют возможность работы через аппаратные прерывания. При этом работа с этими устройствами в программе производится без использования прерываний. Это обусловлено тем, что процессы, использующие эти устройства не критичны по времени реакции, а увеличение числа асинхронно исполняющихся процессов нежелательно, так как требует дополнительных накладных расходов на синхронизацию и повышает риск возникновения гонок.

Хотя используемая в разработанном языке кооперативная форма многозадачности препятствует возникновению гонок на уровне состояний процессов, гонки могут возникать на более высоком уровне. В ходе разработки

управляющего ПО станции пробоподготовки был выявлен пример такой ситуации. Рассмотрим для одного порта связанные с ним процессы:

- вычисления температуры в порту,
- отработки технологической программы,
- регулирования температуры в порту,
- передачи настроечного графика на ПК.

Эти процессы взаимосвязаны имеют разделяемые данные. При этом в их поведении есть общая черта: в штатном режиме работы, каждый из этих процессов чередует два состояния – вычисление и ожидание. В случае процесса вычисления температуры в порту, ожидание обусловлено необходимостью накопления достаточного количества отсчетов для фильтрации. Процессы интерполяции уставки регулятора и регулирования температуры исполняются в соответствии с заданным периодом квантования, а частота передачи точек настроечных графиков на ПК специально ограничивается для удобства восприятия. Хотя все эти процессы исполняются в одном фоновом гиперпроцессе с общим периодом активации, у каждого их них есть свой период работы на более высоком уровне, причем эти периоды обусловлены разными факторами и могут быть различны.

Это создает предпосылки для возникновения некорректного поведения системы. Например, в случае, если интерполяция уставки регулятора опережает вычисление значений температуры, периодически будет возникать ситуация, когда процесс регулирования будет использовать старое значение температуры и новое значение уставки, что приведет к скачкам пропорциональной составляющей регулятора и некорректному поведению алгоритма. Аналогично, различия в периоде исполнения процесса передачи данных на ПК и остальных перечисленных процессов могут приводить к некорректному отображению графиков с периодическим пропусканием отсчетов. Таким образом, обеспечение корректной работы системы потребовало дополнительной синхронизации между процессами, исполняемыми в фоновом гиперпроцессе. Вероятность возникновения

подобных ситуаций следует учитывать также при использовании методов статической балансировки вычислительной нагрузки [74]. В этом случае причины ошибок могут быть менее явными для программиста, поскольку различные периоды активации процессов определяются транслятором и не отражены в коде программы.

При разработке программы в нескольких случаях возникала необходимость организации передачи сообщений (событий) между двумя процессами, либо между одним и многими процессами. Например, в случае событий единичного нажатия и длительного зажатия клавиши каждое из этих событий генерируется одним процессом, обрабатывающим данную клавишу – процесс-производитель события. При этом такое событие может одновременно обрабатываться несколькими процессами – процессы-потребители. Для корректной работы системы необходимо, чтобы все процессы-потребители обработали событие ровно один раз. В общем случае для выполнения этого требования необходимо использовать отдельные переменные-флаги для каждой пары событие-потребитель, и процесс-производитель события должен одновременно модифицировать все эти переменные, что сопряжено с большими накладными расходами. Однако, в случае, когда все участвующие процессы активируются одним гиперпроцессом, например, фоновым, модель исполнения гиперпроцесса позволяет использовать для каждого такого события только одну переменную. При этом процесс-производитель события выставляет флаг события и переходит в состояние, в котором этот флаг сбрасывается. Таким образом, флаг остается выставленным в течение ровно одного полного цикла активации гиперпроцесса, и все активные процессы-потребители отреагируют на него ровно один раз, после чего процесс-производитель сбросит его при следующей активации. Такой механизм позволяет существенно сократит накладные расходы при передаче сообщений-событий между процессами.

Результирующая программа микроконтроллера включает 47 процессов и три гиперпроцесса, включая фоновый гиперпроцесс и два гиперпроцесса

прерываний – внешнего прерывания по переднему фронту сигнала синхронизации сети и прерывания Data Register Empty аппаратного контроллера USART. На рисунке представлена схема взаимодействия процессов ПО SorbiPrep. Объем используемого ПЗУ Flash – 50 КБ. Объем используемого ПЗУ EEPROM – 83Б. Объем постоянно используемого ОЗУ SRAM – 2.2 КБ.

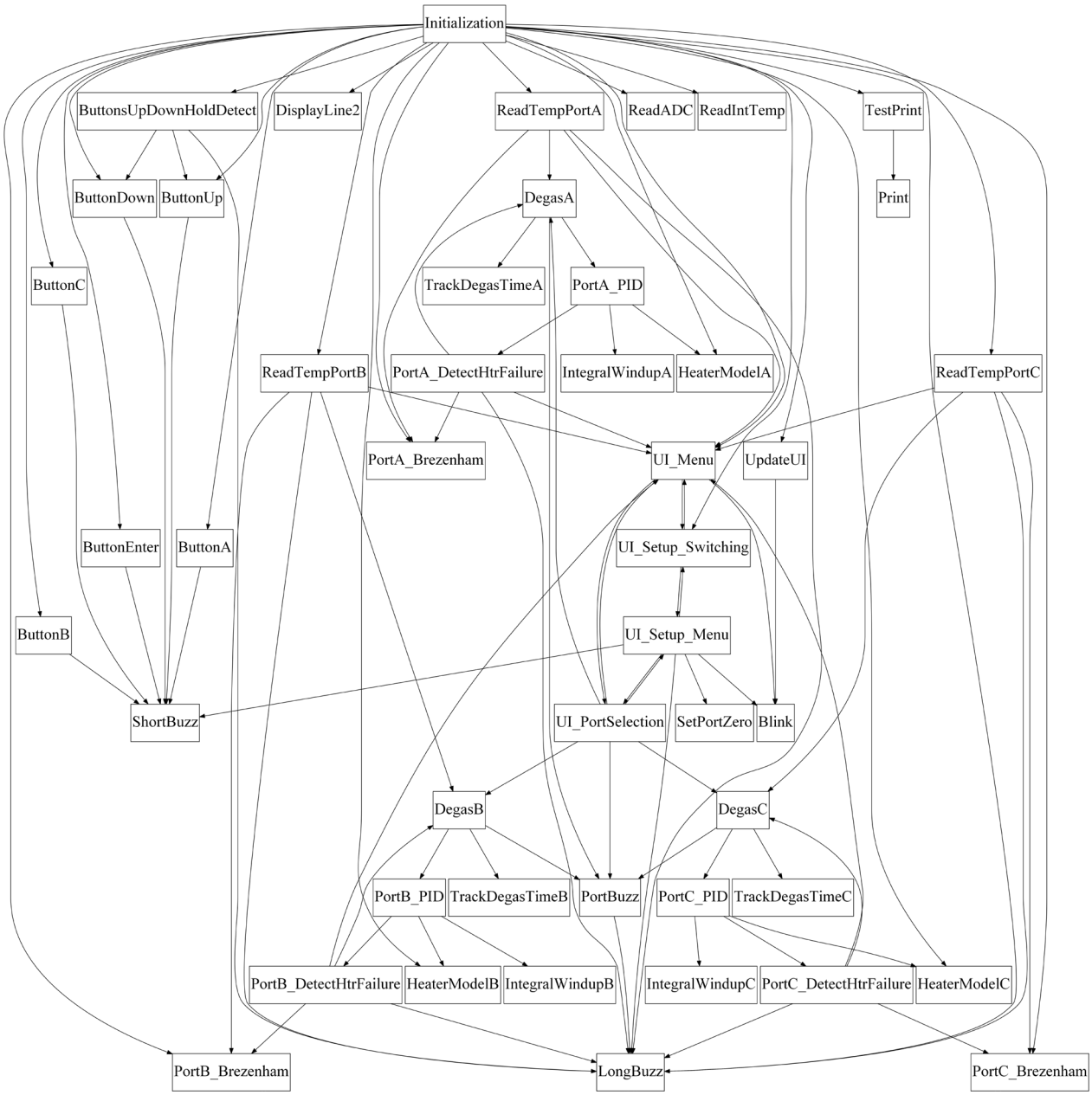


Рис. 12. Архитектура ПО станции пробоподготовки SorbiPrep. Стрелками обозначены операции запуска/остановки процессов

Основные выводы главы

Разработанный формализм описания ПО микроконтроллера, основанный на нем язык IndustrialC, его эквивалентное представление на языке C, а также созданные инструментальные средства – транслятор языка и среда разработки – были опробованы при решении задач разработки встраиваемых систем на микроконтроллерных платформах: разработка метеосервера Института автоматики и электрометрии СО РАН, автоматизация установки термовакуумного напыления УВН-71 П3 и разработка системы управления станции пробоподготовки SorbiPrep.

В результате апробации выявлено, что разработанный язык IndustrialC в совокупности с созданными инструментальными средствами позволяет эффективно описывать программное обеспечение микроконтроллеров во встраиваемых системах. При этом обеспечиваются:

- простота модификации и расширяемость системы – выявлена высокая локальность правок при изменении и расширении функциональности системы, изменении используемых внешних периферийных устройств. Добавление в систему новых процессов и гиперпроцессов для обеспечения дополнительной функциональности не влияет на работу существующих в системе процессов.

- масштабируемость системы – увеличение количества одинаковых контролируемых устройств в системе преимущественно сводится к копированию и переименованию соответствующего набора процессов и гиперпоцесоов. Как пример, рассматривалось увеличение количества портов станции пробоподготовки SorbiPrep с трех до шести.

- простота переиспользования кода – процессы работы с USART, управления нагревателями и взаимодействия с АЦП переиспользовались между ПО SorbiPrep и системой управления установки термовакуумного напыления.

- надежность программы за счет преимущественного использования кооперативной многозадачности, что значительно снижает вероятность

возникновения гонок, по сравнению с вариантом вытесняющей многопоточности.

- низкая вероятность ошибки программиста за счет структурирующих свойств процесс-ориентированного подхода, прозрачности синтаксиса языка, а также производимого транслятором семантического анализа кода, позволяющего выявлять возможные ошибки на этапе трансляции с выдачей предупреждающих сообщений.

ВЫВОДЫ И РЕЗУЛЬТАТЫ

В работе предложен оригинальный поход к процесс-ориентированной разработке управляющего ПО встраиваемых систем на базе открытых микроконтроллерных платформ с использованием разработанного специализированного языка IndustrialC в совокупности с инструментальными средствами - транслятором в язык С и интегрированной средой на базе Notepad++.

В ходе работы получены следующие основные результаты:

1. Проведен анализ специфики управлящюего ПО ВС и программирования открытых микроконтроллерных платформ, сформулированы требования к математическим, языковым и инструментальным средствам разработки ПО ВС МПОА.

2. Разработан формализм описания управляющего ПО на микроконтроллерных платформах, предусматривающий возможность работы с аппаратными прерываниями за счет представления системы набором раздельно активируемых гиперпроцессов, определения механизмов взаимодействия между гиперпроцессами и синхронизации доступа к разделяемым данным.

3. Разработан и формализован синтаксис процесс-ориентированного языка программирования IndustrialC, имеющий низкий порог вхождения за счет

пересечения с языком C и согласованности синтаксиса специфических конструкций с C-подобными языками.

4. Задана трансляционная семантика языка IndustrialC, выраженная через правила трансляции IndustrialC-программ в эквивалентный код на языке C.

5. Разработан и реализован транслятор IndustrialC в язык C, обеспечивающий поддержку и анализ значительной части конструкций языка C за счет механизма чтения и расстановки меток препроцессора. Выявлены критерии автоматической расстановки критических секций для служебных операций языка и выдачи предупреждающих сообщений о возможном асинхронном доступе к разделяемым данным.

6. Разработана и реализована интегрированная среда разработки на базе редактора Notepad++.

7. Полученные средства использованы в практических задачах разработки ВС на МПОА и внедрены в учебный процесс.

В ходе исследования выявлено, что использование языка IndustrialC позволяет существенно снизить временные затраты на разработку, модификацию, расширение и масштабирование ПО МПОА по сравнению с другими применяемыми подходами. Интеграция аппаратных прерываний в понятийный аппарат процесс-ориентированного программирования дает возможность описания всей программной составляющей системы на одном языке. C-подобный синтаксис языка IndustrialC обуславливает низкий порог вхождения. За счет соответствия понятийного аппарата языка области встраиваемых систем управления и специфике программирования микроконтроллеров снижается вероятность ошибки программиста. Преимущественное использование кооперативной многозадачности в совокупности со статическим обнаружением асинхронного доступа к разделяемой памяти обуславливает высокую надежность и устойчивость результирующего ПО.

Список литературы

1. Официальный сайт проекта Arduino [Электронный ресурс] // URL: http://www.arduino.cc (дата обращения: 02.05.2019)
2. Страница платформы Itead Maple [Электронный ресурс] // URL: https://www.itead.cc/leaf-maple.html (дата обращения: 02.05.2019)
3. Шалыто, А. А., and Н. И. Туккель. "SWITCH-технология-автоматный подход к созданию программного обеспечения «реактивных» систем." Программирование 27.5 (2001): 45-62.
4. Harel D. Statecharts: A visual formalism for complex systems // Science of computer programming. 1987. vol. 8, issue 3. pp. 231–274.
5. Kenneth C. Crater. State Language for Machine Control // White Paper. Control Technology Corporation, Hopkinton, MA, Jul. 9, 1999, pp. 1-11
6. Зюбин В. Е. Язык Рефлекс. Математическая модель алгоритмов управления // Датчики и системы. 2006. № 5. С. 24-30.
7. Wagner F., Schmuki R., Wagner T., Wolstenholme P. Modeling software with finite state machines: a practical approach // ISBN 9780849380860 Auerbach Publications, 2006.
8. Samek, M. and Montgomery, P.Y., 2000. State-oriented programming. Embedded Systems Programming, 13(8), pp.22-43.
9. Cassandras, Christos G., and Stephane Lafortune. Introduction to discrete event systems // Springer Science & Business Media, 2009.
10. Gilles, KAHN. The semantics of a simple language for parallel programming. // Information processing 74, 1974, pp. 471-475.
11. Kavi, Krishna M., Bill P. Buckles, and U. Narayan Bhat. Isomorphisms between petr nets and dataflow graphs. // IEEE Transactions on Software Engineering 10, 1987, pp.1127-1134.
12. F. Boussinot and R. de Simone. The ESTEREL Language // Proceedings of the IEEE, 79(9):1293–1304, Sept. 1991

13. A. Benveniste, P. L. Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics // Science of Computer Programming, 16(2):103–149, Sept. 1991.
14. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE // Proceedings of the IEEE, 79(9):1305–1320, September 1991.
15. Satoh, Ichiro, and Mario Tokoro. "Semantics for a real-time object-oriented programming language." In Proceedings of 1994 IEEE International Conference on Computer Languages (ICCL'94), pp. 159-170. IEEE, 1994.
16. T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Embedded Control Systems Development with Giotto // In Proceedings of the ACM Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES), pages 64–72, June 2001.
17. Z. Wan, W. Taha, and P. Hudak. Event-Driven FRP // In Proceedings of the Internation Symposium on Principles of Declarative Languages (PADL), volume 2257, pp. 155–172, 2001.
18. Schneider, Klaus. The synchronous programming language Quartz. // Vol. 292. Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, 2009.
19. N. Halbwachs, Synchronous Programming of Reactive Systems. Kluwer, 1993.
20. Kormanyos, Christopher Object-Oriented Techniques for Microcontrollers // Real-Time C++. 2018. Springer, Berlin, Heidelberg. pp. 61-84.
21. plcLib - PLC Software for the Arduino. [Электронный ресурс] // URL: http://www.electronics-micros.com/software-hardware/plclib-arduino/ (дата обращения: 02.05.2019)
22. Зюбин В. Е. К пятилетию стандарта IEC 1131-3. Итоги и прогнозы // Приборы и системы. Управление, контроль, диагностика. 1999. № 1. С. 64–71.

23. Crater K. C. When Technology Standards Become Counterproductive. Control Technology Corporation. 1992. [Электронный ресурс] // URL: http://support.ctc-control.com/index.php?option=com_content&view=article&id=188&Itemid=null (дата обращения: 02.05.2019)
24. Белоконь С. А., Золотухин Ю. Н., Филиппов М. Н. Архитектура комплекса полунатурного моделирования систем управления летательными аппаратами // Автометрия. 2017. Т. 53, № 4. С. 44-50.
25. Simulink - Simulation and Model-Based Design - MATLAB & Simulink (Электронный ресурс) // URL: https://www.mathworks.com/products/simulink.html (дата обращения: 02.05.2019)
26. Wroldsen, Torstein, and Ståle Tveitane A real time operating system for embedded platforms // MS thesis. Høgskolen i Agder, 2004.
27. Bichu, T., et al. RTOS based software architecture for intelligent unmanned systems // Int. Conf. on Intelligent Unmanned Systems, 2013.
28. Rahman, Habibur & Muthukumaraswamy, Senthil Arumugam Preemptive Multitasking on Atmel AVR Microcontroller // 9th International Conference on Computer Engineering and Applications (CEA '15), Dubai, 2015.
29. Simonović, Mirela, and Lazar Saranovac Power management implementation in FreeRTOS on LM3S3748 // Serbian Journal of Electrical Engineering. 2013. 10.1. pp. 199-208.
30. Short, Michael, Michael J. Pont, and Jianzhong Fang Exploring the Impact of Task Preemption on Dependability in Time-Triggered Embedded Systems: a Pilot Study // ECRTS'08. Euromicro Conference on. IEEE, 2008.
31. Davis, Robert, Nick Merriam, and Nigel Tracey How embedded applications using an RTOS can stay within on-chip memory limits. // 12th EuroMicro Conference on Real-Time Systems, 2000, pp. 71-77.
32. Samek, M. RTOS considered harmful. (Электронный ресурс) // URL: https://embeddedgurus.com/state-space/2010/04/i-hate-rtoses/ (дата обращения: 02.05.2019)

33. Д. Бушенко. ООП, ФП, параллелизм и смена парадигмы [Электронный ресурс] // URL: https://www.kv.by/archive/index2010301105.htm (дата обращения: 02.05.2019)
34. Бобренок, Сергей Иванович, Сергей Михайлович Егоров, Павел Сергеевич Василевич, and Евгений Александрович Загурских. Решение проблемы синхронизации процессов при обращении к разделяемой памяти под управлением Boost. // Весник НГУ, Серия Информационные технологии. Том 13, №. 2., 2015. С. 28–33.
35. Levis, Philip, et al. TinyOS: An operating system for sensor networks // Ambient intelligence. Springer, Berlin, Heidelberg, 2005, pp. 115-148.
36. Gay, David, Philip Levis, David Culler, and Eric Brewer. nesC 1.1 language reference manual, 2003.
37. B. Warneke, M. L. andl B. Liebowitz, and K. Pister. Smart dust: Communicating with a cubic-millimeter computer. IEEE Computer Magazine, pages 44–51, January 2001.
38. David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation (PLDI '03). ACM, New York, NY, USA, pp. 1-11.
39. QP state machine framework for Arduino. [Электронный ресурс]. URL: http://www.state-machine.com/qp/ (дата обращения: 02.05.2019.
40. Samek, Miro. Practical UML statecharts in C/C++: event-driven programming for embedded systems. CRC Press, 2008.
41. Zyubin V. E. Hyper-automaton: A Model of Control Algorithms // Proceedings of the IEEE International Siberian Conference on Control and Communications (SIBCON-2007), Tomsk, 2007, pp. 51–57.
42. Зюбин В. Е. Процесс Чохральского: создание системы управления на основе пакета LabVIEW // VIII Международная конференция по актуальным проблемам физики, материаловедения, технологии и

диагностики кремния, наноразмерных структур и приборов на его основе «Кремний-2011» (Москва, Россия, 5–8 июля, 2011). М.: Изд. дом «МИСиС». С. 96–97.

43. Степанова Т. Н., Зюбин В. Е. Автоматизация исследований роста монокристаллов методом Чохральского на физическом имитаторе // VIII Международная конференция по актуальным проблемам физики, материаловедения, технологии и диагностики кремния, наноразмерных структур и приборов на его основе «Кремний-2011» (Москва, Россия, 5–8 июля, 2011). М.: Изд. дом «МИСиС». С. 119–120.
44. Ковадло П. Г., Лубков А. А., Бевзов А. Н., Будников К. И., Власов С. В., Зотов А. А., Колобов Д. Ю., Курочкин А. В., Котов В. Н., Лылов С. А., Лях Т. В., Максимов А. С., Перебейнос С. В., Петухов А. Д., Пещеров В. С., Попов Ю. А., Русских И. В., Томин В. Е. Система автоматизации Большого солнечного вакуумного телескопа // Автометрия. 2016. Т. 52, вып. 2. С. 97–106.
45. Лях Т. В., Зюбин В. Е., Сизов. М. М. Опыт применения языка Reflex при автоматизации Большого солнечного вакуумного телескопа // Промышленные АСУ и контроллеры. 2016. № 7. С. 37-43.
46. Halbwachs, Nicolas, and Siwar Baghdadi. "Synchronous modelling of asynchronous systems." In International Workshop on Embedded Software, pp. 240-251. Springer, Berlin, Heidelberg, 2002.
47. Balarin F, Giusto P, Jurecska A, Chiodo M, Hsieh H, Passerone C, Sentovich E, Lavagno L, Tabbara B, Sangiovanni-Vincentelli A, Suzuki K. Hardware-software co-design of embedded systems: the POLIS approach // Springer Science & Business Media; 1997 May 31.
48. Benveniste, Albert, Benoît Caillaud, and Paul Le Guernic. From synchrony to asynchrony. // In International Conference on Concurrency Theory, pp. 162-177. Springer, Berlin, Heidelberg, 1999.

49. Berry G, Sentovich E. Embedding synchronous circuits in GALS-based systems // In Sophia-Antipolis conference on Micro-Electronics (SAME 98), Oct. 1998.
50. Hoare, Charles Antony Richard. Communicating sequential processes. // In The origin of concurrent programming, pp. 413-443. Springer, New York, NY, 1978.
51. Ellison, Chucky, and Grigore Rosu. An executable formal semantics of C with applications // In ACM SIGPLAN Notices, vol. 47, no. 1, pp. 533-544. ACM, 2012.
52. Gurevich, Yuri, and James K. Huggins. The semantics of the C programming language. // In International Workshop on Computer Science Logic, pp. 274-308. Springer, Berlin, Heidelberg, 1992.
53. Krebbers, Robbert, Xavier Leroy, and Freek Wiedijk. Formal C semantics: CompCert and the C standard // In International Conference on Interactive Theorem Proving, pp. 543-548. Springer, Cham, 2014.
54. Slonneger K. and Kurtz B. L. Formal syntax and semantics of programming languages // AddisonWesley Reading, 1995, – 340 p
55. Sadilek, Daniel A. Prototyping domain-specific languages for wireless sensor networks. // Workshop on Software Language Engineering, Nashville, Tennessee, USA. 2007.
56. Cleenewerck, Thomas, and Ivan Kurtev. "Separation of concerns in translational semantics for DSLs in model engineering." In Proceedings of the 2007 ACM symposium on Applied computing, pp. 985-992. ACM, 2007.
57. Anureev, Igor, Natalia Garanina, Tatiana Liakh, Andrei Rozov, Horst Schulte, and Vladimir Zyubin. "Towards Safe Cyber-Physical Systems: the Reflex Language and Its Transformational Semantics." // 2019 International Siberian Conference on Control and Communications (SIBCON), pp. 1-6. IEEE, 2019.
58. Klop, Jan Willem, and J. W. Klop. Term rewriting systems // Centrum voor Wiskunde en Informatica, 1990.

59. The Apache Ant Project (Электронный ресурс) // URL: https://ant.apache.org (дата обращения: 02.05.2019)
60. Eysholdt, Moritz, and Heiko Behrens. "Xtext: implement your language faster than the quick and dirty way." In Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion, pp. 307-309. ACM, 2010.
61. Bettini, Lorenzo. Implementing domain-specific languages with Xtext and Xtend // Packt Publishing Ltd, 2016.
62. Parr, Terence J., and Russell W. Quong. ANTLR: A predicated-LL (k) parser generator // Software: Practice and Experience 25, no. 7, 1995, pp. 789-810.
63. Campagne, Fabien. The MPS Language Workbench // Campagne Lab., 2013.
64. Savic, Dušan, Alberto Rodrigues da Silva, Siniša Vlajic, Saša Lazarevic, Ilija Antovic, Vojislav Stanojevic, and Miloš Milic. Preliminary experience using JetBrains MPS to implement a requirements specification language // In 2014 9th International Conference on the Quality of Information and Communications Technology, pp. 134-137. IEEE, 2014.
65. Voelter, Markus, Daniel Ratiu, Bernhard Schaetz, and Bernd Kolb. "mbeddr: an extensible C-based programming language and IDE for embedded systems." In Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity, pp. 121-140. ACM, 2012.
66. Ratiu, Daniel, and Markus Voelter. "Automated testing of DSL implementations: experiences from building mbeddr." In Proceedings of the 11th International Workshop on Automation of Software Test, pp. 15-21. ACM, 2016.
67. Aaby AA. Compiler construction using flex and bison // Walla Walla College. 2003.
68. Levine, John. Flex & Bison: Text Processing Tools. // O'Reilly Media, Inc., 2009.

69. Parr, Terence, and Kathleen Fisher. LL (*): the foundation of the ANTLR parser generator. // In ACM Sigplan Notices, vol. 46, no. 6, pp. 425-436. ACM, 2011.
70. mbeddr for Arduino (Электронный ресурс) // URL: https://github.com/coolya/mbeddr.arduino (дата обращения: 02.05.2019)
71. Levine, John R., John Mason, John R. Levine, John R. Levine, Paul Levine, Tony Mason, and Doug Brown. Lex & yacc. // O'Reilly Media, Inc., 1992.
72. Notepad++ Homepage (Электронный ресурс) // URL: https://notepad-plus-plus.org (дата обращения: 02.05.2019)
73. Buyya R, Dastjerdi AV, editors. Internet of Things: Principles and paradigms. Elsevier; 2016 May 11.
74. Зюбин В. Е. Статическая балансировка вычислительной нагрузки в процесс-ориентированном программировании при многопоточной реализации // Вестник Новосибирского государственного университета. Серия: Информационные технологии 10, №. 2, 2012.

Приложение 1. Структурная грамматика языка IndustrialC

```
program: program_items_list

program_items_list: program_items_list program_item
                  | program_item

program_item: var_declaration
            | mcu_declaration
            | proc_def
            | hp_definition
            | c_code
            | func_definition

hp_definition: "hyperprocess" "identifier" "{" "vector" "="
"identifier" ";" "register" "=" "identifier" ";" "bit" "="
"identifier" ";" "}"

proc_def: "process" "identifier" ":" "identifier" "{" proc_body
"}"

proc_body: proc_body state
         | proc_body var_declaration
         | state
         | var_declaration

state: "state" "identifier" "{" state_body "}"

state_body: %empty
          | state_items_list

state_items_list: state_items_list state_block_item
                | state_block_item
                | state_items_list error ";"
                | error ";"
```

```
state_block_item: block_item
                | timeout

block_item: var_declaration
          | statement
          | c_code

block_items_list: block_items_list block_item
                | block_item

c_code: "c code line"

statement: "set" "state" "identifier" ";"
         | "start" "process" "identifier" ";"
         | "stop" "process" "identifier" ";"
         | "stop" "process" ";"
         | expression_statement
         | "start" "hyperprocess" "identifier" ";"
         | "stop" "hyperprocess" "identifier" ";"
         | "stop" "hyperprocess" ";"
         | compound_statement
         | "if" "(" expr ")" statement "else" statement
         | "if" "(" expr ")" statement
         | "switch" "(" expr ")" statement
         | "case" expr ":" statement
         | "default" ":" statement
         | "for" for_prep_scope "(" for_init_statement
expression_statement expr ")" statement
         | "for" for_prep_scope "(" for_init_statement
expression_statement ")" statement
         | "atomic" statement
         | "reset" "timeout" ";"
         | "return" expr ";"
         | "return" ";"
```

```
          | "break" ";"
          | "continue" ";"
          | "restart" ";"

for_prep_scope: %empty

for_init_statement: expression_statement
                  | var_declaration

expression_statement: expr ";"
                    | ";"

compound_statement: "{" prep_compound block_items_list "}"
                  | "{" prep_compound "}"

prep_compound: %empty

timeout: "timeout" "(" expr ")" "{" block_items_list "}"

expr: assignment_expr

assignment_expr: binary_expr
               | unary_expr assignement_op assignment_expr

binary_expr: cast_expr
           | binary_expr "||" binary_expr
           | binary_expr "&&" binary_expr
           | binary_expr "|" binary_expr
           | binary_expr "^" binary_expr
           | binary_expr "&" binary_expr
           | binary_expr "==" binary_expr
           | binary_expr "!=" binary_expr
           | binary_expr "<" binary_expr
           | binary_expr ">" binary_expr
           | binary_expr "<=" binary_expr
```

```
            | binary_expr ">=" binary_expr
            | binary_expr "<<" binary_expr
            | binary_expr ">>" binary_expr
            | binary_expr "+" binary_expr
            | binary_expr "-" binary_expr
            | binary_expr "*" binary_expr
            | binary_expr "/" binary_expr
            | binary_expr "%" binary_expr

unary_expr: postfix_expr
          | "++" unary_expr
          | "--" unary_expr
          | "-" cast_expr
          | "~" cast_expr
          | "!" cast_expr

postfix_expr: primary_expr
            | postfix_expr "[" expr "]"
            | "identifier" "(" ")"
            | "identifier" "(" arg_expr_list ")"
            | postfix_expr "++"
            | postfix_expr "--"

arg_expr_list: arg_expr_list "," assignment_expr
             | assignment_expr

primary_expr: "true"
            | "false"
            | "integer constant"
            | "double constant"
            | "hex constant"
            | "binary constant"
            | "identifier"
            | "(" expr ")"
            | "identifier" "active"
```

```
            | "identifier" "passive"
            | "c code expression"
            | "string literal"

assignement_op: "="
              | ">>="
              | "<<="
              | "+="
              | "-="
              | "*="
              | "/="
              | "%="
              | "&="
              | "^="
              | "|="

cast_expr: unary_expr
         | "(" type_name ")" cast_expr

type_name: decl_specs
         | decl_specs abstract_declarator

abstract_declarator: pointer
                   | direct_abstract_declarator
                   | pointer direct_abstract_declarator

direct_abstract_declarator: "(" abstract_declarator ")"
                          | "[" "]"
                          | "[" "integer constant" "]"
                          | direct_abstract_declarator "[" "]"
                          | direct_abstract_declarator "["
"integer constant" "]"

pointer: "*"
       | "*" pointer
```

```
func_definition: decl_specs func_declarator func_body

func_body: compound_statement
         | ";"

func_declarator: "identifier" "(" ")"

func_declarator: "identifier" "(" param_list ")"

param_list: param_list "," param_declarator
          | param_declarator

param_declarator: decl_specs direct_declarator

var_declaration: decl_specs init_declarator_list ";"

decl_specs: decl_specs type_spec
          | type_spec

init_declarator_list: init_declarator_list "," init_declarator
                    | init_declarator

init_declarator: direct_declarator
               | direct_declarator "=" initializer

direct_declarator: "identifier"
                 | direct_declarator "[" binary_expr "]"
                 | direct_declarator "[" "]"

initializer: assignment_expr
           | "{" initializer_list "}"
           | "{" initializer_list "," "}"

initializer_list: initializer_list "," initializer
```

```
                | initializer

type_spec: "void"
         | "char"
         | "int"
         | "short"
         | "long"
         | "float"
         | "double"
         | "signed"
         | "unsigned"
         | "bool"
         | "const"
         | "volatile"
         | "inline"

mcu_declaration: "vector" "identifier" ";"
               | "register" "identifier" ";"
               | "bit" "identifier" ";"
```