УДК 004.031.6

ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ МИКРОКОНТРОЛЛЕРНОЙ СИСТЕМЫ

Тыманович Н.А., студент, Скудняков Ю.А., канд. техн. наук

Белорусский государственный университет информатики и радиоэлектроники
Институт информационных технологий
г. Минск, Республика Беларусь

Скудняков Ю.А.– канд. техн. наук, доцент каф. ИСиТ

Аннотация. В данной работе для автоматизации функционирования микроконтроллерной системы разработано программное обеспечение, использование которого позволяет повысить производительность и качество обработки информации системой.

Ключевые слова. автоматизация, микроконтроллерная система, производительность, качество, обработка информации.

Введение. Высокопроизводительная, надежная и высокоточная работа современной микроконтроллерной системы (МКС) возможна только при использовании эффективного программного обеспечения (ПО). Создание высококачественной МКС, выполняющей заданные функции в автоматическом режиме, является актуальной задачей. В рамках данной работы разработка ПО выполнялась для МКС, реализующей функции мониторинга и управления объектами и процессами разного назначения.

Основная часть. Цель работы состоит в исследовании, разработке и практическом применении ПО построения и функционирования МКС для повышения эффективности контроля состояний управляемых объектов и процессов и корреляции их характеристик и параметров на основе использования обратной связи и референсных значений.

При выполнении работы проведен анализ существующих проблем и подходов, имеющих место в области автоматического мониторинга, управления и стабилизации динамических величин, метрик и параметров, которые не могут быть линейно и предсказуемо установлены.

В результате проведения исследований разработано ПО для эффективного мониторинга и управления нелинейных динамических величин, которые нельзя статически задать при изготовлении изделия.

Применение полученных результатов разработки и последних достижений в области автоматизации мониторинга и управления различных объектов и процессов позволяет создавать автономные комплексы, которые могут эффективно сохранять и стабилизировать параметры системы, несмотря на изменяющиеся внешние факторы.

При разработке ПО МКС, с привязкой к аппаратным решениям, используются компиляция и линковка, которые имеют свою специфику при работе с bare metal проектами, учитывая проводные и беспроводные интерфейсы, датчики и микроконтроллеры (МК), которые непосредственно взаимодействуют с ними. За основу взят официальный SDK производителя на базе проекта Zephyr. Zephyr – это операционная система реального времени встраиваемых систем для МК. Zephyr включает в себя ядро, компоненты и библиотеки драйверов устройств, стеков протоколов, файловых систем и обновления встроенного ПО, необходимые для разработки полноценного прикладного ПО. Использовать SDK производителя вынуждает не только то, что туда, помимо, непосредственно, драйверов периферии МК, входят некоторые драйверы датчиков, но и то, что актуальная реализация bluetooth low energy стека выполнена именно на базе данной ОСРВ. Но это, в свою очередь, накладывает некоторые ограничения на использование сторонних языков программирования, которые изначально несовместимы с C, такие как, например, Rust.

Чтобы использовать SDK производителя, имея полный доступ ко всем возможностям МК, но не ограничиваться C, его библиотеками и т.п., а можно поступить следующим образом: основа сборки и уровень доступа к периферии, вместе с ее драйверами, будут использоваться из SDK производителя, но логика работы приложения может быть выполнена на любом другом компилируемом языке программирования без сборки «мусора», который может иметь совместимость с C ABI. Достаточно реализовать некоторые логические компоненты на нужном языке программирования, собрать его как статическую библиотеку в C ABI, после чего использовать полученные функции непосредственно в нужных местах проекта на базе SDK, предварительно указав статическую библиотеку в соответствующих файлах конфигурации сборки. Таким образом, можно использовать произвольный инструментарий и почти нет ограничений в наборе технологий, с которыми можно работать. Почему почти? Работа с прерываниями должна проходить только через SDK производителя, если он хоть где-то участвует в проекте. Дело в том, что, если появится необходимость использовать прерывания на другом языке программирования, придется создавать таблицу векторов и оперировать ей, но SDK производителя будет также иметь свою версию таблицы векторов, часть которых он уже реализовал, что, в свою очередь, приведет к ошибке линковки. При удалении из SDK производителя таблицы

векторов не гарантируется работоспособность всех его компонентов, включая BLE стек, планировщик ОСРВ и т.п.

Рассмотрим особенности линковки и сборки прошивки под встраиваемые системы.

Языки программирования представляют собой средство описания вычислений для людей и машин. Прежде чем запустить программу, ее необходимо преобразовать в форму, которая может выполняться на компьютере. Генерация переносимой программы на машинном языке (часто именуемой объектным модулем (object module)) обеспечивает возможность раздельной компиляции подпрограмм. Набор переместимых объектных модулей может быть скомпонован в одно целое и загружен для выполнения [1].

Создание скрипта компоновщика – неотъемлемый процесс при создании экосистемы для новых микропроцессоров [2,3]. Без него не будет возможности выполнения исполняемой прошивки, так как для каждого МК необходимо, в зависимости от архитектуры, предпочтение производителя и т.п., особая подготовка конечного бинарного файла в соответствующую форму.

Процесс компиляции под bare metal (процессоры без установленной OS), в общем, справедлив и для x86/amd64 процессора,за исключением некоторых особенностей: скомпилированная программа выполняется не на том же устройстве, на котором она компилируется, появляются дополнительные опции для указания модели памяти и таблицы векторов прерываний. По этой причине, хоть компоновщик и является частью компилятора, его стоит отдельно рассматривать в контексте, зависящем от таких факторов, как, например, наличие/отсутствие OS, архитектуры, программной модели и т.п.

В большинстве случаев компилятор может сам получить всю необходимую информацию для подготовки конечного исполняемого объекта, но иногда, всё же, приходится вмешиваться в процесс его работы. Хоть принято считать стартовой точкой программы функцию main(), но это, на самом деле, не совсем корректно. До вызова «главной функции» происходит довольно большое количество операций.

Когда запускается сборка проекта, каждый модуль собирается в объектный файл. По большей части он состоит из машинного кода под заданную архитектуру, но он не является «автономным», т.е. не может быть запущен. Компилятор помещает туда дополнительную информацию: ссылки на функции и переменные, определённые вне модуля в виде таблицы.

На рисунке 1 представлена схема этапов, начиная от проектирования ПО и заканчивая вызовом main функции.

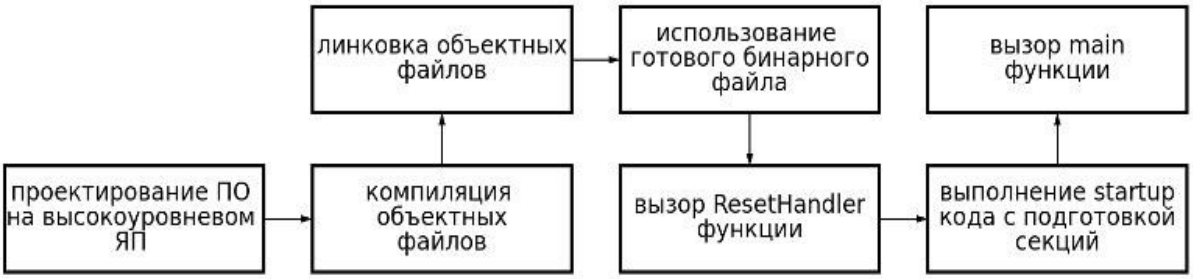

Рисунок 1 – Этапы проектирования ПО

Каждый объектный файл состоит из одной или нескольких секций, в которых хранится либо код, либо данные. Для GCC программный код складывается в секцию .text, проинициализированные глобальные переменные с их значениями складываются в секцию .data, обнуленные - в секцию .bss. Выходной файл компоновщика – это такой же объектный файл, с точно таким же форматом хранения кода и данных. Другими словами, компоновщик группирует код и данные по секциям, пытаясь разрешить внешние связи. Такой файл, однако, не может быть исполнен на целевой платформе. Проблема в том, что в нём нет информации об адресах, где данные и код должны храниться. Следующим в игру вступает локатор.

Любая программа, на любом языке, имеет некоторые требования к среде. Для Java это виртуальная машина, для Python – интерпретатор, а для C – наличие памяти для стека (грубое упрощение). Место под стек должно быть выделено до начала выполнения программы, и этим занимается startup-файл. Он же выполняет и другие задачи, например, производит определенные действия с прерываниями, перемещает нужные данные в оперативную память, и только после этого вызывает функцию main (причем, функция может называться по-другому).

В некоторых компиляторах предусмотрена отдельная утилита, которая занимается локацией адресов, т.е. сопоставлению физических адресов в памяти целевой платформы соответствующим секциям. В GCC она является частью компоновщика. Информация, необходимая для данной процедуры, хранится в специальном файле – в скрипте компоновщика. При работе с интегрированной средой разработки данный файл генерируется автоматически, исходя из указанных параметров МК. В

некоторых случаях бывает полезно изменить его, чтобы добиться желаемого результата: например, поместить код в оперативную память и выполнять программу оттуда.

Рассмотрим основные составляющие данного файла. Текстом «ENTRY (Reset Handler)» указывается компоновщику, с чего начинать запуск. Функция «ResetHandler» определена в startup-файле, её смысл пояснен чуть выше, startup-файл берёт значение этой переменной именно отсюда. Затем определяется минимальный размер для кучи и стека. Эти данные используются в самом конце сборки, когда компоновщик проверяет, достаточно ли места в памяти. В скрипте также должны быть перечислены все виды памяти, доступные в системе.

Рассмотрим подробно механизм реализации и обработки прерываний Interrupt and Events. В МК архитектуры Cortex-M есть два понятия, которые часто путают Interrupt и Event. Event – это событие (аппаратное или программное), на которое могут реагировать ядро или периферийные блоки. Одним из вариантов реакции может быть – прерывание. Interrupt – это прерывание работы программы и переход управления в специализированный участок – обработчик прерывания. Взаимосвязь между Event и Interrupt заключается в следующем: каждый Interrupt вызывается Event, но не каждый Event вызывает Interrupt.

Помимо прерываний, события могут активировать и другие возможности МК. Управление и обработка прерываниями производится контроллером приоритетных векторных прерываний NVIC (Nested Vectored Interrupt Controller).

При возникновении некоторого события контроллер прерываний автоматически прерывает выполнение основной программы и вызывает соответствующую функцию обработки прерываний. После выхода из функции обработчика прерываний программа продолжает выполнение с того места, где произошло прерывание. Все происходит автоматически (при правильной настройке NVIC, но об этом ниже). Из самого названия видно, что контроллер NVIC поддерживает вложенность прерываний и приоритеты. Каждому прерыванию при настройке NVIC присваивается свой приоритет. Если во время обработки низкоприоритетного прерывания возникает высокоприоритетное, то оно, в свою очередь, прервет обработчик низкоприоритетного прерывания.

При инициации прерывания NVIC переключает ядро в режим обработки прерывания. После перехода в режим обработки прерывания регистры ядра помещаются в стек. Непосредственно во время записи значения регистров в стек осуществляется выборка начального адреса функции обработки прерывания. В стек перемещается регистр статуса программы (Program Status Register (PSR)), счетчик программы (Program Counter (PC)) и регистр связи (Link Register (LR)). Благодаря этому запоминается состояние, в котором находилось ядро перед переходом в режим обработки прерываний. Также сохраняются регистры R0 – R3 и R12. Эти регистры используются в инструкциях для передачи параметров, поэтому, помещение в стек делает возможным их использование в функции обработки прерывания, а R12 часто выступает в роли рабочего регистра программы.

По завершении обработки прерывания все действия выполнятся в обратном порядке: извлекается содержимое стека и, параллельно с этим, осуществляется выборка адреса возврата. С момента инициации прерывания до выполнения первой команды обработчика прерывний проходит 12 тактов, такое же время необходимо для возобновления основной программы после завершения обработки прерывания. NVIC поддерживает прерывания с различными приоритетами, которые могут прерывать друг друга. При этом могут возникнуть различные ситуации, обработка которых по-разному оптимизирована. Например, приостановка низкоприоритетного прерывания. В этой ситуации обработка низкоприоритетного прерывания прекращается. В следующие 12 циклов выполняется сохранение в стек нового набора данных и запускается обработка высокоприоритетного прерывания. После его обработки, содержимое стека автоматически извлекается и возобновляется обработка низкоприоритетного прерывания. Больших отличий от прерывания основной программы не наблюдается.

Также возможен вариант непрерывной обработки прерываний. Эта ситуация может возникнуть в двух случаях: 1) если два прерывания имеют одинаковый приоритет и возникают одновременно, 2) если низкоприоритетное прерывание возникает во время обработки прерывания с более высоким приоритетом, то в этом случае, промежуточные операции над стеком не производятся: происходит только загрузка адреса обработчика низкоприоритетного прерывания и переход к его выполнению. Отказ от операций над стеком экономит 6 тактов. Переход к следующему прерыванию происходит не за 12 тактов, а всего за 6. Иногда происходит запаздывание высокоприоритетных прерываний. Ситуация возникает, если высокоприоритетное прерывание происходит во время перехода к обработке низкоприоритетного (за те самые 12 тактов).В этом случае переход к высокоприоритетному прерыванию будет происходить не менее 6 тактов с момента его возникновения (время, необходимое для загрузки адреса обработчика прерывания и перехода к нему).

Возврат в низкоприоритетное уже описан выше.

Помимо простой установки приоритета прерываний, NVIC реализует возможность группировки приоритетов. Прерывания в группе с более высоким приоритетом могут прерывать обработчики прерываний группы с более низким приоритетом. Прерывания из одной группы, но с разным приоритетом внутри группы, не могут прерывать друг друга. Приоритет внутри группы определяет только порядок вызова обработчика, когда были активизированы оба события.

Для того, чтобы включать/выключать различные векторы прерываний,существует маскирование прерываний. Если прерывание замаскировано, это не означает, что периферия не генерирует события. Просто NVIC не вызывает обработчик этого события. Все возможные прерывания, поддерживаемые NVIC, записываются в таблицу векторов прерываний.

По сути своей, таблица векторов прерываний есть ничто иное как список адресов функций обработчиков прерываний.Номер в списке соответствует номеру прерывания. NVIC поддерживает до 240 различных векторов прерываний. Но реализация уже зависит от конкретного производителя. В описании ядра стандартизованы только прерывания исключений ядра: Reset; NMI; HardFault; MemManage; BusFault; UsageFault; SVCall; PendSV; SysTick.

Из начала флеш-памяти ядро считывает значение SP (stack top addres) и PC (reset routine location).

Таким образом, автоматически начинает выполняться функция с адресом, считанным в регистр PC. Это может быть, например main. Кроме наличия обязательных четырех компонентов, может находиться дальнейшая таблица векторов прерываний. Можно разместить таблицу векторов прерываний в другой области памяти, но тогда необходимо сообщить NVIC, куда передвинута таблица.

Заключение. В итоге поведенного исследования:

— разработана схема этапов проектирования ПО МКС;

— рассмотрены особенности компилятора для разработки и применения ПО в процессах проектирования и функционирования МКС;

— ПО МКС разработано на языках C [4] и Rust [5,6];

— код на языке C используется для взаимодействия с датчиками и BLE, по которому может осуществляться передача данных. SDK (набор инструментов) производителя микроконтроллера использует C, а стало быть, если потребуется использовать проприетарные протоколы, такие как BLE, придется использовать их SDK на C.

Язык Rust является более мощным и комфортным в использовании, так как он предоставляет более гибкие механизмы при написании кода, такие как, например, замыкания, дженерики (generics) и др. Поэтому было принято решение использовать его для написания на нем ПО ПИД – регулятора. Так как Rust может использовать C ABI, можно раздельно применять оба языка в одном проекте.

Список использованных источников:

1.Ахо, А. Компиляторы. Принципы, технологии и инструментарий / А. Ахо, Д. Ульман, М. Лам. – М.: Изд-во Диалектика, 2019. – 29 с.

2.Руководство новичка по эксплуатации компоновщика [Электронный ресурс]. – Режим доступа: https:// habr.com/ru/post/150327/. – Дата доступа : 17.09.2022.

3.Создание скриптов компоновщика [Электронный ресурс]. – Режим доступа : https://www.tuneit. ru/web/ myaut/home/-/blogs/25692/. – Дата доступа: 17.09.2022.

4.Демидович, Е.М. Основы алгоритмизации и программирования. Язык СИ / Е.М. Демидович. – Минск: Бест¬принт, 2001. – 440 с.

5.Клабник, С. Программирование на Rust / С. Клабник, К. Николс. – Питер: ЛитРес, 2021. – 592 с.

6.Тымanович, Н.А. Программирование на языке Rust в Embedded-системах / Н.А. Тымanович, Ю.А. Скудняков // Материалы XXV Международной научно-технической конференции «Современные средства связи», Минск, 22-23 октября 2022 года. – Минск: БГАС, 2022. – С.174.

UDC 004.031.6

MICROCONTROLLER SYSTEM SOFTWARE

Tymanovich N.A., Skudnyakov Yu.A.

Institute of Information Technologies of the Belarusian State University of Informatics and Radioelectronics, Minsk, Republic of Belarus

Skudnyakov Yu.A. – Candidate of Engineering Sciences, Associate Professor

Annotation. In this paper, software has been developed to automate the functioning of the microcontroller system, the use of which allows to increase the productivity and quality of information processing by the system.

Keywords. Automation, microcontroller system, productivity, quality, information processing.