Санкт-Петербургский Государственный Университет
Математико-Механический факультет
Кафедра системного программирования

Программная платформа для встраиваемых решений

Курсовая работа студента 445 группы
Козлова Антона Павловича

Научный руководитель Бондарев А. В.

Санкт-Петербург

2012

Оглавление

Введение

Нас окружает множество маленьких устройств, каждое со своей логикой, может быть, довольно простой, но её реализация с помощью базовых электрических компонентов и интергральных схем являлась бы слишком большой и сложной в разработке. Поэтому, вместо схем с использованием разных компонент используются микроконтроллеры, MCU в противовес CPU, так как помимо процессора, MCU содержит набор переферии:

- RAM и Flash память для программ;
- Контроллер GPIO (Ввод-вывод общего назначения, способен гененировать электрические логические сигналы);
- Последовательные интерфейсы различных стандартов;
- ADC, DAC, PWM, USB, RTC, …, их набор может существенно варьироваться.

Помимо разных наборов переферии, так же существует большой выбор производителей, среди которых отметим Intel, Atmel, ST, Phillips.

MCU достаточно дешевы в производстве, что позволяет широко их применять на практике. Также, с течением времени удешевляется производство и растет вычислительная мощность, это позволяет прогнозировать еще более широкое применение MCU.

Для программиста MCU представляет собой сильно ограниченную в ресурах платформу. Хотя и существует целый ряд, с широким диапазоном вычислительной мощности (и, соответственно, цены), как правило, MCU выбирают с минимальным запасом этой мощности, так как любая избыточность отразится на цене продукта. Если отбросить совсем слабых и мощьных представителй, то типичными являются 1-50 MGhz, 1-100 KB RAM-памятьи, 10-1000 KB Flash памяти.

С учетом сказанного, можно определить трудности на нескольких этапах разработки:

1. Планирование. С учетом требования минимизации избыточной мощности контроллер сложно выбрать, так как излишек нежелателен, а недостаточность вообще не может иметь место. Также, следует аккуратно выбирать производителя. Проблема вырисовывается четче, когда требуемая вычислительная мощность находится на стыке двух линеек MCU, сильно отличающихся в цене.
2. Кодирование. Написанная программа должна быть эффективна. Так же, API выбранного производителя может быть неудобен или не подходить к разработке конкретного продукта.
3. Внедрение/сопровождение/следующая версия продукта. Так как избыточная вычислительная мощность сведена к минимуму, возможно, для требования, которое возникло на этапе внедрения, сопровождения, или было выработано для следующей версии продукта, просто не хватит ресурсов.

Целью курсовой работы является разработка платформы, решающей перечисленные проблемы.

Обзор существующих решений

Как уже говорилось, MCU представляет собой платформу с ограниченными ресурсами. Начальное окружение для разработки, предоставляемое производителем, состовляют MCU (с отладочным устройством), компилятор и документация. Дальше программист сам создает основу для своего приложения. Рассмотрим подходы к осуществлению проектирования архитектуры и определению переферии MCU (включая внешнюю), причем критерием оценки будет возможность и удобство добавления новой функциональности.

Архитектура

Среди архитектур можно выделить несколько типов. Можно выделить классы архитектур, разрабатываемых программистами под задачу.

- Бесконечный цикл. Все ПО состовляет один бесконечный цикл, который состоит из опроса оборудования и осуществления соответствующих действий.

  Достоинства: простота реализации.

  Недостатки: энергопотребление (процессор находится в режиме исполнений инструкций); подходит только для однозадачных приложений.

- Программирование прерываний. Настраиваются прерывания на определённые внешние события

  Достоинства: хорошее энергопотребление, наличие приоритетов.

  Недостатки: приходится явно программировать поддержку вложенных прерываний, также, если одно внешнее событие требует нескольких действий, или несколько внешних событий генерируют одно прерывание, то это требует отдельного рассмотрения и, возможно, усложнение программного кода. Также, следует следить, чтобы обработчик прерывания проводил как можно меньше времени в контексте прерывания.

- Кооперативная многозадачность. Несколько приложений разбиваются на фазы выполнения, высокоуровненые операции. Поток выполнения идет согласно очереди таких операций, причем, операции различных приложений выполняются согласно приоритету приложений, но сами приложения не учитываюся (т. е. операции разных приложений с одним приоритетом могут лежать в очереди в произвольном порядке)

  Достоинства: гибкая система приоритетов, определяемая пользователем.

Недостатки: приложения приходится разбивать на высокоуровневые оперцаии в ручном режиме, отсутствует вытиснение, высокоприоритетное приложение не может прервать низкоприоритетное, пока то выполняется.

- Многозадачность. Приложения организуются в потоки, как в операционных системах общего назначения.

  Достоинства: гибкое API работы с потоками, понятие приоритета, приложения можно писать не задумываясь об остальных частях.

  Недостатки: трудное в реализации.

К сожалению, перечисленные методы разработки собственной архитектуры редко предусматривают какую-либо системность, а пишутся под конкретную задачу. Поэтому, расчитывать на заменяемость, как правило, не приходится.

Если разрабатываемая задача достаточна сложна, а времени на разработку отведено не много, программист может предпочесть адаптировать уже существующую платформу для своих нужд, т. е. операционную систему. Операционная система представляет собой некоторую оточенную основу, системную часть, которая отделена от логики приложения, поэтому является более предпочтительным решением. Сразу отметим, что для использования в разработке с MCU пригоден только класс ОС реального времени.

Но и не все ОС РВ могут быть использованны. Рассмотрим классы ядерной архитектуры ОС РВ.

- Монолитное ядро. Содержит в себе все подсистемы. Трудность использования на MCU заключается в размере кода ядра ядра, так и используемой под данные оперативной памяти, которые слишком велики для микроконтроллеров.
- Микроядро. Содержит только минимальные подсистемы, такие как управление процессами и, возможно, памятью. Все остальные

подсистемы реализованны как службы, единственным средством связи которых между собой и микроядром являются сообщения. Основным препятствием использование микроядра являются накладные расходы на пересылку сообщений. Также, можно отметить, что существует аппартное обеспечение, которое не поддерживает в должной мере механизмы защиты сервисов микроядра.

- Модульные ядра, экзоядра. Наиболее вероятные типы для использования в микроконтроллерах. Правда, следует заметить, что существующие решения на модульной архитектуре все равно слишком громоздки (Linux[2]), а экзоядра обладают слишком большими модулями (Embedded RT OS [3]), что не позволяет тонко их настраивать. Это затрудняет их адаптацию под конкретное приложение.

Лучшим решением с точки зрения гибкости является экзо/модульное ядро с оглядкой на вычислительную мощность платформы, другими словами, после выбора конкретной операцинной системы выбор MCU сужается до тех, которые в принципе могли бы быть поддержаны данной ОС, что негативно отражается на требуемой гибкости. Лучшими с точки зрения потребляемых ресурсов являются методы с разработкой собственной простой архитектуры. Таким образом, сейчас нельзя назвать лучшего подхода.

Внешние соединения

Гораздо меньше в современных подходах уделяется внимание внешним соединениям, хотя должное внимание к этому вопросу позволило бы иметь унифицированный доступ ко внутренней и внешней переферии по строго определённым каналам, что помогло бы, как минимум, легкому переносу на различные аппаратные платформы, а в более широком смысле гарантировало бы, что работа с переферией осуществляется правильным образом.

Как пример полезности описания внешних соединений существует программно-аппаратная платформа Arduino. Среди прочих решений, позволивших Arduino стать хорошим продуктом, является строго

фиксированная схема соединения с внешними устройствами. Так, пользователи смогли к основной платформе по универсальному интерфейсу, подключать различные дополнительные устройства. На самом деле, этот интерфейс является скорее соглашением о расположении контактов, спецификацией, а не каким-либо техническим решением. Таким образом, программистам на Arduino было гарантированно, что конкретный класс устройств будет подключаться известным образом. Такое ограничение можно рассматривать как описание аппаратной платформы, только абсолютно не гибкое. Но даже из такого примитивного описания получилось извлечь выйгрыш, ведь этот механизм избавил программиста от написания собственного средства конфигурирования внешних соединений.

Итого, представленные подходы часто забывают об общих чертах предметной области, т. е. программирования под MCU. Предлагается разработка собственной платформы, которую можно будет использовать во всех случаях, когда требуется одна из названных архитектур приложения. Также, платформа должна предоставлять средства описания аппаратной платформы.

Решение

Как основу для платформы предлагается использовать ОС PB Embox. Описываемые проблемы будут решаться с помощью продвинутой системы сборки, как результат аппробации заявленных свойств гибкости будет осуществленно портирование на новую платформу (MCU производителя ST, архитектуры ARM Cortex-M3).

В пользу этого выбора говорит следующее. Все части ОС представляют собой модули. Система сборки позволяет указать только необходимые для программиста свойства системы, все требуемые для этих свойств операционной системы модули будут выведены и включены в сборку ОС. Эти особенности позволяют гарантировать, что ОС не содержит лишних частей, что важно в условиях ограниченности по памяти во время программирования для MCU.

Как пример, подкрепляющий выбор Embox в качестве основы для разрабатываемой платформы можно привести поддержку многозадачности (многопоточность плюс управление ресурсами, ассоцированными с потоками). Сама по себе Embox является многозадачной системой. Приложения на MCU редко пользуются многозадачностью, хотя порой используют многопоточность. Исключив компоненты зависящие от поддержки многозадачности, такие как POSIX-совместимость, удается получить многопоточную ОС без поддержки многозадачности, в которой потоки не изолированны друг от друга. Причем многозадачность не является единственным подобным примером. Исключить можно и многопоточность, сведя ОС к набору библиотек и сервисов для управления прерываниями.

Итого, на базе Embox можно построить систему, которая будет адаптированна под конкретную задачу, можно указать архитектуру, лучше всего подходящую в данный момент. Таким образом, будут использованны лучшие черты из существующих подходов.

Система сборки

В настоящее время используется для сборки Embox используется система Mybuild. В её отличительные черты входят:

- Ориентированность на разработку в модульном стиле.
- Специальный DSL для описания модулей.
- Модули имеют зависимости, наследование, существует понятие абстрактного модуля.

Сборка описывается доступными модулями и списоком модулей, подлежащих включению в образ.

Рассмотрим пример описания модуля:

module foo extends superFoo{ //описание модуля, наследование

@IncludePath(«../include»)

source «bar.c» // включает исходный код, при сборке

// следует учитывать каталог с .h, определённый выше

option number baz = 10 //опция модуля, будет доступна bar.c, если не

//переопределно, то равно 10.

}

Отметим важные для нас черты:

- Опции. Являются числовыми/булевыми/строковыми переменными. Их значение определяется на этапе конфигурирования (включения в образ). Если не указано, используется значение по умолчанию, как описано выше.
- Аннотации. Здесь используются как указание дополнительного каталога с header-файлами для компилятора. В общем, аннотации могут пр..//именяться к любому элементу описания модуля. Главное назначение аннотаций – расширять язык описания модулей до полной обкатки предоставляемых возможностей. Это позволяет сохранять ядру языка малый размер, что положительно сказывается на скорости разбора.

Так же, рассмотрим пример включения модуля в сборку. Для этого используется простой синтаксис:

@For(superFoo)

include foo(baz=11) // включение foo, переопределив значение baz на 11

Подчеркнем возможность ставить аннотации над описанием включения. В данном случае, @For обозначает, что foo включается как явный потомок superFoo. Запись может быть полезна, так как осуществляет контроль, действительно ли foo является потомком superFoo.

На основе аннотаций будем строить расширения языка,

предназначенные для описания платформенной части. Как модель платформенной части я использовал цифровые контакты процессора. Описание контактов, управляемых конкретным модулем я сделал через числовые опции. Тогда встал вопрос корректности, ведь нескольким модулям не может быть переданно один и тот же номер контакта.

Мною была введена аннотация @Type(«ID») над опциями, идентифицирующе множество опций как один логический тип. Сразу оговоюсь, в дальнейшем появится возможность определять типы естественным образом, например, как в C. Сейчас использование @Type выглядит так:

```
@Type(«pin»)
option number led_pin = 13
```

Здесь мы определяем что опция led_pin на самом деле имеет логический тип «pin». Следущим этапом стала реализация аннотации @Unique, которую можно использовать совместно с @Type. Семантика её в точности следующая: если две опции одного или различных модулей помечены одним и тем же логическим типом, и хотя бы у одного и из них указан @Unique, то будет сгенерированная ошибка. Для примера, если описания модулей таковы:

```
module foo {
    @Type(«barType») @Unique
    option number fooOption = 1
}
module baz {
    @Type(«barType») @Unique
    option number bazOption
},
```

то включение foo(), baz(bazOption=1) или foo(fooOption=2), baz(bazOption=2)

сгенерируют ошибку: Unique type barType assigned second time to one value in baz (previous assignment in foo), причем, будет указана позиция в файле конфигурации, где указано включение этих модулей.

Данный механизм также может с успехом использоваться в дальнейшем не только для описания контактов, но и для определения номера softirq (используется для сообщения обработчик прерывания → приоритеный системный уровень) или номера syscall'a (пользовательский режим →режим ядра).

Следущим шагом стало разработка аннотации @NumConstraint. Её назначение заключается в проверке целевой опции на принадлежность некоторому интервалу. @NumConstraint обладает необязательными параметрами:

- gt – строго больше
- ge – больше либо равно
- lt – строго меньше
- le – меньше либо равно.

@NumConstraint может быть использованна несколько раз над одной опцией. Соответственно, при определении опции foo модуля bar как:

@NumConstraint(ge=0,lt=4)

option number foo = 10

включение bar(), bar(foo=5) сгенерирует ошибку: Option bar.foo constraint check error.

Таким образом, полное описание led_pin могло бы выглядеть так (считаем, всего 32 контакта, не знаем, где будет остановлен светодиод)

@Type(«pin») @Unique

@NumConstraint(ge=0,lt=32)

option number led_pin

Следует подробнее рассказать о внутреннем устройстве системы сборки. Для описания возможностей языка существует некоторый набор базовых элементов. Когда система загружает файл описания модуля, описание превращается в некоторую модель, составленную из упомянутых базовых элементов. Другими словами, происходит *инстанцирование* модели языка в модель модуля.

Похожий подход я применил при создании модели сборки из модели описания модуля. Была разработана система, переводящая модель описания модулей и описание необходимых модулей в модель, содержащюю в себе все необходимую информацию для непосредственно сборки. Подход хорошо подходит для проверки корректности, так как появляется возможность отследить все включенные в сборку модули без непосредственной сборки, также, это позволит создавать несколько сборок одновременно, что поможет автоматизированной системе сборки.

Создание модели сборки происходит поэтапно. Сначала список требуемых модулей пополняется списком их всех зависимостей, затем просходит проверка, что каждый требуемый абстрактный модуль имеет реализацию, далее происходит присваивание значений опциям. Каждый из этих этапов может добавлять модулей в сборку, например, в случае если абстрактный модуль включен без реализации, причем существует единственная реализация, эта реализация будет включена в сборку.

Такой подход позволил легко встроить поддержку @Type, @Unique, @NumConstraint в процесс создания модели сборки. Также, был начат процесс разработки требований для фреймворка аннотаций, позволящему пользователю системы сборки определять собственные аннотации и их семантику.

Таким образом, получилось чётко определить сопоставить контакты и модули. Надо сказать что получившийся подход полностью покрывает подход

фиксированных внешних соединений и существенно его превосходит. Также, использованные алгоритмы обеспечивают нулевой расход ресурсов для поддержания системы, все логика выполняется во время подготовки к сборке системы.

Как утверждалось, описание аппаратной платформы поможет в переносе с одной платформы на другую. Чтобы проверить заявленную гибкость, я осуществил перенос программного кода, управляющего моторами согласно команд, приходящих по Bluetooth интерфейсу, с платформы Lego NXT на свою собственную. Собственная разработанная платформа использует совершенно другой модуль Bluetooth и аппаратный драйвер моторов. Поэтому, удалось осуществить перенос, заменив лишь модули драйвера Bluetooth и драйвера моторов. Можно сделать вывод, что подход, использующий описание аппаратной платформы хорошо работает на практике.

Гибкость в архитектуре

Заявлялось, что разработка на основе Embox достаточно гибка. Чтобы подтвердить это, я произвел портирование на д MCU STM32F1xx фирмы ST, архитектура ARM Cortex-M3. Это решение было выбрано за низкую цену, встроенный отладчик. Данный микрокнтроллер обладает 4 KB RAM памяти и 128 KB Flash для программ, 24 Mghz тактовой частотой и базовым набором переферии, что позволяет опробовать новую платформу на MCU среднего уровня[1].

Cortex-M3 представляет собой процессор профиля ARMv7M, что означает поддержку только системы команд Thumb. Таким образом, первым этапом портирования стало написание загрузочного кода, так как, хотя в Embox присутствует поддрежка ARM систем, это поддержка обычного, не Thumb, набора команд.

Следующим этапом стала реализация драйвера GPIO интерфейса,

1 Название этой линейки Medium Density

позволяющего устанавливать логические значения на контактах процессора. В системе уже имелся интерфейс GPIO, но, оказалось, что он не полностью подоходит. Дело в том, что ранее поддержанные платформы имели количество контактов, такое, что полное описание их состояний умещалось в машинное слово. На новой же платформе количество состояний контактов больше, чем машинное слово. Поэтому, было решено расширить интерфейс и создать множество виртуальных GPIO портов, где каждый порт заведует количеством контактов, все возможные состояния которых умещаются в машинное слово. Такое решение позволило создать полную поддержку всех контактов и сохранить совместимость со старым интерфейсом

Наконец, был реализован драйвер системного таймера. Ввиду простой модели программирования обработки прерывания на данной платформе, проблем с реализацией этой части не возникло.

После реализации перечисленных систем, стало возможно использовать платформу на серии MCU STM32F1xx.

Заключение

В ходе работы над курсовой был создан прототип платформы для разработки на MCU. В силу модульного подхода, платформа содежит только необходимые приложениям части, а также позволяет, при необходимости, эти части заменять на предоставленные в составе платформы или свои собственные. Система сборки занимается описанием этих частей и предоставляет средства описания аппаратной платформы, что выливается в простоту переноса готового приложения с одной аппаратной платформы на другую. Полученное решение достаточно компактно и производительно, что демонстрируется работой на микроконтроллере среднего уровня.

Программный код находится по в репозитории по адресу http://code.google.com/p/embox/source/list.

Дальнейшими направлениями работы станут средства описание

доступной переферии, вклбючая внешнюю, и средств коммуникации, что отказаться от всех неявных путей программирования устройств переферии на MCU.

Список использованной литературы

1. A. Sloss, D. Symes. ARM System Delveoper's Guide. Morgan Kaufmann, 2004.
2. D. Bovet, M. Cesati. Understanding the Linux Kernel. O'Reily, 2000.
3. http://www.smxrtos.com/mt.htm
4. Э. Таненбаум. Архитектура компьютера. «Питер», СПб, 2009.
5. ARMv7-M Architecture Reference Manual. ARM Ltd., 2010.
6. STM32F100 Refernce Manual. STMicroelectronics, 2011.