

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное
учреждение высшего образования
«Пермский государственный национальный
исследовательский университет»
Механико-математический факультет

УДК 004.422.833

*Кафедра математического обеспечения
вычислительных систем*

**Разработка средств автоматизации программирования устройств
Интернета вещей на базе платформы SciVi**
Выпускная квалификационная работа бакалавра

Работу выполнил студент группы ПМИ-
1,2-2019 4 курса механико-
математического факультета
Лукьянов Александр Михайлович

«__» _____ 2023 г.

Научный руководитель:
кандидат физико-математических наук,
доцент кафедры МОВС
Рябинин Константин Валентинович

«__» _____ 2023 г.

Пермь 2023

СОДЕРЖАНИЕ

СОДЕРЖАНИЕ	2
ВВЕДЕНИЕ	4
1 Онтологически управляемые периферийные вычисления. Постановка задач	6
1.1 Общие идеи подхода	6
1.2 Онтологически управляемые периферийные вычисления в системе SciVi	6
1.3 Когнитивное сжатие онтологий. Формат EON	7
1.4 Встраиваемый механизм рассуждений	8
1.5 Существующие проблемы коммуникации механизма рассуждений	9
1.6 Уточнение поставленных задач	10
2 Анализ существующих решений задач, аналогичных поставленным	11
2.1 Анализ наиболее популярных средств управления энергонезависимой памятью	11
2.2 Анализ существующих протоколов самоидентификации устройств Интернета вещей	17
3 Разработка библиотеки менеджера EEPROM	23
3.1 Первая версия библиотеки	23
3.2 Уточнение требований к разрабатываемой библиотеке	23
3.3 Разработка структуры библиотеки	24
3.4 Реализация библиотеки	30
4 Разработка библиотеки для сетевой самоидентификации периферийных устройств	32
4.1 Требования к модулю	32
4.2 Анализ существующих реализаций протокола	34
4.3 Разработка структуры библиотеки	34
4.4 Разработка библиотеки	36
4.5 Возможности для интеграции разработанного модуля в платформу SciVi	41
ЗАКЛЮЧЕНИЕ	44
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	45
ПРИЛОЖЕНИЕ А	47

ПРИЛОЖЕНИЕ Б

48

ПРИЛОЖЕНИЕ В

51

ВВЕДЕНИЕ

В настоящее время активно развивается концепция Интернета вещей (англ. internet of things, IoT) — сетей передачи данных между устройствами без непосредственного участия человека. Интернет вещей находит применение в различных областях: от использования в сфере персональных устройств и систем „умных домов“ до автоматизации производств и систем мониторинга. В то же время программирование устройств IoT остаётся достаточно сложным процессом в связи со множеством трудностей в организации сетевого взаимодействия и реконфигурации устройств. Кроме того, этот процесс обычно требует значительной квалификации разработчика, что затрудняет использование IoT специалистами предметных областей конечных решаемых задач.

Со значительной частью обозначенных проблем помогает справиться применение подхода онтологически-управляемых периферийных вычислений (англ. Ontology-Driven Edge Computing), который позволяет управлять устройствами IoT и их сетями только с помощью онтологий, без необходимости прибегать к классическому программированию[1]. Использование данного подхода также позволяет создавать унифицированные интерфейсы устройств и объединять в общие сети разнородные вычислительные ресурсы[2].

Онтологически-управляемые периферийные вычисления играют важную роль в работе платформы научной визуализации и визуальной аналитики SciVi[3; 4]. Платформа SciVi позволяет декларативно, с помощью графического редактора, описывать алгоритмы сбора, обработки и отображения данных, а также выполнять эти алгоритмы на различных устройствах, храня и передавая их в виде онтологий. Благодаря этому создаваемые алгоритмы могут быть исполнены в том числе на устройствах IoT, а сам процесс создания не требует особой квалификации программиста и может быть выполнен специалистом в области конкретной решаемой задачи. Важными преимуществами платформы являются также расширяемость и адаптируемость: элементарные фрагменты, из которых составляются алгоритмы, также описываются онтологиями, и их список может быть легко пополнен операциями, необходимыми для решения конкретных пользовательских задач.

При этом платформа имеет ряд открытых возможностей для большей автоматизации процесса программирования устройств Интернета вещей. Реализации части таких возможностей и посвящена данная работа.

В работе рассмотрены проблемы автоматизации управления энергонезависимой памятью периферийных устройств и автоматизации поиска таких устройств в сети.

Цель работы: разработать программные средства, позволяющие в большей мере автоматизировать процесс программирования устройств Интернета вещей в рамках

платформы SciVi.

Объект исследования данной работы: автоматизация периферийных вычислений.

Предмет исследования: средства платформы SciVi для организации онтологически-управляемых периферийных вычислений.

Для достижения цели работы были поставлены следующие задачи:

1. Провести анализ литературы по тематике Интернета вещей и онтологически-управляемых периферийных вычислений.
2. Изучить принципы функционирования платформы визуальной аналитики SciVi.
3. Провести анализ литературы и существующих решений в областях решаемых проблем.
4. Спроектировать и разработать решения рассматриваемых проблем с учётом особенностей подхода онтологически-управляемых периферийных вычислений и платформы SciVi.
5. Интегрировать разработанные решения в платформу SciVi.

1 Онтологически управляемые периферийные вычисления.

Постановка задач

1.1 Общие идеи подхода

Онтологически управляемые периферийные вычисления (англ. *Ontology-Driven Edge Computing*) — подход, основанный на внедрении интеллектуальности в периферийные устройства в рамках экосистемы интернета вещей (IoT)[1]. В этом подходе использование онтологий позволяет описывать и управлять функциональностью и поведением периферийных устройств на основе знаний и семантических моделей.

В рамках такого подхода традиционная прошивка (встраиваемая программа) для устройств заменяется на комбинацию трёх компонентов:

1. Доменной онтологии, описывающей доступные для выполнения на устройстве действия, средства коммуникации устройства, его состав и другие знания, жёстко привязанные к устройству.
2. Онтологии задачи, описывающая действия необходимые для решения конкретной задачи.
3. Механизма рассуждений, выполняющий вычисления, описываемы онтологией задачи. Своего рода интерпретатор алгоритмов, записанных в онтологическом виде.

Механизм рассуждений является специальной встраиваемой программой для периферийных устройств и выполняется непосредственно на них. Доменная онтология используется для составления онтологии задач, а также её сжатия и разжатия с использованием формата EON, описанного ниже. За счёт этого, отсутствует необходимость хранить такую онтологию на конечном устройстве, так как она не используется непосредственно в момент вычислений, в отличие от онтологии задачи. Доменная онтология может храниться на отдельном внешнем устройстве, к которому у устройства, производящего вычисления есть сетевой доступ.

1.2 Онтологически управляемые периферийные вычисления в системе SciVi

На основе доменной онтологии средства SciVi генерируют инструменты для получения, обработки и визуализации различных данных. Для использования этих инструментов SciVi предоставляет высокоуровневый графический интерфейс пользователя, также генерируемый на основе доменной онтологии. Данный интерфейс позволяет создавать диаграммы потоков данных (англ. *Data Flow Diagrams, DFD*), описывая узлы обработки данных и то, как между ними должны передаваться данные. В рамках SciVi такие узлы именуются операторами. Операторы описывают различные операции с данными, которые могут выполнены на клиентской или серверной частях

платформы, а также на внешних устройствах. Для реализации последнего варианта и применяется подход онтологически управляемых периферийных вычислений.

На основе DFD, созданной пользователем, SciVi генерирует онтологию задачи, соответствующую операциям с данными для выполнения на периферийном устройстве. Затем эта онтология сжимается и передаётся на конечное устройство, где она исполняется с помощью механизма рассуждений, после чего результаты её выполнения могут быть переданы другим устройствам в соответствии с составленной DFD.

В качестве периферийных устройств SciVi, в основном, использует микроконтроллеры ESP8266, которые будут являться целевой платформой для последующих разработок в рамках данной работы.

1.3 Когнитивное сжатие онтологий. Формат EON

Микроконтроллеры обычно обладают небольшим объёмом оперативной памяти, например, у популярного микроконтроллера ATtiny45 её объём составляет всего 256 байт. В следствии этого хранить на устройстве онтологии задач в обычных форматах для описания онтологий, таких как OWL и RFD, не представляется возможным. Для сжатия онтологий в процессе работы над платформой SciVi был разработан специальный формат — EON (англ. Embedded or Edge ONtology).

Перевод онтологии в формат EON включает в себя два основных этапа:

1. Когнитивное сжатие. На данном этапе из онтологии удаляются избыточные знания, которые в будущем могут быть восстановлены с помощью доменной библиотеки. Затем из онтологии удаляются все связи, кроме `instance_of` (с их помощью онтология задачи связывается с доменной, поэтому они необходимы для разжатия) и `use_for` (эта связь используется для описания направления передачи данных на основе DFD). Важно отметить, что набор сохраняемых связей не является фиксированным и может быть изменён (в том числе расширен) при необходимости. Кроме того все имена узлов и связей заменяются на численные идентификаторы. Для связей и узлов доменной онтологии идентификаторы берутся из неё самой, а для узлов онтологии задач генерируются автоматически и сохраняются во время всего процесса обработки онтологии.
2. Затем полученная онтология сериализуется в особый бинарный формат EON, в котором в дальнейшем передаётся и обрабатывается.

При необходимости всегда может быть проведён обратный процесс разжатия, требующий только доменной онтологии, аналогичной той, что использовалась при сжатии.

Приведённый алгоритм позволяет сжимать онтологии даже в сотни раз по сравнению с используемыми обычно форматами.

1.4 Встраиваемый механизм рассуждений

Механизм рассуждений работает на самих периферийных устройствах и с их точки зрения является обычной прошивкой. Архитектура такого механизма представлена на рисунке 1.1[5].

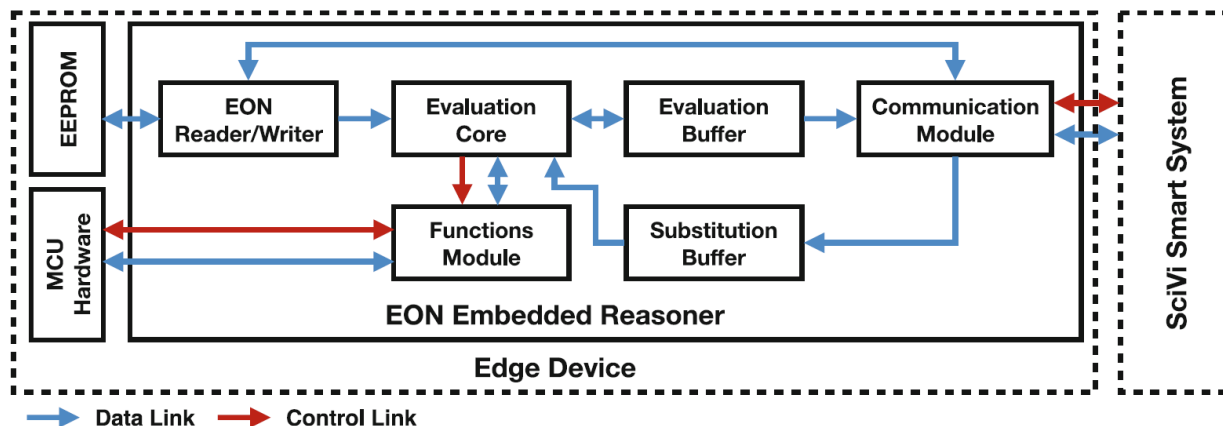


Рисунок 1.1 – Архитектура встраиваемого механизма рассуждений

- Модуль функций (Functions Module на рисунке). Модуль управляет аппаратными ресурсами устройства и определяет доступные для использования операторы SciVi.
- Модуль коммуникации (Communication Module). Модуль обеспечивает передачу данных и команд между этим и другими устройствами системы SciVi. В зависимости от устройства, модуль может включать средства для сетевой связи по протоколу WebSocket, через интерфейс UART и различными иными способами.
- Модуль выполнения (Evaluation Core). Элемент обходит онтологию задачи в формате EON и вызывает операторы.
- Модуль чтения/записи (EON Reader/Writer). Элемент осуществляет чтение и запись онтологии в память устройства.
- Буфер результатов (Evaluation Buffer). Буфер для хранения результатов выполнения операторов.
- Буфер замещения (Substitution Buffer). Буфер для перезаписи результатов выполнения операторов и управления устройством извне через модуль коммуникации.

Данный механизм реализован в рамках платформы SciVi на языке программирования C++ и может исполняться на различных устройствах (протестирован на ESP8266, ATmega328 и ATtiny45). Эта реализация может быть перенесена и на множество других устройств, за счёт использования универсальных инструментов среды разработки Arduino IDE и различных дополнений к ней.

Важно отметить, что различные устройства могут отличаться набором доступных операций и средств связи, поэтому итоговые прошивки, использующие

описанный механизм, для разных устройств должны отличаться. Для решения этой проблемы SciVi имеет специальное средство, генерирующие конкретные прошивки, содержащие встраиваемый механизм рассуждений, автоматизированно на основе онтологического описания целевого устройства, создаваемого пользователем.

1.5 Существующие проблемы коммуникации механизма рассуждений

На момент начала выполнения данной работы описанный механизм имел две значительные проблемы, связанные с коммуникацией с его окружением.

1.5.1 Использование энергонезависимой памяти

Энергонезависимая память — особый вид запоминающих устройств, способный хранить данные при отсутствии электропитания. Такая память используется в составе вычислительных устройств, в том числе для хранения данных, необходимых для их инициализации, и конфигурационных данных между их запусками. В микроконтроллерах для решения этой задачи обычно используются электрически стираемые перепрограммируемые постоянные запоминающие устройства (ЭСППЗУ, англ. Electrically Erasable Programmable Read-Only Memory, EEPROM) — вид устройств энергонезависимой памяти, позволяющих электрическим импульсом стереть сохранённые данные, а затем, при необходимости, записать новые [6; 7].

Механизм рассуждений SciVi использует EEPROM для хранения онтологий задач. Это необходимо для того, чтобы устройство могло автоматически возобновить исполнение полученного им ранее алгоритма после временного отсутствия электропитания. Такое хранение уже было реализовано в SciVi, однако для работы с EEPROM использовались несовершенные и низкоуровневые средства, которые не позволяют удобно использовать EEPROM из независимых программных модулей. Это вызвано необходимостью ручных манипуляций с адресами памяти при каждом обращении к ней. В конечно счёте, использование энергонезависимой памяти микроконтроллера механизмом рассуждений делало невозможным её удобное (а для множества случаев - любое) использование для других целей. Несмотря на возможно достаточный объём памяти.

Описанная особенность использования EEPROM значительно ограничивает возможности автоматической генерации программ для периферийных устройств платформой SciVi. Таким образом платформе было необходимо новое, более высокоуровневое, средство для управления EEPROM, которое бы могло автоматизировать использование адресов памяти.

1.5.2 Обнаружение в сети устройств для периферийных вычислений

Другая проблема связана с сетевой коммуникацией устройств, а именно — с обнаружением их в сети. Изначально клиентская часть SciVi могла подключаться только к одному периферийному устройству, причём по строго фиксированному

IP-адресу. Этот факт сильно ограничивал возможности использования платформы, требуя поиска устройств и подключения к ним от конечного пользователя. В связи с этим, платформе потребовался механизм, позволяющий автоматически в любой сети легко находить устройства для выполнения на них онтологически управляемых периферийных вычислений. После нахождения всех таких доступных устройств, пользователь смог бы выбирать на каком из них выполнять тот или иной оператор SciVi.

1.6 Уточнение поставленных задач

Обобщив вышесказанное, можно уточнить поставленные в работе задачи. Необходимо разработать два программных модуля на языке программирования C++: для высокоуровневого взаимодействия с энергонезависимой памятью микроконтроллеров и для их обнаружения в сети других устройств.

Для каждого из таких модулей необходимо:

1. Изучить существующие средства для решения аналогичных задач.
2. Проанализировать найденные средства с точки зрения их использования в онтологически-управляемых периферийных вычислениях.
3. Спроектировать собственное решение, учитывающее потребности и особенности таких вычислений и платформы SciVi.
4. Реализовать спроектированный модуль.
5. Интегрировать разработанный модуль в платформу SciVi и провести его тестирование в условиях онтологически-управляемых периферийных вычислениях.

2 Анализ существующих решений задач, аналогичных поставленным

2.1 Анализ наиболее популярных средств управления энергонезависимой памятью

2.1.1 Требования к средству управления энергонезависимой памятью

Для программирования микроконтроллеров в проекте SciVi используются инструменты среды разработки Arduino IDE, позволяющие программировать микроконтроллеры, используя язык программирования C++, а также специальное дополнение к этой среде для работы с ESP8266 [8], содержащее, в частности, набор „стандартных“ библиотек.

соответственно, необходимое средство управления энергонезависимой памятью должно являться библиотекой языка C++ и может использовать стандартный набор библиотек Arduino IDE и указанного дополнения к ней. Такая библиотека должна:

1. Предоставлять пользователю возможность сохранять и считывать данные из EEPROM микроконтроллера. При этом:
 - 1.1. Данные могут иметь произвольную структуру.
 - 1.2. Доступ к ним должен производиться по некоторым идентификаторам, уникальным для различных данных и без необходимости ручных манипуляций с адресами EEPROM со стороны пользователя.
2. Автоматически определять факт наличия в EEPROM данных с заданным идентификатором, определять адрес для записи новых данных, сохранять в EEPROM метаданные о хранящихся данных для их использования после перезапуска микроконтроллера.
3. Минимизировать количество операций записи в EEPROM, т.к. каждая ячейка такой памяти может быть перезаписана ограниченное количество раз (обычно производители гарантируют от 100.000 до 1.000.000 циклов перезаписи), после чего выходит из строя.
4. Выполняться на микроконтроллерах серии ESP8266 и, по возможности, на платформе Arduino, так как эта платформа является наиболее популярной и распространённой.

Ключевым требованием является полная автоматизация работы с адресами EEPROM, это необходимо для создания возможности использования EEPROM в различных независимых программных модулях. В противном случае, таким модулям понадобилось бы каким-либо образом обмениваться информацией об используемых ими адресах для избежания чтения и записи разными модулями в одни и те же ячейки EEPROM.

2.1.2 Стандартная библиотека

В стандартный набор библиотек Arduino IDE уже входит библиотека для работы с EEPROM [9]. Как и SciVi в данный момент, большая часть проектов, хранящих какие-либо данные в EEPROM, ограничивается использованием этой библиотеки. Однако она предоставляет только простые функции, такие как записать и считать байт по указанному адресу. Позже в неё были добавлены функции для чтения и записи данных произвольных типов, но также только по явно указанному адресу. Очевидно, это делает стандартную библиотеку нарушающей все поставленные требования, однако её функции можно использовать в качестве низкоуровневого интерфейса EEPROM в разрабатываемой библиотеке. Кроме указанных, стандартная библиотека содержит функцию-обёртку вокруг функции записи, производящую фактическую перезапись данных только тогда, когда они отличаются от хранящихся по указанному адресу в данный момент. В дальнейшем, в большинстве случаев, разумно использовать для записи именно эту функцию с целью уменьшения износа EEPROM.

2.1.3 Библиотека EEPROMEx

Библиотека EEPROMEx (от англ. Extended — Расширенный) — одна из первых разработок для работы с EEPROM микроконтроллеров в среде Arduino IDE [10]. Данная библиотека была создана раньше, чем описанная выше, поэтому часть предоставляемых ими возможностей совпадает, однако реализованы они независимо друг от друга. EEPROMEx содержит функции для чтения и записи в EEPROM данных некоторых стандартных типов: целочисленных беззнаковых чисел длиной в 8, 16 и 32 бита и 32-х и 64-х битных чисел с плавающей точкой. В библиотеке также содержатся функции для чтений и записи отдельных битов и, как и в стандартной библиотеке, аналогичные функции для работы с данными пользовательских типов и аналоги всех функций записи, производящие запись только при отличии данных. Кроме того EEPROMEx содержит и уникальную возможность — с помощью класса EEPROMVar связывать одним объектом переменные в коде программы и данные в EEPROM (их адреса). Причём эти адреса назначаются автоматически, что косвенно соответствует части требований, описанных в пункте 2.1.1. Рассмотрим пример использования EEPROMVar, демонстрирующий большую часть возможностей данного класса:

```
1 EEPROMVar<float> eepromFloat(5.5);  
2  
3 floatVar.restore();  
4  
5 floatVar = 10.5;  
6  
7 float input = floatVar;  
8  
9 floatVar.save();
```

Как можно увидеть в строке 1 примера, при создании объекта класса

EEPROMVar указывается тип данных, которые необходимо хранить (в примере — float), и значение по умолчанию, которое передаётся в конструктор в виде параметра (в примере — 5.5). Метод restore, вызываемый в строке 3, считывает из EEPROM данные, соответствующие данному объекту, и сохраняет в него эти данные. В строке 5 демонстрируется возможность изменения хранимых данных с помощью оператора присвоения, причём в качестве левостороннего значения выступает объект класса EEPROMVar, а правостороннего — значение хранимого объектом типа пользовательских данных. Это достигается за счёт переопределения оператора присвоения для данного класса. В строке 7 показан обратный переход: объект класса EEPROMVar может быть переведён в значение хранимого им типа. В последней строке показан вызов метода save, сохраняющего значение, хранящееся в объекте, в EEPROM.

Также для класса EEPROMVar определены некоторые операторы, такие как оператор инкремента (++), присвоения со сложением (+=) и другие операторы, связанные с изменением хранимого значения. Такие операторы вызывают в своём теле аналогичные для хранимых данных, что, разумеется, требует их существования.

Описанный класс механизм имеет ряд значительных недостатков. Первый из них заключается в способе связывания данных и адреса, по которому они хранятся в EEPROM. Обычно для этого данным выдаются уникальные имена, и каким-либо образом сохраняется информация о том, что данные с определённым именем находятся по определённому адресу. Как можно увидеть в приведённом примере, при использовании библиотеки EEPROMEx присвоения имён не происходит. В этом случае данные располагаются в EEPROM друг за другом в порядке создания объектов класса EEPROMVar. Таким образом, для того, чтобы гарантировать, что при перезапуске микроконтроллера объекты класса EEPROMVar связываются с одними и теми же адресами в EEPROM, необходимо обеспечивать создание этих объектов в строго одинаковом порядке при каждом включении устройства. Эта работа ложится на плечи конечного пользователя — разработчика программы для микроконтроллера. Причём даже в случае, если разработчик не применяет описываемый класс напрямую, но он используется в импортируемых итоговой программой библиотеках.

Второй проблемой является невозможность определения первого вызова конструктора для каких-либо пользовательских данных, то есть момента, в который эти данные отсутствуют в EEPROM, и необходимо записать значение по умолчанию. И вновь разработчик конечной программы должен каким-либо образом самостоятельно определять, когда после создания объекта необходимо проводить сохранение в EEPROM значения по умолчанию, а когда наоборот считывание уже хранящегося в памяти значения.

В конечном итоге можно сделать вывод, что библиотека EEPROMEx частично соответствует предъявляемым требованиям, но при этом её использование нельзя назвать удобным, так как оно требует от конечного пользователя большой работы, которая могла бы быть автоматизирована. При этом идея использовать переопределение

операторов для получения и изменения пользовательских данных в EEPROM является крайне удачной и удобной в использовании. Такой вариант предпочтительнее использования для тех же целей отдельных методов, так как, во-первых, позволяет пользователю библиотеки писать меньше кода и, во-вторых, позволяет ему думать об объекте, описывающем данные в EEPROM, как о самих этих данных. В дальнейшем при разработке собственной библиотеки имеет смысл реализовать аналогичный интерфейс.

2.1.4 Библиотека EEManager

Сравнительно недавно была опубликована новая библиотека для работы с EEPROM — EEManager [11]. Она имеет открытый исходный код (опубликован под лицензией MIT [12]) и документацию на русском языке. Так как данная библиотека значительно новее предыдущей рассмотренной, и в момент её создания уже существовала описанная выше версия стандартной библиотеки, EEManager используют её возможности, в первую очередь — функции-обёртки, уменьшающие износ памяти.

Данная библиотека имеет следующие преимущества:

- Реализован механизм отложенной записи: по умолчанию данные записываются в EEPROM с заданной задержкой после последней команды на запись. Использование такого подхода имеет смысл в ситуациях, когда данные должны перезаписываться много раз за короткий промежуток времени, в действительности же, с таким механизмом данные в EEPROM будут записаны только в последний раз, что значительно замедлит износ памяти. В то же время этот механизм имеет значительный недостаток: если потеря питания произойдёт после команды записи, но до истечения задержки, новые данные записаны не будут. Это делает использование такого механизма оправданным только в устройствах, для которых гарантия записи не является обязательной и точность восстановления состояния после потери питания не представляет критической важности.
- Библиотека также реализует „механизм ключа первой записи“. Вместе с каждым блоком данных в EEPROM хранится специальный однобайтовый ключ. При обращении к блоку данных пользователь указывает придуманный им ключ, который не должен изменяться от запуска к запуску, а из EEPROM считывается записанное значение ключа. Если они совпадают, значит необходимые данные уже находятся в EEPROM и их необходимо только считать, иначе данные никогда не были записаны, в этом случае данные должны быть сохранены в EEPROM.

Работа с библиотекой осуществляется следующим образом:

1. Создаётся переменная в энергозависимой памяти, значение которой необходимо хранить в EEPROM.
2. Создаётся специальный объект, описывающий блок данных EEPROM (некоторый аналог EEPROMVar из библиотеки EEPROMEx). При этом

пользователь указывает переменную в энергозависимой памяти, значение которой необходимо хранить, и адрес в EEPROM, начиная с которого должна быть записана эта переменная.

3. С помощью механизма ключа первой записи либо в EEPROM записывается значение переменной по умолчанию, либо наоборот сохранённое в EEPROM значение считывается в переменную.
4. В дальнейшем, по необходимости, текущее значение переменной записывается в EEPROM. Для этого необходимо сначала обновить значение энергозависимой переменной, с которой связан описанный выше объект, а затем записать это новое значение в EEPROM. Чтобы сделать это, можно воспользоваться одним из двух методов: для немедленной записи и для запуска таймера записи с задержкой.

Несмотря на указанные преимущества перед стандартной библиотекой, библиотека EEManager так же не может быть использована в готовом виде, так как обращение к блокам данных в ней производится только по их адресам. Это делает невозможным независимое использование данной библиотеки из различных программных модулей.

2.1.5 Библиотека EEPROMWearLevel

В большинстве микроконтроллеров, в частности, на платформе Arduino, используется EEPROM с возможностью перезаписи отдельного байта, таким образом одни ячейки памяти могут изнашиваться быстрее других. Для уменьшения скорости общего износа памяти имеет смысл как можно равномернее распределять количество циклов перезаписи по всем ячейкам EEPROM. Классический способ такого распределения — использование кольцевого буфера [13].

Идея этого метода заключается в перезаписи данных не по тому же адресу, а по новому, с некоторым сдвигом вперёд. Когда такой сдвиг становится невозможным из-за недостаточного объёма памяти, запись снова производится по первоначальному адресу. При этом, чтобы иметь доступ к данным, необходимо постоянно хранить их текущий адрес или счётчик циклов перезаписи, и определять адрес по его значению. В самом простом случае, если все данные хранятся в одном блоке, это можно сделать, умножив номер цикла на размер этого блока данных. Однако, если хранить их по фиксированному адресу в EEPROM, соответствующие ячейки будут изнашиваться быстрее, что сделает использование кольцевого буфера бессмысленным.

Один из способов эффективной реализации счётчика — использование двух кольцевых буферов: одного для данных, второго — для счётчика. Если при этом каждый раз при прохождении целого цикла записи (записи данных по изначальному адресу) для второго буфера, заполнять его нулевыми значениями, то после перезапуска микроконтроллера в этом буфере можно найти актуальное значение счётчика тривиальным образом — это будет последнее ненулевое значение в нём.

Другой способ основывается на использовании ещё одной особенности EEPROM микроконтроллеров: после стирания байта все его биты устанавливаются равными единице, и стандартная библиотека Arduino IDE предоставляет возможность устанавливать отдельным битам нулевое значение, без стирания всего байта. За счёт этого можно хранить счётчик в своеобразной унарной системе счисления: десятичное значение счётчика равно количеству нулей в его побитовой записи. Это позволит стирать ячейки EEPROM, хранящие счётчик, только при прохождении полного цикла записи в буфере данных.

Кольцевой буфер с последним из описанных механизмом эффективного счётчика реализует открытая библиотека EEPROMWearLevel [14] (опубликована под лицензией Apache 2.0 [15]). EEPROMWearLevel предоставляет интерфейс, аналогичный стандартной библиотеке, за исключением функции инициализации, в которую в данной библиотеке необходимо подать требуемое количество блоков EEPROM, дополняя этот интерфейс возможностью обращения к блокам данных по их индексам. Указанное отличие в функциях инициализации вызвано тем, что данная библиотека создаёт отдельный кольцевой буфер для каждого блока данных, при этом весь объём EEPROM делится на буферы поровну между всеми хранящимися блоками данных. Такое решение является неэффективным в случае хранения блоков разного размера. В худшем случае размер одного или нескольких блоков может превышать размеры выделенного буфера, однако такой случай обрабатывается библиотекой и приведёт к выводу соответствующей ошибки и отмене записи такого блока. Целостность других блоков при этом нарушена не будет.

Индексы блоков данных, используемые в этой библиотеке для доступа к ним, можно считать уникальными идентификаторами, описанными в требованиях, приведённых в начале главы. Однако необходимость общей инициализации с указанием суммарного количества блоков данных делает невозможным применение данной библиотеки в независимых программных модулях. Такой необходимости можно избежать, например, введя иерархическую структуру разделения EEPROM: сначала разделить весь объём EEPROM на крупные разделы (по одному или несколько на модуль), а затем внутри каждого из них — на кольцевые буферы, аналогично уже реализованному принципу. Чтобы гарантировать, что потребности всех модулей в использовании EEPROM учтены, необходимо:

- Либо проводить разделение EEPROM только после того, как все модули (посредством вызова некоторой специальной функции) сообщат библиотеке управления EEPROM их потребности в использовании EEPROM. Однако такой подход требует обязать конечного пользователя вызывать функцию инициализации библиотеки управления EEPROM строго после инициализации всех модулей, которым она необходима. А разработчиков этих модулей — вызывать функцию, описанную выше, при их инициализации. Такой подход, очевидно, является неудобным, так как требует от конечного

пользователя знания о том, необходимо ли используемым им модулям хранить данные в EEPROM.

- Либо реализовать возможности динамического добавления новых разделов EEPROM, при этом уменьшая размер уже созданных разделов. Такой подход лишён недостатков первого, однако его использование значительно усложнит работу библиотеки из-за необходимости уменьшения размеров каждого кольцевого буфера при добавлении нового раздела и, возможно, сдвига данных, если эти данные в момент добавления выходят за уменьшенные границы буфера.

Можно заключить, что существует возможность дополнения данной библиотеки таким образом, чтобы она удовлетворяла всем поставленным требованиям. В то же время данная библиотека не может напрямую использоваться с микроконтроллерами ESP8266, так как она содержит платформозависимый код, выполнение которого возможно только микроконтроллерами с архитектурой AVR, к которым ESP8266 не относится. Кроме того, микроконтроллеры ESP8266 используют иной тип EEPROM — flash память [16; 17], которая не позволяет стирать и перезаписывать отдельные байты, а только их большие группы (обычно от 512 байт) [18]. Основное назначение этой памяти в данных микроконтроллерах — хранение исполняемых программ, однако её часть специально выделена для хранения пользовательских данных. При этом в ходе анализа исходного кода стандартной библиотеки [16] для работы с EEPROM для ESP8266 было установлено, что для данных микроконтроллеров размер такой группы байт равен всему объёму flash-памяти, выделенному для пользовательских данных.

Таким образом, оказалось, что использование в программах для ESP8266 любых механизмов, подобных кольцевым буферам, не только не приводит к замедлению износа памяти, но и способно ускорять её износ из-за дополнительных операций перезаписи счётчиков и других метаданных.

2.1.6 Вывод

В ходе первого этапа работы были изучены особенности устройства и работы EEPROM. Были составлены требования к программному модулю для работы с EEPROM микроконтроллеров. С учётом этих требований были проанализированы существующие в открытом доступе решения. На основе анализа было принято решение о разработке собственного модуля, а также об использовании в нём стандартной библиотеки для работы с EEPROM и целесообразности заимствования некоторых механизмов из других библиотек.

2.2 Анализ существующих протоколов самоидентификации устройств Интернета вещей

2.2.1 Общее описание протоколов

Протокол самоидентификации — сетевой протокол, позволяющий устройствам:

1. Сообщать другим устройствам о своём присутствии в сети, сообщать свой локальный адрес.
2. Предоставлять в некотором виде описание своей структуры и возможностей.
3. Отвечать на поисковые запросы от других устройств в сети, если это устройство удовлетворяет определённым условиям (на имя, тип, состав и т.д.).

Кроме того, важным требованием является отсутствие необходимости конфигурации, т.е., войдя в какую-либо сеть, устройство сразу должно получить все возможности, описанные выше без дополнительной его настройки или настройки других устройств в сети.

На первом этапе данной работы были рассмотрены следующие существующие прикладные протоколы самоидентификации, удовлетворяющие указанным требованиям:

1. NetBIOS (Network Basic Input/Output System)
2. DNS-SD (DNS Service Discovery)
3. mDNS (multicast DNS)
4. SSDP (Simple Service Discovery Protocol)
5. SLP (Service Location Protocol)

Все рассматриваемые протоколы используют один и тот же транспортный протокол — UDP и одинаковый механизм поиска устройств, этот механизм продемонстрирован на рисунке 2.1. Основой данного механизма является отправка пакетов (датаграмм) на специальные мультитевещательные (англ. Multicast) адреса, т.е. всем устройствам, заявившим о своём вхождении в специальную широковещательную группу (на рисунке 2.1 передача таких пакетов обозначена пунктирными стрелками, а устройства, входящие в мультитевещательную группу жёлтыми кругами, остальные устройства — чёрными). Новое устройство, входя в сеть и вступая в мультитевещательную группу, отправляет устройствам в этой группе информацию о своём появлении в сети и своё описание, в форматах, зависящих от конкретного протокола (Рисунок 2.1.б). Далее, если какому-либо устройству в группе необходимо найти (т.е. узнать его адрес) другого устройства, оно отправляет на мультитевещательный адрес пакет, содержащий описание искомого устройства (Рисунок 2.1.в). Затем, те устройства, которые считают, что они удовлетворяют полученному описанию (на рисунке такие устройства обозначены зелёными кругами), отправляют ответ, содержащий адрес этого устройства и, возможно, его описание (Рисунок 2.1.г), причём делают это не с помощью мультитевещательной передачи данных, а напрямую по обратному адресу, указанному в поисковом пакете [1].

Различаются же эти протоколы между собой наборами используемых для передачи данных сетевых протоколов, возможностями для составления поисковых запросов и наборами дополнительных функций.

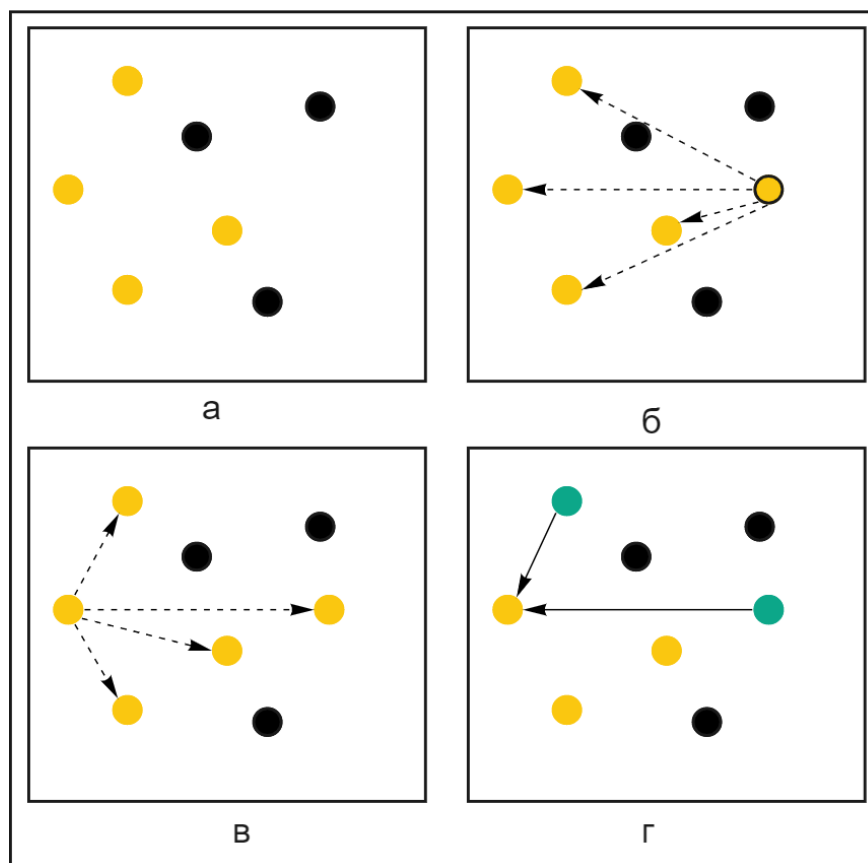


Рисунок 2.1 – Основной алгоритм поиска устройств

2.2.2 Протокол NetBIOS

Данный протокол был разработан в 1983 году и, в данный момент, реализующие его приложения по умолчанию входят во все операционные системы Windows и macOS, а также в некоторые системы семейства Linux. Стандарт NetBIOS описывает три службы, которые должны выполняться на устройствах: служба имён (NetBIOS-NS), служба сеанса (NetBIOS-SSN) и служба рассылки датаграмм (NetBIOS-DGM). Первая из них обеспечивает выделение имени для устройства, по которому его смогут найти другие устройства, а также разрешение конфликтов имён. Служба сеансов позволяет по имени устройства установить с ним сеанс связи и обмениваться данными по TCP. А служба датаграмм позволяет отправить UDP пакет устройству с конкретным именем или всем устройствам в сети [19].

2.2.3 Протоколы DNS-DS и mDNS

Эти протоколы чаще всего встречаются вместе, т.к. оба входят в набор технологий для сетей нулевой конфигурации Zeroconf, разработанный компанией Apple. Они имеют реализации для всех популярных операционных систем. Хотя в названии этих протоколов есть „DNS“ (англ. Domain Name System - система доменных имён), они не требуют наличия отдельных DNS-серверов и существования таблиц доменных имён, однако допускают их использование для уменьшения числа поисковых запросов в сети.

Оба этих протокола передают данные в виде стандартных DNS-пакетов. При этом, mDNS, в качестве искомого имени, передаёт имя устройства в специальном домене первого уровня .local. Очевидно, что имена в одной сети не должны повторяться, а также они должны быть заранее известны как устройствам, осуществляющим поиск, так и самим искомым устройствам [20; 21]. А DNS-DS, вместо имён устройств, использует специальные доменные имена, обозначающие различные сервисы, которые должны предоставлять искомые устройства (например, _printer._tcp.local для сервиса печати или _gaop._tcp.local для AirTunes и т.д.) [20; 22]. Предполагается, что такие имена сервисов берутся из специального реестра имён организации IANA [23].

2.2.4 Simple Service Discovery Protocol

Протокол SSDP (от Simple Service Discovery Protocol) является частью протокола UPnP (англ. Universal Plug and Play), разработанного форумом UPnP и в данный момент поддерживаемого организацией Open Connectivity Foundation. UPnP также реализован для большинства операционных систем, а в некоторые даже встроен. Протокол SSDP позволяет искать устройства по специальным уникальным идентификаторам, типам устройств, а также идентификаторам и типам сервисов, предоставляемых устройствами. Типы устройств и сервисов разделены на домены, каждый производитель устройств в праве задавать свой домен и имена в нём.

По протоколу SSDP данные передаются в текстовом виде в формате запросов HTTP, но без тела запроса. Т.е. каждый передаваемый пакет состоит из заголовка, описывающего этот пакет, и набора пар имя-значение (заголовков HTTP). Документация протокола описывает обязательные заголовки, которые должны содержаться в пакете, оставляя возможность добавлять любые другие, если только их имена уникальны внутри одного пакета [24].

2.2.5 Service Location Protocol

Данный протокол присваивает каждому устройству одну или несколько из следующих ролей: user agent - устройство, которое ищет сервис, service agent - устройство, предоставляющее какие-либо сервисы, и directory agent, кэширующий информацию о сервисах и устройствах, их предоставляющих. Устройства последнего типа могут отсутствовать в сети, однако, если они есть, для поиска сервисов обязаны использоваться они, а не прямой поиск.

Важной отличительной чертой данного протокола является возможность задавать сервисам не только имена, но и произвольные атрибуты, которые могут изменяться со временем. Для формирования пакетов данных Service Location Protocol (SLP) использует свой собственный бинарный протокол, позволяющий эффективно кодировать и передавать информацию о сервисах и устройствах. С помощью SLP возможно проводить поиск сервисов по именам, типам, областям (англ. Scopes), а также предикатам в формате LDAP Search Filters, зависящим от значений атрибутов сервиса.

Помимо этого, SLP включает в себя механизмы защиты подлинности пакетов на основе открытых ключей. В данный момент протокол не имеет особой популярности, так, в частности, компания Apple, продвигавшая его ранее, перешла в своих продуктах на стек Zeroconf [25].

2.2.6 Сравнение протоколов самоидентификации

На основании анализа протоколов была построена сравнительная таблица (Таблица 2.1).

Таблица 2.1 – Сравнение протоколов самоидентификации

Критерий	NetBIOS	DNS-SD	mDNS	SSDP	SLP
Возможность обмена дополнительной информацией	+	-	-	+	-
Возможность поиска по именам	+	+	+	+	+
Возможность поиска по типам	-	+	-	+	+
Возможность поиска с помощью предикатов	-	-	-	-	+
Возможность поиска не только устройств, но и сервисов	-	+	-	+	+
Наличие дополнительных механизмов защиты	-	-	-	-	+
Наличие механизмов уведомления о вхождении устройства в сеть	+	-	-	+	+
Бинарный протокол представления данных	-	+	+	-	+
Год появления актуальной редакции протокола	1987	2013	2013	2015	1999

Помимо критериев, приведённых в таблице, при выборе протокола для реализации учитывалось количество особенностей протокола, которые не являются необходимыми для достижения цели данной работы, но которые необходимо реализовать для соответствия протоколу. С учётом отсутствия других явных преимуществ, это исключает из выбора NetBIOS, т.к., для возможности обмена

данными помимо простого поиска, он требует реализации массы дополнительных функций, связанных, в частности, с поддержкой механизма сеансов.

Протоколы SLP, DNS-SD и mDNS не обеспечивают возможность передачи дополнительной информации, не выходя за рамки протокола, в соответствии с поставленной целью работы, возможность использования протокола для взаимодействия с помощью онтологий является критичной. Поэтому становится необходимым, в случае выбора одного из этих протоколов, разработать некоторый дополнительный механизм поверх него для такого взаимодействия. Очевидно, в случае SLP и DNS-SD, это не имеет особого смысла, т.к. по другим критериям они также проигрывают оставшимся конкурентам.

Каждый из оставшихся протоколов SLP и SSDP обладает и значительными преимуществами, и недостатками. Однако, преимущества протокола SLP являются менее значимыми, потому что:

- Возможность использовать для поиска сервисов предикаты и, соответственно, возможность задавать сервисам произвольные атрибуты являются достаточно сложными в реализации и не менее сложными в интеграции в платформу SciVi. При этом, функциональность, связанную с атрибутами и проверкой их значений, можно переложить на аппарат онтологий и механизмы их обработки.
- Вопросы безопасности на данном этапе не входят в рассмотрение и остаются за рамками исследования.
- Экономией объёма передающихся данных также можно пренебречь, т.к. обмен данными по протоколу самоидентификации осуществляется только в моменты поиска устройств, т.е. достаточно редко.

Таким образом, недостатки протокола SSDP являются незначительными или устранимыми на уровне обработки онтологий, а его преимущества идеально ложатся на рассматриваемую предметную область.

2.2.7 Вывод

В результате сравнения можно сделать вывод, что протокол SSDP - единственный из рассмотренных, который позволяет, полностью оставаясь в рамках одного протокола, удобно для использования совместить в одном решении поиск устройств и передачу онтологий для дальнейшей обработки. Следовательно, для последующей реализации следует выбрать именно его.

3 Разработка библиотеки менеджера EEPROM

3.1 Первая версия библиотеки

После анализа существующих решений, была разработана собственная библиотека. Её исходный код находится в открытом доступе в специальном репозитории, вместе с примерами использования и тестирующими библиотеку программами для микроконтроллеров [26]. Разработанная библиотека основывалась только на библиотеке EEManager, которая на тот момент казалась наиболее удачной из рассмотренных библиотек. Однако после реализации и тестирования было установлено, что логика созданной библиотеки работает в соответствии с поставленными требованиями, но её интерфейс проявил является крайне неудобным и недостаточно высокоуровневым. Общий вид этого интерфейса был продиктован идеей сохранить частичную обратную совместимость с библиотекой EEManager и переиспользовать её код, что привело к предоставлению пользователю недостаточно высокоуровневых функций.

Основным понятием, которой оперировала данная библиотека является EEPROM-переменная — сущность (программный объект), описывающая некоторый единый блок пользовательских данных, хранимых в EEPROM. В дальнейшей работе также использовалось это понятие.

3.2 Уточнение требований к разрабатываемой библиотеке

После признания первой версии библиотеки неудачной, было принято решение о разработке второй со следующими изменениями:

1. Объединить одной сущностью данные в оперативной памяти и объекты, описывающие связанные с ними данные в EEPROM. При использовании такой сущности первоначальное считывание значения из EEPROM (или запись в него значения по умолчанию) возможно производить в конструкторе класса, соответствующего такой сущности. Дальнейшее же получение из неё пользовательских данных и их обновление в EEPROM реализовать за счёт переопределения операторов, аналогично классу EEPROMVar из библиотеки EEPROMEx. Это позволит одновременно (с точки зрения пользователя) работать и с данными в оперативной памяти, и с их представлением в EEPROM.
2. Реализовать механизм отложенной записи для всего EEPROM в целом, а не для каждой EEPROM-переменной по отдельности.
3. Завязать такой механизм на количестве обращений на запись, а не на реальном времени, как в EEManager.
4. Реализовать возможность немедленного обновления данных в EEPROM в случае необходимости.

На основе дополненного списка требований к библиотеке можно составить диаграмму прецедентов работы с ней. Она представлена на рисунке 3.1.

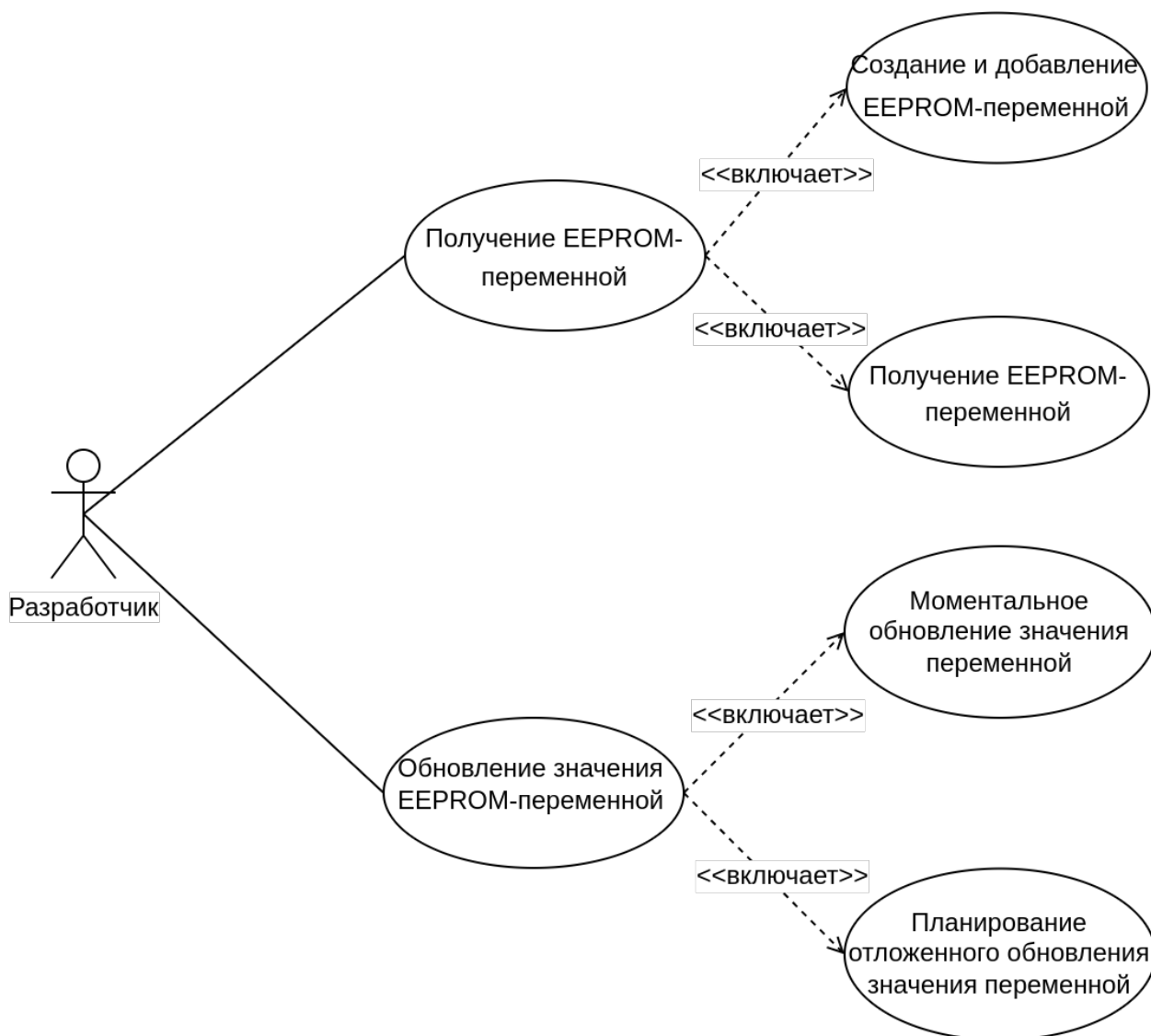


Рисунок 3.1 – Диаграммы прецедентов использования библиотеки

На диаграмме отсутствует прецедент удаления переменной в связи с тем, что, как писалось выше, основная цель применения EEPROM в программах для микроконтроллеров — сохранение состояния устройства между его запусками. Как правило, для запуска микроконтроллера с одной и той же программой необходимы одни и те же данные, следовательно, удалять их нет необходимости.

3.3 Разработка структуры библиотеки

3.3.1 Общая структура библиотеки

В соответствии с общепринятым стилем, библиотека должна быть написана с использованием объектно-ориентированной парадигмы программирования. Основное понятие для разрабатываемой библиотеки — EEPROM-переменная, следовательно, в ней должен содержаться класс, соответствующий этому понятию. Кроме того,

необходим отдельный класс, оперирующий EEPROM в целом. Назовём его классом менеджера памяти. К этому классу должны обращаться EEPROM-переменные для их поиска или добавления в память и обновления их значений.

Важно учитывать, что стандартные библиотеки EEPROM для различных микроконтроллеров обычно работают схожим образом, но всё же могут иметь отличия. В частности незначительно отличаются такие библиотеки и для целевых платформ данной работы — Arduino и ESP8266. В следствии этого необходимо вынести все операции, использующие функции этих библиотек в отдельную сущность с тем, чтобы была возможность легко её изменить для использования других микроконтроллеров. Удобнее всего это сделать, определив абстрактный класс, предоставляющий унифицированные методы-обёртки над стандартными функциями, и создав конкретные реализации такого класса для различных микроконтроллеров. Этот же класс должен реализовывать механизмы уменьшения износа памяти, так как они могут быть различными для различных микроконтроллеров. Такой подход также позволит легче адаптировать разрабатываемую библиотеку для использования с абсолютно другими микроконтроллерами.

Для хранения EEPROM-переменных было решено использовать механизм, схожий с механизмами, используемыми в файловых системах. В описание каждой переменной включается адрес следующей, либо специальное значение, обозначающее, что данная переменная является последней в разделе. Адрес первой же переменной является константой и не изменяется при перезапуске устройства.

Также в энергонезависимой памяти стоит хранить специальное проверочное значение, показывающее, был ли менеджер инициализирован в EEPROM, и, в случае, если не был, инициализировать его. Причём имеет смысл объявить два таких значения, оба из которых свидетельствуют о том, что менеджер инициализирован, но одно из них используется в случае, если в EEPROM была объявлена хотя бы одна переменная, в противном случае, используется второе значение. Использование такого механизма необходимо для избежания чтения „мусорных“ данных, находящихся в EEPROM до первого использования менеджера. Также изменение этого значения в последующих версиях библиотеки можно использовать для принудительной переинициализации менеджера в случае изменения структуры хранения данных при выходе новой версии.

Принудительная переинициализация менеджера также требуется и при перепрошивке микроконтроллера в противном случае в энергонезависимой памяти будут накапливаться неиспользуемые переменные от предыдущих программ, загруженных в устройство. Arduino IDE автоматически создаёт константы, описывающие время компиляции программы и доступные из её кода. Это время можно также сохранять в EEPROM и при запуске микроконтроллера проверять, совпадает ли время компиляции текущей программы с сохранённым временем. Если они отличны, переинициализировать менеджер, считая, что больше в EEPROM нет сохранённых переменных. Для хранения времени компиляции оптимально

использовать формат Unix time — количество секунд, прошедших с момента времени 00:00:00 1 января 1970 года, обычно хранящееся в виде 4-х байтного знакового целого числа. Такой формат является наилучшим, так как позволяет хранить время используя небольшой объём памяти, что является критичным, с учётом ограниченности объёмов EEPROM.

Логическая схема размещения данных в EEPROM показана на рисунке 3.2.

Для идентификации EEPROM-переменных было решено использовать произвольные строковые выражения — имена. Причём хранение в энергонезависимой памяти имён целиком может привести к её большому расходу, если имена будут достаточно длинными. Чтобы избежать этого, было принято решение, хранить в памяти результат вычисления некоторой хеш-функции от имени переменной. Такие значения всегда имеют фиксированную длину, в следствии чего объём используемой памяти будет всегда предсказуем.

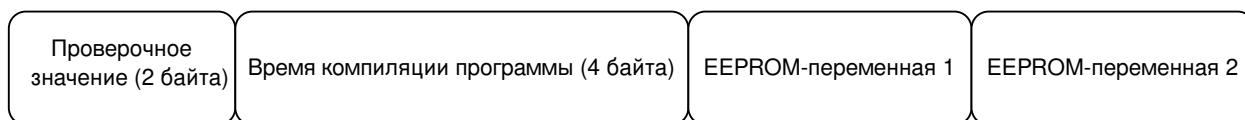


Рисунок 3.2 – логическая схема размещения данных в EEPROM

3.3.2 Диаграмм классов библиотеки

По итогам разработки структуры библиотеки была создана её диаграмма классов представленная на рисунке 3.3.

3.3.3 Внешний интерфейс библиотеки. Класс EEPROM-переменной

Пользователю для использования должен быть доступен всего один класс EEPROMVar, описывающий EEPROM-переменную. Данный класс является шаблонным, и его параметр типа определяет тип хранящихся пользовательских данных.

Этот класс имеет следующие методы

1. Конструктор. Он принимает в качестве аргументов имя переменной и её значение по умолчанию. Если переменная с таким именем уже существует, её значение считывается из энергонезависимой памяти и сохраняется в создаваемый объект, иначе в EEPROM создаётся новая переменная и в неё записывается значение по умолчанию.
2. Update — запланировать запись нового значения в EEPROM. Метод имеет две перегрузки: не принимающую аргументов — для записи текущего значения пользовательских данных, и принимающая новое значение для этих данных, которая планирует запись этого нового значения.
3. UpdateNow — метод для немедленного обновления значения в EEPROM. Так же имеет две перегрузки аналогично предыдущему методу.

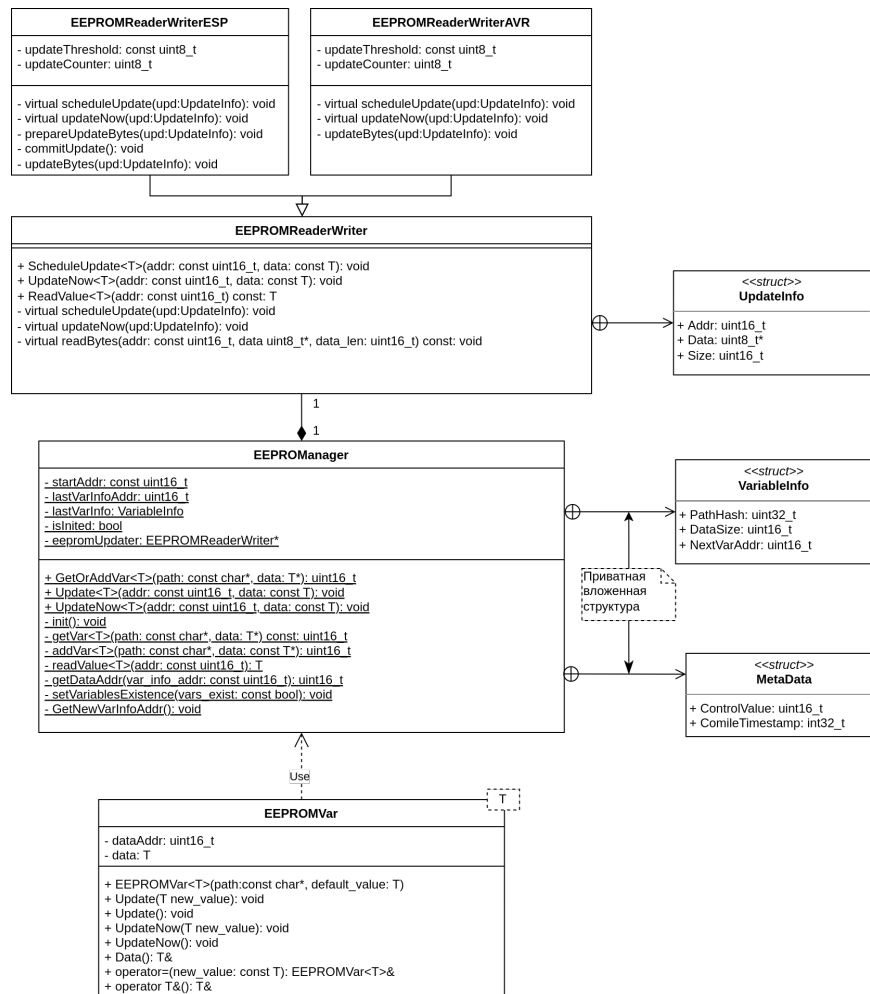


Рисунок 3.3 – Диграмма классов библиотеки

4. `Data` — метод, возвращающий по ссылке пользовательские данные, хранящиеся объектом.
5. Оператор присваивания (`„=“`). Работает аналогично методу `Update`, принимая новое значение пользовательских данных в качестве правостороннего значения. В отличие от аналогичного оператора класса `EEPROMVar` библиотеки `EEPROMEx`, данный сам производит (отложенную) запись новых данных, без необходимости отдельного вызова метода записи.
6. Оператор приведения объекта к типу хранящихся в нём пользовательских данных. Возвращает текущее значение хранимых данных, причём делает это по ссылке, как и метод `Data`.

В виде полей данный класс хранит сами пользовательские данные и адрес в EEPROM для их обновления.

Можно привести следующий пример использования класса `EEPROMVar`:

```

1 EEPROMVar<uint16_t> int_var("int_var", 1);
2
3 int_var = int_var + 1;
4
5 int_var.UpdateNow();

```

```

6
7 EEPROMVar<UserClass> custom_data("custom_data", UserClass());
8
9 custom_data.Data().UpdateField();
10
11 custom_data.Update();
12
13 custom_data = custom_data.Data().UpdateField();

```

1. Сначала пользователь объявляет EEPROM-переменную. Её значение либо считывается из EEPROM, либо наоборот сохраняется в него.
2. Пользователь изменяет значение EEPROM-переменной. Значение в объекте изменяется немедленно, в энергонезависимой памяти — с помощью механизма отложенной записи.
3. При необходимости, пользователь может в любой момент немедленно обновить значение в EEPROM.
4. Пользователь может объявить EEPROM-переменную, хранящую более данные более сложных пользовательских типов (классов или структур). В примере — класс UserClass.
5. Пользователь получает хранящееся значение сложного вызовом метода Data и затем вызывает у него некоторый метод, изменяющий значения его полей — UpdateField. В таком случае изменятся их значения изменятся только в оперативной памяти, так как, в отличие от пункта выше, оператор „=“ не вызывается. В таком случае для обновления значения в EEPROM пользователь может идти различными путями:
 - Вручную вызвать метод Update или UpdateNow у объекта EEPROM-переменной.
 - При возможности реализовать метод UpdateField таким образом, что бы он возвращал ссылку на изменённый объект. В таком случае можно его использовать как показано в трюке 13 примера.
 - Также существует вариант создания временный объект того же типа, что и данные, хранимые EEPROM-переменной, с последующими изменением этого объекта и записью в EEPROM уже этого, временного, объекта. Однако такой вариант проигрывает предыдущим и практически не имеет смысла.

3.3.4 Класс менеджера EEPROM

Данный класс (EEPROMManager) предоставляет методы для поиска переменных в EEPROM по их именам, создания новых переменных и обновления хранимых ими значений. EEPROMManager не используется пользователем напрямую. Его методы вызываются объектами класса описанного выше.

Во-первых данный класс содержит две встроенные структуры:

- **VariableInfo.** Эта структура хранит метаинформацию о EEPROM-переменной в EEPROM: хеш-значение (CRC32) имени переменной, размер хранящихся пользовательских данных, адрес в EEPROM описания следующей переменной (при отсутствии следующей переменной равен нулю). Физически в энергонезависимой памяти всегда хранится такая структура, а начиная со следующего адреса сами пользовательские данные, описываемые структурой.
- **MetaData** — Общие мета-данные об EEPROM. Также сохраняет в энергонезависимой памяти и включает в себя описанные в пункте 3.3.1 специальное проверочное значение и временную метку времени компиляции загруженной в микроконтроллер программы.

К публичным методам класса относятся:

- **GetOrAddVar.** Метод находит переменную в EEPROM по её имени или создает новую.
- **Update** — Планирует обновление данных в EEPROM.
- **UpdateNow** — Немедленно обновляет пользовательских данные.

Также в классе содержатся приватные методы:

- **init** — Инициализирует менеджер. Читает мета-данные о сохраненных в EEPROM данных или записывает эту информацию, если это первый запуск.
- **getVar** — Получает адрес и данные переменной по её имени.
- **addVar** — Добавляет новую переменную в EEPROM и возвращает её адрес.
- **readValue** — Считывает данные пользовательских типов из EEPROM.
- **getDataAddr** — Получает адрес данных переменной по адресу её мета-информации.
- **setVariablesExistence** — Записывает в EEPROM информацию о добавлении первой переменной, если до этого переменных не существовало.
- **getNewVarInfoAddr** — Получает адрес для записи новой переменной.

И поля:

- **startAddr** — Начальный адрес EEPROM для управления данными.
- **lastVarInfoAddr** — Адрес мета-данных последней записанной переменной.
- **lastVarInfo** — Сами такие мета-данные. Используются при добавлении новой переменной: в них обновляется адрес следующей переменной, так как теперь она существует.
- **isInited** — Значение, указывающее, был ли инициализирован менеджер.

3.3.5 Хранение и поиск переменных в EEPROM

В соответствии с описанием структуры **VariableInfo**, в EEPROM данные о каждой переменной хранятся в виде, представленном в таблице 3.1.

Таблица 3.1 – Мета-данные EEPROM-переменной

Размер данных	Сдвиг от начального адреса	Название поля	Описание
4 байта	0 байт	nameHash	Хэш-значение имени переменной. В данный момент используется хэш-функция CRC32 [27].
2 байта	4 байта	dataSize	Размер данных, хранящихся в переменной.
2 байта	6 байт	nextVarAddr	Адрес следующей переменной.
Зависит от типа данных	8 байт	data	Пользовательские данные, хранимые переменной.

3.3.6 Класс для чтения и записи данных

Данный класс (EEPROMReaderWriter) является прослойкой между классом-менеджером и стандартной библиотекой для работы с EEPROM. Он призван изолировать их конкретные реализации от менеджера, предоставляя ему унифицированный интерфейс из трёх методов:

- ScheduleUpdate — Запланировать обновление данных произвольного типа по некоторому адресу.
- UpdateNow — Немедленное обновление данные произвольного типа в EEPROM.
- ReadValue — Чтение данных произвольного типа из EEPROM.

Каждый из таких методов переводит указатель на полученные данные произвольного типа в тип указателя на отдельный байт и в таком виде передаёт эти данные в соответствующий приватный метод. Такие методы являются абстрактными и требуют реализации для конкретных микроконтроллеров. Список абстрактных методов:

- scheduleUpdate — Запланировать обновление данных в EEPROM.
- updateNow — Немедленно обновить данные в EEPROM.
- readBytes — Чтение данных из EEPROM.

В рамках проведённой работы были разработаны реализации этого класса для целевых микроконтроллеров — платформы Arduino и ESP8266.

3.4 Реализация библиотеки

Библиотека была реализована в соответствии с представленной диаграммой. В данном отчёте не приводятся детали реализации отдельных методов, однако исходный код разработанной библиотеки находится в открытом доступе в специальном репозитории, вместе с примерами использования и тестирующими

библиотеку программами для микроконтроллеров [28]. Код также содержит комментарии-аннотации, необходимые для генерации документации средствами Doxygen [29].

В приложении А представлен пример минимальной программы для микроконтроллеров, использующей разработанную библиотеку.

Работа библиотеки была успешно протестирована на микроконтроллерах серии ESP8266 и платформы Arduino. Средства библиотеки корректно решают все поставленные перед ней задачи.

4 Разработка библиотеки для сетевой самоидентификации периферийных устройств

4.1 Требования к модулю

4.1.1 Общие требования

Все требования в этом подпункте основаны на документации к протоколу SSDP[24]. В большинстве случаев (в том числе в условиях решаемой задачи) существует чёткое разделение устройств на производящие поиск и искомые, для встраиваемых устройств достаточно реализовать часть протокола, описывающую поведение искомого устройства. А именно:

- Отправка пакетов-уведомлений, о вхождении устройства и его сервисов в сеть.
- Приём и обработку пакетов, передающих поисковые запросы.
- Ответ на поисковой запрос, в случае если устройство соответствует условию поиска.
- Отправка пакетов-уведомлений перед выходом устройства из сети.

Подробное описание структуры указанных пакетов, а также требований к их обработке и отправке приведено в пункте 6 данной главы.

Помимо этого примем ещё одно допущение. Согласно стандарту SSDP, устройство может иметь в своём составе ещё несколько встроенных устройств, которые, в свою очередь, могут предоставлять свои отдельные сервисы. Однако, выполнение данного модуля предполагается на устройствах Интернета вещей, которые, в большинстве случаев, имеют достаточно простую структуру и не несут в себе отдельных встроенных устройств. Поэтому для данного модуля является допустимой поддержка только головного устройства, которое само предоставляет некоторые сервисы, но не имеет других встроенных устройств.

Разрабатываемый программный модуль должен позволять настроить и в будущем считать все атрибуты устройства, которыми оно должно обладать в соответствии со стандартом SSDP, а именно:

1. Тип устройства в установленном стандартом формате.
2. Перечень сервисов, предоставляемых устройством. Он может быть пустым.
3. Уникальный идентификатор устройства в специальном формате.
4. Ссылка на веб-страницу, содержащую описание устройства в установленном формате.
5. Его человекочитаемое имя.
6. Ссылку на страницу веб-страницу, как-либо презентующую данное устройство.
7. Название его модели.
8. Номер модели.

9. Серийный номер устройства.
10. Название его производителя.
11. Ссылка на веб-страницу, описывающую производителя.
12. Номер HTTP порта устройства, обратившись по которому можно получить веб-страницу, описанную в пункте 4 данного списка.

Для поиска устройства необходимыми являются только атрибуты под номерами 1–3, поэтому, если их значения не будут заданы пользователем, они должны быть проинициализированы значениями по умолчанию. Остальные атрибуты используются для формирования описания устройства на веб-странице из пункта 4 списка выше. Они не чувствуют в процессе поиска и могут иметь нулевые (пустые) значения.

4.1.2 Требования платформы SciVi

Аналогично библиотеке менеджера EEPROM, разрабатываемый модуль должен быть представлен библиотекой классов языка C++, использовать стандартный набор библиотек Ardiono IDE и работать на микроконтроллерах ESP8266. ESP8266 имеет в своём составе встроенный блок для подключения к сетям Wifi, а набор стандартных библиотек для этих устройств включают достаточно высокоуровневые готовые средства для управления блоком WiFi и сетевого взаимодействия с другими устройствами. В том числе для приёма и отправки UDP пакетов на мультивещательные адреса. Устройства платформы по Ardiono не имеют встроенной возможности для сетевой коммуникации. Обычно для этого к ним подключаются другие устройства, например, EPS8266. В следствии этого, разрабатываем модуль не обязан выполняться на других микроконтроллерах и может использовать возможности и средства специфичные только для ESP8266.

По протоколу SSDP данные передаются в виде текста в формате HTTP заголовков, поэтому для передачи онтологий программный модуль должен содержать механизм добавления к отправляемым пакетам дополнительных заголовков. Кроме того, пользователю должна быть предоставлена возможность настройки состава добавляемых заголовков, в зависимости от атрибутов отправляемого пакета: от его типа (уведомление или ответ на поисковой запрос) и типа устройства или сервиса, информацию о котором он передаёт.

После сжатия, объём передаваемых платформой SciVi антологии обычно не превышает нескольких сотен байт, а UDP-пакеты, которые используются протоколом SSDP для передачи данных, имеют ограничение на размер в 65,535 байт. Таким образом, одного UDP пакета должно быть достаточно для передачи устройством всех необходимых данных, включая онтологию. Однако сохраняется теоретическая возможность выхода за пределы указанного ограничения, и подобные ситуации необходимо специально отслеживать.

4.2 Анализ существующих реализаций протокола

На момент выполнения работы для ESP8266 уже существовала реализация протокола SSDP в виде библиотеки для языка C++ [30]. Однако она реализует устаревшую версию протокола, причём не полностью и с нарушениями стандарта. Данная библиотека предоставляет возможность поиска только по типу устройства и его идентификатору, не имея возможности работать с отдельными сервисами. Также она некорректно обрабатывает ряд редких, но возможных ситуаций, связанных с получением нескольких поисковых запросов за малый промежуток времени, и нарушает требования стандарта SSDP по отправке пакетов-уведомления о входе устройства в сеть и выходе из неё. Помимо этого, данная библиотека не предоставляет удобной возможности для настройки атрибутов устройства, требуя от пользователя самостоятельно приводить их к форматам, соответствующим стандарту SSDP.

Также стоит отметить, что структура описанного выше класса не позволяет в значительном объёме переиспользовать его с помощью наследования или вызова существующих методов, однако позволяет переиспользовать отдельные фрагменты самого кода (что не противоречит лицензии, под которой данная библиотека распространяется — MIT).

4.3 Разработка структуры библиотеки

С учётом стиля большинства подобных библиотек, общих современных тенденций, а также уже существующей реализации, оптимальным вариантом является создание модуля в объектно-ориентированном стиле. Разрабатываемая библиотека должна решать одну конкретную задачу, и в ней нет необходимости создавать новых типы данных, реализация которых уже не присутствовала бы в стандартных библиотеках. Поэтому всю функциональность библиотеки можно поместить в один класс.

Причём имеет смысл сохранить обратную совместимость этого класса с соответствующим классом существующей библиотеки, с целью создания возможности лёгкого перехода с неё на более полную реализацию в уже созданных проектах. Этот класс должен содержать следующие публичные методы:

1. Методы для установки и получения значений каждого из атрибутов устройства, описанных в пункте 2.2.1, а также настроек самой библиотеки: времени жизни, отправляемых ей пакетов (англ. Time To Live, TTL), временного интервала повтора уведомлений (см пункт 2.5).
2. Метод для инициализации класса, запуска его работы. Данный метод стоит реализовать отдельно, а не в виде конструктора, т.к. при программировании микроконтроллеров часто используются глобальные переменные (причём именно в виде значений, а не указателей на них), которые затем в специальной функции инициализируются в необходимом программисту порядке.

3. Метод завершения работы объекта и очистки его ресурсов. Использование только деструктора для этих целей также будет неправильным из-за частого использования глобальных переменных.

С учётом использования ООП подхода, механизм расширения функциональности модуля, указанный в требованиях к нему, можно удобно для использования реализовать с использованием наследования. Для этого необходимо определить доступные для переопределения в классах-потомках методы, которые бы вызывались автоматически перед отправкой SSDP пакетов (по одному такому методу на каждый тип SSDP пакетов) для возможности добавления в пакет новых заголовков из этих методов. Кроме того, необходим сам метод добавления нового заголовка в текущий отправляемый пакет. А для большей гибкости ещё и метод, возвращающий значение, которое бы показывало информацию о чём содержит текущий отправляемый пакет (о типе устройства, сервисе и т.д.), для возможности выбора различных действий в зависимости от этого значения.

Главным частным методом является метод обработки получаемых пакетов. Этот метод должен вызываться с определённой периодичностью, и если между его вызовами был получен UDP пакет, проанализировать его и в зависимости от результата анализа отправить ответ или не делать ничего. Также необходимо создать несколько вспомогательных методов: метод, реализующий отправку пакетов другим устройствам, метод, генерирующий значения заголовков для ответов и т.д.

Для периодического вызова метода, описанного выше, существуют 2 очевидных механизма:

- Первый — использующий программный таймер, который раз в заданный промежуток времени вызывал бы этот метод. Класс, предоставляющий возможности такого таймера, содержится в одной из библиотек стандартного набора. Использование такого механизма не требует никаких действий после инициализации объекта и позволяет упростить написание кода, использующего разрабатываемую библиотеку, однако использование таймера может мешать выполнению других частей прошивки. К тому же, оно не соответствует наиболее распространённому среди разработчиков программ для подобных микроконтроллеров стилю вызова подобных методов.
- Альтернативный механизм является более простым и привычным большинству программистов микроконтроллеров. Он заключается в ручном вызове этого метода в специальной функции, циклически повторяющейся в течение всего времени работы устройства.

С целью придания библиотеке большей гибкости предлагается реализовать оба этих механизма, а также дополнительный публичный метод, переключающий объект между этими механизмами.

4.4 Разработка библиотеки

4.4.1 Хранение данных

Большая часть данных, хранящихся в объекте разрабатываемого класса, являются текстовыми атрибутами устройства. И, так как стандарт SSDP накладывает жёсткие ограничения на длину каждого из этих атрибутов, имеет смысл отказаться от динамических строк, а хранить и обрабатывать значения этих атрибутов в виде массива символов. Такое решение обезопасит память от возможного переполнения и увеличит скорость обработки текстовых значений.

Название типов устройств и сервисов должны описываться комбинацией трёх строковых значений: домена имён, самого имени типа и версии, однако, в дальнейшей обработке участвует только комбинация этих значений, и никогда каждое отдельно. Поэтому есть смысл один раз, при получении этих 3 значений для каждого из имён устройства и сервисов, составить из них итоговую комбинацию и далее хранить именно эту комбинацию в виде единого текстового значения в необходимом формате. Описание данных форматов приведено ниже согласно документации к протоколу SSDP.

Формирование пакетов

Любой SSDP пакет содержит:

1. Стартовую строку, описывающую сам пакет.
2. Обязательные заголовки с фиксированными значениями.
3. Обязательные заголовки с значениями, зависящими от имён устройства и его сервисов.
4. Ноль или более необязательных заголовков.

Как упоминалось ранее, все SSDP пакеты делятся на:

1. Пакеты-уведомления о вхождении в сеть
2. Пакеты-уведомления о выходе из сети
3. Поисковые пакеты
4. Пакеты-ответы на поисковые запросы

Структура пакетов разных видов может отличаться. Рассмотрим пакеты-уведомления, их структура представлена на рисунке 4.1.

```
NOTIFY * HTTP/1.1
HOST: 239.255.255.250:1900
CACHE-CONTROL: max-age = seconds until advertisement expires
LOCATION: URL for UPnP description for root device
NT: notification type
NTS: ssdp:alive
SERVER: OS/version UPnP/2.0 product/version
USN: composite identifier for the advertisement
BOOTID.UPNP.ORG: number increased each time device sends an initial announce or an update message
CONFIGID.UPNP.ORG: number used for caching description information
SEARCHPORT.UPNP.ORG: number identifies port on which device responds to unicast M-SEARCH
```

Рисунок 4.1 – Состав пакета-уведомления

Стартовая строка таких пакетов всегда имеет вид „NOTIFY * HTTP/1.1“. Затем идут обязательные заголовки:

- Заголовок HOST, значение которого всегда 239.255.255.250:1900, т.к. IP-адрес 239.255.255.250, представляющий мультивещательную группу, и порт номер 1900 закреплены за данным протоколом.
- Заголовок CACHE-CONTROL со значением „max-age =“ и целым числом, указывающим, в течении какого времени без повторного уведомления устройство или сервис можно считать оставшимися в сети.
- Заголовок LOCATION, хранящий значение веб-адреса страницы с описанием устройства.
- Заголовок NT, он описывает тип устройства или сервиса, о котором оповещает данный пакет. Данный заголовок может принимать различные значения, их формат представлен на рисунке 4.2.
- Заголовок NTS определяет тип сообщения, его значение равно „ssdp:alive“ для уведомлений первого типа и „ssdp:byebye“ для второго.
- Заголовок SERVER описывает систему, с которой пакет был отправлен, для разрабатываемой библиотеки его значение формируется в формате „Arduino/1.0 UPNP/2.0 <название модели устройства>/<номер модели устройства>“ (см. рисунок 4.2).
- Заголовок USN является комбинацией уникального идентификатора устройства в формате „uuid:<значение идентификатора>“ и значения заголовка NT, записанных через два символа двоеточия.
- Заголовок CONFIGID.UPNP.ORG — некоторое положительное 31-битное число, изменяющееся каждый раз, когда меняется состав устройства и его сервисов, либо версия одного из них. Отслеживание подобных изменений является неоправданно сложной в рамках разработки данной библиотеки, поэтому по умолчанию значение этого заголовка будет оставаться нулевым, однако, класс предоставляет возможность пользователю самому вычислять это значение и устанавливать с помощью соответствующего метода.
- Заголовок BOOTID.UPNP.ORG — аналогичное предыдущему значение, но которое должно изменяться при каждом подключении устройства в сети. Данное значение также не обрабатывается библиотекой автоматически.

	NT	USN ^a
1	<u>upnp:rootdevice</u>	uuid:device-UUID:: <u>upnp:rootdevice</u>
2	uuid:device-UUID ^b	uuid:device-UUID (for root device UUID)
3	urn:schemas-upnp-org:device:deviceType:ver or urn:domain-name:device:deviceType:ver	uuid:device-UUID::urn:schemas-upnp-org:device:deviceType:ver (of root device) or uuid:device-UUID::urn:domain-name:device:deviceType:ver

Рисунок 4.2 – Возможные значения заголовков NT и USN (подчёркнутый текст — ключевые слова, курсивный — значения, задаваемые производителем)

Согласно стандарту SSDP, устройство должно отправить ssdp:alive уведомление о головном устройстве и о каждом предоставляемом им сервисе при входе в сеть и аналогично ssdp:byebye уведомления при выходе из неё.

Стартовая строка поисковых запросов отличается только HTTP методом: „M-SEARCH“ вместо „NOTIFY“, а наборы заголовков в таком пакете могут быть очень разнообразными, минимальный состав такого пакета приведён на рисунке 4.3. Важными для поиска в них являются только:

- MAN, значение которого должно быть „ssdp:discover“
- MX — целое число секунд, в течении которого после получения этого пакета, в случайный момент времени устройство должно послать ответ, в случае, если оно удовлетворяет условию поиска.
- ST — цель (условие) поиска, значение формируется аналогично значению заголовка NT пакета-уведомления, либо может являться специальным значением „ssdp:all“.

Если значение заголовка ST совпадает со значением заголовка NT устройства или его сервиса, устройство обязано послать ответный пакет.

```
M-SEARCH * HTTP/1.1
HOST: hostname:portNumber
MAN: "ssdp:discover"
ST: search target
USER-AGENT: OS/version UPnP/2.0 product/version
```

Рисунок 4.3 – Состав минимального поискового пакета

Стартовая строка пакета, являющегося ответом на поисковый запрос, строится по принципу такой строки в HTTP ответе, и её значение равно „HTTP/1.1 200 OK“. Такой пакет должен содержать следующие заголовки:

- CACHE-CONTROL, LOCATION, SERVER, BOOTID.UPNP.ORG, CONFIGID.UPNP.ORG со значениями, аналогичными пакетам-уведомлениям.
- ST, значение которого совпадает со значением одноимённого заголовка в поисковом пакете, на который производится ответ.
- USN, со значением, аналогичным значению этого заголовка в пакете-уведомлении.
- EXT с пустым значением. Этот заголовок необходим для обратной совместимости со старой версией протокола.

4.4.2 Анализ получаемых пакетов

Для синтаксического разбора получаемых поисковых пакетов был использован автомат состояний. На диаграмме его состояний (Рисунок 4.4) состояние „Обработка ошибки в пакете“ означает, что поступивший пакет некорректен или его цель поиска не удовлетворяется данным устройством, поэтому данный пакет

необходимо проигнорировать. Названия других состояний совпадают с названиями обрабатываемых в этот момент лексем пакета.

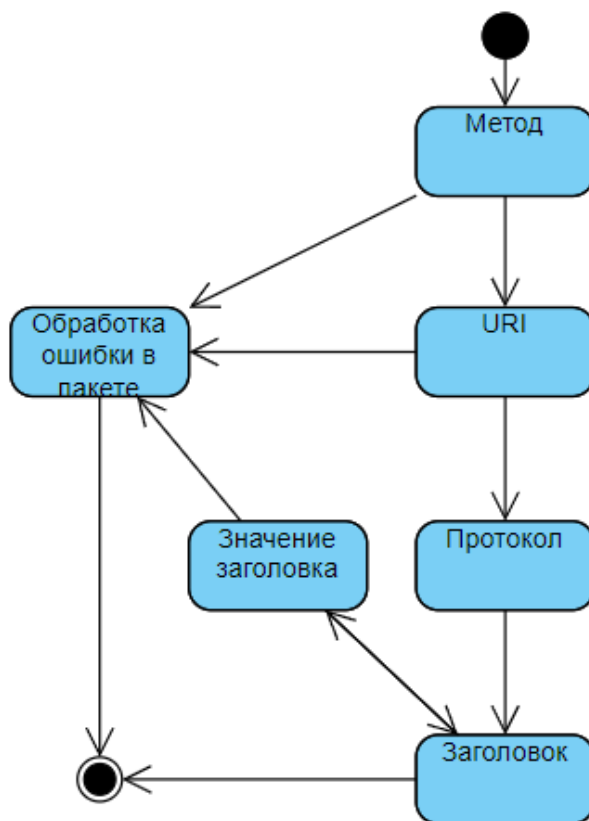


Рисунок 4.4 – Диаграмма состояний синтаксического анализатора SSDP пакетов

4.4.3 Итоговая диаграмма состояний объекта

В соответствии с документацией протокола SSDP, цикл работы объекта, реализующего этот протокол, можно описать в виде диаграммы состояний. Она представлена на рисунке 4.5. Желтым цветом на диаграмме обозначена функциональность, которую необходимо реализовать пользователю библиотеки с помощью механизмов, описанных в пункте 2.4.

4.4.4 Разработка средств тестирования

Для тестирования разрабатываемого модуля были созданы специальная прошивка для микроконтроллера и программа на языке Python, выполнение которой предполагается на другом устройстве. Данная программа в процессе тестирования принимает все SSDP пакеты, отправляемые микроконтроллером, и проверяет их корректность и своевременность. А прошивка микроконтроллера, в свою очередь, задействует все основные возможности библиотеки по отправке и обработке пакетов, то есть отправку мультивещательных пакетов-уведомлений о вхождении в сеть и выходе из неё, ответы на поисковые запросы, которым устройство удовлетворяет, игнорирование

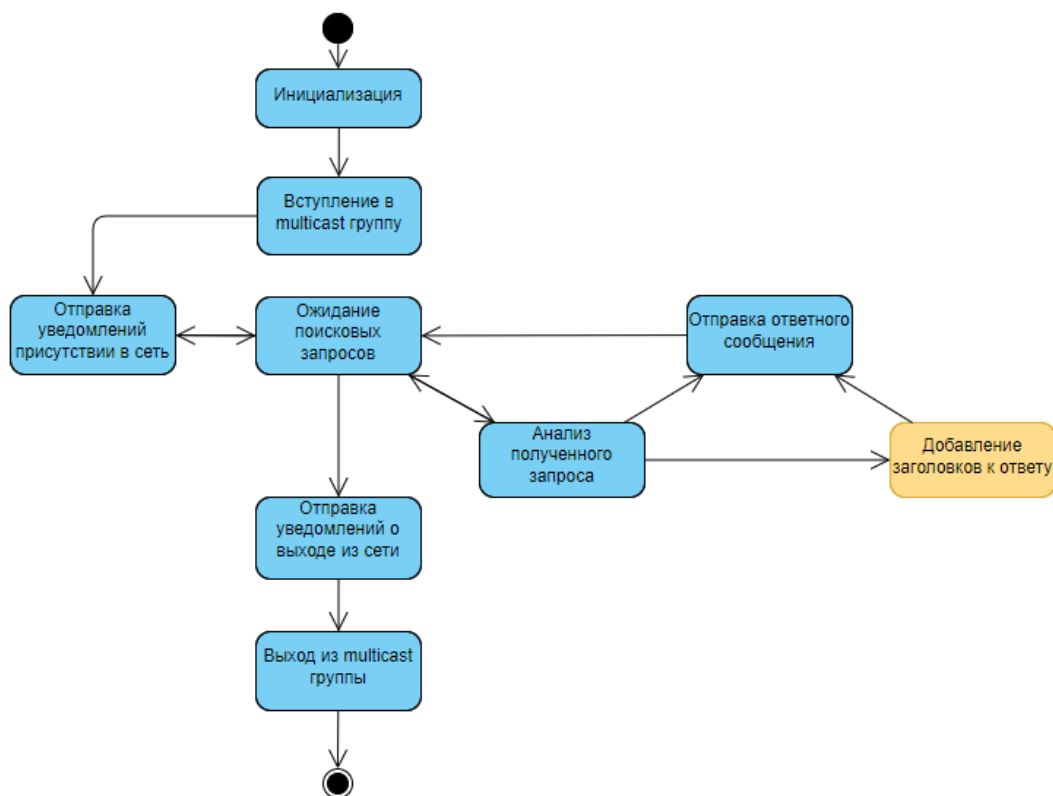


Рисунок 4.5 – Диаграмма состояний объекта

остальных, а также механизм расширения отправляемых пакетов дополнительными заголовками.

4.4.5 Тестирование

Для тестирования разработанной библиотеки на компьютере, находящемся в одной сети с микроконтроллером, была запущена программа, описанная в предыдущем пункте. На самом микроконтроллере была загружена прошивка из того же пункта. В ходе тестирования библиотека отработала корректно, отправив все необходимые пакеты и добавив к верной их части дополнительные заголовки. Результат тестирующей программы представлен на рисунке 4.6.

4.4.6 Итоги разработки

В соответствии с сформулированными требованиями был успешно спроектирован, разработан и протестирован программный модуль, осуществляющий поиск устройств и их самоидентификацию в сети. Его исходный код и примеры использования опубликованы и доступны для свободного использования [31]. Пример использования программного модуля приведён в приложении А.


```
Waiting for notifications...

Waiting for responses to search requests...

Waiting for byebye message...

Notifications: 6/6
Search responses: 9/9
Byebye messages: 6/6
PS D:\учёба\term_work> |
```

Рисунок 4.6 – Диаграмма состояний объекта

4.5 Возможности для интеграции разработанного модуля в платформу SciVi

4.5.1 Основные идеи интеграции

Разработанный модуль позволяет осуществлять поиск устройств по их типу и типам сервисов, которые они предоставляют. Протокол SSDP не накладывает ограничений на типы сервисов, поэтому можно использовать различные операторы SciVi в качестве сервисов, предоставляемых устройствами. Например, устройство может предоставлять функции арифметических вычислений, измерения расстояния или быть аналогово-цифровым преобразователем (АЦП).

В случае необходимости выполнения оператора, серверная часть платформы SciVi должна отправить поисковый SSDP запрос, целью которого будет являться SSDP сервис, соответствующий этому оператору. Кроме того, устройствам можно присваивать человекочитаемые имена, что позволит удобно выбирать необходимое устройство из найденных на интерфейсе платформы.

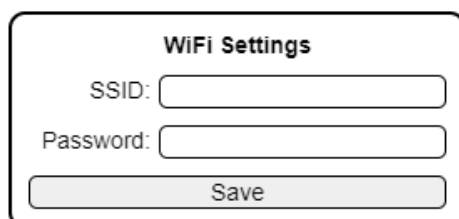
Также в рамках развития платформы SciVi разрабатывается протокол для самоидентификации сенсоров и актуаторов, подключенных к устройствам IoT. Это достигается добавлением к каждому из них маломощного микроконтроллера с прошивкой, реализующей этот протокол. Головной микроконтроллер сможет автоматически определять механизмы работы с подключенными устройствами. Каждый сенсор и актуатор также будет советовать оператору SciVi, формируя описание своего состава в виде онтологий. Затем будет создаваться полное онтологическое описание составного устройства на основе этих данных.

С учетом этих возможностей, поиск устройств можно проводить по их типам, а конкретные возможности каждого устройства передавать в дополнительном заголовке ответа на поисковый запрос в виде сжатой онтологии.

4.5.2 Интеграция в прошивку встраиваемых устройств

Для успешной интеграции разработанного модуля в платформу SciVi необходима возможность подключения нескольких микроконтроллеров к единой существующей Wi-Fi сети. Однако на момент выполнения работы такая возможность отсутствовала в прошивках, используемых в SciVi. Для решения этой проблемы был создан дополнительный программный модуль для микроконтроллеров на основе существующего кода.

Этот модуль предоставляет пользователю веб-интерфейс для подключения к Wi-Fi сети (см. Рисунок 4.7) и позволяет хранить данные для подключения в энергонезависимой памяти, чтобы обеспечить долговременную настройку соединения. Структура класса, реализующего возможность подключения к сети Wi-Fi, представлена на рисунке 4.8.



WiFi Settings

SSID:

Password:

Рисунок 4.7 – Внешний вид интерфейса подключения устройства IoT к сети Wi-Fi

WiFiConnector
-start_eeprom_addr
+Init() +SetAPData() +SetupWebServer() +ConnectWiFi() -saveNetworkInfo() -loadNetworkInfo()

Рисунок 4.8 – Структура класса, реализующего возможность подключения к сети Wi-Fi

После успешного подключения устройства к сети необходимо настроить модуль SSDP. Для первого сценария, описанного в предыдущем разделе, требуется вызов специального метода, где указывается список сервисов, предоставляемых устройством, каждый из которых описывает некоторый оператор SciVi. Для второго

сценария необходимо расширить функциональность класса, реализующего SSDP, чтобы при ответе на определенные поисковые запросы отправлялась сжатая онтология, описывающая состав устройства. Механизм такого расширения описан в главе 2.

4.5.3 Интеграция в серверную часть

Серверная часть платформы SciVi реализована на языке программирования Python с использованием фреймворка Flask. Для добавления возможности поиска устройств к серверу необходимо реализовать функцию поиска на языке Python или на любом другом языке, который может быть использован из Python. Эта функция должна принимать следующие аргументы:

- Название домена типов, к которому принадлежит искомый тип устройства или сервиса, название самого типа и его версию.
- Уникальный идентификатор устройства или сервиса.
- Специальное имя цели поиска, например, „ssdp::all“.

Затем, в зависимости от переданных аргументов, функция должна сформировать поисковый запрос с соответствующей целью, создать SSDP пакет, выполнить поиск устройств и вернуть адреса найденных устройств в сети и заголовки, содержащиеся в их ответах на поисковый запрос для дальнейшей обработки. Упрощенная версия такой функции уже была реализована в рамках данной работы, и ее исходный код приведен в приложении В.

ЗАКЛЮЧЕНИЕ

В ходе работы были изучены принципы работы сетей Интернета вещей, подхода онтологически-управляемых периферийных вычислений и функционирования платформы SciVi. Были проанализированы существующие технологии и средства для управления энергонезависимой памятью микроконтроллеров и самоидентификации и поиска устройств в сетях IoT. На основе проведённого анализа были спроектированы собственные решения, которые были реализованы в виде программных модулей языка C++. Разработанные модули удовлетворяют поставленным в работе требованиям, успешно прошли процесс тестирования и уже частично интегрированы в платформу SciVi.

Таким образом, поставленные в работе задачи были решены, а её главная цель - достигнута.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Ryabinin K., Chuprina S.* Ontology-Driven Edge Computing // Computational Science – ICCS 2020. — Springer, 2020. — С. 312–325.
2. *Konstantin R., Svetlana C., Ivan L.* Tackling IoT Interoperability Problems with Ontology-Driven Smart Approach. — 2021.
3. *Ryabinin K., Chuprina S.* Adaptive Scientific Visualization System for Desktop Computers and Mobile Devices // Procedia Computer Science. — 2013. — Т. 18. — С. 722–731.
4. Ontology-Driven Visual Analytics Platform for Semantic Data Mining and Fuzzy Classification / R. Konstantin [и др.] // Frontiers in Artificial Intelligence and Applications. — 2022. — Т. 358. — С. 1–7.
5. *Ryabinin K.* Ontology reasoning on microcontroller units. — 2020.
6. *Tarui Y., Hayashi Y., Nagai K.* Proposal of electrically reprogrammable non-volatile semiconductor memory // Proceedings of the 3rd Conference on Solid State Devices. — Tokyo : The Japan Society of Applied Physics, 1971. — С. 155–162.
7. *Tarui Y., Hayashi Y., Nagai K.* Electrically reprogrammable nonvolatile semiconductor memory // IEEE Journal of Solid-State Circuits. — 1972. — Т. 7, вып. 5. — С. 369–375.
8. Arduino core for ESP8266 WiFi chip / I. Grokhotkov [и др.]. — URL: <https://github.com/esp8266/Arduino>.
9. *Arduino Software.* EEPROM Library. — URL: <https://docs.arduino.cc/learn/built-in-libraries/eeprom>.
10. *Elenbaas T.* Arduino-EEPROMEx. — 2012. — URL: <https://github.com/thijse/Arduino-EEPROMEx>.
11. *Маѝоров A.* EEManager. — 2021. — URL: <https://github.com/GyverLibs/EEManager>.
12. *Massachusetts Institute of Technology.* MIT License. — 1988. — URL: <https://opensource.org/license/mit/>.
13. *Atmel Corporation.* AVR101: High Endurance EEPROM Storage. — 2002. — URL: <http://ww1.microchip.com/downloads/en/appnotes/doc2526.pdf>.
14. *Rosenberg P.* EEPROMWearLevel. — 2016. — URL: <https://github.com/PRosenb/EEPROMWearLevel>.
15. *The Apache Software Foundation.* Apache License, Version 2.0. — 2004. — URL: <https://www.apache.org/licenses/LICENSE-2.0>.

16. *Grokhotkov I.* ESP8266 EEPROM Library. — 2014. — URL: <https://github.com/esp8266/Arduino/tree/master/libraries/EEPROM>.
17. *Hirnschall S.* ESP8266: Read and Write from/to EEPROM (Flash Memory). — 2021. — URL: <https://blog.hirnschall.net/esp8266-eprom/>.
18. A new flash E2PROM cell using triple polysilicon technology / F. Masuoka [и др.] // 1984 International Electron Devices Meeting. — San Francisco, CA, USA : IEEE, 1984. — С. 464—467.
19. PROTOCOL STANDARD FOR A NetBIOS SERVICE ON A TCP/UDP TRANSPORT: CONCEPTS AND METHODS / A. Aggarwal [и др.]. — 1987. — URL: <https://datatracker.ietf.org/doc/html/rfc1001>.
20. *Cheshire S., Krochmal M.* DNS-Based Service Discovery. — 2013. — URL: <https://datatracker.ietf.org/doc/html/rfc6763>.
21. *Cheshire S., Krochmal M.* Multicast DNS. — 2013. — URL: <https://datatracker.ietf.org/doc/html/rfc6762>.
22. *Steinberg D. H., Stuart C.* Zero Configuration Networking: The Definitive Guide. — Sebastopol : O'Reilly Media, Inc., 2005. — С. 256.
23. Service Name and Transport Protocol Port Number Registry / J. Touch [и др.]. — 2022. — URL: <https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xml>.
24. *Open Connectivity Foundation Inc.* UPnP Device Architecture 2.0. — 2020. — URL: <https://openconnectivity.org/upnp-specs/UPnP-arch-DeviceArchitecture-v2.0-20200417.pdf>.
25. Service Location Protocol, Version 2 / E. Guttman [и др.]. — 1999. — URL: <https://www.ietf.org/rfc/rfc2608.txt>.
26. *Lukianov A.* EEManager. — 2023. — URL: <https://github.com/almiluk/EEManager/>.
27. *Peterson W. W., Brown D. T.* Cyclic Codes for Error Detection // Proceedings of the IRE. — 1961. — Т. 49, вып. 1. — С. 228—235.
28. *Lukianov A.* EEPROManager. — 2023. — URL: <https://github.com/almiluk/EEPROManager/>.
29. Doxygen / D. van Heesch [и др.]. — 1997. — URL: <https://www.doxygen.nl/>.
30. *Gochkov H., Sallemi F.* ESP8266SSDP Library. — 2014. — URL: <https://github.com/esp8266/Arduino/tree/master/libraries/ESP8266SSDP>.
31. *Lukianov A.* almilukESP8266SSDP. — 2022. — URL: <https://github.com/almiluk/almilukESP8266SSDP>.

ПРИЛОЖЕНИЕ А

Пример использования разработанной библиотеки менеджера EEPROM

```
#include <EEPROMVariable.h>
```

```
void setup() {  
    Serial.begin(115200);  
  
    EEPROMVar<uint16_t> boot_cnt("boot_cnt", 1);  
  
    Serial.println(boot_cnt);  
  
    boot_cnt = boot_cnt + 1;  
}  
  
void loop() {  
  
}
```

ПРИЛОЖЕНИЕ Б

Пример использования разработанной реализации протокола SSDP

```
#include <ESP8266WebServer.h>
#include <ESP8266WiFi.h>
#define NO_GLOBAL_SSDP
#include <alnilukESP8266SSDP.h>

#define NETNAME "SSID"
#define PASSWORD "PASSWORD"

class mySSDPClass : public SSDPClass {
public:
    int response_cnt = 0;

    mySSDPClass() {
        setManufacturer("alniluk");
        setManufacturerURL("https://github.com/alniluk");
        setModelName("alnilukESP8266SSDP_test");
        setDeviceType("alniluk-domain", "esp8266-ssdp-test", "1.0");
        SSDPServiceType services[] = {
            {"alniluk-domain", "service1", "v1"},
            {"some-other-domain", "service1", "1.1.0"},
            {"some-other-domain", "service2", "abcd"}
        };
        setServiceTypes(services, 3);
        // You can set all combinations of attributes you want.
    }

protected:

    void on_response() {
        Serial.print("Sending SSDP response for target: ");
        Serial.println(getAdvertisementTarget());
        addHeader("resp_header", "resp_header_val");
        response_cnt++;
    }
}
```



```

void on_notify_alive() {
  Serial.print("Sending SSDP alive notification for target: ");
  Serial.println(getAdvertisementTarget());
  addHeader("alive_header", "alive_header_val");
}

void on_notify_bb() {
  Serial.print("Sending SSDP byebye notification for target: ");
  Serial.println(getAdvertisementTarget());
  addHeader("byebye_header", "byebye_header_val");
}
};

mySSDPClass g_ssdp;
ESP8266WebServer g_webServer(80);

void setup() {
  Serial.begin(115200);

  WiFi.mode(WIFI_STA);
  WiFi.begin(NETNAME, PASSWORD);
  Serial.print("Waiting for WiFi connection");
  while (!WiFi.localIP().isSet()) {
    Serial.print('.');
    delay(500);
  }
  Serial.println("\nConnected to WiFi");

  if (g_ssdp.begin())
    Serial.println("SSDP begun");
  else
    Serial.println("SSDP init failed");

  g_webServer.on("/ssdp/schema.xml", []() {
    g_ssdp.schema(g_webServer.client());
  });
  g_webServer.begin();

  Serial.println("Web server started");
}

```

```
void loop() {  
  if (g_ssdp.response_cnt == 9) {  
    delay(10000);  
    g_ssdp.end();  
    g_ssdp.response_cnt++;  
  }  
  g_ssdp.loop();  
  g_webServer.handleClient();  
  delay(16);  
}
```

ПРИЛОЖЕНИЕ В

Функция поиска устройств и сервисов

```
def get_ssdp_list(st_val: str = "ssdp:all", timeout: int = 3):
    SSDP_ADDR = "239.255.255.250";
    SSDP_PORT = 1900;
    SSDP_MX = timeout;

    ssdpRequest = ("M-SEARCH * HTTP/1.1\r\n"
        + "HOST: %s:%d\r\n" % (SSDP_ADDR, SSDP_PORT)
        + "MAN: \\"ssdp:discover\\"r\n"
        + "MX: %d\r\n" % (SSDP_MX, )
        + "ST: %s\r\n" % (st_val, ) + "\r\n")
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP)
    sock.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_TTL, 5)
    sock.bind(('', 19011))

    sock.settimeout(timeout * 1.1)
    sock.sendto(ssdpRequest.encode(), (SSDP_ADDR, SSDP_PORT))
    ip_set = set()
    while True:
        try:
            data, addr = sock.recvfrom(10240)
            print(data.decode())
        except socket.timeout:
            break
        ip_set.add(addr[0])
    sock.close()
    return list(ip_set)
```