[RO,LE]1 [RE,LO] [RO,LE] © RU FIG, .. <forth@km.ru> ¡a href='mailto: forth@km.ru '¿ forth@km.ru ¡/a¿ , public domain [RE,LO] July 6, 2006

FORTH
Spirit of Babylon

© RU FIG, Понятов Д.А. <forth@km.ru> forth@km.ru , public domain

версия 6 июля 2006 г.

Содержание

1 Введение **5**
1.1 версия 1 . . . 5
1.2 версия 2 . . . 6
1.3 О языке программирования Форт . . . 6
1.4 Из [os] . . . 7
1.5 Из [orange] . . . 7
1.6 Из перевода [thinking] . . . 9
1.7 © Ю. А. Семенов . . . 9
1.8 © ИТФ Технофорт . . . 11
1.9 Достоинства языка Форт (из [ans]) . . . 12

2 История языка **13**
2.1 Из [orange] . . . 13
2.2 Из [ans] . . . 14
2.3 Forth — The Early Years . . . 14
2.3.1 Abstract . . . 15
2.3.2 Forward 1999 . . . 15
2.3.3 Forth . . . 15
2.3.4 MIT, SAO, 1958 . . . 16
2.3.5 Stanford, SLAC, 1961 . . . 17
2.3.6 Free-lance . . . 18
2.3.7 Mohasco, 1968 . . . 19
2.3.8 NRAO, 1971 . . . 21
2.3.9 Moral . . . 22
2.3.10 References . . . 22

3 Основы языка **23**
3.1 Арифметические операции . . . 23
3.1.1 Операторы для работы с небольшими числами . . . 24
3.1.2 Некоторые проблемы операции деления . . . 24
3.1.3 Операции с величинами и знаками чисел . . . 24
3.1.4 Почему используются целые числа ? . . . 24
3.1.5 Масштабирование чисел . . . 24
3.1.6 Числа двойной длины . . . 24
3.1.7 Совместное применение чисел одинарной и двойной длины . . . 24
3.1.8 trash . . . 24

4 Принципы работы форт-системы **25**
4.1 Создание слов-определителей . . . 25
4.2 Память Форта, словари и контекстные словари . . . 25
4.2.1 Кодофайл . . . 25

5 Создание компиляторов и форт-систем **25**
5.1 Создание собственных компиляторов 25
5.2 Целевая компиляция 26
5.2.1 Минимальный ЦК 26
5.2.2 Целевой компилятор для os 27
5.3 Как написать свой (кросс-)ассемблер 32
5.3.1 Введение 32
5.3.2 Зачем использовать ФОРТ ? 33
5.3.3 Простейший пример: ассемблирование NOP 33
5.3.4 Класс наследуемых опкодов 33
5.3.5 Обработка операндов инструкций 34
5.3.6 Обработка режимов адресации 35
5.3.7 Реализация структур управления 36
5.3.8 BEGIN, UNTIL, ... 37
5.3.9 BEGIN, AGAIN, ... 37
5.3.10 DO, LOOP, ... 37
5.3.11 IF, THEN, ... 38
5.3.12 IF, ELSE, THEN, ... 39
5.3.13 BEGIN, WHILE, REPEAT, ... 39
5.3.14 Заголовок ФОРТ-определения 40
5.3.15 Кросс-компиляция 40
5.3.16 Компиляция на диск 40
5.3.17 Безопасная компиляция 40
5.3.18 Метки 41
5.3.19 Табличный ассемблер 41
5.3.20 Префиксные ассемблеры 41
5.3.21 Вывод 42
5.4 Постфиксный форт-ассемблер для MCS-51 42
5.5 Реализация собственной форт-системы 42

6 Реализация ООП **42**
6.1 Детальное описание mini-OOF 42

7 Плавающая точка **43**

8 Сетевые протоколы **43**

9 Операционная система os **43**
9.1 О системе 43
9.2 Многослойная структура 46
9.2.1 Аппаратная платформа и ОС 46
9.2.2 Виртуальная машина (интерпретатор байт-кода) 46
9.2.3 Целевой компилятор 46
9.2.4 Библиотеки 46
9.2.5 Пользовательские расширения и прикладные программы 46
9.3 Архитектура ВМ 46
9.3.1 Память 46
9.3.2 Стек возвратов 46
9.3.3 Стек данных 46
9.3.4 Регистры 46
9.3.5 Стек циклов со счетчиком 46
9.4 Система команд 46

9.4.2 Графический драйвер monoLCD 47
9.5 Библиотеки 47
9.5.1 Библиотека макросов 47
9.5.2 2D полноэкранная графика 47
9.6 Примеры программ 47
9.6.1 empty 47
9.6.2 blinker 47
9.6.3 liner 47
9.6.4 grdemo 47
9.6.5 pautov 47
9.6.6 cmdline 47

10 Заключение **47**
10.1 Констактные адреса 47
10.2 Необходимый софт 48
10.3 Необходмые навыки 48

Список литературы **49**

Необходимо написание новой книги по языку — существующие книги давно устарели. Нужно очень подробно, понятно и в то же время легко читаемо и без традиционной для российских книг зауми (самый rulez в этом плане Броуди) описать сложные технологии форт-программирования: написание ассемблеров, целевых компиляторов, форт-систем для специфических систем типа самодельных компьютеров и промышленных контроллеров, парсеров и синтаксических анализаторов, сложных расширений языка (например ООП), легко переносимых программ и т.д.. Также думаю стоит большое внимание уделить современным технологиям в железе, особенно flex-технологии "гибкой логики" с использованием ПЛИС (CPLD, FPGA) и систем на кристалле (System on Chip, SoC).

Кто готов участвовать в проекте ? Не обязательно автором — нужны также самые начинающие для обкатки разделов начального обучения, рассказы о практическом применении форт-технологий.

Делатся это все будет по той же лицензии, что и документация на Linux (вариант GNU GPL для документации). Проект уже лежит в CVS, желающих участвовать в редактировании книги welcome регистрироваться.

Группа SynergyOS давно предлагала создать CD с коллекцией материалов по Форту — думаю он будет приложением к книге, там будет коллекция форт-систем для нескольких ОС, снимок этого сайта на момент выпуска диска, может быть фирмы, использующие Форт, тоже захотят включить туда свои материалы (копии сайтов, оценочные версии ПО, прайсы и рекламные материалы).

1 Введение

1.1 версия 1

<forth@km.ru> forth@km.ru

Язык программирования Форт[1] широко известен многим современным хакерам, но не нашел широкого применения на обычных компьютерах. Объясняется это достаточно просто: язык первоначально был разработан как замена ассемблеру примерно в те же времена, что и язык C^{++}[2]. Особенность Форта — система строилась как языковая среда, включающая компилятор, редактор, ассемблер, отладчик и средства операционной системы. Фактически классическая форт-система является полнофункциональной ОС[3] со встроенным компилятором и командной оболочкой.

Несмотря на то, что форт-система требовала очень мало аппаратных ресурсов (всего несколько десятков Кб памяти), и язык достаточно полноценен, на обычных персоналках он большого применения не нашел. Для начинающих программистов к сожалению выбрали BASIC, хотя Форт еще проще и в то же время работает намного быстрее. Профессионалы предпочли использовать C^{++}, так как они изучали его при своем обучении, и решили что синтаксис Форта плохо читаем. Основная сложность – язык Форт обладает уникальным свойством саморасширяемости, поэтому для понимания исходных кодов чужих программ нужно понять, как автор программы расширил язык.

Тем не менее во встраиваемых системах (системы управления различным оборудованием, робототехника, бортовое и наземное оборудование в космических исследованиях NASA) Форт до сих пор имеет достаточно широкое применение.

Если вы занимаетесь такими вещами как разработка различных цифровых систем и программного обеспечения для них, вы можете столкнуться с ситуацией, когда вас не будет устраивать по каким-либо причинам обычно используемый в таких случаях инструментарий. Этими причинами может оказаться его дороговизна, неудобство, отсутствие каких-либо утилит, или для вашего железа не оказалось готовой ОС, компилятора и библиотек.

В этом случае хорошим решением может быть использование языка Форт в качестве мощного макроассемблера или операционной системы.

К сожалению, последняя книга на русском языке вышла в 1993 году, существующей литературы очень мало и она явно устарела. С другой стороны, в ней совсем нет примеров практического использования Форта и сложных техник форт-программирования (целевой компиляции, ООП расширений, написания ассемблеров, откомментированных исходников форт-систем для различных платформ).

Очень часто возникают возражения типа "Форт слишком сложен в использовании", но *он проще чем ассемблер, который до сих пор используют очень часто*.

Цель написания этой книги — собрать разрозненную информацию по языку Форт и показать как можно использовать этот язык в качестве мощного средства для создания программного обеспечения встраиваемых систем. Эта задача достаточно сложна, учитывая то что я не форт-гуру, но к сожалению гуру не хватает времени (а чаще желания) писать качественную документацию. Надеюсь что они все же примут активное участие в написании этого руководства.

Буду очень рад, если кто-то захочет присоединится к работе над книгой: она изначально создавалась в on-line версии, для ее редактирования используется система управления контентом в виде CVS[4]-сервера и простейшего скрипта, генерирующего сайт [akps]. Все мои работы по языку также рассылаются через список рассылки comp.soft.prog.forth

---

[1] ФОРТ, Форт, Forth или FORTH

[2] Си или C++

[3] операционной системой, или OS

 comp.soft.prog.forth [5]. Для работы над книгой необходимо участие начаинающих осваивать язык для обкатки учебных разделов, людей использующих или наоборот не использующих Форт по каким-то причинам для написания разделов по практическому применению форт-технологий, и вообще присылайте любые ваши вопросы, комментарии и замечания — мне необходима обратная связь.

1.2 версия 2

<forth@km.ru> forth@km.ru

Эта книга — попытка объединить несколько основных книг по языку Форт, выпущенных на русском языке. После 1995 года не было выпущено не одной книги об этом языке, но даже с учетом того что язык мало распространен[6] и развивается очень медленно, эти книги все равно устарели.

Использовать Форт имеет смысл в достаточно узкой области, где используются маломощные компьютеры с очень небольшими объемами памяти (8- 16 -битные процессоры, десятки и сотни Кб ОЗУ). Язык очень низкоуровневый, поэтому если вам доступен полноценный компилятор C^{++}, лучше использовать его, а не Форт.

Применять Форт можно в том случае, если не доступен готовый компилятор C^{++} — Форт отличается крайней простотой внутреннего устройства и синтаксиса, поэтому написать компилятор Форта можно буквально за полчаса (см. [akps]).

Еще один вариант использования Форта — в качестве операционной системы, написав интерактивную форт-систему. При этом вы получаете командный интерфейс, встроенный компилятор и набор runtime библиотек, объединенные в очень компактную операционную систему.

1.3 О языке программирования Форт

Если вы занимаетесь такими вещами как разработка различных цифровых систем[7] и программного обеспечения для них, вы можете столкнуться с ситуацией, когда вас не будет устраивать по каким-либо причинам обычно используемый в таких случаях инструментарий[8]. Этими причинами может оказаться дороговизна инструментария, его неудобство, отсутствие каких-либо утилит, или в особых случаях когда вы сами сляпали нечто, и для этой железки не оказалось готовой ОС и библиотек.

К сожалению, методика целевой компиляции, применяемая при использовании Форта в таких случаях, описана недостаточно подробно и понятно. В сети много введений для начинающих, но очень мало материалов для опытных фортеров (программистов на Форте), и написание ассемблеров, целевых компиляторов, форт-систем и различных расширений не описано в достаточной степени. Многие разработчики слышали о языке Форт, но не используют его из-за отсутствия хорошей документации. Очень часто возникают возражения типа "Форт слишком сложен в использовании", но *он проще чем ассемблер, который используют очень часто*. Малое распространение Форта определяется двумя причинами:

1. отсутствие документации о программировании на Форте достаточно сложных программ типа компиляторов или ООП-расширений и

2. отсутствие хорошо документированных наработок на Форте в виде его расширений (библиотек) и исходных кодов форт-систем.

---

[5] http://www.subscribe.ru/lists/comp.soft.prog.forth

[6] хотя и широко известен

[7] встраиваемые контроллеры для управления технологическим или лабораторным оборудованием

Учитывая возможности современных технологий программирования, существующих библиотек и возможностей операционных систем, язык Форт имеет также смысл использовать в качестве встраиваемого в программы языка, который удобно использовать для написания конфигурационных файлов и программирования пользователем. Естественно это имеет смысл только если существуют ограничения на ресурсы, которые может использовать такое скрипт-расширение. Если таких ограничений нет, правильнее использовать встраиваемые скрипт-языки с более удобоваримым синтаксисом типа Python. Еще одна область применения — быстрое написание кросс-ассемблеров, которые более удобны чем традиционные в случаях, когда нужна мощная поддержка макросов.

Достоинства Форта, которые могут определить его выбор:

- возможность реализации виртуальной машины в готовых программах на любых языках с минимальными затратами;
- минимальные требования к ресурсам;
- встроенный в ядро компилятор и опционально ассемблер;
- компактность кода;
- расширяемость.

1.4 Из [os]

Язык Форт по своей идеологии очень минималистичен, так как был разработан для программирования систем с очень ограниченными ресурсами — сейчас такие системы можно встретить исключительно в виде блоков управления различным оборудованием. Даже компьютеры внутри мобильников на порядок мощнее по доступным ресурсам — скорости, разрядности процессора, объемам памяти, возможностям вывода графики.

Сам язык практически не защищает программиста от ошибок типа некорректной работы с данными, не поддерживает типизацию, ООП, структуры данных и т.д.. Практически это ассемблер стековой машины, и программирование на нем очень близко к программированию на обычных ассемблерах — программист вынужден реализовывать все вручную, самостоятельно отслеживая состояние стека, корректность адресов и указателей и т.д..

Если вы хотите иметь язык, выполняющий более серьезный контроль вашего кода, Форт для вас не подходит.

С другой стороны у Форта есть одна особенность, которой многие другие языки не обладают, и которая является ключевой — на Форте вы не столько программируете, сколько расширяете язык через уже существующие в нем слова, определяя диалект под вашу задачу. Подробнее см. [orange] и особенно [thinking]. Из-за этой особенности Форт нельзя назвать языком низкого уровня — хотя его ядро и низкоуровневое[9], но его расширение приводит к созданию диалектов сверх-высокого уровня.

Другими словами: Форт — язык для конструирования проблемно-ориентированных языков, работающих на очень простой виртуальной машине. Реализация такой стековой ВМ для реального железа также проста, поэтому эта ВМ легко и быстро реализуется на любом языке или даже на аппаратном уровне в виде специализированных процессоров в FPGA или кремнии.

Кроме того, базовый Форт очень прост как в изучении, так и в самостоятельной реализации, особенно если использовать методику целевой компиляции в интерпретируемый байт-код, описанной в этой книге, и любой чайник вполне в состоянии написать свой работающий вариант Форта буквально за несколько часов для любой платформы и на любом языке.

Естественно при этом у него будут те же проблемы с низкоуровневостью языка и необходимостью расширять язык до приемлемого уровня, но тогда он всегда сможет сказать, что сам написал для себя язык и средства разработки.

В этом случае нужно очень аккуратно следить за совместимостью вашей версии с версиями других хакеров — лучше использовать чужие готовые наработки и делиться своими, чем вариться в собственном соку, используя свой уникальный несовместимый диалект Форта.

1.5 Из [orange]

Материал книги построен для использования ее в качестве справочника так, чтобы охватить весь набор средств и приемов и полный перечень слов и функций языка. Кроме того, приведены примеры расширения языка и использования Форта в качестве средства разработки программ и как операционной системы.

И все же многие утверждают, что Форт труден для изучения. Для этого имеется несколько причин. Опытным программистам Форт зачастую дается труднее, чем новичкам, потому что он отличается от других языков программирования по самой своей природе. Хотя в языке Форт нет каких-либо присущих только ему сложностей, программисты с трудом отвыкают от переменных, подпрограмм, многословного текста на исходном языке, алгебраических обозначений и прочих атрибутов привычных им языков. Если вы знаете другие языки программирования, попробуйте к языку Форт подойти с полной отдачей. Освойте понятия стека и определения слов, прежде чем переходить к более сложным вопросам. Забудьте всякие предубеждения, которые у вас могут возникнуть, вроде того, что для хорошего языка программирования обязательно нужна операционная система и файловая поддержка. И не беспокойтесь о блок-схеме, начните с небольших задач, ваш опыт будет накапливаться постепенно. Форт может показаться трудным, так как это достаточно мощное средство программирования.

Действительно, все возможности языка изучить трудно, но все они и не потребуются, чтобы писать очень полезные программы. На Форте можно научиться писать программы на уровне хорошего программиста, пользующегося языками BASIC и FORTRAN, быстрее, чем на любом другом языке. Так же несложно программирование на форт-ассемблере (определение слов Форта в машинных кодах). Можно описать слова, которые будут создавать в словаре совершенно новые типы данных, использовать Форт для модификации самого языка, для операций с большими объемами данных в памяти компьютера и даже для того, чтобы реализовать новые языки программирования. Конечно, изучение этих вопросов может быть трудным. И, хотя мы осветим большую их часть, вы сможете хорошо программировать на языке Форт раньше, чем овладеете ими всеми. Форт предоставляет вам мощные средства для управления работой компьютера, присущие другим языкам программирования, включая машинный язык, но вам эти мощные средства скорее всего не потребуются.

Существует одна причина, из-за которой Форт иногда оказывается действительно трудным. Дело в том, что в стандартах языка и в поставляемых потребителям реализациях языка отсутствуют некоторые слова для выполнения основных или важных функций, предусмотренных другими языками. В таких случаях программист вынужден сам написать слова, которые должны выполнять эти важные функции. Например, старый стандартный Форт'83 не содержит операций над числами с плавающей запятой, в нем нет трансцендентных функций (тригонометрических, логарифмической); не определены стандартами операции с символами и символьными строками (например, извлечение отдельных слов из текста), работа с файлами данных, графические возможности. Нет в стандарте и слова, позволяющего вводить числа в процессе исполнения программы. К счастью, во многих поставляемых реализациях Форта предусмотрены слова, позволяющие преодолеть эти ограничения, а Форт настолько мощный, что позволяет самому написать такие слова, если

Кроме того, 1994 году вышел новый стандарт [ans], в который включена большая часть из выше перечиленного. И в конце концов всегда остается доступ в Сеть, используя которую вы можете найти исходные тексты программ, сразу подходящих для решения ваших задач или требующих некоторой адаптации. Если эти программы не комплектуются документацией достаточного качества, свяжитесь с их авторами по электронной почте — в этом случае вы можете не только получить консультацию, но возможно появится новая версия программы, в создании которой вы сами примете участие. Именно в этом и зключается смысл движения Open Source[10] *!*

Подводя итог, можно сказать, что Форт — это в то же время мощный и неразвитый язык. Мощный он потому, что программы, написанные на нем, занимают мало места в памяти и исполняются с такой же скоростью или быстрее, чем на других языках, он дает потенциально неограниченные возможности управления компьютером, и писать и отлаживать программы довольно несложно. В то же время стандартный Форт неразвит, ввиду того что не предусматривает выполнения некоторых важных функций, которые являются неотъемлемой частью других языков или операционной системы и доступны для программиста, и кроме того в состав поставки не включены мегабайты библиотек, как это делают поставщики современных коммерческих средств разработки. Введение этих функций и создание библиотек предоставлено программисту, предусматривается в некоторых (коммерческих или расширенных) версиях Форта или просто доступно в Сети в виде отдельных пакетов[11].

Почему же стандартный Форт так бедно определен ? Чтобы понять это, надо немного познакомиться с историей его создания и естественно учесть что в его развитии не участвовала та критическая масса програмистов и поставщиков коммерческого ПО, как это случилось с языками типа C^{++}, Python, Perl, Java и т.п.

1.6 Из перевода [thinking]

Уже несколько лет я использую для программирования язык Форт. С первой же встречи с ним я был очарован и покорен его простотой, элегантностью и логичностью. К сожалению, в нашей стране Форт знают и используют лишь считанные энтузиасты, чему в большой мере способствует отсутствие сколько-нибудь доступной современной литературы и программного обеспечения.

Несколько книг по языку изданы более 10 лет назад, они дают достаточно информации для начинающего *фортера*, но в них не описаны методы и приемы, разработанные и описанные мировым сообществом фортеров.

Необходимо написание новой книги по языку: существующие книги давно устарели, необходимо очень подробно, понятно и в то же время легко читаемо и без традиционной для российских книг зауми (самый rulez в этом плане Броуди) описать сложные технологии форт-программирования: написание ассемблеров, целевых компиляторов, форт-систем для специфических систем типа самодельных компьютеров и промышленных контроллеров, парсеров и синтаксических анализаторов, сложных расширений языка (например ООП), написание легко переносимых программ и т.п. Также возможно стоит уделить внимание современным технологиям в железе, особенно flex-технологии "гибкой логики" с использованием ПЛИС (CPLD, FPGA) и систем на кристалле (System on Chip, SoC).

В качестве базы я решил взять [starting, orange, thinking, green] и написать on-line книгу учитывая современное состояние языка. Первоначально это будет простая компиляция этих книг, которая в дальнейшем будет редактироваться и модифицироваться, в том числе и с вашим участием.

Форт является языком и операционной системой. Но это не все: он также и воплощение философии. Обычно философию не рассматривают как нечто, отдельное от Форта. Она

[10] http://www.fsf.org http://www.fsf.org

не предшествовала Форту и не описывалась где-либо вне рассуждений о Форте, и даже не имеет другого имени, кроме как "Форт". Точнее даже сказать что это не философия, а (информационная) технология и методика разработки программ.

Предлагаемая книга является одновременно учебником и справочником, позволяющим овладеть Фортом независимо от того, начинающий вы программист или опытный.

Форт — достаточно мощный язык программирования, несмотря на его низкоуровневость, но он наиболее эффективен для компьютерных систем с медленным процессором низкой разрядности и небольшим объемом памяти (порядка нескольких Кб).

1.7 © Ю. А. Семенов

Статья является введением к книге: Семенов Ю. А., "Программирование на языке Форт", М., "Радио и связь", 1991 г.

В 1971 г. Чарльз Мур разработал язык для управления оптическим телескопом и, считая его языком четвертого поколения, назвал FOURTH (четвертый). Однако на ЭВМ, на которой он работал, символьные имена могли иметь только пять букв. Так FOURTH стал FORTH (Форт). Несмотря на конкуренцию других языков программирования, в частности языка C^{++}, Форт мало-помалу стал завоевывать популярность, особенно при решении задач управления сложными объектами в реальном масштабе времени.

Язык Форт использовался для математического обеспечения корабля многоразового использования типа Shuttle, разведывательного 1802 (Avco Inc.) и других искусственных спутников Земли, для разработки телеигр (GameFORTH), при создании фильмов Star Wars, Batlle Beyond the Stars и Star Trek, для системы управления полетами в аэропорту Эр-Рияда (400 ЭВМ и 36 000 датчиков) [24].

В 1976 г. Комитет международного астрономического союза принял Форт в качестве стандартного языка программирования. Позднее Форт применялся для создания экспертных систем, систем искусственного зрения, автоматизации анализа крови и кардиологического контроля, систем машинного перевода с 20 языков (Craig M100, карманный переводчик) и т.д..

В СССР этот язык используется для систем управления базами данных экономических задач, для программ управления экспериментом, мониторинга состояния пациентов.

Несмотря на ощутимые успехи в использовании языка Форт, делать прогноз о беспредельном расширении сферы его применения вряд ли можно. В то же время, безусловно, существуют области, где Форт имеет несомненные преимущества перед другими языками. Форт эффективен прежде всего для управления небольшими экпериментами и системами, при диагностике сложной электронной аппаратуры с помощью микроЭВМ или микропроцессоров, для создания дешевых поисковых систем, программ машинной графики, трансляторов с других языков. Это, разумеется, не означает, что Форт неприменим в других областях, его возможности еще не раскрыты полностью.

Язык Форт иногда называют форт-системой, так как он содержит программы для работы с внешними устройствами, файлами, средства обработки прерываний, редактор и т.д.. Преимущество Форта заключается прежде всего в скорости написания и отладки программ, а также в их компактности. Если программу на Фортране или Паскале можно написать и отладить за неделю, то такую же программу на Форте за несколько часов. По сравнению с Бейсиком и некоторыми другими интерпретаторами Форт позволяет составить программу в несколько раз более быстродействующую. Он проигрывает Ассемблеру по скорости исполнения программы не более чем в 1.5–2 раза. Применение же форт-ассемблера позволяет получить еще больший выигрыш в быстродействии.

Экономное использование оперативной памяти ЭВМ и внешней памяти (диска), возможность автономного функционирования (в отсутствие операционной системы), интерактивный характер делают Форт особенно привлекательным в небольших автоматизиро-

ЕС ЭВМ для организации сложных вычислительных процесс сов [36]. Хорошо написанная программа на Форте занимает в памяти меньше места, чем аналогичная, составленная на Ассемблере (здесь не учитывается место, занимаемое процедурами базового словаря Форта).

Модульность интерпретатора и системы в целом позволяет легко адаптировать ее к новым процессорам и задачам. По простоте обучения Форт соперничает с Бейсиком, что делает его привлекательным для непрофессиональных программистов.

Какие же особенности обеспечили Форту шанс выжить в конкурентной борьбе с другими системами ?

Для Форта, включая программы управления терминалом и диском, требуется 5–8 Кбайт оперативной памяти, а для Паскаля 48 Кбайт (здесь, правда, не учитывается место, занимаемое операционной системой). С учетом этого выигрыш по памяти оказывается существенно больше. При необходимости базовый словарь Форта может быть сокращен до 1 Кбайт. Программа на Форте может быть исполнена сразу после написания, так как не требует редактора связей.

Форт допускает рекурсию, т.е. программа может обращаться к самой себе (что недопустимо в Фортране).

После выполнения программы и возврата управления системе Форт сохраняется доступ к любой переменной или массиву с помощью символьных имен, что не допускают многие другие языки.

При работе с форт-ассемблером исполняемая программа практически идентична программе, написанной в машинных кодах, но программист избавлен от длительной трансляции и редактирования связей (программа сразу готова к исполнению). Проигрыш по памяти и скорости исполнения программы в этом случае не превышает 20–30%.

Список операторов Форта открыт для пользователя и может быть расширен по его усмотрению. Это касается самого Форта, форт-ассемблера, редактора и, разумеется, пакетов прикладных программ.

При работе с многозадачным (многопользовательским) Фортом используется общий словарь процедур для нескольких задач.

К недостаткам Форта относят:

- отсутствие в базовой версии операторов для работы с числами с плавающей точкой;
- недостаточные по современным требованиям средства диагностики ошибок (хотя это и компенсируется отчасти другими возможностями, в частности доступностью этих средств для пользователя);
- непривычная для многих обратная польская (постфиксная) нотация, широкое применение стеков;
- отсутствие в базовой версии развитой системы для работы с файлами (отчасти устранен в интерпретаторах для персональных ЭВМ).

Простота расширения списка операторов и легкость модификации интерпретатора сводят эти недостатки к минимуму. Удобная мнемоника и хорошая адаптация ко вкусам программиста могут сделать работу на Форте легким и даже приятным занятием. Далее будет рассматриваться в основном версия FIG-FORTH, как наиболее распространенная.

форт-интерпретатор, как и многие другие, имеет несколько уровней:

- работа системы в качестве настольного калькулятора;
- программирование с использованием базового словаря Форта;
- создание с помощью редактора программ, сохраняемых на магнитном диске, которые

Принадлежность Форта четвертому поколению часто вызывает споры. С одной стороны, он содержит структурные операторы типа DO...LOOP, BEGIN...UNTIL и т.д., с другой — допускает работу с адресами, что характерно для языков низкого уровня. Главное — это баланс достоинств и недостатков, а этот баланс, на мой взгляд, благоприятен для Форта.

В США создано общество пользователей Форта (FORTH Interest Group, FIG) и фирма FORTH Inc. — главный поставщик программных продуктов Форт, в том числе многопользовательской версии PolyFORTH-2.

Существует ряд стандартов: FORTH-79, FORTH-83 [18], FIG [17], MMS, MVP [25, 26]. В какой-то мере этому способствует простота их создания, ведь мало кто решится переделывать или расширять базовый словарь операторов Фортрана просто из-за чудовищной трудоемкости. Наметилась тенденция к созданию процессоров, ориентированных на Форт (фирмы Harris, Silicon Composers, FORTH Inc. США и Институт кибернетики АН ЭССР).

Существуют версии языка Форт для отечественных ЭВМ СМ 1420, "Электроника-60", СМ 1810, ДВК, персональных ЭВМ ЕС1840, ЕС1841, ЕС 1842, микропроцессоров К580 и даже ЭВМ серии ЕС.

1.8 © ИТФ Технофорт

Язык программирования Форт (от английского FORTH) был изобретен Чарльзом Муром в 70-х годах для создания программного обеспечения управляющих устройств. В настоящее время Форт широко используется при решении следующих задач:

- разработка и тестирование встроенного оборудования;
- управление станками, роботами, медицинскими приборами;
- разработка трансляторов и операционных систем;
- системы управления базами данных;
- задачи машинной графики;
- экспертные системы, в том числе экспертные системы реального времени.

В отличие от других языков высокого уровня, Форт обеспечивает программисту полный доступ к машине и не пытается оградить его от ошибок. Однако, модульность, а также расширяемость языка, позволяющая программисту вводить конструкции со встроенными средствами контроля, дает возможность создавать высоконадежные программы.

Форт использует обратную польскую (постфиксную) запись, при которой операнды предшествуют операции. Хотя такая запись непривычна и может показаться неудобной, она существенно уменьшает затраты на организацию вызовов подпрограмм и реализацию языка.

Код, получаемый компилятором Форта, исключительно компактен, даже по сравнению с машинным кодом. Особенно это заметно на больших программах.

форт-система, в основном, написана на самом языке Форт. Она занимает от 8 до 16 Кбайт в зависимости от предоставляемых возможностей (таких, как встроенный ассемблер, экранный редактор, взаимодействие с файловой системой).

Программы на языке Форт реентерабельны, допускают рекурсию. Программист может написать программу в машинных командах на встроенном в форт-систему ассемблере и в дальнейшем использовать ее как обычную подпрограмму. Вследствие этого, Форт можно применять для создания программ непосредственного управления аппаратурой.

форт-система — автономная система. Она может работать как на "голом" оборудова-

Форт является диалоговым языком, то есть команды выполняются форт-системой сразу, как только Вы их введете с клавиатуры и нажмете клавишу ввода. Ответ "ok" является подтверждением того, что запрос выполнен, и приглашением продолжать работу.

1.9 Достоинства языка Форт (из [ans])

Форт предоставляет интерактивную среду разработки (IDE). Его первые применения в научных и промышленных приложениях таких как аппаратура, робототехника, управление процессами, графика и обработка изображений, искуственный интеллект и бинес-приложения. Принципиальные достоинства Форта включают быструю интерактивную разработку программного обеспечения и эффективное использование компьютерного железа.

О Форте часто говорят как о языке, так как это наиболее заметный аспект. Но фактически Форт одновременно больше и меньше чем обычный язык программирования: больше так как все возможности языка, обычно ассоциированные с большим набором отдельных программ[12], включены в язык, и меньше так как в языке (сознательно) отсутствуют сложные синтаксические конструкции, характериные для большинства языков высокого уровня.

Оригинальные реализации Форта были самостоятельными системами, включающими функциональность, обычно обеспечиваемую отдельно операционной системой, редакторами, компиляторами, ассемблерами, отладчиками и другими утилитами. A single simple, consistent set of rules governed this entire range of capabilities. Сегодня до сих пор продаются множество версий самостоятельных[13] форт-систем для многих процессоров, но при этом также существует множество версий, работающих поверх обычных операционных систем типа MS-DOS или UNIX.

Форт не является ответвлением какого-либо другого языка. В результате его внешний вид и внутренние характеристики могут показаться непривычными для новых пользователей. Но простота Форта, его крайняя модульность и интерактивная природа отодвигают его начальную непривычность, делая простым его изучение и использование. Новый фортер[14] должен потратить некоторое время на изучение большого набора команд языка. Примерно после месяца использования Форта фортер может понять больше внутренних механизмов языка, чем это возможно для обычных операционных систем и компиляторов.

Наиболее интересное свойство Форта — его расширяемость. Процесс программирования на Форте состоит из определения новых *слов* — фактически новых команд языка. Они могут быть определены в терминах ранее определенных слов, что сильно похоже на обучение ребенка объясняя новые понятия через ранее изученные. Такие слова называются *высокоуровневыми определениями*. С другой стороны, новые слова могут также быть определены на уровне машинных команд, так как большинство реализаций Форта включают ассемблры для используемых процессоров.

Эта расширяемость опеспечивает разработку специальных прикладных языков для конкретных проблемных областей или дисциплин.

Расширяемость Форта дает больше чем добавление новых команд в язык. С той же простотой вы можете также добавлять новые типы слов. Например, вы можете создать слово, которое само будет определять слова. При создании таких *определяющих слов* программист может задать особое поведение для создаваемых слов, которое будет эффективно во время компиляции, исполнения или в обоих случаях. Эта возможность позволяет вам определить специализированные типы данных с полным контролем над их структурой и поведением. Так как поведение таких слов во время исполнения также может быть задано на высоком уровне или в машинном коде, созданные таким определяющим словом новые слова эквивалентны по производительности другим словам Форта. Так же просто доба-

[12] компиляторами, редакторами и т.д.

[13] stand-alone

вить новые компилирующие команды для реализации особых видов циклов или других управляющих структур.

Большинство профессиональных реализаций Форта сами написаны на Форте. Многие форт-системы включают *мета-компилятор*, который позволяет пользователю модифицировать внутренний код самой форт-системы.

2 История языка

2.1 Из [orange]

В отличие от других языков программирования Форт не является плодом коллективного труда какого-либо комитета или коллектива ученых, он родился в голове одного человека — Чарльза Х. Мура. В начале 60-х годов Мура стало все больше не удовлетворять время и затраты труда, требовавшиеся для разработки программ на существовавших тогда ЭВМ. В течение нескольких лет он создал основы прототипа языка Форт, пользуясь для этого такими языками, как Алгол, Кобол, PL/I и ассемблер для IBM/360. Мур дал своему языку название FORTH, считая, что это будет язык для ЭВМ четвертого (fourth) поколения, однако ему приходилось работать на ЭВМ, которая допускала только пять букв названия, этим и объясняется такое необычное имя языка Форт. В 1971 г. Мура пригласили на работу в Национальную Радиоастрономическую обсерваторию для разработки программ сбора и обработки данных, получаемых с радиотелескопа. В процессе этой работы и появилась первая современная реализация языка Форт (а вторым в мире программистом на этом языке стала сотрудница Мура Элизабет Ратер), который был принят в качестве основного языка программирования в Американском астрономическом обществе. К 1973 г. потребность в языке стала настолько большой, что Мур и Ратер создали новую фирму Forth Inc., президентом которой стала Э. Ратер. Фирма разработала несколько версий языка Форт для различных марок ЭВМ, мини- и микро-ЭВМ, в том числе наиболее современную версию под названием PolyFORTH. По мнению специалистов фирмы Forth Inc., наиболее важные применения языка Форт в настоящее время связаны с работой программно-управляемого оборудования в реальном масштабе времени, хотя еще в 1974 г. фирма разработала на языке Форт административную систему для управления базой данных объемом 300 млн. бит информации. Сейчас фирма Forth Inc. сосредоточила свои усилия на обработке изображений, робототехнике и управлении сервоприводами. Недавно появились программы для таких применений, как автоматизация проверки накопителей на гибких магнитных дисках, для первого прототипа коммерческого устройства по электрофорезному разделению биологических материалов на космическом корабле многоразового использования, для сервосистем роботов, управляемых голосом, а также для контроля почти всех операций в новом главном аэропорте Саудовской Аравии.

Однако популярностью язык Форт обязан не только своим авторам Муру и Ратер и фирме Forth Inc.. Базовый язык Форт общедоступен и бесплатно распространяется заинтересованной группой FORTH Interest Groop (FIG), множество фирм поставляет различные по своим возможностям коммерческие версии языка, но еще важнее то, что организована группа по стандартизации языка Форт для того, чтобы написанные на нем программы могли бы работать на различных компьютерах с минимальными затратами на их адаптацию. Группа FIG была создана в конце 70-х гг. активными программистами и почитателями языка, которые хотели сделать его еще более популярным. Существует множество организаций, в том числе несколько отделений этой группы в США и других странах мира, однако основная деятельность группы направлена на распространение базового диалекта языка, FIGFORTH (которая реализована на многих мини- и микроЭВМ) и издание журнала FORTH Dimensions, выходящего раз в два месяца. Содержание журнала показывает, что редакция проявляет интерес как к модификации и расширению самого языка,

каждый год, начиная с 1980-го, группа FIG созывает конференцию под названием FORTH Modification Laboratory (FORML), целью которой является встреча пользователей и системных программистов для обсуждения вопросов дальнейшего развития языка. Кроме этой конференции Институт прикладных исследований фирмы Applied FORTH Research ежегодно организует в Рочестере (США) конференцию по применению Форта (Rochester FORTH Application Conference). Институт публикует труды конференции и, кроме того, профессиональный журнал Journal of FORTH Application and Research, содержащий рефераты, библиографические ссылки на материалы по языку Форт и материалы студенческих исследований.

Группа по стандартизации первоначально возникла в рамках Международного объединения астрономов. На встрече в Национальной обсерватории Китта (США) в мае 1977 г. был выработан глоссарий языка Форт под шифром AST.01, а после нескольких встреч в Европе, наконец, в феврале 1978 г. в Утрехте (Голландия) был разработан стандарт 1977 г. (FORTH-77), адресованный прежде всего пользователям микроЭВМ. В октябре 1979 г. встреча на острове Каталина закончилась разработкой стандарта FORTH-79, который распространяется на ЭВМ всех типов. Осенью 1983 г. состоялась встреча по разработке стандарта 1983 г, утвержденного в 1984 г, как FORTH-83. Стандарт Форт-83 отличается от стандарта Форт-79 некоторыми деталями, но не отличается от него по существу. Некоторые специалисты, в том числе и поставщики коммерческих версий языка, считают, что изменения 1983 г. в лучшем случае не привели к совершенствованию языка, в худшем же — внесли некоторую путаницу, поэтому как Форт-83, так и Форт-79 имеют равное распространение, и мы рассматриваем здесь обе версии. Несмотря на то, что некоторые изменения могут привести к смешению обеих версий, были все же некоторые разумные, хотя и не очень важные причины внесения этих различий.

С момента издания книги был выпущен новый стандарт ANS FORTH '94, и теперь ситуация аналогична описанной выше — существуют множество форт-систем, полностью или частично соответствующих стандарту 83 или 94. В новом стандарте было добавлено множество расширений базового словаря (плавающая точка, файлы, динамическая память и т.д.), но до сих пор есть множество фортеров, которых не устраивают какие-то особенности нового стандарта.

Целью стандартизации было создание единого набора слов (глоссария), чтобы можно было легко переносить программы с одного компьютера на другой (особенно если они сильно отличаются по ОС и железу, например ПК на интеловских процессорах и ZX Spectrum). Стандарт определяет минимальный набор обязательных слов и необязательный набор расширяющих слов, например ассемблер и слова из контролируемого списка, выполняющие точно определенные функции. К сожалению, в стандарте нет никаких указаний на такие важные компоненты, как числа, символьные строки, на организацию файлов данных (см. замечание выше). Концепции группы стандартизаторов частично были приведены в статье ее председателя У. Рэгсдейла: "Самой трудной задачей разработки языка является принятие решения о том, что следует отбросить. Стандартизаторы сталкиваются с задачей установления равновесия между практически полезными функциями и простейшими функциями, обеспечивающими пользователю возможность дополнить программные средства, чтобы решать прикладные задачи". В результате получилось так, что во всех поставляемых версиях Форта возникает необходимость в добавлении собственных функционально важных слов, что противоречит идее свободной переносимости программ между различными компьютерами.

Почему же все-таки в Форт не включены слова для выполнения функций, безусловно предусмотренных в других языках программирования ? Частично это объясняется традиционными применениями языка, отчасти интересами группы FIG и многочисленных пользователей языка Форт, в особенности любителей, и частично возражениями группы по стандартизации. Во-первых, традиционной областью применения языка было управление установками в реальном масштабе времени, где Форт подходит наилучшим образом.

рационной системой и другие качества здесь не столь важны, как при решении более практических задач, например при проведении инженерных расчетов или в деловой сфере. Во-вторых, основной задачей группы FIG было распространение простейшей версии языка FIGFORTH и, как отмечается в журнале FORTH Dimensions, большинство членов FIG проявляет интерес скорее к модификациям языка, чем к применению его в повседневной жизни, и поэтому не имеют побудительного толчка к снабжению Форта такими функциями. Наконец, группа стандартизации предоставляет пользователю возможность самому выполнить разработку дополнительных инструментальных средств, сведя к минимуму набор стандартных слов. При этом бремя разработки возлагается на пользователей и поставщиков промышленных версий языка, лишь в последние три года некоторые необходимые функции были введены в поставляемых реализациях Форта.

2.2 Из [ans]

2.3 Forth — The Early Years

Chuck Moore: The Invention of Forth

chipchuck@colorforth.com
1991

2.3.1 Abstract

Forth is a simple, natural computer language. It has achieved remarkable acceptance where efficiency is valued. It evolved in the 1960s on a journey from university through business to laboratory. This is the story of how a simple interpreter expanded its abilities to become a complete programming language/operating system.

2.3.2 Forward 1999

This paper was written for the HOPL II (History of programming languages) conference. It was summarily rejected, apparently because of its style. Much of the content was included in the accepted paper [Rather 1993].

This HTML version was reformatted from the original typescript. Minimal changes were made to the text. Examples of source code were suggested by reviewer Phil Koopman. They've not yet been added.

2.3.3 Forth

Forth evolved during the decade of the 60s, across America, within university, business and laboratory, amongst established languages. During this period, I was its only programmer and it had no name until the end. This account is retrieved from memory, prompted by sparse documentation and surviving listings.

Forth is hardly original, but it is a unique combination of ingredients. I'm grateful to the people and organizations who permitted me to develop it — often unbeknownst to them. And to you, for being interested enough to read about it.

Forth is a simple, natural computer language. Today it is accepted as a world-class programming language. That it has achieved this without industry, university or government support is a tribute to its efficiency, reliability and versatility. Forth is the language of choice when its efficiency outweighs the popularity of other languages. This is more often the case

A number of Forth organizations and a plethora of small companies provide systems, applications and documentation. Annual conferences are held in North America, Europe and Asia. A draft ANSI standard will soon be submitted [ANS 1991].

None of the books about Forth quite capture its flavor. I think the best is still the first, Starting Forth by Leo Brodie [Brodie 1981]. Another window is provided by JFAR's invaluable subject and author index [Martin 1987].

The classic Forth we are discussing provides the minimum support a programmer needs to develop a language optimized for his application. It is intended for a work-station environment: keyboard, display, computer and disk.

Forth is a text-based language that is essentially context-free. It combines 'words' separated by spaces, to construct new words. About 150 such words constitute a system that provides (with date of introduction)

SAO	1958	Interpreter
SLAC	1961	Data stack
RSI	1966	Keyboard input
		Display output, OK
		Editor
Mohasco	1968	Compiler
		Return stack
		Dictionary
		Virtual memory (disk)
		Multiprogrammer
NRAO	1971	Threaded code
		Fixed-point arithmetic

Such a system has 3–8K bytes of code compiled from 10–20 pages of source. It can easily be implemented by a single programmer on a small computer.

This account necessarily follows my career. But it is intended to be the autobiography of Forth. I will discuss the features listed above; and the names of the words associated with them. The meaning of many words is obvious. Some warrant description and some are beyond the scope of this paper.

Interpreter						
	WORD	NUMBER	INTERPET	ABORT		
	HASH	FIND	'	FORGET		
	BASE	OCTAL	DECIMAL	HEX		
	LOAD	EXIT	EXECUTE	(		
Terminal						
	KEY	EXPECT	EMIT	CR		
	SPACE	SPACES	DIGIT	TYPE		
	DUMP					
Data stack						
	DUP	DROP	SWAP	OVER		
	+	-	*	/		
	MOD	NEGATE				
	ABS	MAX	MIN			
	AND	OR	XOR	NOT		
	0<	0=	=			
	@	!	+!	C@	C!	
	SQRT	SIN.COS	ATAN	EXP	LOG	
Return stack						
	:	;	PUSH	POP	I	
Disk						
	BLOCK	UPDATE	FLUSH			
	BUFFER	PREV	OLDEST			
Compiler						
	CREATE	ALLOT	,	SMUDGE		
	VARIABLE	CONSTANT				
	[	]	LITERAL	."	COMPILE	
	BEGIN	UNTIL	AGAIN	WHILE	REPEAT	
	DO	LOOP	+LOOP	IF	ELSE	THEN

2.3.4 MIT, SAO, 1958

October, 1957 was Sputnik — a most exciting time. I was a sophmore at MIT and got a part-time job with SAO (Smithsonian Astrophysical Observatory, 14 syllables) at Harvard.

SAO was responsible for optical tracking of satellites — Moonwatch visual observations and Baker-Nunn tracking cameras. Caught off-guard by Sputnik, they hired undergraduates to compute predictions with Friden desk calculators. John Gaustad told me about MIT's IBM EDPM 704 and loaned me his Fortran II manual. My first program, Ephemeris 4, eliminated my job [Moore 1958].

Now a Programmer, I worked with George Veis to apply his method of least-squares fitting to determine orbital elements, station positions and ultimately the shape of Earth [Veis 1960]. Of course, this part-time job was at least 40 hours, and yes, my grades went to hell.

At MIT, John McCarthy taught an incredible course on LISP. That was my introduction to recursion, and to the marvelous variety of computer language. Wil Baden has noted that LISP is to Lambda Calculus as Forth is to Lukasewcleicz Postfix.

APL was also a topical language, with its weird right-left parsing. Although I admire and emulate its operators, I'm not persuaded they constitute an optimal set.

The programming environment in the 50s was more severe than today. My source code filled 2 trays with punch cards. They had to be carried about to be put through machines, mostly by me. Compile took 30 minutes (just like C) but limited computer time meant one

So I wrote this simple interpreter to read input cards and control the program. It also directed calculations. The five orbital elements each had an empirical equation to account for atmospheric drag and the non-spherical Earth. Thus I could compose different equations for the several satellites without re-compiling.

These equations summed terms such as P2 (polynomial of degree 2) and S (sine). 36-bit floating-point dominated calculation time so overhead was small. A data stack was unnecessary, and probably unknown to me.

The Forth interpreter began here with the words

```
WORD   NUMBER   INTERPRET   ABORT
```

They weren't spelled that way because they were statement numbers.

INTERPRET uses WORD to read words separated by spaces and NUMBER to convert a word to binary (in this case, floating-point). Such free-format input was unusual, but was more efficient (smaller and faster) and reliable. Fortran input was formatted into specific columns and typographic errors had caused numerous delays.

This interpreter used an IF . . . ELSE IF construct, coded in Fortran, finding a match on a single character. Error handling consisted of terminating the run. Then, as now, ABORT asked the user what to do. Since input cards were listed as they were read, you knew where the error was.

2.3.5 Stanford, SLAC, 1961

In 1961 I went to Stanford to study mathematics. Although Stanford was building its computer science department, I was interested in real computing. I was impressed that they could (dared ?) write their own Algol compiler. And I fatefully encountered the Burroughs B5500 computer.

I got another 'part-time' job at SLAC (Stanford Linear Accelerator Center, 12 syllables) writing code to optimize beam steering for the pending 2-mile electron accelerator. This was a natural application of my least-squares experience to phase-space. Hal Butler was in charge of our group and the program, TRANSPORT, was quite successful.

Another application of least-squares was the program CURVE, coded in Algol (1964). It is a general-purpose non-linear differential-corrections data-fitting program. Its statistical rigor provides insight into agreement between model and data.

The data format and model equations were interpreted and a push-down stack used to facilitate evaluation. CURVE was an impressive precursor to Forth. It introduced these words to provide the capability to fit models much more elaborate than simple equations:

```
+       -       *           NEGATE
IF      ELSE    THEN        <
DUP     DROP    SWAP
:       ;       VARIABLE    !          (
SIN     ATAN    EXP         LOG
```

Spelling was quite different:

```
  NEGATE  was  MINUS
    DROP       ;
    SWAP       .
       !       <
VARIABLE       DECLARE
       ;       END
```

The interpreter used IF ... ELSE IF to identify a 6-character input word called ATOM (from LISP). DUP DROP and SWAP are 5500 instructions; I'm surprised at the spelling change. The word : was taken from the Algol label format, flipped for left-right parsing (to prevent the interpreter encountering an undefined word):

```
  Algol  —  LABEL:
CURVE    —  : LABEL
```

In fact, : marked a position in the input string to be interpreted later. Interpretation was stopped by ; . A version of : was named DEFINE .

The store operator (!) appeared in connection with VARIABLE . But fetching (@) was automatic. Note the input had become complex enough to warrant comments. The sometime-criticised postfix conditional dates from here:

```
  Algol  —  IF expression THEN true ELSE false
CURVE    —  stack IF true ELSE false THEN
```

True is interpreted if stack is non-zero. THEN provides unique termination, the lack of which always confused me in Algol. Such expressions were interpreted: IF would scan ahead for ELSE or THEN.

The word < introduces the convention that relations leave a truth value on the stack, 1 for true and 0 for false. The transcendental functions are, of course, library calls.

2.3.6 Free-lance

I left Stanford in 1965 to become a free-lance programmer in the New York City area. This was not unusual, and I found work programming in Fortran, Algol, Jovial, PL/I and various assemblers. I literally carried my card deck about and recoded it as necessary.

Minicomputers were appearing, and with them terminals. The interpreter was ideal for teletype input, and soon included code to handle output. So we aquire the words

```
KEY     EXPECT  EMIT  CR
SPACE   SPACES  DIGIT TYPE
```

EXPECT is a loop calling KEY to read a keystroke. TYPE is a loop calling EMIT to display a character.

With the TTY came paper-tape and some of the most un-friendly software imaginable — hours of editing and punching and loading and assembling and printing and loading and testing and repeating. I remember a terrible Sunday in a Manhattan skyscraper when I couldn't find splicing tape (nothing else works) and swore that 'There must be a better way'.

I did considerable work for Bob Davis at Realtime Systems, Inc (RSI). I became a 5500 MCP guru to support his time-sharing service (remote input to a mainframe) and wrote a Fortran–Algol translator and file editing utilities. The translator taught me the value of spaces between words, not required by Fortran.

The interpreter still accepted words with the first 6 characters significant (the 5500 had 48-bit words). The words

```
LIST   EDIT   BEGIN   AGAIN   EXIT
```

appear, with BEGIN ... AGAIN spelled START ... REPEAT and used to bracket the editor commands

```
T   TYPE   I   INSERT   D   DELETE   F   FIND
```

later used in NRAO's editor. The word FIELD was used in the manner of Mohasco and

One of Forth's distinctive features comes from here. The rule is that Forth acknowledge each line of input by appending OK when interpretation is complete. This may be difficult, for when input is terminated by CR a blank must be echoed, and the CR included with OK. At RSI, OK was on the next line, but it still conveyed friendly reassurance over an intimidating communications line:

```
56 INSERT ALGOL IS VERY ADAPTABLE
OK
```

This postfix notation suggests a data stack, but it only had to be one deep.

2.3.7 Mohasco, 1968

In 1968 I transformed into a business programmer at Mohasco Industries, Inc in Amsterdam NY. They are a major home-furnishing company — carpets and furniture. I had worked with Geoff Leach at RSI and he persuaded me to follow him up-state. I had just married, and msterdam has a lovely small-town atmosphere to contrast with NYC.

I rewrote my code in COBOL and learned the truth about business software. Bob Rayco was in charge of Corporate data processing and assigned me two relevant projects:

He leased an IBM 1130 minicomputer with a 2250 graphic display. The object was to see if computer graphics helped design patterned carpets. The answer was 'not without color' and the 1130 went away.

Meanwhile I had the latest minicomputer environment: 16-bit CPU, 8K RAM, disk (my first), keyboard, printer, card reader/punch, Fortran compiler. The reader/punch provided disk backup. I ported my interpreter again (back to Fortran) and added a cross-assembler to generate code for the 2250.

The system was a great success. It could draw animated 3-D images when IBM could barely draw static 2-D. Since this was my first real-time graphics, I coded Spacewar, that first video game. I also converted my Algol chess program into Forth and was duely impressed how much simpler it became.

The file holding the interpreter was labeled FORTH, for 4th (next) generation software — but the operating system restricted file names to 5 characters.

This environment for programming the 2250 was far superior to the Fortran environment, so I extended the 2250 cross-assembler into an 1130 compiler. This introduced a flock of words

DO	LOOP	UNTIL				
BLOCK	LOAD	UPDATE	FLUSH			
BASE	CONTEXT	STATE	INTERPRET	DUMP		
CREATE	CODE	;CODE	CONSTANT	SMUDGE		
@	OVER	AND	OR	NOT	0=	0<

They were still differently spelled

LOOP	was	CONTINUE
UNTIL		END
BLOCK		GET
LOAD		READ
TYPE		SEND
INTERPRET		QUERY
CREATE		ENTER
CODE		the cent symbol

The only use I've ever found for the cent symbol. The loop index and limit were on the

BLOCK manages a number of buffers to minimize disk access. LOAD reads source from a 1024-byte block. 1024 was chosen as a nice modular amount of disk, and has proven a good choice. UPDATE allows a block to be marked and later rewritten to disk, when its buffer is needed (or by FLUSH). It implements virtual memory and is concealed in store (!) words.

BASE allows octal and hex numbers as well as decimal. CONTEXT was the first hint of vocabularies and served to isolate editor words. STATE distinguished compiling from interpreting. During compilation, the count and first 3 characters of a word were compiled for later nterpretation. Strangely, words could be terminated by a special character, an aberration quickly abandoned. The fetch operator (@) ppeared in many guises, since fetching from variables, arrays and disk had to be distinguished. DUMP became important for examining memory.

But most important, there was now a dictionary. Interpret code now had a name and searched a linked-list for a match. CREATE constructs the classic dictionary entry:

link to previous entry
count and 3 characters
code to be executed
parameters

The code field was an important innovation, since an indirect jump was the only overhead, once a word had been found. The value of the count in distinguishing words, I learned from the compiler writers of Stanford.

An important class of words appeared with CODE . Machine instructions followed in the parameter field. So any word within the capability of the computer could now be defined. ;CODE specifies the code to be executed for a new class of words, and introduced what are now called objects.

SMUDGE avoided recursion during the interpretation of a definition. Since the dictionary would be searched from newest to oldest definitions, recursion would normally occur.

Finally, the return stack appeared. Heretofor, definitions had not been nested, or used the data stack for their return address. Altogether a time of great innovation in the punctuated evolution of Forth.

The first paper on Forth, an internal Mohasco report, was written by Geoff and me [Moore 1970]. It would not be out of place today.

In 1970 Bob ordered a Univac 1108. An ambitious project to support a network of leased lines for an order-entry system. I had coded a report generator in Forth and was confident I could code order-entry. I ported Forth to the 5500 (standalone !) to add credibility. But corporate software was COBOL. The marvelous compromise was to install a Forth system on the 1108 that interfaced with COBOL modules to do transaction processing.

I vividly recall commuting to Schenectady that winter to borrow 1107 time 3rd shift. My TR4-A lacked floor and window so it became a nightly survival exercise. But the system was an incredible success. Even Univac was impressed with its efficiency (Les Sharp was project liason). The ultimate measure was response time, but I was determined to keep it maintainable (small and simple). Alas, an economic downturn led Management to cancel the 1108. I still think it was a bad call. I was the first to resign.

1108 Forth must have been coded in assembler. It buffered input and output messages and shared the CPU among tasks handling each line. Your classic operating system. But it also interpreted the input and PERFORMed the appropriate COBOL module. It maintained drum buffers and packed/unpacked records. The words

```
BUFFER   PREV       OLDEST
TASK     ACTIVATE   GET       RELEASE
```

date from here. BUFFER avoided a disk read when the desired block was known empty.

management. TASK defines a task at boot time and ACTIVATE starts it when needed. GET and RELEASE manage shared resources (drum, printer). PAUSE is how a task relinquishes control of the CPU. It is included in all I/O operations and is invisible to transaction code. It allows a simple round-robin scheduling algorithm that avoids lock-out.

After giving notice, I wrote an angry poem and a book that has never been published. It described how to develop Forth software and encouraged simplicity and innovation. It also described indirect-threaded code, but the first implementation was at NRAO.

I struggled with the concept of meta-language, language that talks about language. Forth could now interpret an assembler, that was assembling a compiler, that would compile the interpreter. Eventually I decided the terminology wasn't helpful, but the term Meta-compile for recompiling Forth is still used.

2.3.8 NRAO, 1971

George Conant offered me a position at NRAO (National Radio Astronomy Observatory, 15 syllables). I had known him at SAO and he liked Ephemeris 4. So we moved to Charlottesville VA and spent summers in Tucson AZ when the radio-telescope on Kitt Peak was available for maintainance.

The project was to program a Honeywell 316 minicomputer to control a new filter-bank for the 36' millimeter telescope. It had a 9-track tape and Tektronix storage-tube terminal. George gave me a free hand to develop the system, though he wasn't pleased with the result. NRAO was a Fortran shop and by now I was calling Forth a language. He was right in that organizations have to standardize on a single language. Other programmers now wanted their own languages.

Anyhow, I had coded Forth in assembler on the IBM 360/50 mainframe. Then I cross-compiled it onto the 316. Then I re-compiled it on the 316 (Although I had a terminal on the 360, response time was terrible). The application was easy once the system was available. There were two modes of observing, continuum and spectral-line. Spectral-line was the most fun, for I could display spectra as they were collected and fit line-shapes with least-squares [Moore 1973].

The system was well-received in Tucson, where Ned Conklin was in charge. It did advance the state-of-the-art in on-line data reduction. Astronomers used it to discover and map inter-stellar molecules just as that became hot research.

Bess Rather was hired to provide on-site support. She had first to learn the Forth system and then explain and document it, with minimal help from me. The next year I reprogrammed the DDP-116 to optimize telescope pointing. The next, Bess and I replaced the 116 and 316 with a DEC PDP-11.

The development that made all this possible was indirect-threaded code. It was a natural development from my work at Mohasco, though I later heard that DEC had used direct-threaded code in one of their compilers. Rather than re-interpret the text of a definition, compile the address of each dictionary entry. This improved efficiency for each reference required only 2 bytes and an address interpreter could sequence through a definition enormously faster. In fact, this interpreter was a 2-word macro on the 11:

```
: NEXT   IP )+ W MOV  W )+ ) JMP ;
```

Now Forth was complete. And I knew it. I could write code more quickly that was more efficient and reliable. Moreover, it was portable. I proceeded to recode the 116 pointing the 300' Green Bank telescope, and the HP mini that was inaugurating VLBI astronomy. George gave me a ModComp and I did Fourier transforms for interferometry and pulsar search (64K data). I even demonstrated that complex multiply on the 360 was 20% faster in Forth than assembler.

NRAO appreciated what I had wrought. They had an arrangement with a consulting firm

software patents were controversial and might involve the Supreme Court, NRAO declined to pursue the matter. Whereupon, rights reverted to me. I don't think deas should be patentable. Hindsight agrees that Forth's only chance lay in the public domain. Where it has flourished.

Threaded-code changed the structure words (such as DO LOOP IF THEN). They acquired an elegant implementation with addresses on the data stack during compilation.

Now each Forth had an assembler for its particular computer. It uses post-fix op-codes and composes addresses on the data stack, with Forth-like structure words for branching. The manufacturer's mnemonics are defined as word classes by ;CODE . Might take an afternoon to code. An example is the macro for NEXT above.

Unconventional arithmetic operators proved their value

M*	*/	/MOD	SQRT
SIN.COS	ATAN	EXP	LOG

M* is the usual hardware multiply of 2 16-bit numbers to a 32-bit product (arguments, of course, on the data stack). */ follows that with a divide to implement rational arithmetic. /MOD returns both quotient and remainder and is ideal for locating records within a file. SQRT produces a 16-bit result from a 32-bit argument. SIN.COS returns both sine and cosine as is useful for vector and complex arithmetic (FFT). ATAN is its inverse and has no quadrant ambiguity. EXP and LOG were base 2.

These functions used fixed-point arithmetic — 14 or 30 bits right of a binary point for trig, 10 for logs. This became a characteristic of Forth since it's simpler, faster and more accurate than floating-point. But hardware and software floating-point are easy to implement.

I'd like to applaud the invaluable work of Hart [Hart 1978] in tabulating function approximations with various accuracies. They have provided freedom from the limitations of existing libraries to those of us in the trenches.

The word DOES> appeared (spelled ;:). It defines a class of words (like ;CODE) by specifying the definition to be interpreted when the word is referenced. It was tricky to invent, but particularly useful for defining op-codes.

Nonetheless, I failed to persuade Charlottesville that Forth was suitable. I wasn't going to be allowed to program the VLA. Of any group, 25% like Forth and 25% hate it. Arguments can get violent and compromise is rare. So the friendlies joined forces and formed Forth, Inc. And that's another story.

2.3.9 Moral

The Forth story has the making of a morality play: Persistant young programmer struggles against indifference to discover Truth and save his suffering comrades. It gets better: Watch Forth. Inc go head to head with IBM over a French banking system.

I know Forth is the best language so far. I'm pleased at its success, especially in the ultra-conservative arena of Artificial Intelligence. I'm disturbed that people who should, don't appreciate how it embodies their own description of the ideal programming language.

But I'm still exploring without license. Forth has led to an architecture that promises a wonderful integration of software and silicon. And another new programming environment.

2.3.10 References

[ANS 1991] Draft Proposed ANS Forth, document number X3.215-199x, available from Global Engineering Documents, 2805 McGaw Ave., Irvine CA 92714.

[Brodie, 1981] Brodie, Leo, Starting FORTH, Englewood Cliffs NJ: Prentice-Hall, 1981, ISBN 0 13 842930 8.

[Hart, 1968] Hart, John F. et al, Computer Approximations. Malabar FL: Krieger, 1968;

[Martin, 1987] Martin, Thea, A Bibliography of Forth References, 3rd Ed, Rochester NY: Institute for Applied Forth Research, 1987, ISBN 0 914593 07 2.

[Moore, 1958] Moore, Charles H. and Lautman, Don A., Predictions for photographic tracking stations — APO Ephemeris 4, in SAO Special Report No. 11, Schilling G. F., Ed., Cambridge MA: Smithsonian Astrophysical Observatory, 1958 March.

[Moore, 1970] — and Leach, Geoffrey C., FORTH — A Language for Interactive Computing, Amsterdam NY: Mohasco Industries, Inc. (internal pub.) 1970.

[Moore, 1972] — and Rather, Elizabeth D., The FORTH program for spectral line observing on NRAO's 36 ft telescope, Astronomy & Astrophysics Supplement Series, Vol. 15, No. 3, 1974 June, Proceedings of the Symposium on the Collection and Analysis of Astrophysical Data, Charlottesville VA, 1972 Nov. 13–15.

[Moore, 1980] —, The evolution of FORTH, an unusual language, Byte, 5:8, 1980 August.

[Rather, 1993] Rather, Elizabeth D., Colburn, Donald R. and Moore, Charles H., The Evolution of Forth, in History of Programming Languages-II, Bergin T. J. and Gibson, R. G., Ed., New York NY: Addison-Wesley, 1996, ISBN 0-201-89502-1.

[Veis, 1960] Veis, George and Moore, C. H., SAO differential orbit improvement program, in Tracking Programs and Orbit Determination Seminar Proceedings, Pasadena CA: JPL, 1960 February 23–26.

3 Основы языка

3.1 Арифметические операции

Наконец, в этом месте вы можете задать вопрос: должны ли мы работать только с целыми числами ? Сам по себе Форт, определенный в соответствии со стандартами, не применяет арифметики с плавающей запятой.

Многие считают, что целочисленной арифметики вполне достаточно для разработки программ управления оборудованием, автоматики, компьютерных игр и обработки текстовой информации.

Но для решения технических и научных задач этого явно недостаточно. Хотя действия с целыми числами выполняются быстрее, а в других языках программирования использованием чисел с плавающей запятой иногда чрезмерно увлекаются, все же бывает, что целые числа при решении практических задач оказываются неприменимыми. Далее будут рассмотрены приемы вычисления с применением целых чисел, а также специальных аппаратных средств или их программной замены. И, кроме, того узнаем значительно больше о других арифметических операциях в языке Форт.

Мы уже не раз говорили о том, что компьютеры используются не только для вычислений, но и во многих других областях, однако, что бы они ни делали, они имеют дело с числами. Эта большая глава представляет собой что-то вроде попурри на тему чисел. Однако если в предыдущих и большинстве следующих глав для любой серьезной работы по программированию был нужен весь материал, то некоторые части этой главы потребуются вам только в том случае, если вы будете заниматься "перемалыванием"чисел[15], например обработкой результатов экспериментов или моделированием.

Как минимум, вы должны изучить это введение и разделы 3.1.1, 3.1.2, 3.1.3. Если вы будете работать с очень большими числами и дробями, вам нужно изучить разделы 3.1.4, 3.1.5, 3.1.6 и 3.1.7. А если вас интересует работа с числами с плавающей запятой, вы должны прочитать раздел 7.

К настоящему времени вы изучили четыре арифметические операции: сложение, вычитание, умножение и деление и соответствующие слова Форта +, -, * и /. Нам остается мало что добавить, кроме как упомянуть о потенциально возможной проблеме арифметического переполнения.

Предположим, вы складываете числа 36000 и 37000. Результат должен равняться 73000; т.е. быть больше, чем 16-разрядное число, которое может быть в стеке 16-битного Форта. Аналогичная ситуация возникает и в системах любой другой разрядности (32- и 64-) — только в этом случае эти числа на многие порядки больше, поэтому вы можете и не столкнуться с такой ситуацией, если не пишете соответствующих программ для работы с такими большими числами. Например, если вы пишете бухгалтерскую программу, выполняющую все вычисления в копейках или десятичных долях копеек[16], ваша программа будет нормально работать во всех мелких фирмах типа Вася Пупкин и Ко, но при попытке ее использования какой-нибудь транснациональной корпорацией с огромным бюджетом может вылезти ошибка переполнения.

Это и есть переполнение, которое приводит к неверному результату. Во всех 16-битных версиях Форта в результате сложения 36000 и 37000 будет получаться 7464, что, очевидно, неверно.

Переполнение может возникнуть также при операциях * и -. Некоторые языки могут обнаружить ошибку переполнения и либо прекратить исполнение программы, либо выдать сообщение об ошибке, Форт этого не сделает. Почему ? Потому что проверка на наличие ошибок занимает время ЭВМ, и обычно немалое, поэтому, чтобы не снижать скорости работы, заботу об избежании ошибок Форт предоставляет программисту. Вторая причина — желание, чтобы программа работала, даже если произошла ошибка.

Нас интересует проверка возникновения переполнения и его предотвращение. Как быть, если вы хотите работать с числами больше 32767 или меньшими -32767, т.е. с числами больше 65535, если не учитывать знака. Имеется два способа решения этой проблемы. Один из них — это использование так называемых чисел двойной длины (или, как иногда говорят, двойной точности или просто двойных чисел), которые представляются 32 разрядами. Другой — в использовании чисел с плавающей запятой (при этом вы получите большую потерю точности, именно поэтому использование таких чисел в финансовых расчетах воспрещается).

3.1.1 Операторы для работы с небольшими числами

В стандартах описываются несколько специальных операций для работы с небольшими числами, типа Это 1+, 1-, 2+ и 2-, 2*, 2/ и т.п. Эти операции выполняют то, что и следует ожидать, работая с числом, находящимся на вершине стека, т.е. производят умножение, деление, сложение и

3.1.2 Некоторые проблемы операции деления

3.1.3 Операции с величинами и знаками чисел

3.1.4 Почему используются целые числа ?

3.1.5 Масштабирование чисел

3.1.6 Числа двойной длины

3.1.7 Совместное применение чисел одинарной и двойной длины

3.1.8 trash

Арифметический стек — основное поле для выполнения арифметических действий и хранения промежуточных результатов вычислений. Надо только помнить, что знак операции (точнее слово, обозначающее операцию) пишется ПОСЛЕ того, как аргументы в стеке уже размещены.

Текст

```
13  3  -
```

помещает в стек число 10, так как слово "-" (минус) извлекает из стека два числа, сперва вычитаемое, потом уменьшаемое, и помещает в стек их разность:

```
...  a  b   -->        ...  a-b
```

С другими операциями все обстоит аналогично

```
+            ...  a  b   -->        ...  a+b
*            ...  a  b   -->        ...  a*b
ABS          ...  a      -->        ...  |a|
NEGATE       ...  a      -->        ...  -a
/            ...  a  b   -->        ...  целая часть
MOD          ...  a  b   -->        ...  остаток
/MOD         ...  a  b   -->        ...  остаток    целая часть
```

В трех последних словах имеются в виду остаток и целая часть частного от деления a на b. Так, при делении 26 на 7 имеем:

```
26  7         /    -->  3
26  7        MOD   -->  5
26  7        /MOD  -->  5  3
```

Имеются специальные слова для действий с 1 и 2 (они выполняются немного быстрее)

```
1+                ...  a      -->  ...  a+1
```

4 Принципы работы форт-системы

4.1 Создание слов-определителей

4.2 Память Форта, словари и контекстные словари

4.2.1 Кодофайл

Кодофайлом будем называть участок памяти, в котором располагаются набор слов Форт-системы и новые скомпилированные слова, написанные пользователем. Здесь же размещаются константы и переменные. Память занимается в направлении возрастания адресов, при этом свободная память находится в конце словаря. Иногда два соседних байта называют ячейкой. Тогда адресом ячейки считается адрес младшего байта (то есть байта с меньшим адресом). Мы будем называть ВЕРШИНОЙ СЛОВАРЯ первый свободный байт памяти. От программиста требуется особая осторожность при работе с памятью: изменения, записанные в ячейку с ошибочным адресом, могут нарушить функционирование Форт-системы так, что потребуется ее перезагрузка!

Вот некоторые стандартные слова для работы с кодофайлом:

```
HERE                ...       -->   ...   addr
```

На стек кладется адрес вершины кодофайла. С помощью этого слова можно определить, какой объем памяти требуется для любого фрагмента Вашей программы - надо сравнить значения HERE до компиляции и после нее.

```
ALLOT               ...   n    -->   ...
```

Резервируются n байтов свободной памяти: адрес вершины кодофайла увеличивается на n (а при n<0 уменьшается).

```
,                         ...   n    -->   ...
```

Занятие двух байтов в кодофайле и запись туда n.

```
!                   ...   a          addr    -->   ...
```

Это слово (восклицательный знак, читается "запомнить") служит для записи значения по данному адресу.

```
@                   ...   addr       -->   ...   a
```

Слово @ (читается "взять") кладет в стек значение, хранящееся по адресу, лежащему на стеке. Сам адрес из стека при этом убирается.

```
+!                  ...   a    addr     -->   ...
```

К числу, расположенному по адресу addr, прибавляется значение a. Результат сохра-

4.3 Интерпретация, компиляция и исполнение

5 Создание компиляторов и форт-систем

5.1 Создание собственных компиляторов

Если вам приспичило написать собственный компилятор языка высокого или сверхвысокого уровня, но при этом хочется это дело максимально упростить — используйте модификацию языка Форт.

Форт — самый простой по синтаксису и реализации из ЯВУ, написать компилятор для Форта сможет любой чайник, особенно если будет использовать эту книгу как справочник.

Именно простотой Форт и одновременно его крайне низкими требованиями к компьютеру объясняется то, что созданы десятки компиляторов Форта для всех существующих компьютеров, даже для самых хилых микроконтроллеров с ОЗУ всего несколько Кб.

В этой книге описана методика, которая упростит написание своего Форта на порядок — теперь вам даже не нужно писать свой интерпретатор и стандартный набор слов, достаточно взять готовую чужую форт-систему и добавить к ней простейший компилятор в байт-код, а интерпретатор байт-кода написать на любом другом языке, ассемблере или даже сделать собственный стековый процессор, используя перепрограммируемые логические матрицы фирм Altera или Xilinx.

5.2 Целевая компиляция

Значение термина *компиляция* в Форте отличается от его традиционного применения. Обычно под компиляцией подразумевается обработка текстовых файлов с текстами программ и генерация машинного или байт-кода. В Форте компиляцией называется добавление байтов в словарь в процессе определения новых слов и структур данных. Компилятором в Форте являются слова C, и , , которые добавляют на вершину словаря соответственно байт или cell (машинное слово). Остальные компилирующие слова Форта работают через эти слова.

Штатный компилятор ФОРТа способен только добавлять код к коду ФОРТ-системы. Более интересная для разработчиков техника[17] — целевая компиляция. Она используется в случаях когда необходимо генерировать машинный код для другой архитектуры (кросс-компиляция) или для той же самой, но не включать в код ФОРТ-систему, то есть создавать объектный код с нуля. При этом объем генерируемого кода можно значительно сократить, не включая в него никакой дополнительной информации (словарей).

5.2.1 Минимальный ЦК

Механизм работы целевого компилятора (ЦК) полностью аналогичен обычному, отличие заключается в том что формируемый код компилируется не в память ФОРТ-системы, а в специально выделенный буфер целевого компилятора. Текущую позицию компиляции в этом буфере указывает value-перемененная THERE, аналогичная HERE в ФОРТ-системе.

```
0x10000 CONSTANT Msz \ размер буфера ЦК (максимальный размер целевого кода)
CREATE M Msz ALLOT   \ буфер ЦК
0 VALUE THERE        \ указатель ЦК
```

Определяем основные слова ЦК. При этом нужно учитывать порядок байт в машинном слове. Например, в архитектуре 80x86 первым в памяти находится младший байт, у MCS-51 и AVR старший. В примере показаны слова для 80x86 и os, для MCS-51 нужно будет дополнительно выполнять переупорядочение байтов. Для упрощения понимания кода мы

не будем переопределять стандартные слова C, и , , а будем использовать для слов ЦК нестандартные имена.

```
: b, ( byte -- ) THERE M + C! THERE 1 + TO THERE ; \ скомпилировать байт
: w, ( word -- ) THERE M + W! THERE 2 + TO THERE ; \ скомпилировать слово
: d, ( dword -- )  THERE M +  ! THERE 4 + TO THERE ; \ скомпилировать
  \ 32-битное слово
```

Для сохранения наработанного кода в двоичный файл используем слово save"(также неплохо было бы определить слово hex" для сохранения в формате Intel HEX для загрузки в симуляторы и целевые системы)

```
: save" \ program.bin"
[CHAR] " WORD COUNT W/O CREATE-FILE DROP \ открыть файл
M THERE ROT WRITE-FILE DROP               \ записать код в файл
BYE                                       \ выход с автоматическим
 \ закрытием файла
;
```

Приведенные три секции кода являются минимальным целевым компилятором, на основе которого пишутся компиляторы и форт-ассемблеры для любых платформ (с учетом порядка байт в машинном слове см. выше). Использование техники целевой компиляции позволяет очень быстро написать ассемблер с очень мощной поддержкой макросов[18], но имеет значительный недостаток, который может оказаться критичным при выборе между готовым ассемблером и форт-ассемблером: при написании форт-ассемблера чаще всего используют постфиксный форт-подобный синтаксис, так как его очень просто реализовать, но при этом возникает проблема использования наработок с традиционным синтаксисом.

5.2.2 Целевой компилятор для os

Простейшим примером использования ЦК является ЦК для os. Его простота определяется тем, что виртуальная машина os[19] имеет всего два вида команд с простым форматом: опкод имеет фиксированный размер в один байт, команды безоперандные или с одним операндом фиксированного размера байт или 16-битное слово. В других архитектурах формат команды намного сложнее из-за использования битовых полей, в которых кодируются операнды и режимы адресации. Основную сложность в ЦК для os создают слова для формирования словарной структуры и особенно слова, компилирующие управляющие структуры. Если у вас возникают вопросы, напишите мне чтобы я более подробно описал структуру кода управляющих структур и механизм работы слов, компилирующих этот код.

Эта версия ЦК написана для конкретной форт-системы, которая является регистрочувствительной, то есть отличает слова `dup` и `DUP`. Я пробовал использовать другие форт-системы, но они не отличают регистр слов, поэтому этот вариант ЦК для них не годится.

Хорошим упражненением (которое вам все равно придется сделать, если строго следовать идеологии os) будет написание ЦК для вашей любимой форт-системы.

Есть подводный камень — в примерах программ используются макросы, которые на самом деле являются обычными форт-словами, поэтому они могут отказаться работать с вашей форт-системой, и их тоже придется адаптировать.

Самый неприятный вариант, если вы решите написать целевой компилятор не на Форте, в этом случае макросы работать вообще не будут, так как их работа целиком обеспечивается форт-системой.

---

[18] для определения макросов доступны все функции форт-системы

```
\ стековый ассемблер (целевой компилятор) для SP-FORTH 4 [http://spf.sf.net
```

В этом варианте ЦК не используются словари, а этот код только убирает лишние предупреждения о переопределении слов.

```
VOCABULARY TC       ONLY FORTH ALSO TC       TC DEFINITIONS
```

По молчанию ЦК компилирует байт-код с созданием словарной структуры (включением метаинформации с именами всех слов ваших программ). Если вам по какой-то причине нужно выключить генерацию словаря, после загрузки ЦК нужно выполнить код `0 TO ?VOC`.

```
1 VALUE ?VOC            \ флаг включения в байт-код словарной структуры
                         \ (по умолчанию влючен, для минимизации кода
                       \ после S" tc.4th" INCLUDED добавить 0 TO ?VOC )
```

Мы компилируем код для 16-битной виртуальной машины, поэтому нужно переопределить слова `CELL` и `CELLS`:

```
2 CONSTANT CELL          \ разрядность ВМ 8*cell бит
: CELLS CELL * ;
```

Этот код описан в примере минимального ЦК выше.

```
0x10000 CONSTANT Msz     \ максимальный размер программы, байт
CREATE M Msz ALLOT       \ буфер компилятора
0 VALUE THERE            \ указатель компилятора
```

В начале байт-кода находятся несколько полей, адреса которых нам будут нужны в словах ЦК (компиляцию этих полей см. последние строки ЦК)

По адресу 0000 в байт-коде находится команда перехода на точку входа `jmp _entry`:

```
1 CONSTANT _entry                  \ стартовый адрес
```

Затем идет 2-байтное поле с адресом последнего форт-слова в словаре, который представляет собой односвязный список словарных статей (см. далее). Это поле используется для перехода на начало этого списка.

```
_entry CELL + CONSTANT _latest     \ адрес последнего слова в словаре
```

Последнее поле — значение указателя компиляции на момент сохранения байт-кода (аналог значения, возвращаемого словом HERE для обычного Форта).

```
_latest CELL + CONSTANT _here      \ адрес начала кучи (размер байт-кода +1)
```

Для чтения/записи буфера ЦК, компиляции байта, 16-битного слова и счетной байтной строки определяются специальные слова, которые вы можете также использовать в своих макросах и расширениях ЦК.

Записать байт, слово, прочитать слово:

```
: b!      ( byte addr -- ) M + C! ;
: w!      ( cell addr -- ) M + W! ;
```

Компилирующие слова

```
: b,      THERE b! THERE 1+ TO THERE ;  \ скомпилировать байт
: w,      THERE w! THERE 2+ TO THERE ;  \ -"- 16-битное слово
```

Скомпилировать счетную строку

```
: s, ( addr n -- )                      \ -"- счетную 8-битную строку
    DUP b, 0 DO
     DUP I + C@ b,
    LOOP DROP
;
```

Для завершения работы ЦК нужно выполнить слово `save" filename"`, указав имя файла, в который записать байт-код:

```
: save" \ program"
\ сохранить скомпилированный байт-код в файл и выйти из форт-системы
    THERE _here w!                              \ сохранение размера кода
     [CHAR] " WORD COUNT W/O CREATE-FILE DROP     \ открытие файла
    M THERE ROT WRITE-FILE DROP BYE            \ запись байт-кода и выход
;
```

В os используется два типа команд (Полное описание системы команд os см. раздел 9.4):

```
: 0op     CREATE C, DOES> C@ b, ;        \ безоперандные команды
: 1op    CREATE C, DOES> C@ b, w, ;    \ команды с одним операндом (16 бит)
```

команды управления выполнением программы

```
0x00 0op nop
0x01 1op jmp       0x02 1op ?jmp      0x03 1op call      0x04 0op ret
0x05 1op lit   : # lit ;
0x06 0op exec
0x07 0op bye
0x0F 0op cell
```

Самое сложное в ЦК – компиляция управляющих структур. Присылайте вопросы, если нужно более подробное описание.

Циклы со счетчиком

```
0x08 0op do    0x09 0op loop      0x0A 0op i     0x0B 0op j     0x0C 0op k
```

```
: entry!  THERE _entry w! ;   \ модифицировать стартовый код (точку входа)
```

Слова для компиляции словарной статьи.

Каждое слово в словаре состоит из нескольких полей:

LFA
Поле связи
CELL
адрес следующего элемента списка словаря, 0 признак конца списка.

При компиляции LFA текущее значение указателя сохраняется в поле `_latest`, а его

```
: LFA,     _latest w@ THERE _latest w! w, ;
```

NFA
Поле имени
счетная строка
содержит имя слова.

Имя получается из входного потока, затем позиция во входном потоке восстанавливается.

```
: NFA,     >IN @ BL WORD COUNT s, >IN ! ;
```

AFA
Поле атрибутов слова
байт
Хранит флаги слова типа SMUDGE и IMMEDIATE, в ЦК не устанавливаются.

```
: AFA,     0 b, ;
```

Слово компилирует заголовок словарной статьи (LFA, NFA, AFA), если установлен флаг ?VOC.

```
: HEADER,                    \ скомпилировать заголовок слова если ?VOC != 0
     ?VOC IF
          LFA, NFA, AFA,
     THEN
;
```

Начало определения слова

Компилируется заголовок словарной статьи, текущая позиция компиляции (поле кода, CFA) запоминается в виде специального слова в словаре форт-системы. При выполнении этого слова будет скомпилирована команда call на запомненный CFA.

```
: {       HEADER, entry! CREATE THERE , DOES> @ call ; \ начало {} определени
```

Конец определения слова

Просто компилируется команда ret.

```
: }       ret ;                                  \ конец {} определения
```

Определение константы 0x1234 const A

Компилируется слово A, которое при выполнении положит на стек указанную константу командой lit.

```
: const   ( n -- )  { ( n ) # } ;              \ определить константу
```

Определение переменной 0x5678 var B

Компилируется слово B и следом (PFA, поле параметров) начальное значение переменной. Слово будет класть на стек адрес поля параметров.

Скомпилировать буфер размером n байт `0x120 buffer C`

Работа аналогична `var`, но поле параметров – компилируется 0 байт указанное число раз.

```
: buffer  CREATE THERE , 0 ?DO 0 b, LOOP DOES> @ # ;
```

Управляющая структура if else then

if комплирует команду условного перехода на фиктивный адрес -1, запоминая адрес параметра команды на стеке.

```
: if      -1 ?jmp THERE CELL - ;
```

then берет со стека адрес, и записывает по этому адресу текущее значение указателя компиляции. адрес указывает на параметр команды jmp или ?jmp (метод backpatching при однопроходной компиляции)

```
: then    THERE SWAP w! ;

: else    -1 jmp THERE CELL - SWAP then ;
```

в итоге из

```
a (flag ) if b else c then d
```

должен скомпилироваться код

```
        a
:if     ?jmp else
        b
        jmp then
:else
        c
:then
        d
```

Циклы с условием

Реализация аналогична if else then

```
: begin   THERE ;
: again   jmp ;
: until   ?jmp ;
: while   if ;
: repeat  SWAP jmp THERE SWAP w! ;
```

Остальные команды описываются элементарно через слова 0op и 1op.

Если вы будете добавлять в систему команд BM свои команды, дописывайте их на опкоды F0..FF, можно также использовать многобайтные команды, включая дополнительные наборы опкодов командой-префиксом (в качестве примера см. графическое расширение).

стек

память

Для использования слова @ из форт-системы (оно переопределяется новым словом @ компилирующим соответствующубю команду) сначала определяется новое алиасное слово ~@.

```
                                        : ~@ @ ;
0x20 0op c@         0x21 0op c!         0x22 0op @          0x23 0op !
```

арифметика

```
: ~+ + ;             : ~- - ;            : ~* * ;
0x30 0op +          0x31 0op -          0x32 0op *          0x33 0op /
0x34 0op */          0x35 0op rnd
```

логика и битовые операции

```
0x40 0op =          0x41 0op <>         0x42 0op >          0x43 0op <

0x48 0op not        0x49 0op or         0x4A 0op and        0x4B 0op xor
0x4C 0op lshift      0x4D 0op rshift
```

консольный ввод/вывод

```
0x70 0op key         0x71 0op emit
0x72 0op ?key        0x73 0op ?emit
```

интерфейс к отладчику

```
0x80 0op s.          0x81 0op h.          0x82 0op dump
0x83 0op .           0x84 0op u.
```

строки

```
0x90 0op (")         0x91 0op count       0x92 0op print
```

Для компиляции `" строка"` и `." вывод строки"` используются макросы:

```
: "         (") [CHAR] " WORD COUNT s, ;
: ."        " print ;
```

Быстрый поиск в словаре

```
0xA0 0op find
```

Так как у нас опкод команды занимает всего один байт, общее число команд не может превышать 256 команд, что явно недостаточно даже для Форта во встраиваемой системе. Чтобы расширить диапазон команд, используются специальные однобайтные команды-модификаторы (префиксы), после выполнения которых в виртуальной машине запускается дополнительный цикл выборки/выполнения команды, при этом опкод этой команды декодируется по другой (расширенной) таблице соответствия опкод–команда. Используя один или несколько команд-префиксов, мы получаем неограниченное количество команд ВМ.

В качестве иллюстрации — расширенный набор команд графического драйвера, встроенного в виртуальную машину:

Для определения gr/ команд используем слово, аналогичное 0op, но компилирующее перед опкодом префикс включения графического расширения системы команд (gr/ не использует команды формата 1op):

```
: gr CREATE C, DOES> 0xF0 b, C@ b, ;

0x00 gr gr/on       0x01 gr gr/off
0x02 gr gr/Xsz      0x03 gr gr/Ysz
0x04 gr gr/set      0x05 gr gr/clr      0x06 gr gr/get
0x07 gr gr/pointer  0x08 gr gr/?pointer
```

Соответственно команда gr/on скомпилирует два байта: `F0 00`.

Этот код скомпилирует заголок байт-кода (стартовый `jmp` и поля `_here` и `_latest`)

```
-1 jmp    \ jmp _entry
0 w,      \ _latest
0 w,      \ _here
```

5.3 Как написать свой (кросс-)ассемблер

© Brad Rodriguez

5.3.1 Введение

В предыдущем выпуске журнала я описал как "загрузить" себя на новый процессор используя простой отладочный монитор. Но как вы сможете писать код для этого нового процессора, если вы не можете найти ассемблер или он вас не устраивает ? Напишите свой !

ФОРТ — идеальный язык для этого. Для TSM320 например я написал кросс-ассемблер всего за два часа, включая длинный обеденный перерыв. Обычно это занимает около двух дней, но для одного из процессоров (Zilog Super8) потребовалось пять дней. Но когда у вас больше времен чем денег, это не важно.

В части 1 этой статьи я опишу основные принципы ФОРТ-ассемблеров — структурированные, однопроходные, постфиксные. Многое из этого применимо для любого процессора, и это концепции почти любого ФОРТ-ассемблера.

В части 2 я покажу как писать ассемблер для специфичного процессора Motorola 6809. Этот ассемблер прост но не тривиален, занимает 15 страниц исходного кода. Кроме всего прочего, этот пример покажет как реализовывать инструкции с множественными режимами адресации. Изучив этот пример, вы сможете понять как использовать особенности вашего процессора.

5.3.2 Зачем использовать ФОРТ ?

Мне кажется что ФОРТ – простейший язык для написания ассемблера. Прежде всего ФОРТ имеет текстовый интерпретатор для разбора текстовых строк и выполнения соответствующих команд. Преобразование текстовых строк в байты кода – как раз и есть задача ассемблера. Операнды и режимы адресации также реализуются через ФОРТ-слова.

ФОРТ также включает определяющие слова, которые позволяют легко описать большие наборы слов с общим действием. Эта возможность очень полезна при определении мнемоник ассемблера.

Так как при работе ассемблера доступны все слова ФОРТ а, они могут быть использованы при использовании ассемблера не только для вычисления адресов и операндов, и и для выполнения более сложных действий.

В общем так как ассемблер полностью реализуется через слова ФОРТ а, определения

5.3.3 Простейший пример: ассемблирование NOP

Для понимания того как ФОРТ транслирует мнемоники в машинный код, рассмотрим простейший случай: инструкцию NOP (0x12 для 6809, 0x90 для 80x86).

Обычный ассемблер при нахождении строки "NOP" должен добавить соответствующий байт в выходной файл и увеличить указатель на 1. Операнды и комментарии игнорируются. Я также пока буду игнорировать метки.

В ФОРТ е выходным файлом обычно является словарь в памяти или при кросс-ассемблировании блок памяти: образ памяти целевой системы. Так, определим слово NOP соответствующим образом: "скомпилировать опкод NOP и увеличить указатель".

```
: NOP,    0x12 C, ;
```

Обратите внимание что в этом примере используется возможность SP-FORTH воспринимать числа с префиксом 0x (шестнадцатеричные числа) независимо от текущей системы счисления, заданной в переменной BASE. Если ваша ФОРТ-система не имеет такой возможности, для удобства в начале кода ассемблера можно использовать слово HEX. Также следует обратить внимание на то, что при использовании целевого компилятора для кросс-разработки слово C, должно быть переопределено так, чтобы оно компилировало байт в образ памяти целевой системы.

Для ассемблерных опкодов часто задаются имена ФОРТ-слов имеющие в конце символ ",", как это сделано выше. Это делается из-за того что многие ФОРТ-слова (например AND, XOR, OR) конфликтуют с мнемониками ассемблера. Простейшее решение – слегка изменить мнемоники (символ "," в ФОРТ е обозначает что что-то компилируется (добавляется) в словарь).

5.3.4 Класс наследуемых опкодов

Большинство процессором имеют много инструкций подобных NOP, которые не требуют операндов. Все они могут быть определены в ФОРТ е через двоеточие, но это значительно увеличивает объем исходного кода. Намного более эффективным является использование механизма *определяющих слов* для того, чтобы задать для всех таких слов общее поведение. В терминах ООП это значит создание экземпляров единственног класса.Это далется с использованием слов CREATE и DOES>. В приведенном примере параметр (разный для каждого созданного слова) обычный опкод, который должен быть скомпилирован для каждой инструкции.

```
: cmd0ops                  \ определение имени класса команды без опреандов
    CREATE                  \ этот код выполняется при создании слов
       C, ( byte -- )       \ сохранить байт для нового определяющего слова
    DOES>  ( -- addr )      \ этот код определяющего слова (общее действие)
       C@ ( addr -- byte)   \ получить сохраненный при создании слова опкод
        C, ( byte -- )        \ скомпилировать опкод
;
```

Примеры использования определяющего слова cmd0ops:

```
0x12 cmd0ops NOP,
0x3A cmd0ops ABX,
0x3D cmd0ops MUL,
```

Обратие особое внимание на то, что этот пример дан для обычного форт-ассемблера, а не кросс-ассемблера, то есть генерируемый код компилируется в основной словарь форт-

несколько финтов ушами — например нужно использовать *разные* варианты слова `C,` в блоках `CREATE` и `DOES>` слова `cmd0ops`, которые должны компилировать опкод соответственно в словарь форт-системы и в область памяти целевой системы.

Эта техника дает некоторую экономию памяти почти без потери скорости, но реальное удобство будет видно при определении сложных инструкций, которые требуют параметры и модификаторы режимов адресации.

5.3.5 Обработка операндов инструкций

Большинство команд ассемблера требуют один или более операндов. Для этих команд ассемблер должен быть способен разбирать текст из входного потока и интерпретировать его как операнды. Для постфиксного ФОРТ-ассемблера используется более простой способ, использующий готовый механизм интерпретации ФОРТ-системы.

Итак ФОРТ будет использоваться для разбора операндов. Числа обрабатываются как обычно (в любой системе счисления), для EQU выражений могут использоваться обучные константы ФОРТ CONSTANT. Так как операнды определяют формат ассемблируемой команды, они должны быть обработаны до команды ассемблера. Результаты разбора операндов обычно оставляются на стеке данных и используются командами ассемблера для определения формата команды и используемого оп-кода. Для ФОРТ-ассемблера используется уникальный[20] постфиксный формат команд: операнды, за которыми следует команда ассемблера.

Для примера рассмотрим инструкцию ORCC процессора 6809, которая использует простой числовой параметр:

```
: ORCC,   0x1A C,  C, ; \ пример использования: 0x34 ORCC,
```

Выполнение этой команды состоит из двух этапов:

1. компилируется опкод инструкции ORCC 0x1A;
2. компилируется параметр инструкции 0x34, взятый со словаря.

Эта команда предполагает что на стеке уже лежит числовой параметр, полученный в результате разбора операндов, которые в нашем случае имеют вид простого hex числа 0x34 в исходном коде.

Достоинство такого ассемблера – доступна вся мощь ФОРТ а, которая может быть использована для формирования операндов, например:

```
ONSTANT CY-FLAG    \ используем константы ФОРТа для задания операндов
0x02 CONSTANT OV-FLAG
0x04 CONSTANT Z-FLAG

CY-FLAG Z-FLAG + ORCC,  \ команда ORCC проверит флаги CY и Z
```

Из примера видно что расширение разбора операндов для определяющих слов ассемблера достаточн просто.

5.3.6 Обработка режимов адресации

Для современных процессоров для одной команды используется несколько форматов и различные оп-коды в зависимости от режима адресации. ФОРТ-ассемблеры решают эту проблему несколькими способами в зависимости от типа процессора. Все эти способы

используют методологию ФОРТ а: операторы определения режима адресации являются ФОРТ-словами. Когда эти слова исполняются, они изменяют формат ассемблируемой инструкции. 1 помещение дополнительных параметров на стек.

Это наиболее удобно если всегда должен указываться режим адресации. Слова режима адресации оставляют на стеке некоторые константы, которые обрабатываются словами ассемблера. Иногда эти значения могут быть "магическим числом", добавляемым к опкоду для изменения режима адресации команды. Кодга это неприменимо, используется выбор формата команды в блоке CASE. В общем случае скомпилированные инструкции могут иметь разнуб длину в зависимости от режима адресации.

2 установка флагов или значений в фиксированных переменных.

Это наиболее удобно если режим адресации опционален. Не зная что был указан режим адресации, вы не можете знать что значение на стеке "магическое число" или просто значение операнда. Решение этой пролемы: режим адресации указывать помещая соответствующую ему константу в определенную переменную (часто называемую MODE). После выполнения каждой команды ассемблера эта переменная инийиализируется значением по умолчанию. Если используется команда ассемблера, у которой может быть несколько режимов адресации, то она определяет содержимое этой переменной.

3 модификация значений параметров непосредственно на стеке.

Это иногда возможно для реализации слов указывающих режим адресации, которые работают модифицируя значения операндов. Этот метод используется редко.

Все эти три метода я использовал с некоторыми расширениями для реализации ассемблера 6809.

Для большинства процессоров имена регистров задаются обычными константами Форта, и оставляют на стеке свои значения. Для некоторых процессоров также удобно иметь имена регистров, определяющие режим "регистровой адресации". Это легко сделать, определяя имена регистров как определяющие слова, которые создают слова, устанавливающие режим адресации (а стеке или в переменной `MODE`).

Некоторые процессоры позволяют использовать несколько режимов адресации в одной инструкции. Если чисто режимов адресации фиксировано, они могут оставляться на стеке. Если чисто переменное, необходимо знать сколько их был указано, и нужно использовать несколько переменных MODE_n. Для процессора Super8 я должен был отслеживать не только сколько режимов адресации указано, но и сколько операндов. Я сделал это сохраняя позицию стека отдельно для каждого режима адресации.

Рассмотрим инструкцию 6809 ADD. Для упрощения игнорируем индексированные режимы адресации, и реализуем только три режима: непосредственный (immediate), прямой (direct) и расширенный (extended):

```
           исходный код     ассемблируется как
-----------------------------------------------
immediate  <число> # ADD,   8B nn
direct     <адрес> <> ADD,  9B nn
extended   <адрес> ADD,     BB aa aa
```

Так как режим extended не имеет оператора режима адресации, режим адресации оказывается уже определенным. Слова ФОРТ а # и <> устанавливают режим адресации. Рассмотрим систему команд 6809. Если опкод immediate является основным значением, то опкод в direct режиме равен базовому +0x10, в indexed режиме +0x20 и в extended режиме +0x30. Это справедливо для почти всех инструкций, использующих эти режимы адресации. Исключение составляют те опкод, для которых опкоды в режиме direct имяют форму 0x0?.

Такие особенности системы команд нужно использовать. Это общее правило при написании ассемблеров: найдите или сделайте сами таблицу опкодов, и найдите закономерности – особенно те, которые применимы для режимов адресации или других модификаторов

В нашем случае нужно использовать следующие значения переменной MODE:

```
VARIABLE MODE
: #     0x00 MODE ! ;
: <>    0x10 MODE ! ;
: RESET 0x30 MODE ! ; \ значение по умолчанию
```

Значение по умолчанию 0x30 (extended режим) устанавливается словом RESET. Это слово будет использоваться после того как будет ассемблирована каждая инструкция.

Теперь может быть написана команда ассемблера ADD,. Давайте посмотрим ее старую версию и напишем определяющие слова для создания команд ассемблера.

```
: GENERAL-OP    ( <базовый опкод> -- )
     CREATE C,
     DOES>              \ ( <операнд> -- )
     C@                 \ получить базовый опкод
     MODE @ +           \ добавляем "магическое число"
     C,                 \ ассемблируем опкод
     MODE @ CASE
       0x00 OF C, ENDOF \ операнд байт
       0x10 OF C, ENDOF \ операнд байт
       0x30 OF ,  ENDOF \ операнд слово (2 байта)
     ENDCASE
     RESET \ выбор режима адресации по умолчанию
;

8B GENERAL-OP ADD,
```

Каждый экземпляр GENERAL-OP будет иметь различный базовый опкод. Когда ADD, выполняется, он будет получать этот базовый опкод, добавлять к нему значение MODE, и комплировать полученный байт. Дале он будет брать операнд, находящийся на стеке, и компилировать его в зависимости от выбранного режима адресации как байт или слово. В конце MODE будет сброшен в значение по умолчанию. Отметим что уже определен весь код, который нужно использовать для определения инструкций того же семейства что и ADD:

```
0x89 GENERAL-OP ADC,
0x84 GENERAL-OP AND,
0x85 GENERAL-OP BIT,
```

Сохранение памяти при использовании определяющих слов по сравнению с обучными словами определенными через двоеточие очевидно. Все команды ассемблера, приведенные выше, используют единственный блок кода после DOES>, вызов которого из этих слов занимает всего несколько байт.

На самом деле это не мой код реального ассемблера 6809 – существуют дополнительные особые случаи, которые необходимо реализовать. Но он показывает что сохраняя что достаточно сделать чтобы сохранить информацию о режиме и как свободно использовать конструкцию CASE чтобы реализовать наиболее простые наборы команд.

5.3.7 Реализация структур управления

Структурное программирование наделало очень много шума, и было написано множество макропакетов "структурного ассемблирования" для распространенных ассембле-

структурный ассемблерный код, что объяснятся той причиной что *на ФОРТ е проще реализовать структурный ассемблер чем метки.*

Структуры, обычно включаемые в ФОРТ-ассемблеры, аналогичны структурам высокоуровневого ФОРТ а. Для их отличия к ним добавляется запятая, как это было сделано для ассемблерных команд.

5.3.8 BEGIN, UNTIL,

Эта самая простая для понимания ассемблерная структура. Ассемблерный код должен защиклиться до метки BEGIN до того как будет удовлетворено некоторое условие. Синтаксис для ФОРТ-ассемблера:

```
BEGIN, <код> <условие> UNTIL,
```

Код условия предположительно определен как операнд или режим адресации для интсрукции перехода.

Очевидно, что `UNTIL` должен комплировать условный переход. Условия перехода должны быть инвертированы так, что, если условие удовлетворяется, то переход игнорируется, в отличие от перехода, когда условие ложно. Ассемблерный код для обычного ассемблера должен иметь вид:

```
xxx: ...
     ...
     ...
     JR  ~cc,xxx     \ ~cc то же что и NOT(cc)
```

Существует два свойства, помогающие реализовать эту структуру. Слово HERE[21] возвращает текущий указатель компиляции. Числа могут храниться на стеке не влияя на реботу ФОРТ а, и доставаться по необходимости.

Итак BEGIN, должен запоминать позицию указателя компиляции помещая ее в стек, а UNTIL, будет ассемблировать условный переход по запомненному на стеке адресу.

```
: BEGIN, ( -- a )         HERE ;
: UNTIL, ( a cc -- )      NOTCC JR, ;  \ флаг перехода инвертируется
```

Как видно по стековой нотации BEGIN, оставляет на стеке текущий адрес, и UNTIL, использует запомненный адрес и код условия для компиляции условного перехода. Слово NOTCC предварительно инвертирует код условия. Слово JR, использует адрес перехода и (инвертированный) код условия для формирования соответствущего кода условного перехода 6809.

Такая реализация позволяет использовать вложенные условные циклы:

```
BEGIN,
     ...
     BEGIN,
...
     cc UNTIL,
     ...
cc UNTIL,
```

Вложенный UNTIL, ссылается на вложенный BEGIN„ формируая код цикла внутри кода охватывающего цикла.

5.3.9 BEGIN, AGAIN,

ФОРТ также поддерживает конструкцию бесконечного цикла BEGIN AGAIN. Определение этой конструкции для ассемблера аналогично, за тем исключением что код условия не используется, и компилируется безуслоный переход на начало цикла.

5.3.10 DO, LOOP,

Многие процессоры предоставляют некоторые инструкции цикла. Так как их нет у 6809, расмотрим Zilog Super8. Он имеет инструкцию DJNZ (декремент и переход если не ноль), которая может использовать любой из 16 регистров как счетчик цикла. Для процессора 80x86 эта инструкция работает только с регистром (E)CX. Цикл на структурном ассемблере будет иметь вид:

```
DO, ... r LOOP,
```

где r регистр используемый как счетчик цикла.

```
: DO,   ( -- a)     HERE ;
: LOOP, ( a r -- )  DJNZ, ;
```

В некоторых ФОРТ-ассемблерах DO, также ассемблирует код инициализации счетчика цикла, но это уменьшает гибкость.

5.3.11 IF, THEN,

Эта конструкция – простейшая конструкция, использующая ссылки вперед. Если условие истинно, она должна выполняться, иначе управление передается на первую инструкцию после THEN,.

Ассемблерный код для ФОРТ-синтаксиса

```
cc IF,  ... ... ...  THEN,
```

будет иметь вид

```
     JP   ~cc,xxx
     ...
     ...
     ...
xxx:
```

Обратите внимание что код условия должен быть инвертирован аналогично UNTIL,.

В однопроходном ассемблере необходимый переход вперед не может быть скомпилирован сразу, так как адрес перехода еще неизвестен. Эта проблема решается путем ассемблирования пустого перехода типа JMP 0 и помещения на стек адреса операнда команды JMP. Позже слово THEN, сможет взять этот адрес и пропатчить скомплирированный JMP, соответствующм образом модифицировав его операнд.

```
: IF, (cc -- a)
     NOT 0 SWAP JP,       \ условный переход
         HERE 2 -           \ помещаем на стек HERE-<размер операнда>
  ;

  : THEN, (a --)
      HERE SWAP !         \ запоминаем по адресу операнда JP текущий
       HERE
```

IF, инвертирует код условия и компилирует условный переход на нулевой адрес, кладя адрес операнда на стек. После компиляции условного перехода HERE указывает на байт после него, поэтом нужно вычесть размер операнда условного перехода. THEN, затем модифицирует команду условного перехода, изменяя адрес перехода на реальный.

Приведенный вариант THEN, — самый простой вариант для процессоров, использующих условный переход по абсолютному 16-битному адресу. У многих процессоров есть только команда условного перехода по относительному адресу ±127 байт от адреса команды перехода. В этом случае размер операнда равен одному байту, а THEN, должен предварительно вычесть из HERE адрес на стеке.

5.3.12 IF, ELSE, THEN,

Улучшенный вариант IF, THEN, дополняется блоком кода, выполняемом если условие не выполняется:

```
cc IF,  ... ...  ELSE,  ... ...  THEN,
```

Ассемблерный код этой конструкции имеет вид:

```
    JP   ~cc,xxx
         ...            ; код "если"
      ...
           JP   yyy
  xxx: ...             ; код "иначе"
       ...
 yyy:
```

ELSE, должен модифицировать действия IF, и THEN, следующим образом:

1 переход вперед в IF, долен быть модифицирован чтобы выполнялся переход на начало блока ELSE, ("xxx");

2 адрес, положенный на стек THEN, должен быть записан в операнд безусловного перехода в конце блока IF, ("JP yyy").

ELSE, также должен компилировать безусловный переход.

```
: ELSE (a -- a)
    0 T JP,         \ безусловный переход
       HERE 2 -       \ поместить в стек адрес операнда перехода для THEN,
      SWAP            \ получить адрес перехода IF,
           HERE SWAP !     \ заменить его на текущий адрес
   ;
```

Условие перехода T обозначает TRUE, то есть безусловный переход. При определении ELSE, может быть использован код IF, и THEN, если таже определено условие F FALSE:

```
: ELSE (a -- a)     F IF, SWAP THEN, ;
```

SWAP адресов в стеке инвертирует последовательность модификации инструкций перехода так, что THEN, модифицирует переход внутри кода ELSE,:

```
IF,(1)   ...   IF,(2)  THEN,(1)   ...   THEN,(2)
               \_______________/
```

5.3.13 BEGIN, WHILE, REPEAT,

Наиболее сложной ассемблернй структурой является цикл ПОКА, в котором условие проверяется в начале цикла, а не в конце.

```
BEGIN, <код> cc WHILE, <код цикла> REPEAT,
```

На практике между BEGIN, и WHILE, может быть вставлен любой код, а не только задание условия.

WHILE, должен скомпилировать условный переход по инверсному условию на код за REPEAT,. Если код условия cc удовлетворяется, этот переход должен игнорироваться и выполняться код цикла.

REPEAT, должен компилировать безусловный переход на BEGIN,.

```
BEGIN,(1)  ...  cc IF,(2)  ...  AGAIN,(1)  THEN,(2)
```

Это может быть сделано с использованием существующих слов:

```
: WHILE, (a cc -- a a)   IF, SWAP ;
: REPEAT, (a a -- )      AGAIN, THEN, ;
```

5.3.14 Заголовок ФОРТ-определения

В большинстве приложений машинный код, созданный ФОРТ-ассемблером, помещается в словарь с помощью CODE <имя>, который создает словарную статью с именем <имя> и связывает ее со словарным списком.

Слово CODE получает имя слова из входного потока, создает определение в словаре с этим именем, и настраивает указатель словаря на начало поля кода этого имени.

Стандартный ФОРТ использует слово CODE не только для выделения начала ассемблерного определения в словаре, но и дополнительной инициализации ассемблера (установке переменных типа MODE).

5.3.15 Кросс-компиляция

До сих пор мы предполагали что в словарь компилируется машинный код системы, на которой и исполняется.

Для кросс-компиляции обычно выполняется компиляция в отдельную область памяти. Эта область может иметь или не иметь словарной структуры, но она отделена от словаря хост-машины, и скомплированный код не может быть исполнен.

Чаще всего для этого используется набор слов для доступа к целевому пространству памяти по аналогии с обычным доступом к памяти в Форте. Для этого могут быть использованы обычные слова с префиксом "T":

TDP	указатель компиляции DP целевой системы
THERE	аналог HERE для целевой системы (если VALUE-переменная, TDP не нужен)
TC,	
TC@	
TC!	
T@	
T!	

Иногда вместо использования префиска "T" эти слова определяются в отдельном словаре целевого компилятора. Словарная структура ФОРТ а позволяет иметь несколько

5.3.16 Компиляция на диск

Целевая компиляция также может выполняться сразу на диск (в файл) без использования буфера памяти целевой системы, но из-за использования прямого побайтного доступа к файлу это может значительно замедлить работу компилятора. Особенно это заметно если не используется кеширование дисковых операций.

5.3.17 Безопасная компиляция

Некоторые реализации ФОРТ а используют безопасную компиляцию, которая пытяется отловить ошибки типа несбалансированных структур управления типа

```
IF, ... cc UNTIL,
```

В этом примере UNTIL, некорректно использует адрес, помещенный на стек IF,.

Обычный метод проверки сбалансированности структур управления – помещение на стек данных или на отдельный стек констант, уникальных для каждого управляющего слова, которые проверяются другими словами:

```
IF, помещает 1;
THEN, проверяет на 1;
ELSE, проверяет на 1 и оставляет 1;
BEGIN, оставляет 2;
UNTIL, проверяет 2;
AGAIN, проверяет 2;
WHILE, проверяет 2 и оставляет 3;
REPEAT, оставляет 3
DO, оставляет 4;
LOOP, проверяет 4.
```

Стоимость такой безопасности – увеличение сложности манипуляций со стеком в таких словах как ELSE, и WHILE,. ТАкже программист может захотеть последовательность в которой управляющие структуры are resolved вручную манипулируя стеком. Безопасность делает это более сложным.

5.3.18 Метки

Даже в эру структурного программирования некоторые программисты используют метки в ассемблерном коде.

Принципиальная проблема с именоваными метками в ФОРТ-ассемблере – так как метки это ФОРТ-слова, они должны быть скомпилированы в словарь во время незавершенной компиляции другого слова в машинном коде, например:

```
CODE TEST  ...  <машиннный код>  ...
HERE CONSTANT LABEL1
...  <машинный код>  ...
LABEL1 NZ JP,
```

Как видно из примера определение метки LABEL1 должно создавать словарную статью внутри серидины CODE. Вот несколько решений 1 метки определяются только вне машинного кода. 2 используются некие предопределенные переменные для временного хранения меток. 3 для меток используется отдельное словарное пространство, например как это сделано по схеме TRANSIENT

$$tj3$$

. 4 для машинного кода используется отдельное пространство словаря. Это наиболее часто используемый метод при мета-компиляции. Большинство мета-компиляторов ФОРТ а

5.3.19 Табличный ассемблер

Большинство ФОРТ-ассемблеров могут отрабатывать режимы адресации и инструкции используя структуры типа CASE. Их можно назвать процедурными ассемблерами.

Некоторые процессоры типа 68000 имеют настолько сложные наборы инструкций и режимов адресации, что становится сложным строить деревья выбора при ассемблеировании их команд. В таких случаях более подходящим является использование таблиц, что уменьшает используемую память и процессорное время.

Я не буду описывать такие процессоры – табличные ассемблеры писать намного сложнее[22].

5.3.20 Префиксные ассемблеры

Иногда использование префиксных ассемблеров необходимо. Например мне однажды пришлось переводить многие килобайты ассемблерного кода для Super8 для обычного Zilogовского ассемблера для ФОРТ-ассемблера. Существует трюк позволяющий имитировать префиксный ассемблер при использовании методов, описанных в этой статье.

В основном этот трюк можно определить как отложенное выполнение ассемблирования после того как были обработаны операнды. Это делается следующим словом ассемблера.

Так каждое слово ассемблера модифицируется так что оно

1 сохраняет свой код или адрес кода, компилирующего команду;

2 выполняет компилирующий команду код для предыдущего слова ассемблера.

```
... JP  operand    ADD  operands  ...
```

JP сохраняет указатель на свой код в какой-либо переменной. Далее обрабатываются операнды. ADD берет информацию, запомненную JP и выполняет действие JP, которое использует операнды на стеке. Когда код JP завершается, ADD сохраняет свой адрес, и процесс продолжается.

Особо должны отрабатываться первые и последние команды в ассемблерном блоке. Если используются переменные режима, то их сохранение и восстановление станоится кошмаром.

5.3.21 Вывод

Я рассмотрел наиболее простые методики используемые в ФОРТ-ассемблерах. Изучение готового ассемблера может дать вам информацию как писать собственный ассемблер, но лучший способ изучения – действие.

5.4 Постфиксный форт-ассемблер для MCS-51

5.5 Реализация собственной форт-системы

6 Реализация ООП

6.1 Детальное описание mini-OOF

Объектно-ориентированные (ОО) системы с поздним связыванием обычно используют метод VTABLE: первая переменная[23] в каждом объекте[24] является указателем на описание

[22] пример такого ассемблера приведен в следующем разделе

[23] поле

класса в виде таблицы указателей на функции[25]. Эта VTABLE может также содержать некоторую другую информацию, например статические переменные.

Объект	→	vtable-ptr	→	число переменных
		переменная		число методов
		переменная		xt метода
		...		...

Во-первых, определим методы:

: method (m v – m+1 v) CREATE OVER , SWAP CELL+ SWAP DOES> (... o — ....) @ OVER @ + @ EXECUTE ;

При декларации метода со стека берется общее число методов и переменных класса. method создает один метод и увеличивает на 1 число методов. Для исполнения метода берется объект, из него получается указатель на VTABLE, добавляется смещение, и выполняется xt[26] метода, который находится по полученному адресу. Каждый метод определяет объект, для которого он вызывается, по адресу на стеке — указателю this. Метод должен работать с объектом самостоятельно и удалять this со стека перед выходом из метода.

Теперь мы должны определить переменные класса

: var (m v size – m v+1) CREATE OVER , + DOES> (o – addr) @ + ;

так ка, как и method, слово создается с текущим смещением. Переменные могут иметь различные размеры[27], так что все что мы делаем — берем размер и добавляем его к смещению. Если ваша машина имеет требования по выравниванию[28], перед переменной добавляем требуемые выравнивающие байты словами ALIGNED или FALIGNED, также корректирующими смещение переменной. Именно по этой причине смещение находится на вершине стека.

Для начала определения объекта нам нужно задать начальное состояние[29] и выполнить некоторые дополнительные действия:

CREATE object 1 CELLS , 2 CELLS ,

: class (class – class methods vers) DUP 2@ ;

Для наследования при декларации нового наследуемого[30] класса должна быть скопирована VTABLE родительского объекта. Это копирование дает все методы родительского объекта, которые далее могут быть переопределены.

: end-class (class methods vars –) CREATE HERE >R (1) (vars) , (methods) , DUP , (methods) 2 CELLS ?DO ['] NOOP , 1 CELLS +LOOP (2) CELL+ DUP CELL+ R> (here) ROT @ 2 CELLS /STRING MOVE ;

(1) создает VTABLE, инициализированную NOOPами;

(2) реализует механизм наследования — из родительской VTABLE копируются xt методов.

У нас все еще нет способа определения новых методов, поэтому определим слово

: defines (xt class –) ' >BODY @ + ! ;

Для выделения памяти под новый объект нам также нужно слово

: new (class – o) HERE OVER @ ALLOT SWAP OVER ! ;

Многда дочерние классы требуют доступа к методам родительского класса. Существует два способа сделать это в этом варианте реализации OOF: вы можете

1. использовать именованные слова;

---

[25] методы класса

[26] execution token, обычно адрес поля кода CFA слова

[27] целые, числа с плавающей точкой одинарной и двойной точности, символы и т.п.

[28] например на границу машинного слова

[29] пустой объект

2. использовать их xt из VTABLE родительского класса.

: :: (class "name ' >BODY @ + @ COMPILE, ;

Пример Ничто не может быть лучше чем хороший пример, так что вот он. Для начала определим текстовый объект (налее называемый button), который хранит текст и позицию:

object class cell var text vell var len cell var x cell var y method init method draw end-class button

Теперь реализуем два метода:

:NONAME (o –) >R R@ x @ R@ y @ AT-XY R@ text @ R> len @ TYPE ; button defines draw

:NONAME (addr u o –) >R 0 R@ x ! 0 R@ y! R@ len ! R> text ! ; button defines init

В качестве примера наследования определим класс bold-button без новых переменных и методов

button class end-class bold-button

: bold 27 EMIT ."[1m"; : normal 27 EMIT ."[0m";

:NONAME bold [button :: draw] normal ; bold-button defines draw

И наконец некоторый код, демонстрирующий создание объектов и применение методов:

button new CONSTANT foo s"thin foo"foo init foo draw

bold-button new CONSTANT bar s"fat bar"bar init bar draw

7 Плавающая точка

8 Сетевые протоколы

9 Операционная система os

Этот раздел книги является документацией к проекту [os]. Документация исключена из проекта, чтобы не делать двойную работу, дублируя одни и те же темы в документации проекта и книге. Кроме того, как раз [os] является хорошим подробным примером использования Форта и практически не описанного в литературе метода *целевой компиляции*.

9.1 О системе

В последние несколько лет производительность компьютеров настолько возрасла, что появился интерес к реализациям языков программирования, использующих интерпретацию. Самый распространенный подход при реализации таких систем программирования — компиляция программ с одного или нескольких ЯВУ в более или менее низкоуровневый промежуточный код, который затем интерпретируется при работе программ в run-time.

Вид такого промежуточного кода может отличаться системой команд и возможностями *виртуальной машины* (ВМ), их кодированием, но чаще всего такой код называется *байт-кодом*. Соответственно *интерпретатор байт-кода* называют также *виртуальной машиной* (ВМ), *движком* или *(интерпретирующим) ядром*.

Для ускорения работы интерпретатора часто[31] используются методы частичной компиляции байт-кода в *нативный*[32] машинный код реального компьютера.

У меня еще с институтских времен была мечта написать свою операционную систему на Форте, но обдумав требования к такой системе я остановился на варианте интерпретатора байт-кода, который может работать на всех платформах и ОС, которые я использую

[31] особенно в тяжелых коммерческих языковых системах типа Java или .NET

(Windows, Linux и DOS на i386 ПК, наладонник Palm IIIxe, мобильный телефон с виртуальной машиной Java). Кроме того, я планирую заняться встраиваемыми системами на 8-битных микроконтроллерах, и есть вариант использовать чисто интерпретируемый вариант Форта для части кода, который не требует высокой скорости работы.

В течение нескольких месяцев экспериментов и была создана [os]: ОС для встраиваемых систем, построенная на основе виртуальной стековой машины, система команд которой оптимизирована под язык программирования Форт.

Текущий вариант os искуственно сделан 16-битным, чтобы программы для нее могли работать на самодельных специализированных маломощных управляющих компьютерах. Кроме того, это одновременно является экспериментом — даст ли использование языка Форт ту компактность программ, о которой часто говорят фортеры.

Использование интерпретирующего движка имеет несколько важных достоинств:

- крайне высокая переносимость между разными компьютерами и ОС — ядро системы (интерпретатор байт-кода) переписывается под любую *хост-систему* всего за несколько часов;
- ядро может быть легко написано на любом языке программирования или ассемблере;
- ядро элементарно встраивается внутрь любых других программ, что позволяет добавить в программы *программирование пользователем* — в систему команд ВМ добавляются команды для взаимодействия с программной ("ручки"), а пользователь может писать свои скрипты и расширения и компилировать их в байт-код;
- еще один вариант (пока нереализованный) — вместо двухступенчатого процесса компиляции пользовательских скриптов в байт-код и их последующего выполнения внутри основной программы на встроенном интерпретаторе использовать форт-систему в байт-коде; в этом случае пользовательские скрипты на Форте будут загружаться непосредственно в текстовом виде без предварительной компиляции в байт-код;
- программы, скомпилированные в байт-код, выполняются без перекомпиляции (правильно написанные программы) или с минимальной настроечной модификацией исходного кода под конкретную хост-систему или требования пользователя;
- неограниченные возможности отладки — отладочный код добавляется в исходный код ядра или в целевой компилятор;
- за счет чистой интерпретации возможно использовать любые типы многозадачности, защиты данных, контроля работы программ в процессе выполнения.

Вообще использование интерпретации одинакового для всех хост-систем байт-кода позволяет применять методики, не работающие для традиционной компиляции в машинный код

- самомодификация программ;
- использование метаданных об исходном коде программ, также скомпилированных в байт-код[33];
- разнообразные виды многозадачности с защитой задач друг от друга, в том числе и на хост-системах, не поддерживающих ее аппаратно, типа 8086 и 8-битных микроконтроллеров;

- ресурсы хост-системы (файловая система, сетевые функции, память, устройства ввода/вывода и т.д.) либо вообще недоступны для программ в байт-коде, либо доступ обеспечивается с помощью специализированных команд виртуальной машины, поэтому программы выполняются в изолированной среде и не имеют возможности повлиять на работу других программ и системы, а интерпретатор при этом может отслеживать и ограничивать использование всех ресурсов в процессе выполнения программ;

- полная независимость формата байт-кода от платформы и его изолированность позволяет выполнять программы на гетерогенных распределенных сетях, состоящих из разнотипных компьютеров, соединенных между собой любым способом — можно повысить надежность программно-аппаратных систем, повысить эффективность их использования и масштабируемость за счет распределения нагрузки.

Основное средство разработки — ассемблер стековой машины (*целевой компилятор*, ЦК), написанный в видее расширения форт-системы SP-FORTH[34]. Система команд стековой машины естественно заточена под Форт.

К сожалению пока (?) не нашлось компиляторщика, способного и желающего написать для os компиляцию в байт-код с других языков программирования, особенно C^{++}. С другой строны, если будет написан компилятор C^{++}, в сообщество хакеров-разработчиков os набежится ламерье и любители высокоуровневых языковых примочек типа ООП, динамических структур данных и сборщиков мусора, и получится еще одна Java.

При необходимости вы можете написать собственную версию целевого компилятора, который будет вместо байт-кода генерировать настоящий машинный код целевой платформы, при условии что ваши программы не работают напрямую с байт-кодом и словарем.

Это условие не выполняется для самомодифицирующихся программ, в том числе и для интерактивной форт-системы, которая в дальнейшем возможно кем-то будет написана. Если написать такую форт-систему, она совместит в себе интерфейс командной строки, монитор, средство on-board разработки, загрузчик конфигурационных файлов (если движок их поддерживает) и скриптов и т.д..

В процессе развития системы она естественно будет постепенно распухать, но я надеюсь все же сохранить ее простоту за счет многослойности.

Самый лучший способ освоения os — полностью написать с нуля ее клон, сохранив совместимость.

Для движка желательно использовать один или несколько языков программирования и библиотек, которые вам больше всего нравятся или лучше подходят для решаемых задач. В российских ВУЗах особенно популярен в этом плане Pascal, ну и конечно найдется масса народа, желающего написать движок на ассемблере.

Еще один кандидат на переписывание – целевой компилятор, так как он заточен на регистро-зависимый SP-FORTH, и не будет работать на большинстве форт-систем, не различающих слова `dup` и `DUP`. Кроме того, в новом стандарте появился раздел по целевой компиляции, а ЦК в дистрибутиве os ему совершенно не соответствует.

Собственно говоря os — это специфическая методика программирования на Форте с использованием целевой компиляции в байт-код и интерпретации, и по большей части написана как иллюстрация этой методики. То, что эта методика оказывается более эффективной, чем традиционное программирование на Форте с раскруткой системы от уровня машинного кода, просто побочный эффект. Жаль только, что мне не попалось хорошего описания целевой компиляции типа статьи Brad Rodriguez лет этак 10 назад.

Будущее покажет, но если вдруг методика os окажется популярной, нам опять придется решать основную Вавилонскую проблему языка Форт — расползание на десятки частично

несовместимых диалектов Форта, появление программ, зависимых от особенностей конкретных форт-систем, и самое главное проблемы понимания кодов чужих программ из-за языкового перенасыщения (см. **??**).

9.2 Многослойная структура

Система имеет многослойную архитектуру:

9.2.1 Аппаратная платформа и ОС

Текущий вариант дистрибутива работает только в гостевом режиме поверх DOS на ПК с процессорами 80x86.

DOS дает ощущение встраиваемой системы — низкая разрядность[35]), прямой доступ к железу[36], отсутствие многих библиотек[37].

os разрабатывалась как гостевая ОС, работающая поверх хост-ОС, но никто не запрещает взять в руки ассемблер и/или компилятор C^{++} и линкер, способный генерировать бинарный файл с машинным кодом, и написать движок os, который будет загружаться как stand-alone ОС и работать напрямую с железом. Правда при этом вам придется вложить в это чудо несколько человеко-лет кодинга, чтобы написать драйвера, сетевые протоколы, библиотеки программ и т.д..

Если вы хотите работать в привычной среде, но при этом экспериментировать с системным программированием, есть более простой вариант — напишите ВМ, которая будет эмулировать низкоуровневый доступ к устройствам, и пишите свою ОС на Форте.

9.2.2 Виртуальная машина (интерпретатор байт-кода)

Написан на C^{++} с использованием левого нелицензионного компилятора Borland C++ 3.1. Распространяется в исходных текстах в качестве примера для написания вашего собственного интерпретатора байт-кода на любом языке или ассемблере, очень желательно без нарушений лицензионности.

9.2.3 Целевой компилятор

Принципы целевой компиляции и подробное описание 16-битного ЦК для os подробно описано в разделе 5.2.

---

[35] 16 бит

[36] видео, контроллеры периферии

9.2.4 Библиотеки

9.2.5 Пользовательские расширения и прикладные программы

9.3 Архитектура ВМ

9.3.1 Память

9.3.2 Стек возвратов

9.3.3 Стек данных

9.3.4 Регистры

9.3.5 Стек циклов со счетчиком

9.4 Система команд

os использует подпрограммный шитый код, причем используются не машинные команды, выполняемые аппаратно, а команды виртуальной машины (байт-код). Эта технология также используется некоторыми форт-системами, и называется tokenized threaded code (TTC) или switched threaded code.

Набор команд и архитектура ВМ оптимизирована для языка Форт, в частности все команды выполняют действия, аналогичные небольшому набору слов из CORE WORDSET стандарта языка.

9.4.1 Базовые команды

Управление выполеннием программы

Память

Стек

Целочисленная арифметика

9.4.2 Графический драйвер monoLCD

9.5 Библиотеки

9.5.1 Библиотека макросов

9.5.2 2D полноэкранная графика

9.6 Примеры программ

9.6.1 empty

9.6.2 blinker

9.6.3 liner

9.6.4 grdemo

9.6.5 pautov

9.6.6 cmdline

10 Заключение

Работа над книгой ведется постоянно, поэтому та версия, которую вы сейчас читаете, может оказаться не самой свежей.

Последнюю версию этой книги и другие материалы по Форту и разработке железа вы можете скачать с [akps].

10.1 Констактные адреса

Пожалуйста, направляйте ваши замечания и предложения:

Dmitry Ponyatov <forth@km.ru>

FidoNet: 2:5057/18.29 или в конференцию SU.FORTH

Лучше всего если вы присоединитель к работе над книгой и другими материалами сайта — это могут быть как ваши письма с описанием различных методик, исходными текстами программ и т.д., так и полноценная on-line работа над материалами используя доступ с использованием CVS (см. http://www.cvs.ru).

10.2 Необходимый софт

- Любая Форт-система для вашей ОС. Примеры приведены для форт-систем, соответствующих стандартам FORTH-83 и ANS FORTH '94. Для Win я использую SP-FORTH. См. подробный список форт-систем. В книгу планируется включить разделы, посвященные использованию нескольких самых распространенных форт-систем (в смысле использования специфичных для каждой системы возможностей).

- Реальное железо или эмуляторы не-80x86 платформ (Palm, ZX Spectrum, средства разработки для встраиваемых систем на микроконтроллерах или не-80x86 процессорах[38], симулятор VHDL).

- Компилятор C(++), ассемблер, компилятор или языковая среда для вашего любимого языка – для написания интерпретирующего движка FVM и т.п. Для DOS-движка я использую Borland C++ 3.1. Из скриптовых языков очень рекомендую Python.

- Клиент для CVS-сервера, если хотите участвовать в реадктировании книги или другого раздела сайта. Для win качайте cvsnt/WinCVS или cvs-sfx .

- Редактор, способный работать с текстами в кодировках DOS cp866, win-1251 и koi8-r — я пользуюсь средой DOS Navigator и редактором vim.

10.3 Необходмые навыки

- Для работы над текстом книги — владение издательской системой LaTeX, желательно под любым UNIXом (Linux, *BSD).

- Для работы над кодом — владение C(++), стандартным FORTH и (или) диалектом FVM.

Список литературы

[os]

16-битная OS для встраиваемых систем
система построена на базе интерпретатора байт-кода
архитектура виртуальной машины и система команд взяты из Форта
http://akps.ssau.ru/forth/os/ http://akps.ssau.ru/forth/os/

[rufig]

сайт группы RU FIG (Russian FORTH Interests Group)
http://www.forth.org.ru http://www.forth.org.ru

[akps]

сайт Лаборатории аэрокосмического приборостроения (АКПС) СГАУ
http://akps.ssau.ru http://akps.ssau.ru

[orange]

М. Келли, Н.Спайс
Язык программирования ФОРТ
Перевод с английского Е. В. Куркова, Ю. А. Семенова
Москва, "Радио и связь", 1993
http://akps.ssau.ru/forth/orange/ http://akps.ssau.ru/forth/orange/

[green]

С.Н. Баранов, Н.Р. Ноздрунов
Язык Форт и его реализации
Ленинград, "Машиностроение" Ленинградское отделение, 1988
http://akps.ssau.ru/forth/green/ http://akps.ssau.ru/forth/green/

[starting]

Лео Броуди
Начальный курс программирования на языке Форт
Перевод с английского В.А. Кондратенко
Под редакцией Б.А. Кацева, В.А. Кириллина
Предисловие И.В. Романовского
Москва, "Финансы и статистика", 1990

[thinking]

Лео Броуди
Способ мышления — Форт. Язык и философия для решения задач
Перевод с английского С.Н. Дмитренко
Москва, 1993
http://akps.ssau.ru/forth/thinking/ http://akps.ssau.ru/forth/thinking/

[ans]

American National Standard for Information Systems
Programming Languages
Forth
Computer and Business Equipment Manufacturers Association
Approved: March 24, 1994
American National Standards Institute, Inc.

оригинал http://akps.ssau.ru/forth/ans/en/DPANS.HTM http://akps.ssau.ru/forth/ans/en/DPANS.HTM
перевод http://akps.ssau.ru/forth/ans/ru/DPANS.HTM http://akps.ssau.ru/forth/ans/ru/DPANS.HTM