ISSN 2782-2001
Системы анализа и обработки данных
том 90, № 2, 2023, с. 43–58

http://journals.nstu.ru/vestnik
Analysis and data processing systems
Vol. 90, No. 2, 2023, pp. 43–58

ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ И ТЕЛЕКОММУНИКАЦИИ

INFORMATION TECHNOLOGIES AND TELECOMMUNICATIONS

УДК 004.05

DOI: 10.17212/2782-2001-2023-2-43-58

Разработка архитектурного решения программного обеспечения для устройств интернет-вещей*

В.К. ШПЕРЛИНГ[a], А.А. ЯКИМЕНКО[b]

630087, Новосибирская область, г. Новосибирск, ул. Немировича-Данченко, 136, Новосибирский государственный технический университет

[a] *vladimir-shperling@yandex.ru* [b] *yakimenko@corp.nstu.ru*

Представлена разработка архитектурного решения программного обеспечения для устройств интернет-вещей (IoT), реализующая функционал автоматического дозатора медицинских препаратов, на базе аппаратной платформы ESP32 с использованием возможностей существующих операционных систем реального времени (RTOS).

Архитектура программного обеспечения для устройств IoT была спроектирована с учетом масштабируемости и отказоустойчивости. Все компоненты системы взаимодействуют друг с другом через асинхронные callback-функции, что обеспечивает гибкость и расширяемость архитектуры. Было проведено тестирование на отказоустойчивость системы. Архитектура может быть внедрена и использована в основе любого устройства IoT, что позволяет обеспечить поддержку современных стеков безопасности и функциональности при реализации данного функционала единожды в любом из устройств.

В работе описывается процесс проектирования архитектуры программного обеспечения, включая выбор подходящих технологий и библиотек. Особое внимание уделено обеспечению безопасности и надежности работы устройства, в том числе защите от несанкционированного доступа и ошибок в работе. Результаты экспериментальных испытаний показывают высокую эффективность и точность работы автоматического дозатора медицинских препаратов на базе разработанного программного обеспечения.

В практической части приводятся примеры реализации предложенной архитектуры на языках C и C++ с примерами и основными диаграммами взаимодействия компонентов друг с другом. При написании реализации на языке C++ также была использована библиотека **rxcpp**, которая позволила упростить написание кодовой базы для взаимодействия с ресурсами операционной системы и переиспользовать многопоточное взаимодействие с системой.

Ключевые слова: IoT, разработка C++, архитектура ПО, ESP-IDF, FreeRTOS, ReactiveX, архитектура интернет-вещей, ПО IoT

---

* *Статья получена 10 апреля 2023 г.*

Исследование выполнено при финансовой поддержке РФФИ в рамках научного проекта № 20-08-00550 А.

ВВЕДЕНИЕ

В системах с использованием технологий интернет-вещей конечными узлами являются смарт-устройства. При разработке программного обеспечения (ПО) для смарт-устройств существующие архитектурные паттерны в других системах не учитывают специфику работы данных устройств. Это приводит к необходимости для каждого устройства разрабатывать уникальное ПО, которое в дальнейшем необходимо поддерживать и обновлять. При выпуске множества различных устройств возникает проблема с поддержкой актуального ПО устройства и короткого жизненного цикла продукта.

Рассмотренные альтернативы предлагают решения на уровне операционной системы реального времени (ОСРВ), которая бы диктовала парадигму, предусмотренную разработчиками. Однако такое решение накладывает аппаратные ограничение на конечное устройство, поскольку в различных аппаратных платформах может использоваться только ограниченный круг ОСРВ.

На текущий момент существуют архитектурные решения в сфере IoT [1, 2], представляющие комплекс, в котором присутствуют как конечные устройства, так и центры обработки данных. В этих решениях зачастую отсутствуют описания работы конечных устройств и способы их взаимодействия с другими узлами, хотя это может быть решающим аспектом при выборе и построении сети. Помимо этого, не учтены и затраты на разработку программного обеспечения для этих конечных узлов (несмотря на то что зачастую они представляют собой лишь сенсоры с минимальным функционалом передачи данных), которые имеют немаловажную роль и требуют взаимодействия в зависимости от выбранной сети IoT. Разработка ПО для каждого устройства ведется индивидуально, что требует реализации одних и тех же протоколов, алгоритмов для каждой аппаратной платформы, что критично при масштабной сети с множеством разнообразных устройств. Очень важно вовремя отслеживать безопасность ПО и своевременно доставлять обновление, что невозможно реализовать при таком подходе. Единая архитектура позволила бы решить данную проблему путем создания единой среды для работы с различными протоколами и уровнями абстракции при реализации программного обеспечения для устройства. При такой архитектуре для каждой новой аппаратной платформы необходимо реализовать лишь аппаратный слой. Обновление безопасности, если уязвимость находится в программном алгоритме или конфигурации, необходимо внести только в единой системе, которую с помощью аппаратных реализаций можно масштабировать на все используемые платформы.

1. ПОСТАНОВКА ЗАДАЧИ

Целью настоящей работы является разработка архитектуры программного обеспечения устройств интернет-вещей, описывающих следующие ключевые возможности:

1) масштабировать функционал системы без необходимости изменения текущего:
2) минимизировать отклик системы;
3) минимизировать связанность компонентов друг с другом.

Такая архитектура предоставит возможность системного подхода к решению задач, связанных с разработкой программного обеспечения для интернет-вещей и поддержания актуального программного обеспечения за счет универсальности архитектур. Это позволит частично переиспользовать или уменьшить время внедрения нового протокола или устранения уязвимости.

Для анализа полученных результатов будет проведено сравнение программного обеспечения дозатора медицинских препаратов с обновленным программным обеспечением, которое будет написано исходя из предлагаемой архитектуры. Сравнение будет содержать несколько этапов.

Первый этап будет включать в себя проведение тестов быстродействия ответов на сетевые запросы устройства, в котором будут замерены следующие характеристики: скорость ответа, количество одновременных подключений, загрузка микроконтроллера. Количеством одновременных подключений будет являться абсолютное число подключенных устройств, которые получили корректный ответ на сетевой запрос в течение 10 секунд.

Второй этап включает в себя оценку возможности покрытия исходного кода средствами автоматического тестирования программного обеспечения. За 100 % будет приниматься количество алгоритмических конструкций, использованных в программном обеспечении. Такая единица будет считаться протестированной, если в ходе прохождения автоматического теста алгоритмическая конструкция была исполнена [3].

Третий этап заключается в оценке скорости разработки нового функционала программного обеспечения с помощью методологии COCOMO [4].

2. КОМПОНЕНТЫ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ IOT

При разработке программного обеспечения для устройств интернет-вещей (IoT) необходимо учитывать специфику аппаратной платформы, которая имеет небольшое количество ресурсов для работы с полноценной операционной системой, позволяющей включать в себя несколько уровней абстракции, а также такие функции, как планировщики задач и подсистемы для переключения потоков. Для работы таких устройств используются операционные системы реального времени (RTOS – Real-Time Operation System), ядро которых обеспечивает функционирование промежуточного абстрактного уровня, которое скрывает аппаратные особенности конечного устройства и связанного с ним аппаратного обеспечения [5]. В настоящее время существует несколько разновидностей RTOS.

1. Предоставление доступа только к базовым функциям аппаратной платформы. Данный тип реализует базовые сервисы системы реального времени [6]: управление задачами, динамическое распределение памяти, управление таймерами, синхронизация и контроль ввода-вывода.

2. Предоставление доступа к базовому функционалу аппаратной платформы, а также к программным реализациям различных алгоритмов. Помимо всего перечисленного функционал первого типа включает в себя дополнительные библиотеки различных стеков (TCP/IP, Bluetooth), шифрования и хэширования (MD5, AES256).

3. Включает в себя второй тип, а также реализации дополнительных функций аппаратной платформы, такие как реализация работы с модулем Wi-Fi, поддержка сопроцессоров сверхнизкого энергопотребления.

Исходя из такого разнообразия функций необходимо выделить отдельную единицу архитектурного компонента, который связывает аппаратно-программные зависимости системы и позволяет скрыть выбор наиболее приоритетного способа реализации от компонентов конечной реализации.

3. АРХИТЕКТУРНЫЕ ВОЗМОЖНОСТИ IOT

Архитектурные возможности устройств интернет-вещей зависят от инструментов при разработке программного обеспечения. Эти инструменты определяются RTOS, которая представляет возможности сборки различными компиляторами, что в итоге представляет доступ к языкам программирования. На основе языка программирования есть возможность выбрать наилучший подход к проектированию и спроецировать его непосредственно на область разработки IoT. Исходя из этого прежде всего нам необходимо выбрать язык, который поддерживается операционными системами реального времени, работающими на большинстве архитектур аппаратных устройств.

3.1. ВЫБОР ОПЕРАЦИОННОЙ СИСТЕМЫ РЕАЛЬНОГО ВРЕМЕНИ

На современном этапе рынок информационных технологий предлагает большое количество операционных систем реального времени для смарт-устройств, позволяющих решать поставленные задачи. При выборе стоит учитывать следующие факторы: поддерживаемые языки разработки; минимальное количество оперативной и постоянной памяти, необходимой для функционирования RTOS; поддерживаемый стек параллелизации задач; тип лицензирования; количество поддерживаемых аппаратных платформ; объем поддерживаемых функций аппаратной платформы.

При реализации программного обеспечения для микроконтроллеров и смарт-устройств требуется работа с окружением операционной системы реального времени, которая предоставляет ресурсы аппаратной платформы и позволяется взаимодействовать с потоками данных. Приведем сводную таблицу наиболее распространенных ОСРВ с основными техническими характеристиками.

Для более детального анализа выделим несколько систем, которые поддерживают наибольшее число архитектур: RIOT, FreeRTOS, Zephyr. Возможности каждой системы будут рассмотрены для платы ESP32, работающих на ядрах XTENSA.

Система RIOT предоставляет основной функционал для работы с аппаратными возможностями системы, ядро которого написано на языке C. Из дополнительного функционала можно выделить следующее [7]: I2C, SPI, UART, CAN интерфейсы; CPU ID доступ; RTC модуль; PWM, ADC и DAC каналы; SPI RAM, Flash Drive; управление энергопотреблением; протокол ESP-NOW; протокол ESP Ethernet MAC (EMAC). К недостаткам можно отнести отсутствие поддержки bluetooth-стека, работу только в одноядерном режиме и

невозможность использования аппаратного шифрования постоянной памяти устройства.

Таблица 1

Table 1

Обзорная таблица основных характеристик операционных систем реального времени

An overview table of the main characteristics of real-time operating systems

ОСРВ	Минимальная RAM (КБ)	Минимальная ROM (КБ)	Язык программирования	Количество архитектур	Лицензия
RIOT	1,5	5	C, C++*	7	Открытая
FreeRTOS	1	10	C, C++*	35	Открытая
Nano-RK	2	18	C	4	Открытая
LiteOS	4	128	C++*	3	Открытая
Apache Mynewt	16	128	C, Go	2	Открытая
Zephyr	8	128	C, C++*, Kotlin	9	Открытая
Windows IoT	256 МБ	200 МБ	Множество	4	Закрытая
* – частичная поддержка					

Система Zephyr выделяется среди остальных тем, что поддерживает язык разработки Kotlin, но сегодня этот язык доступен только для одной платы. Поддержка же возможностей платы включает следующие возможности: интерфейсы SPI, I2C, Serial; каналы PWM, ADC, DAC; SPI RAM, Flash Drive; управление энергопотреблением; поддержка Wi-Fi.

FreeRTOS – это система, которая предоставляет лишь базовую функциональность ОСРВ, и для реализации поддержки функционала существует библиотека ESP-IDF, которая является официальной от производителя и поддерживает полную функциональность микроконтроллера.

По результатам сравнения функционала систем можно сделать вывод о том, что при разработке решения будет использована ОСРВ FreeRTOS, поскольку за счет ESP-IDF можно реализовать весь потенциал контроллера при решении задачи. При этом языком разработки будет выбран язык C или C++, так как большинство систем поддерживают именно эти языки разработки.

3.2. ПРЕДЛАГАЕМАЯ АРХИТЕКТУРА РЕШЕНИЯ

Перед тем как рассматривать архитектурные возможности языков, необходимо определить целевое решение и концепцию архитектуры, которую мы пытаемся достичь. По сравнению с обычным прикладным ПО необходимо учитывать также системные компоненты, которые могут изменятся в зависимости от компонентной базы аппаратного комплекса. Поэтому были выдвинуты следующие требования к архитектуре компонентов.

Компоненты по использованию можно разделить на следующие типы (рис. 1):

– с общим инициализатором. При работе необходимо проинициализировать единственный раз общую часть, после чего можно использовать компоненты;

– с единственным экземпляром. Такие компоненты можно создавать лишь единственный раз, а в дальнейшем до деинициализации необходимо использовать всегда созданный экземпляр для работы с потоком данных;

– с независимыми экземплярами. Для каждых данных необходимо создавать новый экземпляр, при этом их количество неограниченно.

Рис. 1. Типы компонентов

Fig. 1. Component types

Для создания компонента должен всегда использоваться конструктор, который гарантирует его инициализацию начальными значениями. После создания компонент имеет следующие сущности:

- данные – внутренняя информация для нормального функционирования компонента. Доступно только на уровне компонента;
- потоки вывода – информация, к которой предоставляет доступ данный компонент. При этом каждый поток вывода должен поддерживать подключение множества подписчиков на изменение.

Если компонент более не используется, он должен быть уничтожен с использованием деструктора, который уведомляет всех подписчиков о том, что данный компонент больше не будет отправлять данные, а также отписаться от источников данных (при их наличии). Такой жизненный цикл представлен на рис. 2.

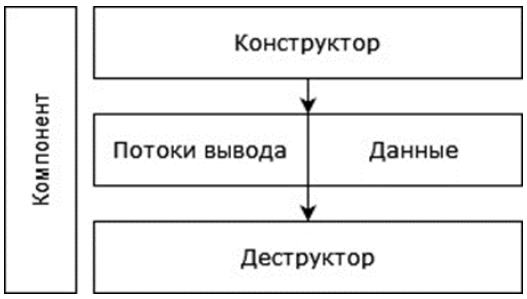

Рис. 2. Жизненный цикл компонент

Fig. 2. Component lifecycle

Все компоненты не должны быть связаны друг с другом (рис. 3). Связь между ними необходима только при условии, если без этого компонента невозможна функциональность другого.

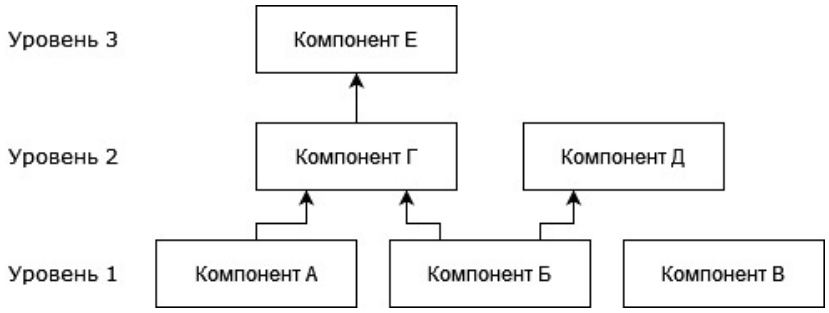

Рис. 3. Граф связанности компонентов

Fig. 3. Component dependency graph

После проектирования компонентов, которые могут быть как полностью программными, так и аппаратно зависимыми, появляется необходимость связать эти компоненты для получения необходимого функционала устройства. К этим сущностям будут точно такие же архитектурные требования, как и к компонентам, за исключением следующего:

– сущность не должна иметь аппаратно-зависимых частей кода. Всё обращение к аппаратным возможностям происходит через компоненты;

– сущность необходима для объединения компонентов и реализации функциональность возможности с помощью компонентов.

3.3. АРХИТЕКТУРНЫЕ ВОЗМОЖНОСТИ ЯЗЫКА С

Исходя из выбранных языков программирования и необходимых архитектурных единиц рассмотрим, каким образом можно реализовать каждое требование.

Определим все реализации архитектурных компонентов согласно возможностям языка [8].

Конструктор – заранее оговоренная функция (например, init), которую необходимо вызвать для того, чтобы проинициализировать компонент. В качестве результата функция вернет контекст компонента, который необходимо будет передавать в качестве аргумента в любые последующие функции по работе с ним.

Потоки вывода. Для подписки и прекращения отслеживания необходимо создание двух соответствующих функций, на вход которые получают, помимо контекста, и функцию с необходимыми параметрами, которую необходимо вызывать при обновлении данных.

Данные или контекст – структура, которая будет хранить все необходимые данные для работы с компонентом (например, инициализированный номер порта, список всех подписчиков на данные).

Деструктор – функция, обратная конструктору (например, deinit), которая на вход принимает контекст и деинициализрует его, после чего использовать его далее невозможно.

В итоге шаблонный заголовок будет иметь следующий вид (рис. 4).

```
// component.h
#ifndef _COMPONENT_H_
#define _COMPONENT_H_

struct component_t;

component_t* component_init();
void component_deinit(component_t* context);

typedef void (*component_stream_callback) (
    component_t* context, const void* data
);
void component_subscribe_stream(
    component_t* context, component_stream_callback* callback
);
void component_unsubscribe_stream(
    component_t* context, component_stream_callback* callback
);

#endif
```

Рис. 4. Заголовок шаблонного компонента на языке С

Fig. 4. The template component header in the C language

Для реализации потоков данных воспользуемся паттерном проектирования OBSERVER (рис. 5). Паттерн позволяет реализовать принцип стабильных зависимостей [9].

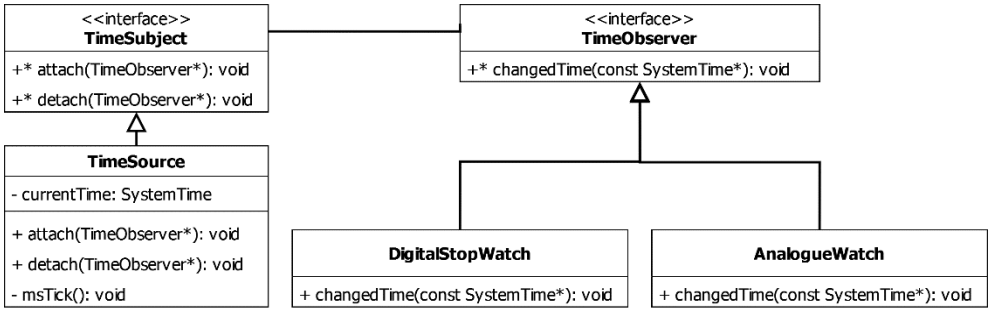

Рис. 5. Структура шаблона проектирования OBSERVER

Fig. 5. Structure of the OBSERVER design pattern

Реализация различных типов компонентов будет выполнена следующим образом.

1. Компоненты с единственным инициализатором (рис. 6). Фабрика в данном случае при первом обращении на получение экземпляра должна создать его, а также вести подсчет количества отданных экземпляров. При деинициализации последнего элемента уничтожать и созданный экземпляр для освобождения памяти. Из-за отсутствия у языка элементов синхронизации операций при обращении к фабрике необходимо учитывать синхронность обращений.

```
// component.h
#ifndef _COMPONENT_H_
#define _COMPONENT_H_

struct component_t;

component_t* component_init();
void component_deinit(component_t* context);

typedef void (*component_stream_callback) (
    component_t* context, const void* data
);
void component_subscribe_stream(
    component_t* context, component_stream_callback* callback
);
void component_unsubscribe_stream(
    component_t* context, component_stream_callback* callback
);

#endif
```

Рис. 6. Фабрика, реализующая компонент с единственным экземпляром на языке C

Fig. 6. A factory that implements a component with a single instance in the C language

2. Компоненты с независимыми экземплярами – самый простой вариант реализации, так как нет необходимости создавать дополнительную фабрику, поскольку она будет полностью дублировать функционал конструктора и деструктора.

3. Компоненты с общим инициализатором (рис. 7) – объединение двух предыдущих способов реализации. Ответственность фабрики заключается в том, чтобы контролировать количество созданных экземпляров, а когда они все уничтожены – выполнять общую деинциализацию. Если же инициализатор при инициализации одного функционала блокирует другой, тогда под него необходимо выделить отдельный компонент первого типа, реализовав функцию проверки возможности работы в том или ином режиме.

Связи же между компонентами могут осуществляться одним из следующих шаблонов проектирования.

1. Шаблон проектирования STATE (рис. 8). Самый простой паттерн в реализации, побуждает разграничить все действия и функции класса на определенные конечные состояния. Для реализации конечных автоматов необходимо использовать таблицу переходов, чтобы решение хорошо масштабировалось, было легко читаемым и не дублировало код. Ниже приведена структура паттерна.

Данное решение очень похоже на объектно-ориентированное решение, однако оно таковым не является. Чтобы это понять, приведем описание каждой сущности:

– DigitalStopWatch – экземпляр структуры, который определяет текущее конечное состояние и реализует интерфейс взаимодействия для остальных сущностей;

– WatchState – интерфейс, который определяет все поддерживаемые конечные состояния;

– StoppedState/StartedState – конкретные состояния, каждое из которых инкапсулирует поведение, которое оно представляет.

```
// fabric_component.c

static int instance_count = 0;
static bool is_init = false;

void fabric_component_global_init()
{
    //YOUR CODE
    is_init = true;
}

void fabric_component_global_deinit()
{
    //YOUR CODE
    is_init = false;
}

component_t* get_instance()
{
    if (!is_init) fabric_component_global_init();
    instance_count++;
    return component_init();
}

void clear_instance(component_t* context)
{
    component_deinit(context);
    if (--instance_count == 0)
    {
        fabric_component_global_deinit();
    }
}
```

Рис. 7. Фабрика, реализующая компонент с общим инициализатором на языке C

Fig. 7. A factory that implements a component with a common initialize in the C language

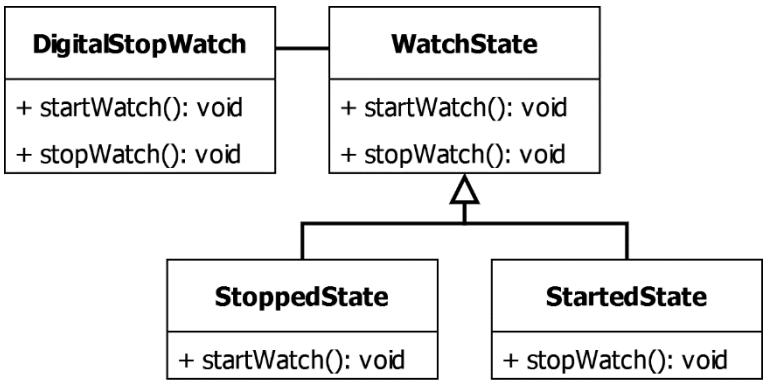

Рис. 8. Структура шаблона проектирования STATE

Fig. 8. Structure of the STATE design pattern

Основная идея заключается в том, чтобы разделить каждое состояние на отдельный объект, тем самым переход в различные состояния будет являться просто изменением ссылки в экземпляре из одного состояния в другое.

2. Шаблон проектирования STRATEGY (рис. 9). Позволяет абстрагироваться от реализации определенного алгоритма с целью инкапсулировать реализацию и сделать ее взаимозаменяемой.

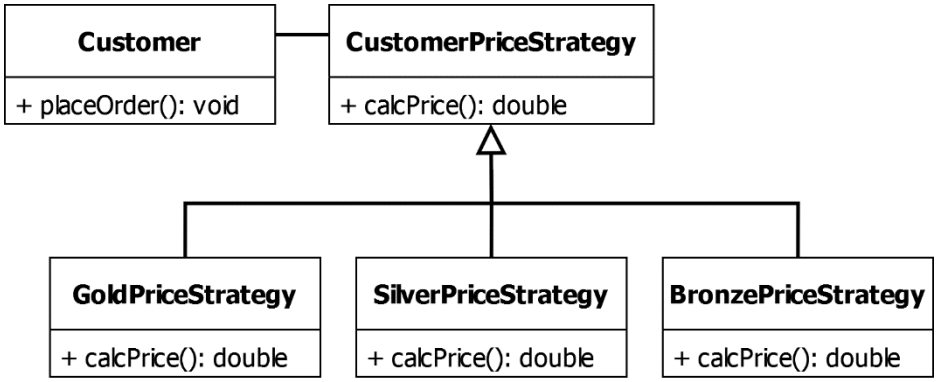

Рис. 9. Структура шаблона проектирования STRATEGY

Fig. 9. Structure of the STRATEGY design pattern

Рассмотрим пример реализации данного шаблона. Например, нам необходимо внедрить бонусную систему, при этом у нас есть три статуса: бронзовый, серебряный и золотой. Каждый из статусов имеет сложную формулу расчета итоговой стоимости. Если описывать всё в одной функции через условные конструкции, то данное решение будет немасштабируемым и нечитаемым. В этом случае для реализации необходимо применить шаблон STRATEGY. Для этого создается дополнительный слой абстракции, который реализует набор необходимых для работоспособности функций. Это позволяет полностью отделить реализацию пользовательских процедур от процедур расчёта стоимости.

3.4. АРХИТЕКТУРНЫЕ ВОЗМОЖНОСТИ ЯЗЫКА C++

C++ расширяет возможности языка C и дополняет их. Исходя из этого все реализации, приведенные на языке C, можно использовать и в C++, поэтому рассмотрим возможности улучшения существующей архитектуры благодаря возможностям языка [10].

Благодаря наличию объектно-ориентированного программирования жизненный цикл компонента можно описать с помощью встроенных средств. При этом сам компонент будет являться объектом, и соответственно нет необходимости в создании дополнительной структуры для хранения контекста. При использовании дополнительной библиотеки реактивного подхода **rxcpp** потоки вывода также можно реализовать с помощью системы подписок [11]. При этом благодаря дополнительным модификаторам можно заранее фильтровать данные как перед получением, так и после него, что в итоге уменьшает количество шаблонного кода (рис. 10).

```
// component.h
#ifndef _COMPONENT_H_
#define _COMPONENT_H_

#include "rxcpp/rx.hpp"

class Component
{
public:
    Component();
    ~Component();

    rxcpp::observable<void*> observe_state();

private:
    rxcpp::subjects::behavior<void*> behavior;
};
#endif
```

Рис. 10. Заголовок шаблонного компонента на языке C++

Fig. 10. Template component header the C language

При замене функционального подхода фабрик на объектно-ориентрованный появится необходимость в реализации паттернов хранения единого экземпляра фабрики, что ухудшит читаемость и не даст преимуществ. Если проанализировать работу фабрик в текущей реализации, то можно сказать, что у них есть следующие основные недостатки.

• Асинхронная работа с объектами. При управлении объектами асинхронно есть возможность обратиться к промежуточному значению переменной, тем самым будут некорректно выполнены операции по уничтожению или созданию объектов, что приведет к критическим ошибкам программного обеспечения. Чтобы избежать конфликтов при доступе к переменным из разных потоков, необходимо применять атомарные операции, которые гарантируют блокировку доступа к переменной до окончания работы с первым потоком [12]. Для реализации этого функционала в языке C++ есть библиотека std::atomic.

• Необходимо вызывать функцию уничтожения объекта в фабрике. Поскольку компонент написан с помощью подхода ООП, то деструктор вызывается автоматически, когда ссылок на компонент ни у кого нет и, следовательно, необходимо вызывать функцию, которая уведомила бы фабрику об уничтожении компонента. Этот вариант нежелателен, поскольку в этом случае компоненту необходимо дополнительно иметь зависимость в виде фабрики, что обязует использовать промежуточный интерфейс, чтобы обеспечить возможность создавать компоненты разными фабриками. Альтернативным вариантом является создание дополнительных потоков данных жизненного цикла, которые будут извещать о текущем состоянии компонента (рис. 11). Тем самым фабрика будет подписываться на жизненный цикл, и когда придет событие об уничтожении компонента, она выполнит все необходимые инструкции.

```
// base_component.hpp
#pragma once

#if !defined(_BASE_COMPONENT_H_)
#define _BASE_COMPONENT_H_

#include "rxcpp/rx.hpp"

class BaseComponent
{
public:
    BaseComponent();
    ~BaseComponent();

    enum lifecycle {create, destroy}
    rxcpp::observable<lifecycle> observe_lifecycle();

private:
    rxcpp::subjects::behavior<lifecycle> behavior_lifecycle;
};
#endif
```

Рис. 11. Заголовок базового компонента с жизненным циклом на языке C++

Fig. 11. Header of base component with a lifecycle in the C++ language

Для связи компонентов можно использовать все паттерны проектирования, используемые в языке С, а также множество паттернов объектно-ориентированного программирования при необходимости разделения функционала одного компонента на несколько.

ЗАКЛЮЧЕНИЕ

По результатам проведенного исследования при разработке программного обеспечения для устройств интернет-вещей необходимо учитывать следующее:

– проектирование компонентов должно учитывать аппаратно-зависимые компоненты, поскольку при смене контроллера управления в большинстве случаев придется изменять код именно в этих компонентах;

– при выборе ОСРВ наиболее поддерживаемыми различными микроконтроллерами являются RIOT, FreeRTOS, ZephyrRTOS. Каждая из данных систем поддерживает написание ПО на языках С, C++;

– разработка архитектуры ПО должна строиться на компонентах с жизненным циклом, приведенным на рис. 2. Если компоненты не являются независимыми, то при их создании необходимо пользоваться вспомогательными фабриками, которые контролируют экземпляры компонентов и предотвращают возможность получения недействительных или блокирующих функциональность устройства компоненты. Для соединения функциональности необходимо применять аналогичные сущности, в которых используются необходимые компоненты для реализации совместного взаимодействия.

Реализованная архитектура на примере дозатора медицинских препаратов позволила внедрить поддержку нового протокола взаимодействие на 40 % быстрее, чем в предыдущем решении. Помимо этого, данная архитектура

позволила реализовать потенциал многоядерной аппаратной платформы и более эффективно использовать ресурсы платформы, что повлияло на быстродействие и отказоустойчивость системы (в 1.3 раза больше обработано соединений).

Результаты работы можно использовать для построения программного обеспечения любых смарт-устройств с использованием подходов тестирования и системности.

СПИСОК ЛИТЕРАТУРЫ

1. *Sutar S., Mekala P.* An Extensive review on IoT security challenges and LWC implementation on tiny hardware for node level security evaluation // International Journal of Next-Generation Computing. – 2022. – Vol. 13 (1). – DOI: 10.47164/ijngc.v13i1.424.

2. A smart wearable system for sudden infant death syndrome monitoring / A.G. Ferreira, D. Fernandes, S. Branco, J.L. Monteiro, J. Cabral, A.P. Catarino, A.M. Rocha // 2016 IEEE International Conference on Industrial Technology (ICIT). – Taipei, Taiwan, 2016. – P. 1920–1925. – DOI: 10.1109/ICIT.2016.7475060.

3. *Myers G.J., Sandler C., Badgett T.* The Art of Software Testing. – Hoboken, NJ: John Wiley & Sons, 2011.

4. *Jorgensen M., Shepperd M.* A systematic review of software development cost estimation studies // IEEE Transactions on Software Engineering. – 2007. – Vol. 33 (1). – P. 33–53. – DOI: 10.1109/TSE.2007.256943.

5. *Kalinsky D.* Basic concepts of real-time operating systems (18 November, 2003). – URL: https://linuxdevices.org/basic-concepts-of-real-time-operating-systems-a/ (accessed: 29.05.2023).

6. *Laplante P.A., Ovaska S.J.* Real-time systems design and analysis: tools for the practitioner. – 4th ed. – Wiley-Interscience, 2012. – 536 p.

7. *Schorcht G.* Documentation. ESP32 SoC Series // RIOT: website. – URL: https://doc.riot-os.org/group__cpu__esp32.html#esp32_features (accessed: 29.05.2023).

8. *Kernighan W.B., Ritchie M.D.* The C programming language. – 2nd ed. – Englewood Cliffs, NJ: Prentice-Hall, 1988. – 272 p.

9. *Tornhill A.* Patterns in C: patterns, idioms and design principles. – Leanpub, 2015.

10. *Schildt H.* C++: the complete reference. – 3rd ed. – Osborne: McGraw-Hill, 1998. – ISBN 978-0-07-882476-0.

11. *Ševc B.* Library for Handling. Asynchronous Events in C++: bachelor's thesis. – Brno, 2019. – URL: https://is.muni.cz/th/lx0et/thesis_Archive.pdf (accessed: 29.05.2023).

12. *Schweizer H., Besta M., Hoefler T.* Evaluating the cost of atomic operations on modern architectures // 2015 International Conference on Parallel Architecture and Compilation (PACT). – IEEE, 2015. – P. 445–456. – DOI: 10.1109/PACT.2015.24.

13. *Garg R., Kumar R.* A review of real-time operating systems for Internet of Things applications // Journal of Ambient Intelligence and Humanized Computing. – 2019. – Vol. 10 (7). – P. 2665–2681.

14. *Yang Y., Li Y., Li X.* A survey on real-time operating systems for embedded systems in Internet of Things // Journal of Ambient Intelligence and Humanized Computing. – 2019. – Vol. 10 (6). – P. 2423–2437.

15. RIOT OS: towards an OS for the Internet of Things / E. Baccelli, O. Hahm, M. Gunnes, M. Wählisch, T.C. Schmidt // 2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). – Turin, Italy, 2013. – P. 79–80. – DOI: 10.1109/INFCOMW.2013.6970748.

Шперлинг Владимир Константинович, магистрант, Android-разработчик, эксперт Академии Samsung по компетенции интернета вещей. Научные интересы: информационные технологии, интернет вещей, системное программирование, мобильная разработка. E-mail: vladimir-shperling@yandex.ru

Якименко Александр Алесандрович, доцент, кандидат технических наук, заведующий кафедрой вычислительной техники Новосибирского государственного технического университета. Научные интересы: информационные технологии, компьютерные системы, компьютерное моделирование, параллельные вычисления. Автор и соавтор более 60 научных работ. E-mail: yakimenko@corp.nstu.ru

Shperling Vladimir K., MSc student, an Android developer, Samsung Academy expert on the Internet of Things competence. His research interests include information technology, Internet of things, system programming, and mobile development. E-mail: vladimir-shperling@yandex.ru

Yakimenko Alexander A., associate professor, PhD (Eng.), Head of the Department of Computer Engineering, NSTU. His research interests include information technologies, computer systems, computer modeling, and parallel computing. He is an author and co-author of more than 60 scientific papers. E-mail: yakimenko@corp.nstu.ru

DOI: 10.17212/2782-2001-2023-2-43-58

Development of an architectural software solution for Internet of Thing devices [*]

V.K. SHPERLING[a], A.A. YAKIMENKO[b]

Novosibirsk State Technical University, 136 Nemirovich-Danchenko Street, Novosibirsk, 630087, Russian Federation

[a] vladimir-shperling@yandex.ru [b] yakimenko@corp.nstu.ru

Abstract

The article presents the development of an architectural software solution for Internet of Things (IoT) devices that implements the functionality of an automatic medical drug dispenser, based on the ESP32 hardware platform and utilizing the capabilities of the existing real-time operating systems (RTOS). The software architecture for IoT devices was designed with scalability and fault tolerance in mind. All components of the system interact with each other through asynchronous callback functions, which provides flexibility and extensibility to the architecture. Testing for system fault tolerance was conducted. The architecture can be implemented and used as the basis for any IoT device, allowing for support of modern security and functionality stacks, by implementing this functionality once in any of the devices.

The process of designing the software architecture is presented, including the selection of suitable technologies and libraries. Particular attention was paid to ensuring the safety and reliability of the device, including protection against an unauthorized access and errors in operation. The experimental results show high efficiency and accuracy of the automatic medical drug dispenser based on the developed software.

The practical part provides examples of implementing the proposed architecture in the C and C++ languages, with examples and basic interaction diagrams between the components. The rxcpp library was also used in writing the C++ implementation, which made it easier to write the code base for interacting with the operating system resources and reusing the multithreaded interaction with the system.

Keywords: IoT, C++ development, software architecture, ESP-IDF, FreeRTOS, ReactiveX, Internet of Things architecture, IoT software

[*] *Received 10 April 2023.*

REFERENCES

1. Sutar S., Mekala P. An Extensive review on IoT security challenges and LWC implementation on tiny hardware for node level security evaluation. *International Journal of Next-Generation Computting*, 2022, vol. 13 (1). DOI: 10.47164/ijngc.v13i1.424.

2. Ferreira AG, Fernandes D, Branco S., Monteiro J.L., Cabral J., Catarino A.P., Rocha A.M. A smart wearable system for sudden infant death syndrome monitoring. *2016 IEEE International Conference on Industrial Technology (ICIT)*, Taipei, Taiwan, 2016, pp. 1920–1925. DOI: 10.1109/ICIT.2016.7475060.

3. Myers G.J., Sandler C., Badgett T. *The Art of Software Testing*. Hoboken, NJ, John Wiley & Sons, 2011.

4. Jorgensen M., Shepperd M. a systematic review of software development cost estimation studies. *IEEE Transactions on Software Engineering*, 2007, vol. 33 (1), pp. 33–53. DOI: 10.1109/TSE.2007.256943.

5. Kalinsky D. *Basic concepts of real-time operating systems* (18 November, 2003). Available at: https://linuxdevices.org/basic-concepts-of-real-time-operating-systems-a/ (accessed 29.05.2023).

6. Laplante P.A., Ovaska S.J. *Real-time systems design and analysis: tools for the practitioner*. 4th ed. Wiley-Interscience, 2012. 536 p.

7. Schorcht G. Documentation. ESP32 SoC Series. *RIOT*. Website. Available at: https://doc.riot-os.org/group__cpu__esp32.html#esp32_features (accessed 29.05.2023).

8. Kernighan W.B., Ritchie M.D. *The C programming language*. 2nd ed. Englewood Cliffs, NJ, Prentice-Hall, 1988. 272 p.

9. Tornhill A. *Patterns in C: patterns, idioms and design principles*. Leanpub, 2015.

10. Schildt H. *C++: the complete reference*. 3rd ed. Osborne, McGraw-Hill, 1998. ISBN 978-0-07-882476-0.

11. Ševc B. *Library for Handling. Asynchronous Events in C++*. Bachelor's thesis. Brno, 2019. Available at: https://is.muni.cz/th/lx0et/thesis_Archive.pdf (accessed 29.05.2023).

12. Schweizer H., Besta M., Hoefler T. Evaluating the cost of atomic operations on modern architectures. *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 2015, pp. 445–456. DOI: 10.1109/PACT.2015.24.

13. Garg R., Kumar R. A review of real-time operating systems for Internet of Things applications. *Journal of Ambient Intelligence and Humanized Computing*, 2019, vol. 10 (7), pp. 2665–2681.

14. Yang Y., Li Y., Li X. A survey on real-time operating systems for embedded systems in Internet of Things. *Journal of Ambient Intelligence and Humanized Computing*, 2019, vol. 10 (6), pp. 2423–2437.

15. Baccelli E., Hahm O., Gunes M., Wählisch M., Schmidt T. C. RIOT OS: towards an OS for the Internet of Things. *2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, Turin, Italy, 2013, pp. 79–80. – DOI: 10.1109/INFCOMW.2013.6970748.

Для цитирования:

Шперлинг В.К., Якименко А.А. Разработка архитектурного решения программного обеспечения для устройств интернет-вещей // Системы анализа и обработки данных. – 2023. – № 2 (90). – С. 43–58. – DOI: 10.17212/2782-2001-2023-2-43-58.

For citation:

Shperling V.K., Yakimenko A.A. Razrabotka arkhitekturnogo resheniya programmnogo obespecheniya dlya ustroistv internet-veshchei [Development of an architectural software solution for Internet of Thing devices]. *Sistemy analiza i obrabotki dannykh = Analysis and Data Processing Systems*, 2023, no. 2 (90), pp. 43–58. DOI: 10.17212/2782-2001-2023-2-43-58.