

## **Аннотация**

Разработка модуля автогенерации платформно-зависимых констант  
для целей кросс-компиляции  
*Кофанов Даниил Сергеевич*

В данной работе определена практическая ценность кросс-компиляции в контексте виртуальных машин. На примере работы абстрактной виртуальной машины дано определение платформно-зависимых констант и сформулирована связанная с ними проблема, возникающая при кросс-компиляции. Предложена идея гибкого и масштабируемого решения этой проблемы, основанного на создании некоторого интерфейса и использовании системного (т. е. не относящегося к виртуальным машинам) кросс-компилятора. Произведено сравнение с альтернативным подходом. Дан обзор имплементации найденного решения в некой виртуальной машине, основанного на модуле ExternalProject утилиты cmake. Отмечены и обоснованы преимущества, включающие высокую степень автоматизации, удобства в использовании разработанного модуля. Также выделены недостатки текущей имплементации, предложены пути их устранения.

# Оглавление

<b>1</b>	<b>Введение</b>	<b>3</b>
1.1	Контекст проблемы . . . . .	3
1.2	Описание проблемы . . . . .	5
1.3	Терминология . . . . .	6
<b>2</b>	<b>Постановка задачи</b>	<b>8</b>
2.1	Цель работы . . . . .	8
2.2	Решённые задачи . . . . .	8
<b>3</b>	<b>Сравнение подходов к проблеме</b>	<b>9</b>
3.1	Фиксирование констант . . . . .	9
3.2	Вычисление констант . . . . .	10
3.3	Итог сравнения . . . . .	11
<b>4</b>	<b>Реализация модуля</b>	<b>12</b>
4.1	Описание основной идеи решения . . . . .	12
4.2	Схема вычисления гостевых констант . . . . .	13
4.3	Детали и особенности реализации . . . . .	17
4.4	Валидация . . . . .	18
<b>5</b>	<b>Недостатки реализации</b>	<b>19</b>
5.1	Рассинхронизация флагов конфигурации . . . . .	19
5.2	Издержки по времени сборки . . . . .	19
<b>6</b>	<b>Заключение</b>	<b>21</b>
	<b>Литература</b>	<b>22</b>
	<b>Приложение</b>	<b>23</b>
6.1	Основной скрипт . . . . .	23
6.2	Шаблон генерации численных значений . . . . .	25
6.3	Шаблон генерации геттер-функций . . . . .	26

# Глава 1

## Введение

### 1.1 Контекст проблемы

На момент написания этой работы сложно переоценить влияние мобильных устройств на повседневную жизнь. Смартфоны предоставляют огромный спектр возможностей, начиная от коммуникации и игр, заканчивая рабочими потребностями, навигацией и отслеживанием физического состояния человека. Количество используемых телефонов значительно преобладает над количеством ПК и ноутбуков. Однако такой массовый характер был бы невозможен в отсутствии многообразия устройств на рынке. Действительно, развитие технологий безусловно исходит из потребности в новых возможностях, однако финансирование этого развития поступает из уже существующих и отлаженных технологий. Кроме того, только что изобретённой технологии требуется некоторое время, чтобы спрос на неё появился и вырос. Исходя из этого, а также готовности потребителя пойти на компромисс между ценой за устройство и количеством и качеством предоставляемых им функций, представленные на рынке устройства коренным образом отличаются между собой.

С другой стороны, имея такое разнообразие, разрабатывать приложения и отлаживать под каждое устройство было бы слишком дорогой задачей. Чтобы решить эту проблему, вводится понятие виртуальной машины, то есть некоторого абстрактного вычислителя, способного исполнять инструкции из определённого набора, тем самым обеспечивая разработчиков приложений некоторым количеством гарантий. Кажется очевидным, что этот подход аналогичен договорённости между программистами и разработчиками микропроцессоров в виде архитектуры системы команд и почти в той же степени необходим. Главное отличие заключается в более высоком уровне абстракции, что позволяет переложить ответственность за управление ресурсами приложения, например выделение и освобождение памяти, на саму виртуальную машину, тем самым упрощая код и снижая требовательность к опыту разработчика. В случае мобильных устройств, виртуальная машина рассматривается практически как неотъемлемая часть операционной системы, которая, помимо обеспечения гарантий разработчикам приложений, служит барьером безопасности для пользователей, устанавливающих и использующих самые разные приложения.

Высокоуровневость виртуальных машин позволяет выражать программы более компактно в сравнении с представлением в нативном коде. В то же время, нативный код, полученный в результате анализа программы целиком, несравнимо быстрее поочерёдного исполнения инструкций виртуальной машины (интерпретации байткода). Выходом из этой ситуации стало использование Just-In-Time компиляции, то есть генерации оптимизированного нативного кода для часто вызываемых функций, происходящей параллельно выполнению основного алгоритма. Использование как

классических оптимизаций (межпроцедурной оптимизации, раскрутки циклов) так и специфичных JIT-оптимизаций (профилирование типов в случае динамических языков) позволяет достичь производительности, нивелирующей накладные расходы вносимые этим уровнем абстракции. В виртуальных машинах так же используется Ahead-Of-Time компиляция, производящаяся до запуска приложения. Это позволяет избежать излишних затрат энергии при повторных запусках приложений связанных с JIT-компиляцией, скомпилировать большее количество методов, и, возможно, применить большее количество оптимизаций, тем самым экономя заряд батареи мобильных устройств, ускоряя время запуска и улучшая производительность приложения ценой хранения результатов компиляции (АОТ-файлов) на устройстве.

Очевиден тот факт, что множество приложений в своих подзадачах часто используют идентичные алгоритмы, такие как сортировка, работа со строками и т. д., и что множественная имплементация того или иного алгоритма ведёт к практической сложности поддержки такого кода. Ввиду этого, многие алгоритмы, оптимизированные с целью быть приемлемо эффективными для самых различных нагрузок, образуют стандартную библиотеку языка и поставляются вместе с виртуальной машиной. Будучи часто переиспользуемой, стандартная библиотека АОТ-компилируется и хранится на устройстве, что позволяет любому приложению гарантировано использовать оптимизированный нативный код, не совершая излишнюю работу по JIT-компиляции.

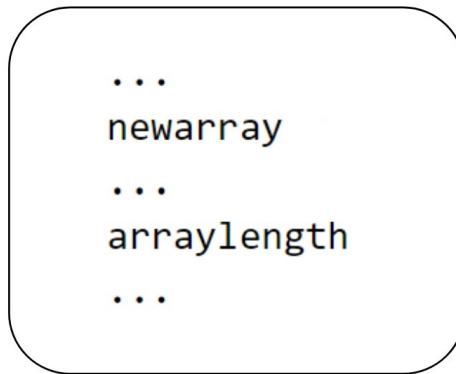
Выпуская огромное количество устройств, запуск АОТ-компиляции стандартной библиотеки на каждом из них был бы нерационален и сопровождался бы ощутимыми издержками производства. Их можно избежать путём единовременной компиляции и последующей загрузки полученного образа на устройства, например при прошивке. Для подготовки образа виртуальной машины и операционной системы используется специальный набор утилит, включающий в себя кросс-компилятор и опции компиляции, который работает на отличной от целевого устройства платформе и генерирующий исполняемые файлы в виде нативного кода целевого устройства. Для такого набора утилит принят термин таргет-тулчейн, в то время как для аналогичного набора утилит, предназначенных для получения исполняемых файлов для той же платформы на которой они и были сгенерированы, используется термин хост-тулчейн.

Однако, таргет-тулчейн не подходит для непосредственной подготовки АОТ-файла стандартной библиотеки. Это связано с тем, что стандартная библиотека рассчитана на исполнение в среде виртуальной машины, отличающейся от окружения создаваемого операционной системой и может проявляться в том, что тулчейн-компилятор и АОТ-компилятор виртуальной машины могут использовать разное соглашение о вызовах. Поэтому приобретает смысл вопрос о кросс-компиляторе в терминах виртуальной машины, что на самом деле означает возможность генерации АОТ-файлов виртуальной машины для определённого устройства на другой платформе, например на сервере. Эта задача имеет множество тонкостей, одной из которых посвящена данная работа - проблема зависимости некоторых численных значений от платформы (платформено-зависимых констант). Наиболее ярким примером является размер указателя. Однако в этом случае зависимость от платформы достаточно легко «предсказать»: на AMD64-совместимой платформе это скорее всего 8 байт, а на ARM32 - 4. Другим, более трудноразрешимым и содержательным примером является смещение до полей в структурах данных. Даже в процессе компиляции на одной и той же платформе одним и тем же компилятором, в зависимости от опций компиляции результат может быть различным. В последующих главах излагается подробное описание этой проблемы, на примере абстрактной виртуальной

машины демонстрируется один из источников таких значений, а также предлагается решение, основанное на использовании таргет-тулчейна, нашедшее применение в некоторой виртуальной машине. Как будет показано в дальнейшем, эта имплементация отличается высоким уровнем автоматизации и масштабируемости, делающей её малозаметной и лёгкой в использовании. Кроме того, помимо основной функции, она вносит дополнительную степень верификации АОТ-файлов.

## 1.2 Описание проблемы на практическом примере

Рассмотрим более подробно язык программирования, который поддерживает какая-либо представляющая интерес виртуальная машина, реализованная, для определённости, на языке C++. Положим также, что в нём существуют некоторые встроенные типы, это могут быть числа, строки массивы, словари и т.д., а операции с этими сущностями, например их создание, модификация, получение внутреннего состояния, выражаются в специальных инструкциях виртуальной машины. Тогда при исполнении программы созданию массива будет соответствовать выполнение некой инструкции *newarray*, а получению длины этого массива – выполнение *arraylength* (рис. 2.1).



```
...  
newarray  
...  
arraylength  
...
```

Рис. 1.1: Пример инструкций в программе

С другой стороны, каждый встроенный тип находит отражение в некотором C++-классе. При этом, при исполнении инструкции *newarray* будет происходить аллокация экземпляра этого класса, а при исполнении *arraylength* будет произведено чтение поля *size\_* одного из экземпляров (рис. 2.2).

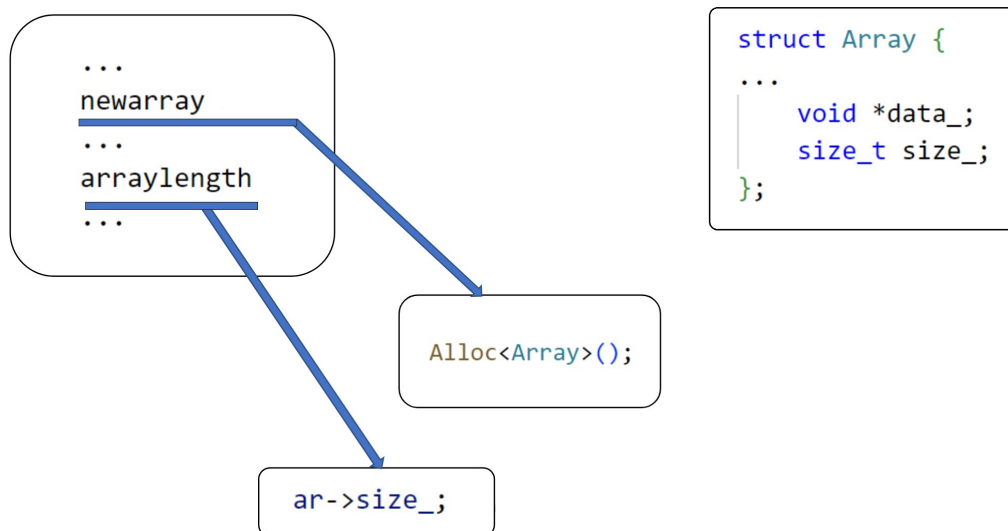


Рис. 1.2: Реализация семантики языка

В нативном формате обращение к этому полю будет выражено в виде машинной инструкции загрузки по указателю с константным смещением. Возвращаясь к кросс-компилятору виртуальной машины, эта инструкция будет создана на финальном этапе, кодогенерации, из соответствующей инструкции промежуточного представления. Фактически, эта инструкция промежуточного представления абстрагирует бинарный формат машинной инструкции загрузки по смещению конкретной архитектуры, отделяя зависимость от платформы. Однако, такой абстракции для отделения оказывается недостаточно.

Дело в том, что смещение до поля класса (эквивалентное в рассматриваемой ситуации смещению в инструкции загрузки), в общем случае, зависит от платформы. Это может происходить по разным причинам, например – из-за зависимости размера какого-либо элемента структуры данных от платформы, выравнивания по разным адресам. Так, рассматриваемое ранее смещение до поля `size_` структуры `Array` на 64-битной архитектуре AMD64 будет равно 8 байтам, хотя на архитектуре ARM32 – 4. Если упустить из рассмотрения этот факт, то кросс-компилятор виртуальной машины, запущенный на AMD64-сервере, при генерации ARM32-инструкций будет использовать соответствующее текущей платформе значение смещения. Тогда, при запуске такого кода на ARM32-телефоне, исполнение такой инструкции приведёт к некорректному поведению и будет являться программной уязвимостью. Вопрос корректности и безопасности генерируемого кода является первостепенным в процессе компиляции, что подчёркивает значимость этой проблемы.

Можно выделить два подхода, позволяющие разрешить этот конфликт: первый – сделать смещения для всех платформ одинаковыми, второй – каким-либо образом сообщать компилятору значение константы на платформе, для которой происходит компиляция. Данная работа посвящена главным образом реализации второго из них, однако в следующих главах также производится сравнение этих подходов, обосновывающее этот выбор.

### 1.3 Терминология

Для дальнейшего повествования удобно ввести следующие определения.

- *Тулчейн* – набор утилит, позволяющий транслировать C++-программы в некоторый бинарный код.
- *Платформа* – предполагаемое тулчейном окружение, в котором будет исполняться сгенерированный им код. Подразумевается, что каждая платформа в достаточной степени однозначно определяется некоторым тулчейном. В качестве синонима, в данной работе может также использоваться термин *архитектура*.
- *Хост-тулчейн* – тулчейн предназначенный для генерации кода, совместимого с той же платформой, на которой и генерируется.
- *Таргет-тулчейн* – тулчейн предназначенный для генерации кода, совместимого с платформой отличной от той, на которой был сгенерирован.
- *Платформозависимая константа* – именованное, определённое в терминах C++ константное выражение, численное значение которой каким-либо образом зависит от платформы. Под этим термином может пониматься как множество численных значений какой-либо величины на всех платформах, так и само значение на какой-то конкретной платформе.
- *Гостевая константа* – значение платформозависимой константы, соответствующее некоему таргет-тулчейну.

# Глава 2

## Постановка задачи

С учётом принятой терминологии цель работы и решённые задачи выражаются следующим образом.

### 2.1 Цель работы

Целью работы является разработка модуля, способного каким-либо образом вычислять заранее обговорённый набор платформо-зависимых констант для интересующих разработчика платформ, а так же имеющего интерфейс, предоставляющий доступ к значению любой из констант на каждой из архитектур.

### 2.2 Решённые задачи

Для достижения этой цели были решены следующие задачи.

1. Найден способ вычисления гостевых констант на хост-платформе (с одновременной поддержкой нескольких платформ).
2. Разработан интерфейс, предоставляющий компилятору виртуальной машины доступ к множеству вычисленных гостевых констант.
3. Обеспечена синхронизация между процессом вычисления констант и интерфейсом, использующимся компилятором.

При решении этих задач особенно важным являлся вопрос масштабируемости (касательно как поддержки новых платформ, так и добавления новых/удаления неиспользуемых платформо-зависимых констант) и автоматизации, то есть минимизации ручной работы, требующейся для корректного функционирования этого модуля при модификации виртуальной машины. Также произведена оптимизация задержек, вносимых процессом обеспечения вышеупомянутой синхронизации. Подробное описание этих аспектов можно найти в главах 4 и 5, посвящённых деталям и недостаткам реализации.



## Глава 3

# Сравнение подходов к проблеме платформо-зависимых констант

В этой главе описываются два подхода к решению проблемы платформо-зависимых величин – фиксирование величин между всеми платформами и, контрастное ему, вычисление величин для каждой платформы, а также производится сравнение этих подходов. Хочется отметить, что данная работа посвящена последнему из них, а так же что её возникновение обязано успешной попытке устранить недостатки, связанные с первым подходом.

### 3.1 Фиксирование констант

Этот подход заключается в поддержании кода в таком состоянии, при котором величины будут иметь одно и то же значение на каждой из платформ. Из его достоинств можно отметить простоту идеи, однако на практике, следование такому подходу сопровождается нетривиальными сложностями.

Например, к этому может привести использование атрибутов, не относящихся к стандарту языка. При использовании разных тулчейнов, хотя и относящихся к одной и той же платформе (например GCC[1] и LLVM [2]), может получаться отличающийся результат.

Далее, остаётся открытым вопрос самого фиксирования этих величин: ранее был рассмотрен лишь один из примеров – смещения полей в структуре данных, однако фиксирование величин другой природы может быть связано со значительными сложностями, если будет возможно в принципе.

Кроме этого, единственным способом достоверной верификации совпадения значений между платформами является сравнение реального значения с числом, то есть каждой константе должна соответствовать статическая проверка на то, что текущее значение величины совпадает с некоторым жёстко закодированным числом. Из-за этого исходный код проекта приобретает избыточные и малоинформативные конструкции, необходимые лишь для проверки корректности реализации подхода, причём на фоне большого числа таких констант сложно гарантировать действительное присутствие такой проверки для каждой из них. Более того, при активной разработке и модификации виртуальной машины фактические значения констант могут меняться (что можно легко проследить на примере изменения смещений при добавлении нового поля в структуру), инициируя необходимость модификации уже существующих проверок. Разработчик неизбежно будет сталкиваться с последствиями такого подхода, поддержка проекта будет становиться всё сложнее.

Обращаясь к известным проектам, виртуальная машина *ART* [3], использующаяся в ОС *Android* [4], на момент написания работы использует данный подход. Изучая имплементацию встроенных типов поддерживаемого ею языка (*mirror*-классов), можно увидеть, что все они имеют выравнивание полей на 4 байта, а размер каждого поля строго составляет 4 байта. Такой подход призван обеспечить требуемую идентичность платформу-зависимых констант, благоприятствуя процессу кросс-компиляции, однако документация утилиты *dex2oat* [5], производящей АОТ-файлы для виртуальной машины *ART*, не указывает на такую возможность, явно позиционируя этот инструмент как предназначенный для локальных запусков на устройствах. Несмотря на это и перечисленные выше недостатки, популярность и успешность устройств на базе *Android* доказывает пригодность такого подхода.

## 3.2 Вычисление констант

Альтернативой является более консервативный подход, описываемый как вычисление величин для каждой платформы и имплементированный в ходе данной работы. Под консервативностью следует понимать отсутствие каких-либо требований и ограничений, описанных ранее, на значения констант и отсутствие вмешательств в исходный код, связанных с этими ограничениями. Хотя на первый взгляд способ вычисления гостевых значений кажется неочевидным, предположив что он существует, множество описанных ранее проблем устраняются сами собой или вовсе лишены смысла.

Так, не появляются затруднения при использовании платформу-зависимого функционала какого-либо тулчейна. Также, не приходится изобретать способ фиксирования для констант разного происхождения, что делает такой подход более однородным и гибким. Не возникает и вопроса сверки констант на разных платформах, так как разрешается их различие. Это в свою очередь допускает решения, являющимися локально оптимальными для каждой из платформ. Например, рассмотрим структуру данных состоящую из двух 32-битных полей. Выравнивание на 8 байт на некой 32-битной архитектуре будет связано с накладными расходами на количество используемой памяти, в то время как выравнивание на 4 байта может повлечь снижение производительности на 64-битной платформе. Данный подход позволяет использовать 4-байтное выравнивание на 32-битной платформе и 8-байтное выравнивание на 64-битной платформе для одной и той же структуры данных.

Таким образом, этот подход, а именно отказ от «модификации» констант, устраняет большинство, если не все, недостатки, описанные в предыдущей секции. С другой стороны, формально он допускает ранее описанный подход, то есть в определённом смысле не противоречит ему: значения каждой из величин на всех платформах могут (искусственным или естественным образом) совпасть. С практической точки зрения это означает, что миграция с использования первого подхода на использование второго может происходить поэтапно, то есть сначала можно наладить инфраструктурную часть, а затем, по мере необходимости, ослаблять введённые искусственно ограничения.

Важно сделать следующее замечание. Необходимо, чтобы полученный кросс-компилятором, рассчитывающим на какую-то конкретную платформу, код действительно запускался в контексте виртуальной машины, собранной для той же платформы. В описываемой ниже имплементации эта проблема устраняется путём вычисления контрольной суммы CRC32 из значений констант на таргет-платформе и последующей её записи как в каждый АОТ-файл, генерируемый кросс-компилятором виртуальной машины, так и в сам таргет-образ виртуальной машины. Во время инстал-

ляции АОТ-файла происходит сверка этих значений, что позволяет обнаружить потенциальные ошибки, связанные с несовместимостью между кросс-скомпилированным кодом и самой виртуальной машиной. Однако в таком виде, эта особенность не является недостатком. Действительно, значение какой-либо константы может изменяться в зависимости от версии виртуальной машины, а совпадение версии кросс-компилятора, находящегося на сервере, и самой виртуальной машины, находящейся на устройстве, при ошибках инфраструктуры может нарушиться. Такая контрольная сумма вносит дополнительную степень верификации, косвенно проверяя совместимость версий и несёт дополнительную полезную нагрузку, никак не связанную с выбранным подходом.

### 3.3 Итог сравнения

Исходя из вышесказанного, второй подход, связанный с вычислением констант для каждой платформы, является более предпочтительным ввиду своей универсальности. Характерная ему однородность, определённая выше, позволяет в высокой степени автоматизировать весь процесс, связанный с вычислением значений и поддержанием их в консистентном состоянии (как в случае чистой, так и инкрементальной сборки), фактически отделяя его реализацию в независимый, инкапсулирующий модуль. Разработчик сталкивается с этим модулем лишь через интерфейс и у него нет необходимости вручную вносить изменения в этот модуль: достаточно лишь определения платформи-зависимой константы по аналогии с другими, после чего её можно использовать в основной части проекта.

Дальнейшее повествование раскрывает детали реализации выбранного подхода.

## Глава 4

# Реализация модуля автогенерации платформо-зависимых констант

### 4.1 Описание основной идеи решения

В дальнейшем для определённости предполагается, что исходный код виртуальной машины собирается с помощью утилиты CMake [6]. Данная утилита поддерживает кросс-сборки, позволяя выбрать не только архитектуру микропроцессора, но также уточнить операционную систему целевого устройства (или её отсутствие). Это позволяет генерировать образы программ под большое множество устройств. В случае виртуальной машины, подготовив на сервере соответствующий образ и загрузив его на телефон, получается корректно работающая виртуальная машина, способная исполнять байткод, каким-либо образом попадающий на устройство в дальнейшем, а также имеющая JIT- и АОТ-компилятор (рис. 4.1).

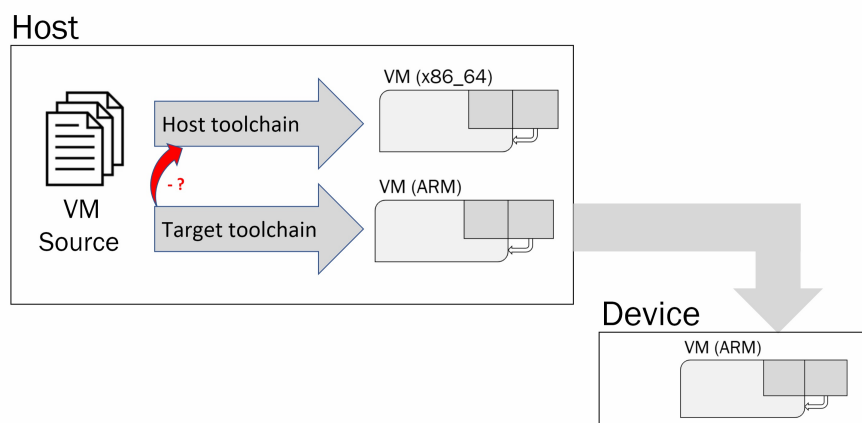


Рис. 4.1: Процесс появления виртуальной машины на таргет-устройстве. Под красной стрелкой можно понимать проблему вычисления гостевых констант.

Чтобы иметь возможность подготавливать АОТ-файлы для телефона на сервере (мотивация чего описана в Главе 1), можно каким-либо образом извлечь значения гостевых констант из процесса сборки образа для гостевого устройства и добавить их в некотором виде в исходный код хост-сборки АОТ-компилятора. Идея компактной «псевдо-сборки», которая не полностью подготавливает образ виртуальной машины, а служит лишь для извлечения необходимой информации, и является основной идеей описываемого решения.

## 4.2 Схема вычисления гостевых констант

Для начала, все платформо-зависимые константы объединяются в специальную единицу трансляции, формат которой приведён на рис. 4.2.

```
#include "array.h"
#define DEFINE_VALUE(name, value)
void AsmDefinition_##name()
{
    asm volatile("\n.ascii \"^^\" #name \" %0^^\" ::\"i\"(static_cast<int64_t>(value))); \n");
}
DEFINE_VALUE(ARRAY_DATA_OFFSET, offsetof(Array, data_))
DEFINE_VALUE(ARRAY_SIZE_OFFSET, offsetof(Array, size_))
```

Рис. 4.2: Формат описываемой единицы трансляции с примером платформо-зависимых констант (смещения до полей структуры *Array*).

Помимо своего основного предназначения, речь о котором пойдёт далее, она обладает важным документирующим свойством: все платформо-зависимые константы оказываются определены в общем, предназначенном для них месте, а для добавления новой константы требуется лишь добавить строчку формата **DEFINE\_VALUE**(/\*id\*/, /\*constant expression \*/).

Далее, рассмотрим подробнее процесс сборки виртуальной машины из исходного кода для хост-устройства. На стадии CMake-конфигурации помимо основной, хост-сборки, также конфигурируется с использованием соответствующего таргет-тулчейна один или несколько вспомогательных сборок, каждая из которых находится в отдельной директории внутри корневой директории сборки (рис. 4.3).

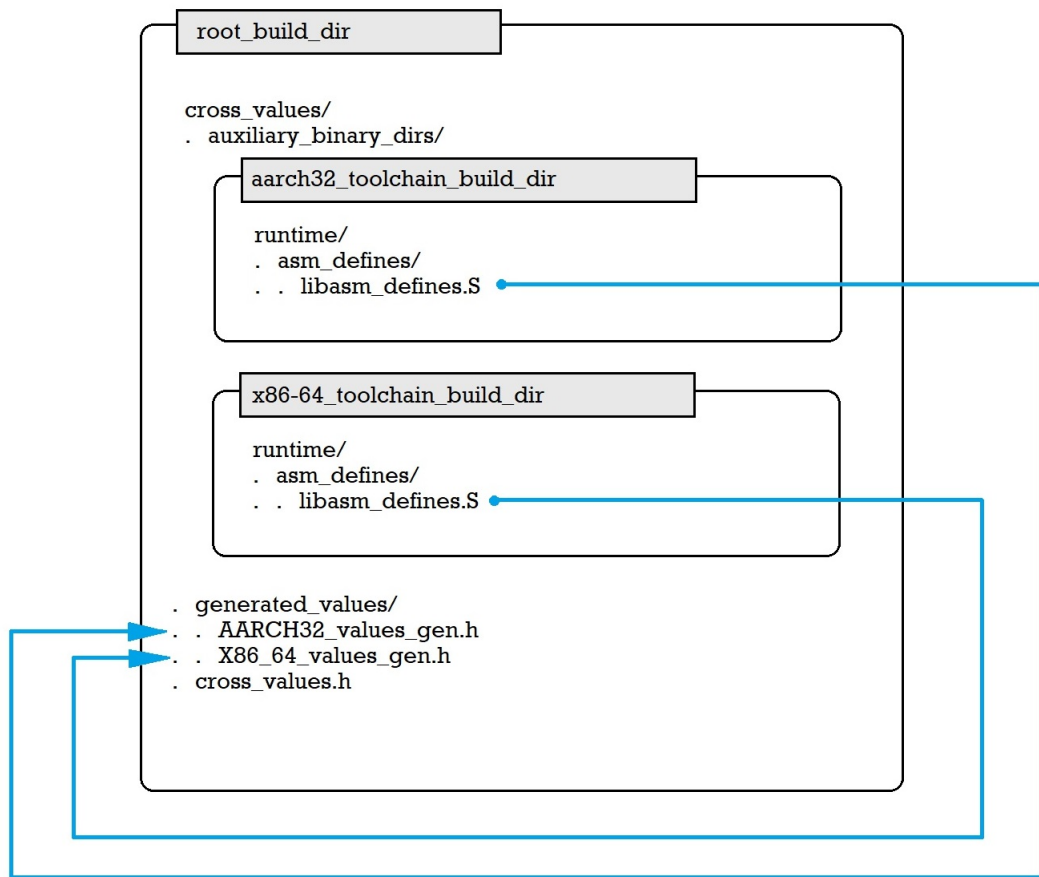


Рис. 4.3: Схема директории сборки виртуальной машины. Голубым цветом обозначены зависимости сборки.

Стоит отметить, что хост-платформа также может рассматриваться как одна из таргет-платформ, что показано на схеме выше. Это позволяет унифицировать подход к выбору целевой платформы в случае поддержки АОТ-компилятором виртуальной машины нескольких целевых платформ. В каждой вспомогательной сборке, после того как она была сконфигурирована, происходит кросс-компиляция описанной выше единицы трансляции, однако не в бинарный, а ассемблерный вид. Часть такого файла изображена на рисунке 4.4.

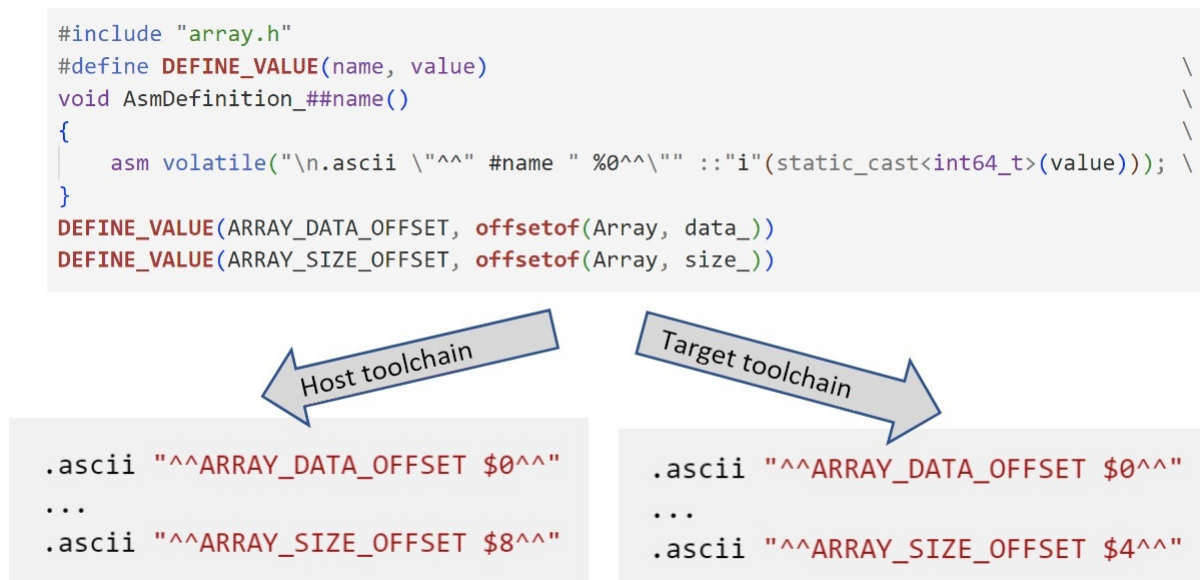


Рис. 4.4: Процесс генерации ассемблерного файла с помощью таргет-тулчейнов.

Особый формат ассемблерных вставок, в которые преобразуются макросы, соответствующие платформо-зависимым константам, позволяют достаточно просто проанализировать результат компиляции и извлечь гостевые константы в численном виде. Полученные таким образом данные используются для определения C++-констант, имеющих одинаковые имена и разделённые пространством имён, соответствующие обозначению целевой платформы. Они оформляются в виде заголовочных файлов, которые уже могут встраиваться в платформо-специфичный код виртуальной машины, предоставляя тем самым доступ к платформо-зависимым константам (рис. 4.5).

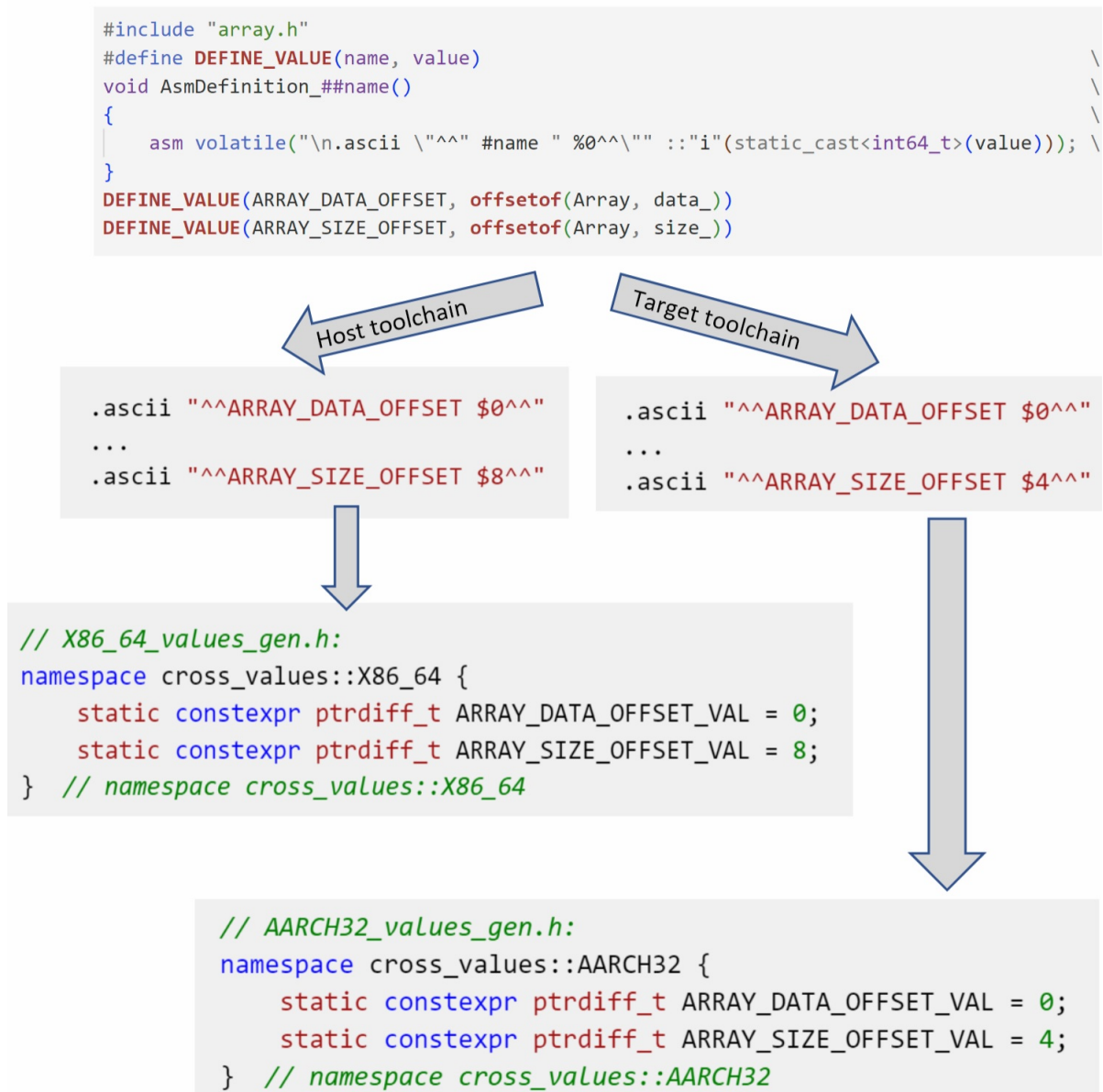


Рис. 4.5: Схема генерации файлов, содержащих определения констант.

После того как последний заголовочный файл сгенерирован, необходимо объединить их в один для удобства, а также сгенерировать для каждой платформозависимой константы геттер-функцию, позволяющую различать значения каждой из констант, принимаемые ими на целевых платформах, по некоторой переменной перечисляемого типа (рис. 4.6).



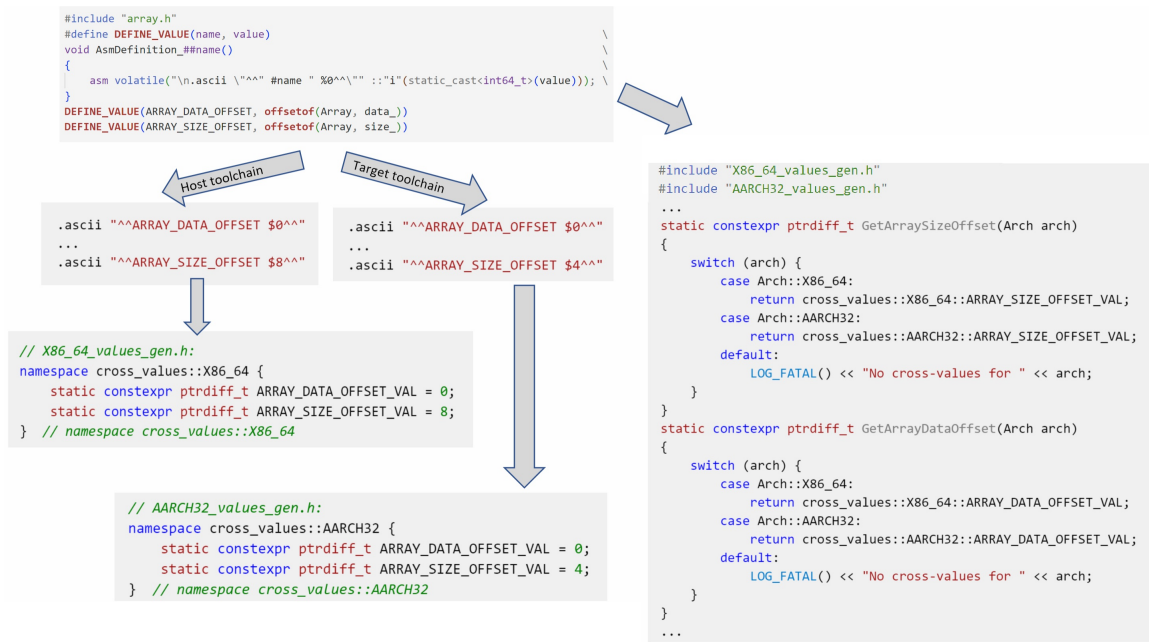


Рис. 4.6: Схема генерации геттер-функций.

Затем, конфигурация хост-сборки завершается и можно приступать к непосредственной сборке АОТ-компилятора с помощью вызова утилиты Make [7] или Ninja [8]. Собранный таким образом компилятор располагает необходимыми значениями платформу-зависимых констант и способен генерировать корректный код для запуска в среде виртуальной машины, собранной тем же таргет-тулчейном, что и соответствующая вспомогательная директория.

### 4.3 Детали и особенности реализации

Основным принципом, соблюдаемым при разработке этого модуля являлось стремление к минимизации работы, связанной с его обслуживанием в будущем. В первую очередь это выражается в высоком уровне автоматизации процесса, описанного в предыдущей секции.

Конфигурация вспомогательных директорий сборки происходит с использованием встроенного в *СMake* модуля *ExternalProject* [9]. Данный модуль позволяет встроить процессы скачивания, конфигурации, сборки и т.д. какого-то стороннего проекта в процесс сборки основного проекта, и предоставляя контроль за ними в терминах зависимостей и целей сборки. Используя в качестве директории исходного кода стороннего проекта директорию, являющуюся деревом исходного кода для основной сборки виртуальной машины, а также указав соответствующий таргет-тулчейн образуются необходимые зависимости сборки для *libasm\_defines.S* файлов (см. рис. 4.3), автоматически поддерживающие их консистентное состояние при модификациях в исходном коде. В свою очередь, эти файлы являются входными для скриптов (см. ниже), осуществляющих генерацию файлов с определением констант в процессе сборки, а создание такой зависимости также влечёт консистентное состояние для них, в том числе. Тот факт, что они являются файлами-заголовками, позволяет построить необходимые зависимости к конечным целям сборки, включая АОТ-компилятор. За счёт этого достигается корректность не только чистой сборки, но и инкрементальной, делая разработку виртуальной машины более удобной.

Автоматизация обеспечивается не только с помощью *CMake*, но и с помощью шаблонной генерации *ERB* (сокращённо от *Embedded Ruby*) [10]. Этот инструмент позволяет генерировать текстовые файлы любого формата, включая C++-код. Такой подход находит эффективное применение в обобщении однообразного кода, путём вынесения нетривиальной информации в один документ (например, *JSON*- или *YAML*-формата), а затем генерируя исходный код на основе его содержания по шаблону специального формата (*.erb*-шаблон). К наиболее ярким его достоинствам относятся явное выделение однообразия в коде (что упрощает его понимание и позволяет избежать многих ошибок, например из-за копирования кода), дешёвая с точки зрения разработки масштабируемость, возможность быстрого изменения функционала. В качестве основного документа, на основе которого производится весь описанный ранее процесс генерации, используется сама единица трансляции, изображённая на рисунке 4.2, а также список *cmake-toolchain* файлов, в том числе определяющий количество поддерживаемых АОТ-компилятором платформ и сами платформы.

Созданный C++-интерфейс представляет собой обычные C++-функции, имена которых основаны на именах соответствующих платформо-зависимых констант, принимающие в качестве аргумента переменную перечисляемого типа, отражающую выбранную платформу, и возвращающие значение константы на платформе согласно аргументу. Кроме того, сами константы доступны для использования непосредственно, так как находятся в отдельных пространствах имён. Это позволяет использовать данный модуль в как в платформо-определённых участках кода, так и обобщённых от платформы участках кода.

## 4.4 Валидация

На основе сгенерированных файлов с определениями констант вычисляется контрольная сумма. В случае поддержки АОТ-компилятором нескольких целевых платформ, контрольная сумма вычисляется для каждой из них, и соответствующее результату компиляции значение записывается в каждый выходной файл. Аналогично, контрольная сумма записывается в образ самой виртуальной машины. При инициализации АОТ-файла происходит сравнение контрольной суммы, записанной в виртуальную машину и в АОТ-файл. Это позволяет отследить ситуации расхождения констант и предотвратить ряд трудно обнаруживаемых ошибок.

## Глава 5

# Недостатки реализации

После имплементации и интеграции описанного решения в основную ветвь проекта, проявились некоторые его тонкости, которые вполне можно отнести к недостаткам. В этой главе дан обзор этим моментам и предложены пути их сглаживания.

### 5.1 Рассинхронизация флагов конфигурации

Одна из существенных проблем, потенциально влекущая неконсистентность платформозависимых констант между кросс-компилятором и самой виртуальной машиной исходит из рассинхронизации флагов *CMake*-конфигурации. Для корректной конфигурации вспомогательных сборок важно передать все флаги, переопределённые во время иницилирующего вызова *stake* из командной строки. Если упустить какой-нибудь из них, то при конфигурации вспомогательной сборки он примет значение по-умолчанию, при этом такое различие в конфигурациях может повлиять, например, на наличие поля в какой-то структуре, смещения в которой отслеживаются рассматриваемым модулем, что повлечёт расхождение констант. На момент написания работы, такая ошибка выявляется хотя и гарантировано, но достаточно поздно - в момент запуска приложения на устройстве. Желательно более раннее детектирование, а в лучшем случае - предотвращение такой ошибки. Поскольку *stake* не предоставляет интерфейса по определению флагов, указанных при вызове команды, решение может быть основанным на анализе файла *CMakeCache.txt*, предназначение которого описывается в документации *CMake*, содержащего действительные значения флагов.

### 5.2 Издержки по времени сборки

При недостаточно аккуратно указанных зависимостях сборки проекта, возможна ситуация, при которой генерируемые файлы пересоздавались, но их содержание было бы тем же самым. Так как утилиты осуществляющие сборку (*Make*, *Ninja*) определяют модификацию файлов в терминах временных меток, а не их содержимого, это существенно замедляет инкрементальную сборку из-за лишней работы связанной с перекомпиляций, производящей в точности тот же самый результат. Грубым решением этой проблемы будет использование утилиты на подобии *ccache*. Более правильным будет анализ зависимостей сборки и устранение ложных зависимостей. Отсылаясь к опыту реализации и поддержки этого модуля, оказалось так, что эта проблема исходила из другой части проекта и проявлялась ранее, хотя и реже. Таким образом, анализ и исправление зависимостей привела к более предсказуемой

инкрементальной сборке, устранив «случайные» перекомпиляции, занимающие время сравнимое с чистой сборкой проекта (т.е. около 10-20 минут).

На данном этапе разработки, на конфигурацию одной вспомогательной сборки приходится около 10 секунд. При сборке проекта с поддержкой нескольких платформ последовательная настройка может вносить некоторый дискомфорт. Эту проблему можно решить с помощью параллелизации ценой потери информативности логов процесса конфигурации вспомогательныхборок, возможно даже совместно с их переносом из стадии конфигурации в стадию сборки основного проекта. Это приводит к снижению наглядности процесса автогенерации и несколько усложняет трассировку проблем при ошибках конфигурации (например, связанных с инфраструктурой) для разработчика, не знакомого с этим модулем. В этих условиях, предпочтение было сделано в пользу наглядности процесса.

## Глава 6

### Заключение

В процессе работы был разработан и интегрирован модуль автогенерации платформозависимых констант, отличающегося высокой степенью автоматизированности и масштабируемости, а также обладающего легковесным и интуитивно понятным C++-интерфейсом. Его применение позволяет эффективно разрешить одну из проблем кросс-компиляции, то есть генерации нативного кода для гостевой платформы, в контексте АОТ-компилятора виртуальной машины. Также произведено сравнение этого решения с альтернативным, использующимся для кросс-компиляции ранее. Выделенные достоинства и недостатки обоих подходов являются существенными доводами в пользу предлагаемого подхода. Результатом является ожидаемое устранение критических недостатков предшествующего подхода, связанных как с вопросом корректности генерируемого кросс-компилятором кода, так и долгосрочной поддержкой проекта и внесением в него изменений. Важно отметить, что разрешение этих проблем в первую очередь достигается именно благодаря автоматизации всего процесса. Также выделены выявленные недостатки созданного модуля, предложены пути их устранения.

Данная работа также была представлена в рамках 65-ой Всероссийской научной конференции МФТИ.

# Литература

- [1] *Free Software Foundation, Inc.* GCC, the GNU Compiler Collection - GNU Project. — 2021. <https://gcc.gnu.org/>.
- [2] *LLVM Foundation.* The LLVM Compiler Infrastructure Project. — 2021. <https://llvm.org/>.
- [3] *Google LLC.* Android Runtime (ART) and Dalvik. — 2023. <https://source.android.com/docs/core/runtime>.
- [4] *Google LLC.* Android Open Source Project. — 2023. <https://source.android.com/>.
- [5] *Google LLC.* Configuring ART. — 2023. <https://source.android.com/docs/core/runtime/configure>.
- [6] *Kitware, Inc.* CMake Reference Documentation. — 2021. <https://cmake.org/cmake/help/latest/index.html>.
- [7] *Free Software Foundation, Inc.* GNU make. — 2023. <https://www.gnu.org/software/make/manual/make.html>.
- [8] *Martin, Evan.* Ninja. — 2023. <https://ninja-build.org/>.
- [9] *Kitware, Inc.* ExternalProject. — 2021. <https://cmake.org/cmake/help/latest/module/ExternalProject.html>.
- [10] *Britt, James.* Ruby-Doc.org: Documenting the Ruby Language. ERB – Ruby Templating. — 2021. <https://ruby-doc.org/stdlib-2.7.1/libdoc/erb/rdoc/ERB.html>.

## Приложение

В приложении приводится основной Ruby-скрипт и использующиеся в нём ERB-шаблоны, на основе которых производится генерация C++-сущностей, описанных ранее в работе.

### 6.1 Основной скрипт

```

1  #!/usr/bin/env ruby
2
3  require 'erb'
4
5  class CrossValues
6    def initialize
7      @target_gen_dir = ARGV[0]
8      @constants_gen_dir = @target_gen_dir + "/generated_values"
9      @asm_defines = File.read(ARGV[1]).scan(/DEFINE_VALUE\((\w+)\),/).sort_by
10     (&:first)
11     @consts_tmpl = ERB.new(File.read(ARGV[2]), trim_mode: "_")
12     @getters_tmpl = ERB.new(File.read(ARGV[3]), trim_mode: "_")
13
14     @archs = []
15     (4..ARGV.size - 1).step(2).each do |i|
16       pair = []
17       pair.append(ARGV[i])
18       pair.append(File.read(ARGV[i + 1]).scan(/"^\^(\\w+) [#\$]?([+]?\\d+)
19       \^\^"/).sort_by(&:first))
20       @archs.append(pair)
21     end
22     @arch_cursor = 0
23     Dir.mkdir(@target_gen_dir) unless File.exists?(@target_gen_dir)
24     Dir.mkdir(@constants_gen_dir) unless File.exists?(@constants_gen_dir)
25
26     def update_arch_cursor
27       @arch_cursor = (@arch_cursor + 1) % @archs.size
28     end
29
30     def asm_defines
31       @asm_defines
32     end
33
34     def libasm_defines
35       @archs[@arch_cursor][1]
36     end
37
38     def arch
39       @archs[@arch_cursor][0]
40     end
41
42     def archs
43       @archs
44     end
45
46     def generate_constants
47       @archs.each do |arch|
48         File.open(@constants_gen_dir + "/#{arch[0]}_values_gen.h", "w") do |
49           file |

```

```
48         file.write(@consts_tmpl.result)
49     end
50     update_arch_cursor
51 end
52 end
53
54 def generate_getters
55     File.open(@target_gen_dir + "/cross_values.h", "w") do |file|
56         file.write(@getters_tmpl.result)
57     end
58 end
59 end
60
61 # ARGV[0] - target_gen_dir
62 # ARGV[1] - asm_defines.def
63 # ARGV[2] - consts_template.erb
64 # ARGV[3] - getters_template.erb
65 # ARGV[n + 0] - arch name
66 # ARGV[n + 1] - arch's libasm_defines.S
67 CV = CrossValues.new;
68 CV.generate_constants
69 CV.generate_getters
```

code/main.rb



## 6.2 Шаблон генерации численных значений

```

1 #ifndef CROSS_VALUES_GENERATED_VALUES_<%= CV.arch %>_VALUES_GEN_H
2 #define CROSS_VALUES_GENERATED_VALUES_<%= CV.arch %>_VALUES_GEN_H
3
4 namespace cross_values::<%= CV.arch %> {
5
6 <% CV.libasm_defines.each do | asm_def | -%>
7     static constexpr ptrdiff_t <%= asm_def[0] %>_VAL = <%= asm_def[1] %>;
8 <% end %>
9
10 } // namespace cross_values::<%= CV.arch %>
11 #endif // CROSS_VALUES_GENERATED_VALUES_<%= CV.arch %>_VALUES_GEN_H

```

code/gen\_constants.erb

## 6.3 Шаблон генерации геттер-функций

```

1 #ifndef CROSS_VALUES_CROSS_VALUES_H
2 #define CROSS_VALUES_CROSS_VALUES_H
3
4 <% CV.archs.each do |arch| -%>
5 #include "generated_values/<%= arch[0] %>_values_gen.h"
6 <% end %>
7 #include <stddef>
8 #include <stdlib>
9
10 namespace cross_values {
11
12 <% CV.asm_defines.each do |define| -%>
13 [[maybe_unused]] static constexpr ptrdiff_t Get<%= define[0].split('_')
14 .collect(&:capitalize).join%>(Arch arch)
15 {
16     switch (arch) {
17         <%- CV.archs.each do |arch| -%>
18             case Arch::<%= arch[0] %>:
19                 return <%= arch[0] %>::<%= define[0] %>_VAL;
20         <%- end -%>
21         default: {
22             std::cerr << "No cross-values generated for arch(" << arch
23             << ")" << std::endl;
24             abort();
25         }
26     }
27 } // namespace cross_values
28 #endif // CROSS_VALUES_CROSS_VALUES_H

```

code/gen\_getters.erb