Санкт-Петербургский государственный университет

Программная инженерия
Кафедра системного программирования

Мелентьев Кирилл Игоревич

Разработка системы программирования гетерогенных архитектур с использованием LLVM

Бакалаврская работа

Научный руководитель:
ст. преп. Я. А. Кириленко

Рецензент:
инженер-программист "Cogent Embedded" Р.В.Мешкевич

Санкт-Петербург
2017

SAINT-PETERSBURG STATE UNIVERSITY

Software Engineering
Software Engineering Department

Kirill Melentev

Development of LLVM-based system for heterogeneous architectures programming

Bachelor's Thesis

Scientific supervisor:
senior lecturer Iakov Kirilenko

Reviewer:
software engineer "Cogent Embedded" Roman Meshkevich

Saint-Petersburg
2017

Оглавление

Введение

Многие современные киберфизические системы (например, роботы, системы управления "умным домом", "интернет вещей") имеют гетерогенные архитектуры, то есть включают в себя процессоры и микроконтроллеры различных аппаратных архитектур. Такие системы обычно включают в себя более мощный процессор, осуществляющий общее управление, и дополнительные вспомогательные процессоры или микроконтроллеры, часто включающие в себя такие компоненты, как АЦП, ШИМ-контроллер и выполняющие специальные функции, например, такие, как работа с датчиками, моторами, манипуляторами. Типичными примерами конфигураций таких систем являются: STM32 discovery (периферийный микроконтроллер) + Odroid XU4 (управляющий), Arduino с Atmel AVR (периферийный) + Raspberry Pi (управляющий), MSP430 (периферийный) + OMAP-L138 (управляющий).

Традиционно каждый из компонентов такой гетерогенной системы программируется отдельно с использованием специфичных для платформы инструментов, программное обеспечение для него компилируется статически. Необходимость использования для каждого аппаратного компонента системы своих средств разработки (IDE и другие инструменты) и вести разработку в отдельных проектах усложняет процессы разработки и поддержки программного обеспечения.

Разработка дополнительно осложняется тем, что необходимо учитывать особенности работы каждого компонента. Например, вспомогательный микроконтроллер зачастую обладает малым количеством памяти и низкой вычислительной мощностью, поэтому для него часто создают базовую прошивку, способную выполнять лишь простейшие действия: чтение значений подключенных датчиков и отправка их на управляющий процессор; запись значений в регистры устройств, инициируемая со стороны управляющего процессора. Такая организация имеет недостаток в виде постоянной передачи данных между компонентами системы, поскольку в базовой прошивке вспомогательного контроллера нет даже части логики алгоритма, который полностью испол-

няется управляющим процессором.

Контроллер TRIK [9], предназначенный для разработки киберфизических систем, имеет гетерогенную архитектуру. Он включает в себя управляющий процессор OMAP-L138 (архитектура ARM) и периферийный микроконтроллер msp430f5510.

В данной работе предлагается решение для программирования различных киберфизических гетерогенных систем, а также описывается его апробация на платформе TRIK.

1. Постановка задачи

Целью данной работы является разработка прототипа системы программирования киберфизических систем с гетерогенными архитектурами (включающими в себя процессоры различных аппаратных архитектур), позволяющая пользователю (программисту приложений для гетерогенной системы) использовать единые средства разработки для создания и поддержки программного обеспечения. Для достижения данной цели были поставлены следующие задачи.

1. Провести обзор существующих решений для программирования гетерогенных систем (не только киберфизических).
2. Спроектировать архитектуру системы программирования киберфизических гетерогенных архитектур.
3. Реализовать прототип спроектированной системы.
4. Провести апробацию прототипа системы на платформе TRIK.

2. Обзор

В данном разделе рассматриваются некоторые известные существующие решения для разработки программного обеспечения в системах, состоящих из разнородных компонентов.

WebSharper

Одним из примеров такого программного обеспечения являются веб-приложения. Веб-приложение включает в себя серверную и клиентскую части, которые обычно программируются раздельно. Фреймворк WebSharper [1] позволяет создавать цельное веб-приложение на F#, благодаря трансляции кода клиентских компонентов в JavaScript, а также обеспечивает прозрачное взаимодействие между клиентским и серверным кодом, например, транслируя вызов серверной функции из клиентского компонента в AJAX-запрос.

GWT

Фреймворк GWT [3] аналогичным образом облегчает разработку веб-приложения — он позволяет как клиентскую, так и серверную логику реализовать на языке Java, обеспечивая трансляцию клиентских компонентов в JavaScript код, а также делает взаимодействие между клиентом и сервером через AJAX более прозрачным.

N2O

Фреймворк N2O [6] для Erlang также генерирует JavaScript код, но, в отличие от предыдущих, он не только может транслировать исходный код клиентских компонентов в JavaScript заранее, но и на лету генерировать JavaScript код действий в ответ на клиентские события. Это возможно потому, что между клиентом и сервером поддерживается постоянный канал связи через WebSockets.

NVIDIA CUDA

Одним из популярных направлений в области гетерогенных систем является разработка программного обеспечения для систем с графическими ускорителями. Например, платформа CUDA [7] от NVIDIA позволяет разрабатывать приложение, использующее графический ускоритель. Разработка происходит на специальном расширении языка C, добавляющем особые аттрибуты для того чтобы помечать функции, предназначенные для выполнения на gpu, а также синтаксис для их вызова из кода, выполняемого на основном процессоре системы. Таким образом, код и для cpu, и для gpu создается на одном и том же языке и даже может находиться в одном файле с исходным кодом. Данная работа непосредственно не включает в себя программирование систем с графическими ускорителями, но сама идея использовать единую программную платформу для программирования всех компонентов системы также лежит в основе нашей работы.

Brahma.FSharp

Другой интересный пример — библиотека Brahma.FSharp [2], предоставляющая возможность транслировать F# цитирования (F# quotations — инструмент, предоставляющий разработчику доступ к исходному коду конструкций F# в виде синтаксического дерева) в OpenCL, таким образом позволяя разрабатывать на F# приложение, использующее графический ускоритель для выполнения вычислений. Причем в отличие от CUDA такой подход позволяет использовать устройства от различных производителей (не только NVIDIA).

LLVM

LLVM [5] — универсальная система анализа, трансформации и оптимизации программ, реализующая виртуальную машину с RISC-подобными инструкциями. Может использоваться как оптимизирующий компилятор этого байткода в машинный код для различных архитектур. На-

пример, она поддерживает архитектуры ARM, Atmel AVR, TI MSP430, которые часто применяеются во встраиваемых системах.

3. Архитектура системы

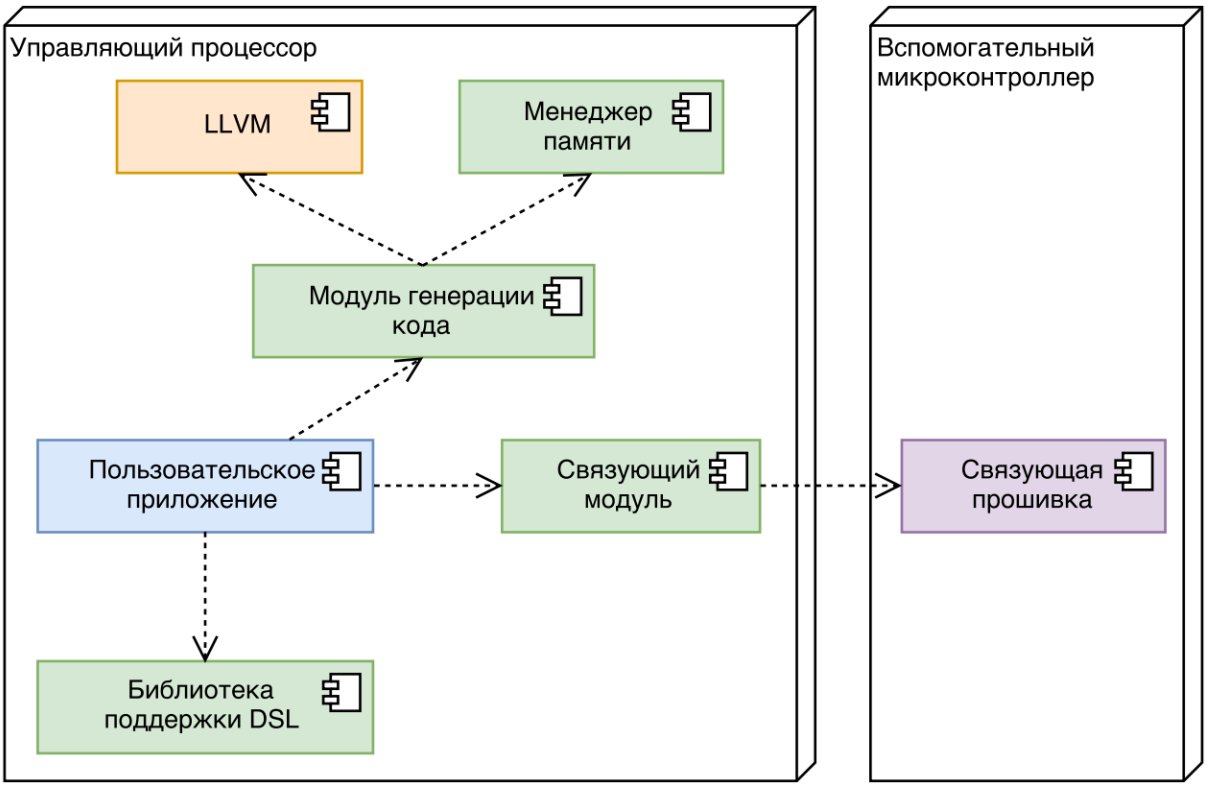

Рис. 1: Схема взаимодействия компонентов системы

В данной работе рассматривается аппаратная конфигурация системы, включающая в себя один мощный управляющий процессор с подключенным к нему вспомогательным микроконтроллером.

Рассмотренные продукты позволяют упростить разработку программного обеспечения для гетерогенных систем, предоставляя возможность создавать единый проект с использованием одного языка программирования.

Таким образом, было решено разработать систему с аналогичными возможностями для разработки на языке C++, как на одном из самых популярных языков разработки для встраиваемых систем [10]. Потенциально система может быть расширена или модифицирована для разработки приложений на других языках — далее в работе будут прокомментированы потенциальные возможности развития в этом направлении, но в целом этот вопрос выходит за рамки данной работы.

На основе обзора рассмотренных продуктов, были определены два

основных подхода, с помощью которых достигается возможность такой разработки.

- Статическое разделение и компиляция исходного кода для разных целевых платформ с помощью специальных аттрибутов или конструкций, например, как CUDA (компилятор nvcc) или GWT. Такой подход возможно организовать, модифицировав существующий компилятор (например, разработать плагин к компилятору), либо внедряя в процесс сборки проекта дополнительные препроцессинговые утилиты, разделяющие исходный код на разные целевые платформы, таким образом усложняя сборку проекта. Однако на практике используется множество различных компиляторов и их версий для разных платформ, и не представляется возможным модифицировать их все. Также введение дополнительных аттрибутов или конструкций в язык может помешать работать с кодом в IDE.

- Динамическая генерация кода для управляемого устройства, происходящая во время работы приложения, так как это делают Brahma.FSharp или N2O. Для реализации такого подхода достаточно создать библиотеку, которая подключается к приложению, выполняющемуся на управляющем процессоре. При таком подходе процесс сборки проекта усложняется лишь подключением библиотеки — не требуется ни расширений компилятора, ни дополнительных утилит для сборки. Возможность динамической генерации кода также предоставляет потенциальную возможность для некоторых оптимизаций кода. Также упрощается процесс развертывания приложения, поскольку нет необходимости обновлять прошивку вспомогательного микроконтроллера — загрузка кода должна управляться приложением со стороны основного процессора.

С учетом выше обозначенных плюсов и минусов обоих подходов, был выбран второй — на основе динамической генерации исполняемого кода.

Поскольку основная идея работы — упростить для пользователя процесс разработки программного обеспечения, система реализована в виде набора библиотек, подключаемых к пользовательскому приложению, которое запускается на управляющем процессоре. Система позволяет в рамках приложения описать код для вспомогательного микроконтроллера при помощи DSL языка, управлять его загрузкой на вспомогательный микроконтроллер и осуществлять контроль за исполнением сгенерированного кода. Для функционирования системы требуется заранее разработать и загрузить на вспомогательный микроконтроллер базовую прошивку, предоставляющую интерфейс для доступа к памяти микроконтроллера, чтобы загружать туда исполняемый код для выполнения. На рисунке 1 изображена общая схема взаимодействия компонентов системы.

В следующих разделах подробно описаны разработанные программные компоненты и приведены дополнительные комментарии по представленной системе.

3.1. Связующая прошивка

Для каждого конкретного вспомогательного процессора или микроконтроллера требуется разработать базовую прошивку, далее именуемую в тексте loader, отвечающую за поддержку соединения с управляющим процессором, например, по шине I2C или USB. Loader должен предоставлять интерфейс для чтения и записи данных в произвольные области памяти контроллера, а также предоставлять точки расширения своей базовой логики. Одной из таких возможностей является расширение протокола взаимодействия или основной функциональности прошивки динамически генерируемым кодом. Точками расширения могут быть фиксированные адреса функций-обработчиков, исполняемый код которых будет присылаться с управляющего процессора.

3.2. Модуль соединения

Связующий модуль поддерживает соединение основного процессора со вспомогательным микроконтроллером и предоставляет пользовательскому приложению интерфейсы для пересылки кода и данных, а также вызова функций на микроконтроллере. В рамках данной работы связующий модуль реализован для I2C шины, но может быть дополнен и для использования другого типа соединения, например такого как USB.

3.3. Менеджер памяти

Менеджер памяти — компонент, ответственный за размещение кода и данных в памяти вспомогательного микроконтроллера. При классическом способе разработки статической прошивки вопросами размещения кода и данных занимаются компилятор и редактор связей. В нашем же случае появляется необходимость контролировать это снаружи, то есть со стороны управляющего процессора. Для того, чтобы разместить сгенерированный код или статические данные в памяти вспомогательного микроконтроллера, необходимо иметь информацию о том, какие участки памяти свободны для использования, а какие использовать нельзя — например, они могут быть уже заняты кодом и данными базовой прошивки, некоторые участки адресного пространства могут использоваться как регистры устройств, отображенными в адресное пространство, вектора прерываний. Помимо этого, генерируемому коду необходимо обращаться к другим, уже размещенным функциям и данным. Следовательно, информация о размещении в памяти необходима уже на этапе генерации кода.

3.4. Модуль генерации кода

Модуль генерации исполняемого кода в описанной реализации — это библиотека, предоставляющая упрощенный интерфейс к LLVM [5], позволяющая на основе кода промежуточного представления LLVM

(LLVM IR) получить машинный код для отправки на вспомогательный процессор. Генерация машинного кода, а не использование интерпретируемого кода, например, на Python или Lua, была выбрана по той причине, что для многих вспомогательных устройств не представляется возможным использовать какие-либо интерпретаторы из-за слишком высоких требований к ресурсам.

Платформа LLVM была выбрана потому, что может быть использована в качестве библиотеки и поддерживает различные целевые архитектуры. Например, такие часто применяемые во встраиваемых системах, как ARM, Atmel AVR, TI MSP430. Также сама по себе платформа LLVM имеет невысокие требования для запуска, например она может быть запущена на ARM процессоре под управлением ОС Embedded Linux.

3.5. Библиотека поддержки DSL

Чтобы избавить пользователя от необходимости работать непосредственно с LLVM IR в приложении, была разработана библиотека функций и классов на языке C++, реализующая DSL язык, транслируемый в LLVM IR. Библиотека позволяет пользователю описать на языке DSL функциональность вспомогательного процессора или микроконтроллера. DSL основан на возможностях мета-программирования и перегрузки операторов в C++, а также использует препроцессор языка C++. Идея разработки DSL поверх C++ не нова и уже рассмотрена ранее в статьях. Например, хорошим примером является “DSL Implementation in MetaOCaml, Template Haskell, and C++” [4].

Разработанный в ходе данной работы DSL является удобной оберткой над LLVM IR и по своим возможностям соответствует языку C, таким образом являясь универсальным для различных целевых платформ, лишь иногда требуя небольших доработок, связанных с особенностями целевой платформы. На данный момент разработанный язык позволяет описать следующие конструкции.

- Объявление функций.

- Глобальные и локальные переменные.
- Целочисленные типы данных, соответствующие типам данных в языке C.
- Арифметические, логические, битовые операции.
- Указатели и работа с памятью через них.
- Операции приведения типов.
- Массивы фиксированной длины.
- Структуры, вложенные структуры и массивы фиксированной длины.
- Оператор if (при этом являющийся выражением).
- Оператор switch-case (при этом являющийся выражением).
- Циклы while и for.
- inline аттрибуты функций.
- volatile переменные, поля структур.

4. Реализация

В рамках данной работы был реализован прототип спроектированной системы, на текущий момент реализация протестирована на одной платформе TRIK. Полная реализация требует доработки компонентов системы, апробации на других аппаратных конфигурациях, а также разработки полноценной документации.

Как уже было сказано, универсальные компоненты системы реализованы в виде библиотек на языке C++. Для сборки библиотеки используется CMake, а также необходим компилятор поддерживающий C++11. На данный момент собирается статическая библиотека, в будущем планируется добавить конфигурацию для сборки динамической библиотеки, которую достаточно будет лишь раз загрузить на управляющее устройство.

Модуль соединения

Связующий модуль для главного процессора представлен интерфейсом `hetarch::conn::IConnection` для различных типов соединения. На данный момент разработаны 2 реализации:

- `hetarch::conn::I2CConnection` для I2C соединения
- реализация `hetarch::conn::TCPTestConnection`, работающая на основе TCP соединения и предназначенная для локального тестирования системы.

Менеджер памяти

Менеджер памяти в существующей реализации конфигурируется вручную — необходимо задать области памяти, которые могут быть использованы для хранения кода и данных, причем на данный момент выделены два класса памяти:

- `hetarch::mem::MemClass::ReadWrite` — RAM память, перезаписываемая без ограничений

- `hetarch::mem::MemClass::ReadPreferred` — память, которую предпочтительно не перезаписывать, например Flash, часто используемая в микроконтроллерах для хранения кода и константных данных.

Сконфигурированный менеджер памяти жадным алгоритмом выделяет области памяти для загрузки функций, определения адресов глобальных переменных и констант, позволяя явно указать класс памяти. В будущем возможно добавление новых классов/правил работы с памятью, изменения алгоритма выделения памяти. Одна из важных планируемых доработок системы — сохранение в некотором виде текущей информации о загруженном коде в памяти микроконтроллера, поскольку важно минимизировать запись во Flash-память. Например, это может быть некоторая хэш-сумма идентификаторов функций, которую менеджер памяти будет читать при загрузке приложения, таким образом, по возможности, избегая перезаписи.

Модуль генерации кода

Модуль генерации кода является оберткой над LLVM. Он ответственен за следующие задачи:

- инициализирует платформу LLVM, конфигурирует ее опции и целевую архитектуру (`llvm::TargetMachine`);
- передает IR модуль для компиляции объекту класса `llvm::orc::SimpleCompiler`, получая `llvm::object::ObjectFile`;
- извлекает из полученного `llvm::object::ObjectFile` требуемый исполняемый код.

Библиотека поддержки DSL

Библиотека поддержки DSL позволяет описывать алгоритмы для вспомогательного микроконтроллера, используя набор классов, функций и переопределенных операторов. DSL типизирован, операции опре-

делены для целочисленных типов (рекомендуется использовать целочисленные фиксированные типы, определенные в `stdint.h` (`cstdint`)), логического типа, указателей и составных типов (массивов и структур).

Пользователь конструирует код из выражений представленных классами `hetarch::dsl::RValue<T>` и `hetarch::dsl::LValue<T>`, где `T` — тип выражения. Простейшие выражения — это переменные (представленные типами `Local<T>`, `Global<T>`, и `Param<T>`) и константы. Выражения можно комбинировать, используя конструкции:

- перегруженные арифметические и побитовые операторы (такие, как "+", "-", "<", "|", "&");
- логические `&&` и `||`, реализующие нестрогий порядок вычислений;
- `LValue<T>` поддерживает присваивание (метод `LValue<T>::Assign(RValue<T>)`);
- конструкции потока управления, например, условное выражение `hetarch::dsl::If`, цикл `hetarch::dsl::While`, переключатель `hetarch::dsl::Switch`, последовательность `hetarch::dsl::Seq` (аналог оператора "запятая" в C);
- преобразование типов `RValue<T>::Cast<T2>()`, неявное преобразование типов отсутствует;
- взятие адреса (`LValue<T>::GetAddr()`) и разыменование указателя оператором "*";
- обращение к элементу массива через оператор `[]`;
- доступ к полям структур (Функция `Fields(RValue<T>)` , где T наследуется от `hetarch::dsl::Struct`);
- вызов функции `Function<T...>::Call(T...)` .

Для описания и работы со структурами используются специальные типы и операции. Пользовательская структура должна наследоваться

от типа `hetarch::dsl::Struct`, определять поля как `SField<T>` и содержать правильный конструктор по умолчанию, который должен инициализировать поля структуры в нужном порядке. Например, следующее определение:

```
using namespace hetarch::dsl;
struct Point2D: Struct<StructLayout16> {
  SField<uint8_t> x;
  SField<uint8_t> y;
}
```

будет соответствовать обычной C структуре, как если бы она была определена так:

```
struct Point2D {
  uint8_t x;
  uint8_t y;
};
```

причем с выравниванием по двухбайтовой границе. Для доступа к полям структуры используется функция `Fields`:

```
Local<Point2D> pt1;
LValue<uint8_t> fieldAssign = Fields(pt1).x.Assign((uint8_t)10);
```

Приведем пример использования DSL и других компонентов системы. Предположим, необходимо вынести логику остановки робота при столкновении с препятствием на вспомогательный процессор. Пусть к вспомогательному процессору подключен датчик расстояния, значение которого можно получить из регистра P3, отображенного в адресное пространство по адресу 0x00C1, а скорость движения моторов определяется значениями в регистрах A8 и A9, отображенных в адресное пространство по адресам 0x01E8 и 0x01E9 соответственно. Листинг 1 демонстрирует, как при помощи библиотек системы описать функцию, выполняющую данную проверку, загрузить код на вспомогательный микроконтроллер и назначить на регулярное исполнение.

Листинг 1: Пример использования системы

```
using namespace hetarch::dsl;
```

```
// memory mapped registers is just like regular global
// variables with fixed address
Global<uint8_t> P3(0x00C1);
Global<uint8_t> A8(0x01E8);
Global<uint8_t> A9(0x01E9);
...

// assume that we reached the wall on this value of sensor
uint8_t COLLISION_THRESHOLD = 5;

// defining function
auto stopOnCollision = MakeFunction("stopOnCollision", memMgr,
  If(P3 < COLLISION_THRESHOLD, Seq(
    A8.Assign(0),
    A9.Assign(0),
  ))
);

// compile to executable code (codeGen: hetarch::cg::ICodeGen*)
stopOnCollision.Compile(codeGen);

// send executable code to helper device (connection: hetarch::
   conn::IConnection*)
stopOnCollision.Send(connection);

// schedule function to execute on each iteration
// of firmware main loop (connection: hetarch::conn::
   IConnection*)
connection->ScheduleRegular(stopOnCollision);
...
```

Дополнительные примеры, а также базовая документация системы доступны в репозитории проекта по адресу
https://github.com/melentyev/hetarch.

Тестовое приложение

В процессе разработки система тестировалась локально — LLVM использовалась для генерации кода для платформы x86/64, было реали-

зовано тестовое приложение (для локального запуска на x86/64 машине под управлением Linux) для имитации работы базовой прошивки вспомогательного микроконтроллера. Приложение выделяет область памяти из которой можно исполнять код (используя функцию mprotect), предоставляет TCP интерфейс для доступа к этой памяти для отправки исполняемого кода. Класс `hetarch::conn::TCPTestConnection`, обозначенный выше, предназначен для взаимодействия с данным приложением.

5. Апробация и анализ решения

Идея данной работы возникла при работе с контроллером TRIK, для которого и было запланировано первое внедрение системы. Существующая в настоящий момент прошивка для микроконтроллера msp430f5510, входящего в состав TRIK реализует I2C соединение, через которое предоставляет управляющему процессору доступ к управлению силовыми моторами и позволяет чтение значений энкодеров и аналоговых датчиков. В рамках работы в данную прошивку были добавлены обработчики команд I2C, предоставляющие доступ к адресному пространству микроконтроллера, то есть доступ к RAM и Flash памяти, а также регистрам устройств, отображенных в адресное пространство. Также были добавлены точки расширения главного цикла (main loop) прошивки и функций-обработчиков I2C соединения. Точка расширения основного цикла - это переменная типа указатель на функцию, находящаяся по заранее известному адресу, вызываемую на каждой итерации цикла. Таким образом, для расширения логики главного цикла, достаточно загрузить код функции, и установить данный указатель на адрес загруженной функции (данную логику инкапсулирует метод `Connection::ScheduleRegular`). Аналогичным образом работают точки расширения в обработчиках I2C. Данные модификации являются основными для функционирования системы. После этого логика работы с силовыми моторами была изъята из кода прошивки, и перенесена на DSL.

Поскольку на данный момент в платформе LLVM для MSP430 отсутствует поддержка генерации объектного кода, необходимого для создания нашей системы, а поддерживается только генерация текстового кода на языке ассемблера, то в рамках данной работы также была добавлена соответствующая поддержка в код платформы LLVM (в объеме, необходимом для данной работы). Для обеспечения генерации исполняемого кода были реализованы интерфейсы `llvm::MCAsmBackend`, `llvm::MCELFObjectTargetWriter`, `llvm::MCCodeEmitter`, в TableGen файлы (.td) для архитектуры MSP430 добавлены бинарные кодировки ин-

струкций. В работе была использована версия LLVM 3.7, поскольку более новые версии имеют проблемы со сборкой под архитектуру arm (управляющий процессор в нашей конфигурации), при этом не имеют каких-либо возможностей, важных для данной работы.

Анализ разработанного решения

Представленная в данной работе система программирования позволяет описать всю основную логику как единое целостное приложение, при этом не используя специальных средств разработки для программирования вспомогательного микроконтроллера за исключением базовой связующей прошивки. В данной работе рассматривается пример с одним подключенным микроконтроллером, но система может быть также использована и в более сложных аппаратных конфигурациях (достаточно использовать несколько экземпляров соединения и менеджера памяти)

Разработанная система основывается на использовании C++ в качестве основного языка программирования при разработке приложений. Однако потенциально система может быть расширена или модифицирована для разработки на другом языке. Например, на основе языка Python также возможно создать DSL язык, транслируемый в LLVM IR (возможно даже более удобный, благодаря, например, такой возможности Python, как рефлексия), при этом возможно переиспользовать библиотеки (менеджер памяти, модуль генерации кода, соединение), написанные на C++.

Помимо удобства для программиста (что может быть субъективным вопросом), представленная в работе система обеспечивает дополнительные возможности для оптимизации приложений. Одна из таких оптимизаций заключается в возможности организовать оверлейную структуру памяти, загружая только необходимые в работе текущего алгоритма функции и данные в память микроконтроллера. Сама по себе динамическая кодогенерация позволяет проводить такие оптимизации, как вынесение редко изменяющихся переменных в константы и переге-

нерацию кода по необходимости.

В случае, когда в системе на вспомогательном микроконтроллере используется прошивка, не имеющая собственной логики, помимо передачи значений сенсоров на управляющий процессор и получения значений для актуаторов с управляющего процессора, при переходе на вышеописанную систему обеспечивается возможность выделить задачи, с которыми вспомогательный микроконтроллер может справиться самостоятельно. Например, это такие задачи, как балансирование, остановка в случае встречи с препятствием, прямолинейное движение и другие задачи, не требующие сложных расчетов и зависящие только от значений датчиков, подключенных к вспомогательному микроконтроллеру. Исключение ответственности управляющего процессора при выполнения таких задач может значительно ускорить и стабилизировать работу системы благодаря избавлению от лишних операций передачи данных между компонентами системы. Такое разделение на уровни ответственности входит в понятие архитектуры Брукса [8].

Заключение

В рамках данной выпускной квалификационной работы были достугнуты следующие результаты.

1. Проведен обзор решений в области программирования гетерогенных систем.

2. Разработана архитектура системы программирования киберфизических гетерогенных архитектур на основе динамической генерации машинного кода с использованием LLVM.

3. Реализован прототип системы.

4. Проведена апробация разработанного прототипа системы на кибернетической платформе TRIK, также проведен анализ разработанного решения и его потенциального развития.

Результаты данной работы также были представлены на конференции SEIM 2017.

Список литературы

[1] Bjornson Joel, Tayanovskyy Anton, Granicz Adam. Composing reactive GUIs in F# using WebSharper // Symposium on Implementation and Application of Functional Languages / Springer. — 2010. — P. 203–216.

[2] Brahma.FSharp. — 2017. URL: https://sites.google.com/site/semathsrprojects/home/brahma-fsharp/.

[3] Chaganti Prabhakar. Google Web Toolkit: GWT Java AJAX Programming: a Practical Guide to Google Web Toolkit for Creating AJAX Applications with Java. — Packt Publishing Ltd, 2007.

[4] DSL implementation in MetaOCaml, Template Haskell, and C++ / K. Czarnecki, J. O'Donnell, J. Striegnitz, W. Taha // Domain-Specific Program Generation. — Springer, 2004. — P. 51–72. — URL: http://www.springerlink.com/index/NL620EL7N94M161H.pdf.

[5] Lattner Chris, Adve Vikram. LLVM: A compilation framework for lifelong program analysis & transformation // Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization / IEEE Computer Society. — 2004. — P. 75.

[6] N2O. — 2012. — [Online; accessed 19-март-2017]. URL: http://synrc.com/apps/n2o/.

[7] Nvidia CUDA. Compute unified device architecture programming guide. — 2007.

[8] R. Brooks. A robust layered control system for a mobile robot // IEEE J. Robot. Automat. 1986. Т. 2, № 1. — 2004. — P. 14–23.

[9] TRIK: project site. — URL: http://www.trikset.com/index_en.html.

[10] The Top Programming Languages 2016. — [Online; accessed 19-март-2017]. URL: http://spectrum.ieee.org/static/interactive-the-top-programming-languages-2016.