Оглавление

Об этом сборнике

В этот сборник (блогбук) я пишу отдельные статьи и переводы, сортированные только по общей тематике, и добавляю их, когда у меняв очередной раз зачешется LaTeX.

Это сборник черновых материалов, которые мне лень компоновать в отдельные книги, и которые пишутся просто по желанию “чтобы было”. Заказчиков на подготовку учебных материалов подобного типа нет, бо̀льшая часть только на этапе освоения мной самим, просто хочется иметь некое слабоупорядоченное хранилище наработок, на которое можно дать кому-то ссылку.

Часть I

ML & функциональное программирование

Глава 1

Основы Standard ML © Michael P. Fourman

http://homepages.inf.ed.ac.uk/mfourman/teaching/mlCourse/notes/L01.pdf

1.1 Введение

ML обозначает “MetaLanguage”: МетаЯзык. У Robin Milner была идея создания языка программирования, специально адаптированного для написания приложений для обработки логических формул и доказательств. Этот язык должен быть **метаязыком** для манипуляции объектами, представляющими формулы на логическом **объектном языке**.

Первый *ML* был метаязыком вспомогательного пакета автоматических доказательств Edinburgh LCF. Оказалось что метаязык Милнера, с некоторыми дополнениями и уточнениями, стал инновационным и универсальным языком программирования общего назначения. Standard ML (SML) является наиболее близким потомком оригинала, другой — CAML, Haskell является более дальним родственником. В этой статье мы представляем язык SML, и рассмотрим, как он может быть использован для вычисления некоторых интересных результатов с очень небольшим усилием по программированию.

Для начала, вы считаете, что программа представляет собой последовательность команд, которые будут выполняться компьютером. Это неверно! Предоставление последовательности инструкций является лишь одним из способов программирования компьютера. Точнее сказать, что **программа — это текст *спецификации вычисления***. Степень, в которой этот текст можно рассматривать как последовательность инструкций, изменяется в разных языках программирования. В этих заметках мы будем писать программы на языке ML, который не является столь явно императивным, как такие языки, как Си и Паскаль, в описании мер, необходимых для выполнения требуемого вычисления. Во многих отношениях ML **проще** чем Паскаль и Си. Тем не менее, вам может потребоваться некоторое время, чтобы оценить это.

ML в первую очередь функциональный язык: большинство программ на ML лучше всего рассматривать как спецификацию **значения**, которое мы хотим вычислить, без явного описания примитивных шагов, необходимых для достижения этой цели. В частности, мы не будем описывать, и вообще беспокоиться о способе, каким значения, хранимые где-то в памяти, изменяются по мере выполнения программы. Это позволит нам сосредоточиться на **организации** данных и вычислений, не втягиваясь в детали внутренней работы самого вычислителя.

В этом программирование на ML коренным образом отличается от тех приемов, которыми вы привыкли пользоваться в привычном императивном языке. **Попытки транслировать ваши программисткие привычки на ML неплодотворны — сопротивляйтесь этому искушению!**

Мы начнем этот раздел с краткого введения в небольшой фрагмент на ML. Затем мы используем этот фрагмент, чтобы исследовать некоторые функции, которые будут полезны в дальнейшем. Наконец, мы сделаем обзор некоторых важных аспектов ML.

Крайне важно пробовать эти примеры на компьютере, когда вы читаете этот текст.[1]

---

[1] Пользовательский ввод завершается точкой с запятой ";". В большинстве систем, ";" должна завершаться нажатием [Enter]/[Return], чтобы сообщить системе, что надо послать строку в ML. Эти примеры тестировались на системе Abstract Hardware Limited's Poly/ML. В **Poly/ML** запрос ввода символ $>$ или, если ввод неполон — #.

Примечение переводчика Для целей обучения очень удобно использовать онлайн среды, не требующие установки программ, и доступные в большинстве браузеров на любых мобильных устройствах. В качестве рекомендуемых online реализаций Standrard ML можно привести следующие:

CloudML https://cloudml.blechschmidt.saarland/
описан в блогпосте B. Blechschmidt как онлайн-интерпретатор диалекта Moscow ML

TutorialsPoint SML/NJ http://www.tutorialspoint.com/execute_smlnj_online.php

Moscow ML (**offline**) http://mosml.org/ реализация Standart ML

- Сергей Романенко, Келдышевский институт прикладной математики, РАН, Москва
- Claudio Russo, Niels Kokholm, Ken Friis Larsen, Peter Sestoft
- используется движок и некоторые идеи из Caml Light © Xavier Leroy, Damien Doligez.
- порт на MacOS © Doug Currie.

Глава 2

Programming in Standard ML'97

http://homepages.inf.ed.ac.uk/stg/NOTES/

© Stephen Gilmore
Laboratory for Foundations of Computer Science
The University of Edinburgh

Часть II

Язык bI

Глава 3

DLR: Dynamic Language Runtime

DLR: Dynamic Language Runtime — может использоваться как runtime-ядро для реализации динамических языков, или только в качестве библиотеки хранилища данных

синтаксический парсер для разбора текстовых данных, файлов конфигурации, скриптов и т.п., необязателен. В результате разбора формируется синтаксическое дерево из динамических объектов DLR. По реализации может быть

конфигурируемым в runtime добавление/изменение/удаление правил правил грамматики в процессе работы программы

статическим неизменный синтаксис, реализация в виде внешнего модуля, в самом простом случае достаточно использования **flex**/**bison**

библиотека динамических типов данных выполняет функции хранения данных, может быть реализована

в *Lisp*-стиле базовый набор скаляров 4.2 (символы, строки и числа) и тип **cons-ячейка** позволяющий конструировать составные структуры данных

***bI*-стиль** универсальный символьный тип 4.1, позволяющий хранить как скаляры, так и вложенные элементы; в базовый тип **AST** заложено хранение типа данных `tag`, его значения `value`, и два способа вложенных хранилищ: плоский упорядоченный список `nest` и именованный неупорядоченный со строковыми ключами `pars`.

От базового символьного типа наследуются

скаляры символ, строка, несколько вариантов чисел (целые, плавающие, машинные, комплексные)[1]

композиты структуры данных и объекты

функционалы объекты, для которых определен *оператор аппликации*

библиотека операций над данными для преобразования данных и символьных вычислений на списках, деревьях, комбинаторах и т.п.

Lisp стандартная библиотека функций языка *Lisp*

bI каждый тип данных имеет набор унарных и бинарных *операторов*, реализованых в виде виртуальных методов классов

подсистема ООП реализация механизмов ООП, наследования от класса и объекта-инстанса, вывод типов, преобразование объектных моделей

реализация механизмов функциональных языков хвостовая рекурсия, pattern matching, динамическая компиляция, автоматическое распараллеливание на map/reduce

менеджер памяти со сборщиком мусора

динамический компилятор функциональных типов — через библиотеку JIT LLVM

[1] критерием скалярности можно считать возможность распознавания элемента данных лексером

статический компилятор

в объектный код через LLVM
кодогенератор C_+^+

Расширенный функционал

подсистема облачных вычислений и кластеризации расширение DLR на кластера: распределение объектов и процессов между вычислительными узлами. Варианты кластера с высокой связностью[2], Beowulf[3] с постоянным составом, интернет-облака с переменным составом: узлы асинхронно подключаются/отключаются, гомо/гетерогенные: по аппаратной платформе узлов и ОС/среде на каждом узле. Распределедение вычислений на одно- и многопроцессорных SMP-системах[4]

прикладные библиотеки GUI, CAD/CAM/EDA, численные методы, цифровая обработка сигналов, сетевые сервера и протоколы,...

подсистема кросс-трансляции между ходовыми языками программирования (C_+^+, *JavaScript*, *Python*, PHP, Паскакаль) через связку: парсер входного языка → система типов DLR → кодогенератор выходного языка

интерактивная объектная среда а-ля *SmallTalk* с виджетами и функционалом GUI, CAD, IDE и визуализациии данных

сервер приложений обслуживающий тонких браузерных клиентов по HTTP/JS

---

[2] аппаратная разделяемая память через сеть InfiniBand — "Сергей Королев"
[3] компьютеры общего назначени (офисные) с передачей сообщений по Gigabit Ethernet
[4] многопоточные вычисления на одном многоядерном узле

Глава 4

Система динамических типов

4.1 sym: символ = Абстрактный Символьный Тип /AST/

Использование класса **Sym** и виртуально наследованных от него классов, позволяет реализовать на C^{+}_{+} хранение и обработку **любых** данных в виде деревьев[1]. Прежде всего этот ***символьный тип*** применяется при разборе текстовых форматов данных, и текстов программ. **Язык** *bI* **построен как интерпретатор AST, примерно так же как язык** *Lisp* **использует списки**.

```
1 // ================================================== ABSTRACT SYMBOLIC TYPE (AST)
2 struct Sym {
```

тип (класс) и значение элемента данных

```
1 // ------------------------------------------------------------------------------
```

[1] в этом ACT близок к традиционной аббревиатуре AST: Abstract Syntax Tree

```
2     string tag;                          // data type / class
3     string val;                          // symbol value
```

конструкторы (токен используется в лексере)

```
1  // ------------------------------------------------------------ constructors
2     Sym(string,string);                  // <T:V>
3     Sym(string);                         // token
```

Хранение вложенных элементов реализовано через указатели на базовый тип **Sym**. Благодаря виртуальному наследованию и использованию RTTI, этими указателями можно пользоваться для работы с любыми другими наследованными типами данных[2]

AST может хранить (и обрабатывать) вложенные элементы

```
1  // ------------------------------------------------------------ nest[]ed elements
2     vector<Sym*> nest;
3     void push(Sym*);
4     void pop();
```

параметры (и поля класса)

```
1  // ------------------------------------------------------------ par{}ameters
2     map<string,Sym*> pars;
3     void par(Sym*);                      // add parameter
```

вывод дампа объекта в текстовом формате

```
1  // ------------------------------------------------------------ dumping
```

[2] числа, списки, высокоуровневые и скомпилированные функции, элементы GUI,..

```
2     virtual string dump(int depth=0);    // dump symbol object as text
3     virtual string tagval();             // <T:V> header string
4     string tagstr();                     // <T:'V'> Str-like header string
5     string pad(int);                     // padding with tree decorators
```

Операции над *символами* выполняются через использование набора *операторов*:

вычисление объекта

```
1 // ------------------------------------------------------ compute (evaluate)
2     virtual Sym* eval();
```

операторы

```
1  // ------------------------------------------------------------------ operators
2      virtual Sym* str();                 // str(A)    string representation
3      virtual Sym* eq(Sym*);              // A = B     assignment
4      virtual Sym* inher(Sym*);           // A : B     inheritance
5      virtual Sym* member(Sym*);          // A % B,C   named member (class slot)
6      virtual Sym* at(Sym*);              // A @ B     apply
7      virtual Sym* add(Sym*);             // A + B     add
8      virtual Sym* div(Sym*);             // A / B     div
9      virtual Sym* ins(Sym*);             // A += B    insert
10 };
```

4.2 Скаляры

4.2.1 str: строка

4.2.2 int: целое число

4.2.3 hex: машинное hex

4.2.4 bin: бинарная строка

4.2.5 num: число с плавающей точкой

4.3 Композиты

4.3.1 list: плоский список

4.3.2 cons: cons-пара и списки в *Lisp*-стиле

4.4 Функционалы

4.4.1 op: оператор

4.4.2 fn: встроенная/скомпилированная функция

4.4.3 lambda: лямбда

Глава 5

Программирование в свободном синтаксисе: FSP

5.1 Типичная структура проекта FSP: *lexical skeleton*

Скелет файловой структуры FSP-проекта = lexical skeleton = skelex

Создаем проект **prog** из командной строки (*Windows*):

```
1 mkdir prog
2 cd prog
3 touch src.src log.log ypp.ypp lpp.lpp hpp.hpp cpp.cpp Makefile bat.bat .gitignore
4 echo gvim -p src.src log.log ... Makefile bat.bat .gitignore >> bat.bat
5 bat
```

Создали каталог проекта, сгенерили набор пустых файлов (см. далее), и запуститили батник-hepler который запустит **(g)Vim**.

Для пользователей GitHub `mkdir` надо заменить на

```
git clone -o gh git@github.com:yourname/prog.git
cd prog
git gui &
...
```

src.src		исходный текст программы на вашем скриптовом языке
log.log		лог работы ядра *bI*
ypp.ypp	**flex**	парсер **??**
lpp.lpp	**bison**	лексер **??**
hpp.hpp	C_+^+	заголовочные файлы **??**
cpp.cpp	C_+^+	код ядра **??**
Makefile	**make**	зависимости между файлами и команды сборки (для утилиты **make**)
bat.bat	*Windows*	запускалка **(g)Vim ??**
.gitignore	**git**	список масок временных и производных файлов **??**

5.1.1 Настройки (g)Vim

При использовании редактора/IDE **(g)Vim** удобно настроить сочетания клавиш и подсветку **синтаксиса вашего скриптовго языка** так, как вам удобно. Для этого нужно создать несколько файлов конфигурации .vim: по 2 файла[1] для каждого диалекта скрипт-языка[2], и привязать их к расширениям через

[1] (1) привязка расширения файла и (2) подсветка синтаксиса

[2] если вы пользуетесь сильно отличающимся синтаксисом, но скорее всего через какое-то время практики FSP у вас выработается один диалект для всех программ, соответсвующий именно вашим вкусам в синтаксисе, и в этом случае его нужно будет описать только в файлах /.vim/(ftdetect|syntax).vim

dot-файлы **(g)Vim** в вашем домашнем каталоге. Подробно конфигурирование **(g)Vim** см. 18.

filetype.vim	**(g)Vim**	привязка расширений файлов (.src .log) к настройкам **(g)Vi**
syntax.vim	**(g)Vim**	синтаксическая подсветка для скриптов
/.vimrc	*Linux*	настройки для пользователя
/vimrc	*Windows*	
/.vim/ftdetect/src.vim	*Linux*	привязка команд к расширению .src
/vimfiles/ftdetect/src.vim	*Windows*	
/.vim/syntax/src.vim	*Linux*	синтаксис к расширению .src
/vimfiles/syntax/src.vim	*Windows*	

5.1.2 Дополнительные файлы

README.md	github	описание проекта для репоитория github
logo.png	github	логотип
logo.ico	*Windows*	
rc.rc	*Windows*	описание ресурсов: логотип, иконки приложения, меню,..

logo.png: Логотип

5.1.3 Makefile

Для сборки проекта используем команду **make** или **ming32-make** для *Windows*/**MinGW**. Прописываем в **Makefile** зависимости:

универсальный Makefile для fsp-проекта

```
1  log.log: ./exe.exe src.src
2      ./exe.exe < src.src > $@ && tail $(TAIL) $@
3  C = cpp.cpp ypp.tab.cpp lex.yy.c
4  H = hpp.hpp ypp.tab.hpp
5  CXXFILES += -std=gnu++11
6  ./exe.exe: $(C) $(H) Makefile
7      $(CXX) $(CXXFILES) -o $@ $(C)
8  ypp.tab.cpp: ypp.ypp
9      bison $<
10 lex.yy.c: lpp.lpp
11     flex $<
```

./exe.exe

префикс ./ требуется для правильной работы **ming32-make**, поскольку в *Linux* исполняемый файл может иметь любое имя и расширение, можем использовать .exe.

Для запуска транслятора используем простейший вариант — перенаправление потоков stdin/stdut на файлы, в этом случае не потребуется разбор параметров командной строки, и получим подробную трассировку выполнения трансляции.

переменные `C` и `H` задают набор исходный файлов ядра транслятора на C_+^+:

cpp.cpp реализация системы динамических типов данных, наследованных от символьного типа AST 4.1. Библиотека динамических классов языка *bI* II компактна, предоставляет достаточных

набор типов данных, и операций над ними. При необходимости вы можете легко написать свое дерево классов, если вам достаточно только простого разбора.

hpp.hpp заголовочные файлы также используем из *bI* II: содержат декларации динамических типов и интерфейс лексического анализатора, которые подходят для всех проектов

ypp.tab.cpp ypp.tab.hpp C_+^+ код синтаксического парсера, генерируемый утилитой **bison** 7.2

lex.yy.c код лексического анализатора, генерируемый утилитой **flex** 7.1

`CXXFLAGS += gnu++11` добавляем опцию диалекта C_+^+, необходимую для компиляции ядра *bI*

Глава 6

Синтаксический анализ текстовых данных

6.1 Универсальный Makefile

Универсальный Makefile сделан на базе 5.1.3, с добавлением переменной `APP` указывающий какой пример парсера следует скомпилировать и выполнить.

Для хранения (и возможной обработки) отпарсенных данных используем ядро языка *bI* 4 — используем файлы **../bi/hpp.hpp** и **../bi/cpp.cpp**. Ядро **очень компактно**, но умеет работать со скалярными, составными и функциональными данными, и содержит минимальную реализацию *ядра динамического языка.*

Универсальный **Makefile**

```
1 APP = minimal
2 $(APP).log: ./$(APP).exe $(APP).src
3 	./$(APP).exe < $(APP).src > $@ && tail $(TAIL) $@
4 C = ../bi/cpp.cpp ypp.tab.cpp lex.yy.c
```

```
5  H = ../bi/hpp.hpp ypp.tab.hpp
6  CXXFILES += -I../bi -I. -std=gnu++11
7  ./$(APP).exe: $(C) $(H) minimal.mk
8      $(CXX) $(CXXFILES) -o $@ $(C)
9  ypp.tab.cpp: $(APP).ypp
10     bison -o $@ $<
11 lex.yy.c: $(APP).lpp
12     flex -o $@ $<
13
14 .PHONY: src
15 src: minimal.src comment.src string.src ops.src brackets.src
16
17 minimal.src: ../bi/cpp.cpp
18     head -n11 $< > $@
19 comment.src: ../bi/cpp.cpp
20     head -n11 $< > $@
21 string.src: ../bi/cpp.cpp
22     head -n11 $< > $@
23 ops.src: ../bi/cpp.cpp
24     head -n5 $< > $@
25 brackets.src: ../bi/cpp.cpp
26     head -n5 $< > $@
```

6.2 C_+^+ интерфейс синтаксического анализатора

```
extern int yylex();             // получить код следующиго токена, и yylval.o
extern int yylineno;            // номер текущей строки файла исходника
```

```
extern char* yytext;            // текст распознанного токена, asciiz
#define TOC(C,X) { yylval.o = new C(yytext); return X; }

extern int yyparse();           // отпарсить весь текущий входной поток токенов
extern void yyerror(string);    // callback вызывается при синтаксической ошибке
#include "ypp.tab.hpp"
```

6.3 Минимальный парсер

Рассмотрим минимальный парсер, который может анализировать файлы текстовых данных (например исходники программ), и вычленять из них последовательности символов, которые можно отнести к ***скалярам*** символ, строка и число. [1]

Лексер **minimal.lpp** /**flex**/

```
1 %{
2 #include "hpp.hpp"
3 %}
4 %option noyywrap
5 %option yylineno
6 %%
7 [a-zA-Z0-9_.]+         TOC(Sym,SYM)
8 %%
```

(../bi/)**hpp.hpp** содержит определения интерфейса лексера 6.2, и ядра языка *bI* 4 для хранения результатов разбора текстовых данных

[1] эти три типа можно назвать атомами computer science

`noyywrap` выключает использование функции `yywrap()`

`yylineno` включает отслеживание строки исходного файла, используется при выводе сообщений об ошибках. В минимальном парсере не используется, но требуется для сборки *bI*-ядра.

`%%..%%` набор правил группировки отдельных символов в элементы данных — ***токены***, правила задаются с помощью *регулярных выражений*

`TOC(Sym,SYM)` единственное правило, распознающее любые группы сиволов как класс **bi::sym**: латинские буквы, цифры и символы _ и . (точка)[2]

Парсер **minimal.ypp /bison/**

```
1  %{
2  #include "hpp.hpp"
3  %}
4  %defines %union { Sym*o; }          /* use universal bI abstract type */
5  %token <o> SYM STR NUM              /* symbol 'string' number */
6  %type <o> ex scalar                 /* expression scalar */
7  %%
8  REPL : | REPL ex { cout << $2->tagval(); } ;
9  scalar : SYM | STR | NUM ;
10 ex : scalar ;
11 %%
```

hpp.hpp заголовок аналогичен лексеру 6.3

[2] точка добавлена, так часто используется в именах файлов

`%defines %union` указывает какие типы данных могут храниться в узлах разобранного ***синтаксического дерева***. Поскольку мы используем *bI*-ядро, нам будет достаточно пользоваться только классами языка *bI*, прежде всего универсальным символьным типом AST 4.1 и его прозводными классами.

`%token` описывает токены, которые может возвращать лексер `??`, причем набор токенов должен быть согласованным между лексером и парсером[3]

`%type` описывает типы синтаксических выражений, которые может распознавать ***грамматика*** синтаксического анализатора,

`REPL` выражение, описывающее грамматику, аналогичную простейшему варианту цикла REPL: Read Eval Print Loop[4]. В нашем случае часть вычисления Eval не выполняется[5], а часть Print выполняется через метод `Sym.tagval()`, возвращающий котороткую строку вида `<класс:значение>` для найденного токена.

`ex` (expression) универсальное символьное выражение языка *bI*, в нашем случае оно должно представлять только `scalar`

`scalar` выражение, представляющиее только распознаваемые скаляры:

`SYM` символ,

`STR` строку **или**

`NUM` число[6]

[3] определение токенов генерируется в файл **ypp.tab.hpp**

[4] чтение/вычисление/вывод/повторить

[5] разобранное выражение не вычисляется, хотя используемое ядро *bI* и поддерживает такой функционал

[6] числа в грамматике языка *bI* по типам не делятся, токен соответствует как **int**, так и **num**

В качестве тестового исходника возьмем C_+^+ код ядра языка *bI*: **../bi/cpp.cpp**:

minimal.src: Тестовый исходник

minimal.log: Результат прогона

```
1  #<sym:include> "<sym:hpp.hpp>"
2  #<sym:define> <sym:YYERR> "\<sym:n>\<sym:n>"<<<sym:yylineno><<":"<<<sym:msg><<"[">>><sym:y
3  <sym:void> <sym:yyerror>(<sym:string> <sym:msg>) { <sym:cout><<<sym:YYERR>; <sym:cerr><<<s
4  <sym:int> <sym:main>() { <sym:return> <sym:yyparse>(); }
5
6  <sym:Sym>::<sym:Sym>(<sym:string> <sym:T>, <sym:string> <sym:V>) { <sym:tag>=<sym:T>; <syn
7  <sym:Sym>::<sym:Sym>(<sym:string> <sym:V>):<sym:Sym>("<sym:sym>",<sym:V>) {}
8
9  <sym:string> <sym:Sym>::<sym:tagval>() { <sym:return> "<"+<sym:tag>+":"+<sym:val>+">"; }
10 <sym:string> <sym:Sym>::<sym:tagstr>() { <sym:return> "<"+<sym:tag>+":'"+<sym:val>+"'>"; }
11 <sym:string> <sym:Sym>::<sym:pad>(<sym:int> <sym:n>) { <sym:string> <sym:S>; <sym:for> (<s
12 <sym:string> <sym:Sym>::<sym:dump>(<sym:int> <sym:depth>) { <sym:string> <sym:S> = "\<sym
13     <sym:return> <sym:S>; }
14
15 <sym:Sym>* <sym:Sym>::<sym:eval>() { <sym:return> <sym:this>; }
```

Как видно по логу **minimal.log**, все группы сиволов, соответствующих правилу лексера `SYM`6.3, распознались как объекты *bI*, остальные остались символами и попали в лог без изменений.

6.4 Добавляем обработку комментариев

В тестах программ и файлов конфигурации очень часто используются *комментарии*. В языке *Python*, *bI* и UNIX shell комментарием является все от символа # до конца строки.

Для обработки таких ***строчных комментариев*** достаточно добавить одно правило лексера, **обязательно первым правилом**:

Лексер со строчными комментариями

```
1 %{
2 #include "hpp.hpp"
3 %}
4 %option noyywrap
5 %option yylineno
6 %%
7  #[^\n]*                {}
8  [a-zA-Z0-9_.]+         TOC(Sym,SYM)
9 %%
```

Группа символов, начинающаяся с символа #, затем идет ноль или более `[]*` любых символов не равных `^` концу строки `\n`. Пустое тело правила: C_+^+ код в `{}` скобках — выполняется и ничего не делает.

Тело правила SYM — вызов макроса `TOC(C,X)`6.2, наоборот, при своем выполнении создает токен, и возвращает код токена `=SYM`.

comment.log: Результат прогона

```
1
2
3 <sym:void> <sym:yyerror>(<sym:string> <sym:msg>) { <sym:cout><<<sym:YYERR>; <sym:cerr><<<s
4 <sym:int> <sym:main>() { <sym:return> <sym:yyparse>(); }
5
6 <sym:Sym>::<sym:Sym>(<sym:string> <sym:T>, <sym:string> <sym:V>) { <sym:tag>=<sym:T>; <sym
7 <sym:Sym>::<sym:Sym>(<sym:string> <sym:V>):<sym:Sym>("<sym:sym>",<sym:V>) {}
8
9 <sym:string> <sym:Sym>::<sym:tagval>() { <sym:return> "<"+<sym:tag>+":"+<sym:val>+">"; }
```

```
10 <sym:string> <sym:Sym>::<sym:tagstr>() { <sym:return> "<"+<sym:tag>+":'"+<sym:val>+"'>";
11 <sym:string> <sym:Sym>::<sym:pad>(<sym:int> <sym:n>) { <sym:string> <sym:S>; <sym:for> (<s
```

Как видно из лога, из вывода исчезли первые 2 строки, начинающиеся на #, причем концы этих строк остались (но не были как-либо распознаны).

6.5 Разбор строк

Для разбора строк необходимо использовать лексер с применением ***состояний***. Строки имеют сильно отличающийся от основного кода синтаксис, и для его обработки нужно **переключать набор правил лексера**.

Лексер с состоянием для строк

```
1  %{
2  #include "hpp.hpp"
3  string LexString;    /* string parser buffer */
4  %}
5  %option noyywrap
6  %option yylineno
7  %x lexstring
8  %%
9  #[^\n]*              {}
10
11 \"                   {BEGIN(lexstring); LexString="";}
12 <lexstring>\"        {BEGIN(INITIAL); yylval.o = new Str(LexString); return STR;}
13 <lexstring>\n        {LexString+=yytext[0];}
14 <lexstring>.         {LexString+=yytext[0];}
15
```

```
16 [a-zA-Z0-9_.]+          TOC(Sym,SYM)
17 %%
```

`string LexString` строковая буферная переменная, накапливающая символы строки

`%x lexstring` создание отдельного состояния лексера `lexstring`

`INITIAL` основное состояние лексера

`<lexstring>\n` правило конца строки позволяет использовать многострочные строки[7]

`<lexstring>.` любой символ в состоянии `<lexstring>`

Лог разбора со строками

```
1
2
3 <sym:void> <sym:yyerror>(<sym:string> <sym:msg>) { <sym:cout><<<sym:YYERR>; <sym:cerr><<<s
4 <sym:int> <sym:main>() { <sym:return> <sym:yyparse>(); }
5
6 <sym:Sym>::<sym:Sym>(<sym:string> <sym:T>, <sym:string> <sym:V>) { <sym:tag>=<sym:T>; <syn
7 <sym:Sym>::<sym:Sym>(<sym:string> <sym:V>):<sym:Sym>(<str:'sym'>,<sym:V>) {}
8
9 <sym:string> <sym:Sym>::<sym:tagval>() { <sym:return> <str:'<'>+<sym:tag>+<str:':'>+<sym:
10 <sym:string> <sym:Sym>::<sym:tagstr>() { <sym:return> <str:'<'>+<sym:tag>+<str:':''>+<sym
11 <sym:string> <sym:Sym>::<sym:pad>(<sym:int> <sym:n>) { <sym:string> <sym:S>; <sym:for> (<s
```

Обратите внимание, что ранее попадавшие в лог строки в двойных кавычках, типа `"]\n\n"`, стали распознаваться как строковые токены `<str:']\n\n'>`. [8]

[7] символ конца строки не распознается метасимволом . (точка) в регулярном выражении, и требует явного указания

[8] использованы 'одинарные кавычки' как в *Python/bI*

6.6 Добавляем операторы

Для разбора языков программирования необходима поддержка операторов, включим общепринятые одиночные операторы, операторы C_+^+ и bI. **Скобки различного вида тоже будет рассматривать как операторы.** Операторы реализованы в ядре bI как отдельный класс **op**, зададим пачку правил разбора операторов, создающих токены `TOC(Op,XXX)`:

Лексер с операторами

```
1  %{
2  #include "hpp.hpp"
3  string LexString;    /* string parser buffer */
4  %}
5  %option noyywrap
6  %option yylineno
7  %x lexstring
8  %%
9  #[^\n]*              {}                /* # line comment */
10
11 \"                   {BEGIN(lexstring); LexString="";}
12 <lexstring>\"        {BEGIN(INITIAL); yylval.o = new Str(LexString); return STR;}
13 <lexstring>\n        {LexString+=yytext[0];}
14 <lexstring>.         {LexString+=yytext[0];}
15
16 [a-zA-Z0-9_.]+       TOC(Sym,SYM)      /* symbol */
17
18 \(                   TOC(Op,LB)        /* brackets */
19 \)                   TOC(Op,RB)
20 \[                   TOC(Op,LQ)
21 \]                   TOC(Op,RQ)
```

```
\{                      TOC(Op,LC)
\}                      TOC(Op,RC)

\+                      TOC(Op,ADD)     /* typical arithmetic operators */
\-                      TOC(Op,SUB)
\*                      TOC(Op,MUL)
\/                      TOC(Op,DIV)
\^                      TOC(Op,POW)
                                        /* bI language specific */
\=                      TOC(Op,EQ)      /* assign */
\@                      TOC(Op,AT)      /* apply */
\~                      TOC(Op,TILD)    /* quote */
\:                      TOC(Op,COLON)   /* inheritance */

%%
```

Парсер с операторами

```
%{
#include "hpp.hpp"
%}
%defines %union { Sym*o; }      /* use universal bI abstract type */
%token <o> SYM STR NUM          /* symbol 'string' number */
%token <o> LB RB LQ RQ LC RC    /* brackets: () [] {} */
%token <o> ADD SUB MUL DIV POW  /* arithmetic operators: + - * / ^ */
%token <o> EQ AT TILD COLON     /* bi specific operators: = @ ~ : */
%type <o> ex scalar             /* expression scalar */
%type <o> bracket operator
%%
```

```
12 REPL : | REPL ex { cout << $2->dump(); } ;
13 scalar : SYM | STR | NUM ;
14 ex : scalar | operator ;
15 bracket : LB | RB | LQ | RQ | LC | RC ;
16 operator :
17     bracket
18     | ADD | SUB | MUL | DIV | POW
19     | EQ | AT | TILD | COLON
20 ;
21 %%
```

Лог уже стал нечитаем, переключаемся на древовидный вывод через метод `Sym.dump()`.

Разбор с операторами

```
1
2
3
4 <sym:void>
5 <sym:yyerror>
6 <op:(>
7 <sym:string>
8 <sym:msg>
9 <op:)>
10 <op:{>
11 <sym:cout><<
12 <sym:YYERR>;
13 <sym:cerr><<
14 <sym:YYERR>;
15 <sym:exit>
```

```
<op:(>
<op:->
<sym:1>
<op:)>;
<op:}>

<sym:int>
<sym:main>
<op:(>
<op:)>
<op:{>
<sym:return>
<sym:yyparse>
<op:(>
<op:)>;
<op:}>
```

6.7 Обработка вложенных структур (скобок)

Обработка вложенных структур возможна только парсером, лексер оставляем без изменений. Хранение вложенных структур в виде дерева — главная фича типа *bI* AST4.1. Заменяем грамматическое выражение `bracket` на отдельные выражения для скобок:

Парсер со скобками

```
%{
#include "hpp.hpp"
%}
```

```
%defines %union { Sym*o; }       /* use universal bI abstract type */
%token <o> SYM STR NUM           /* symbol 'string' number */
%token <o> LB RB LQ RQ LC RC     /* brackets: () [] {} */
%token <o> ADD SUB MUL DIV POW   /* arithmetic operators: + - * / ^ */
%token <o> EQ AT TILD COLON      /* bi specific operators: = @ ~ : */
%token <o> SCOLON GR LS
%type <o> ex scalar              /* expression scalar */
%type <o> operator
%%
REPL : | REPL ex { cout << $2->dump(); } ;
scalar : SYM | STR | NUM ;
ex :
      ex ex                      { $$=$1; $$->push($2); }
    | scalar | operator
    | LB ex RB                   { $$=new Sym("()"); $$->push($2); }
    | LB  RB                     { $$=new Sym("()"); }
    | LQ ex RQ                   { $$=new Sym("[]"); $$->push($2); }
    | LC ex RC                   { $$=new Sym("{}"); $$->push($2); }
;
operator :
    ADD | SUB | MUL | DIV | POW
    | EQ | AT | TILD | COLON
    | SCOLON | GR | LS
;
%%
```

Разбор со скобками

```
1
```

```
2
3
4
5
6
7  <sym:void>
8      <sym:yyerror>
9          <sym:()>
10             <sym:string>
11                 <sym:msg>
12             <sym:{}>
13                 <sym:cout>
14                     <op:<>
15                         <op:<>
16                             <sym:YYERR>
17                                 <op:;>
18                                     <sym:cerr>
19                                         <op:<>
20                                             <op:<>
21                                                 <sym:YYERR>
22                                                     <op:;>
23                                                         <sym:exit>
24                                                             <sym:()>
25                                                                 <op:->
26                                                                     <sym:1>
27                                                                 <op:;>
28             <sym:int>
29                 <sym:main>
```

```
30                                  <sym:()>
31                                      <sym:{}>
32                                          <sym:return>
33                                              <sym:yyparse>
34                                                  <sym:()>
35                                                      <op:;>
```

Глава 7

Синтаксический анализатор

Синтаксис языка *bI* был выбран алголо-подобным, более близким к современным императивным языкам типа C_+^+ и *Python*. Использование типовых утилит-генераторов позволяет легко описать синтаксис с инфиксными операторами и скобочной записью для композитных типов 4.3, и не заставлять пользователя закапываться в клубок *Lisp*овских скобок.

Инфиксный синтаксис **для файлов конфигурации** удобен неподготовленным пользователям, а возможность определения пользовательских функций и объектная система, встроенная в ядро *bI*, дает богатейшие возможности по настройке и кастомизации программ.

Единственной проблемой с точки зрения синтаксиса для начинающего пользователя *bI* может оказаться отказ от скобок при вызове функций, применение оператора явной аппликации `@`, и функциональные наклонности самого *bI*, претендующего на звание универсального **объектного мета-языка** и **языка шаблонов**.

7.1 lpp.lpp: лексер /flex/

lpp.lpp

```
%{
#include "hpp.hpp"
string LexString;                                          // string parse buffer
void incLude(Sym*inc) {                                    // .include processing
    if (!(yyin = fopen((inc->val).c_str(),"r"))) yyerror("");
    yypush_buffer_state(yy_create_buffer(yyin,YY_BUF_SIZE));
}
%}
%option noyywrap
%option yylineno
%x lexstring docstring
S [\-\+]?
N [0-9]+
%%
#[^\n]*             {}                    /* == line comment == */

                                                           /* == .directives == */
^\.end              {yyterminate();}                       /* .end */
^\.inc[ \t]+[^\n]+  {incLude(new Directive(yytext));}      /* .include */
^\.[a-z]+[^\n]*     TOC(Directive,DIR)                     /* .directive */

                                                           /* 'string' parse */
'                   {BEGIN(lexstring); LexString="";}
<lexstring>'        {BEGIN(INITIAL); yylval.o=new Str(LexString); return STR;}
<lexstring>\\t      {LexString+='\t';}
```

```
<lexstring>\\n      {LexString+='\n';}
<lexstring>\n       {LexString+=yytext[0];}
<lexstring>.        {LexString+=yytext[0];}
                                                    /* "docstring" parse */
\"                  {BEGIN(docstring); LexString="";}
<docstring>\"       {BEGIN(INITIAL); yylval.o=new Str(LexString); return DOC;}
<docstring>\\t      {LexString+='\t';}
<docstring>\\n      {LexString+='\n';}
<docstring>\n       {LexString+=yytext[0];}
<docstring>.        {LexString+=yytext[0];}

                                        /* == numbers == */
{S}{N}\.{N}         TOC(Num,NUM)        /* floating point */
{S}{N}[eE]{S}{N}    TOC(Num,NUM)        /* exponential */
{S}{N}              TOC(Int,NUM)        /* integer */
0x[0-9A-F]+         TOC(Hex,NUM)        /* machine hex */
0b[01]+             TOC(Bin,NUM)        /* bin string */

[a-zA-Z0-9_]+       TOC(Sym,SYM)        /* == symbol == */

\(                  TOC(Op,LP)          /* == brackets == */
\)                  TOC(Op,RP)
\[                  TOC(Op,LQ)          /* [list] */
\]                  TOC(Op,RQ)
\{                  TOC(Op,LC)          /* {lambda} */
\}                  TOC(Op,RC)
\<                  TOC(Op,LV)          /* <vector> */
\>                  TOC(Op,RV)
```

```
54
55                                              /* == operators == */
56 \=                    TOC(Op,EQ)
57 \@                    TOC(Op,AT)
58 \~                    TOC(Op,TILD)
59 \:                    TOC(Op,COLON)
60 \.                    TOC(Op,DOT)
61 \,                    TOC(Op,COMMA)
62
63 \+                    TOC(Op,ADD)
64 \-                    TOC(Op,SUB)
65 \*                    TOC(Op,MUL)
66 \/                    TOC(Op,DIV)
67 \^                    TOC(Op,POW)
68
69 [ \t\r\n]+            {}                    /* == drop spaces == */
70
71 <<EOF>>           { yypop_buffer_state();             /* end of .inluded file */
72                     if (!YY_CURRENT_BUFFER)
73                         yyterminate();}
74 %%
```

7.2 ypp.ypp: парсер /bison/

ypp.ypp

```
1 %{
```

```
2 #include "hpp.hpp"
3 %}
4 %defines %union { Sym*o; }          /* universal bI abstract symbolic type  */
5 %token <o> SYM STR NUM DIR DOC      /* symbol 'string' number .directive    */
6 %token <o> LP RP LQ RQ LC RC LV RV  /* () [] {} <>                          */
7 %token <o> EQ AT TILD COLON         /* = @ ~ :                              */
8 %token <o> DOT COMMA                /* . ,                                  */
9 %token <o> ADD SUB MUL DIV POW      /* + - * / ^                            */
10 %type <o> ex scalar list lambda    /* expression scalar [list] {lam:bda}   */
11 %type <o> vector cons op bracket   /* <vector> co,ns operator bracket      */
12
13 %left DOC
14 %left EQ
15 %right COMMA
16 %left ADD SUB
17 %left MUL DIV
18 %left POW
19 %left AT
20 %left PFX
21 %left TILD
22 %left COLON
23 %%                                 /* REPL with full pasre/eval logging */
24 REPL : | REPL ex                   { cout << $2->dump();
25                                    cout << "\n--------------------";
26                                    cout << $2->eval()->dump();
27                                    cout << "\n====================\n"; } ;
28 ex      : scalar | DIR
29         | ex DOC                   { $$=$1; $$->doc=$2->val; }
```

```
30|         | LP ex RP            { $$=$2; }
31|         | LQ list RQ          { $$=$2; }
32|         | LC lambda RC        { $$=$2; }
33|         | LV vector RV        { $$=$2; }
34|         | TILD ex             { $$=$1; $$->push($2); }
35|         | TILD op             { $$=$1; $$->push($2); }
36|         | cons
37|         | ADD ex %prec PFX    { $$=$2->pfxadd(); }
38|         | SUB ex %prec PFX    { $$=$2->pfxsub(); }
39|         | ex EQ ex            { $$=$2; $$->push($1); $$->push($3); }
40|         | ex AT ex            { $$=$2; $$->push($1); $$->push($3); }
41|         | ex ADD ex           { $$=$2; $$->push($1); $$->push($3); }
42|         | ex SUB ex           { $$=$2; $$->push($1); $$->push($3); }
43|         | ex MUL ex           { $$=$2; $$->push($1); $$->push($3); }
44|         | ex DIV ex           { $$=$2; $$->push($1); $$->push($3); }
45|         | ex POW ex           { $$=$2; $$->push($1); $$->push($3); }
46| ;
47| op      : bracket |EQ |AT |TILD |COLON |DOT |COMMA |ADD |SUB |MUL |DIV |POW ;
48| bracket : LP |RP |LQ |RQ |LC |RC |LV |RV ;
49| scalar  : SYM | STR | NUM ;
50|
51| cons    : ex COMMA ex         { $$=new Cons($1,$3); } ;
52| list    :                     { $$=new List(); }
53|         | list ex             { $$=$1; $$->push($2); }
54| ;
55| lambda  :                     { $$=new Lambda(); }
56|         | lambda SYM COLON    { $$=$1; $$->par($2); }
57|         | lambda ex           { $$=$1; $$->push($2); }
```

```
58 ;
59 vector   :                    { $$=new Vector(); }
60         | vector ex           { $$=$1; $$->push($2); }
61 ;
62 %%
```

В качестве типа-хранилища для узлов синтаксического дерева идеально подходит базовый символьный тип *bI* 4.1, причем его применение в этом качестве рассматривалось как основное: гибкое представление произвольных типов данных. Собственно его название намекает.

В качестве токенов-скаляров логично выбираются SYMвол, STRока и число NUM[1]. Надо отметить, что в принципе можно было бы обойтись единственным SYM, но для дополнительного контроля грамматики полезно выделить несколько токенов: это позволит гарантировать что в определении класса **??** вы сможете использовать в качестве суперкласса и имен полей только символы. По крайне мере до момента, когда в очередном форке *bI* не появится возможность наследовать любые объекты.

[1] их можно вообще рассматривать как элементарные частицы Computer Science, правда к ним еще придется добавить PTR: божественный указатель

Часть III

em*Linux* для встраиваемых систем

Структура встраиваемого микро*Linux*а

syslinux Загрузчик

em*Linux* поставляется в виде двух файлов:

1. ядро `$(HW)$(APP).kernel`
2. сжатый образ корневой файловой системы `$(HW)$(APP).rootfs`

Загрузчик считывает их с одного из носителей данных, который поддерживается загрузчиком[2], распаковывает в память, включив защищенный режим процессора, и передает управление ядру *Linux*.

Для работы em*Linux* не требуются какие-либо носители данных: вся (виртуальная) файловая система располагается в ОЗУ. При необходимости к любому из каталогов корневой ФС может быть *подмонтирована* любая существующая дисковая или сетевая файловая система (FAT,NTFS,Samba,NFS,..), причем можно явно запретить возможность записи на нее, защитив данные от разрушения.

Использование rootfs в ОЗУ позволяет гарантировать защиту базовой ОС и пользовательских исполняемых файлов от внезапных выключений питания и ошибочной записи на диск. Еще большую защиту даст отключение драйверов загрузочного носителя в ядре. Если отключить драйвера SATA/IDE и грузиться с USB флешки, практически невозможно испортить основную установку ОС и пользовательские файлы на чужом компьютере.

kernel Ядро *Linux* 3.13.xx

[2] IDE/SATA HDD, CDROM, USB флешка, сетевая загрузка с BOOTP-сервера по Ethernet

ulibc Базовая библиотека языка Си

busybox Ядро командной среды UNIX, базовые сетевые сервера

дополнительные библиотеки

zlib сжатие/распаковка gzip
??? библиотека помехозащищенного кодирования
png библиотека чтения/записи графического формата .png
freetype рендер векторных шрифтов (TTF)
SDL полноэкранная (игровая) графика, аудио, джойстик

кодеки аудио/видео форматов: ogg vorbis, mp3, mpeg, ffmpeg/gsm

К базовой системе добавляются пользовательские программы ***/usr/bin***
и динамические библиотеки ***/usr/lib***.

Процедура сборки

Глава 8

clock: коридорные электронные часы = контроллер умного дурдома

Глава 9

gambox: игровая приставка

Часть IV

GNU Toolchain и C_+^+ для встраиваемых систем

Глава 10

Программирование встраиваемых систем с использованием GNU Toolchain [21]

10.1 Введение

Пакет компиляторов GNU toolchain широко используется при разработке программного обеспечения для встраиваемых систем. Этот тип разработки ПО также называют ***низкоуровневым***, ***standalone*** или ***bare metal*** программированием (на Си и C_+^+). Написание низкоуровневого кода на Си добавляет программисту новых проблем, требующих глубокого понимания инструмента разработчика — **GNU Toolchain**. Руководства разработчика **GNU Toolchain** предоставляют отличную информацию по инструментарию, но с

[1] © http://bravegnu.org/gnu-eprog/
[2] © https://github.com/ponyatov/gnu-eprog/blob/ru/gnu-eprog.asciidoc

точки зрения самого **GNU Toolchain**, чем с точки зрения решаемой проблемы. Поэтому было написано это руководство, в котором будут описаны типичные проблемы, с которыми сталкивается начинающий разработчик.

Этот учебник стремится занять свое место, объясняя использование **GNU Toolchain** с точки зрения практического использования. Надеемся, что его будет достаточно для разработчиков, собирающихся освоить и использовать **GNU Toolchain** в их embedded проектах.

В иллюстративных целях была выбрана встроенная система на базе процессорного ядра ARM, которая эмулируется в пакете **Qemu**. С таким подходом вы сможете освоить **GNU Toolchain** с комфортом на вашем рабочем компьютере, без необходимости вкладываться в "физическое" железо, и бороться со сложностями с его запуском. Учебник не стремиться обучить работе с архитектурой ARM, для этого вам нужно будет воспользоваться дополнительными книгами или онлайн-учебниками типа:

- ARM Assembler http://www.heyrick.co.uk/assembler/
- ARM Assembly Language Programming http://www.arm.com/miscPDFs/9658.pdf

Но для удобства читателя, некоторое множество часто используемых ARM-инструкций описано в приложении 10.18.

10.2 Настройка тестового стенда

В этом разделе описано, как настроить на вашей рабочей станции простую среду разработки и тестирования ПО для платформы ARM, используя **Qemu** и **GNU Toolchain**. **Qemu** это программный[3] эмулятор нескольких распространенных аппаратных платформ. Вы можете написать программу на ассемблере и C_{+}^{+}, скомпилировать ее используя **GNU Toolchain** и отладить ее в эмуляторе **Qemu**.

[3] для i386 — программно-аппаратный, использует средства виртуализации хост-компьютера

10.2.1 Qemu ARM

Будем использовать **Qemu** для эмуляции отладочной платы **Gumstix connex** на базе процессора PXA255. Для работы с этим учебником у вас должен быть установлен **Qemu** версии не ниже 0.9.1.

Процессор[4] PXA255 имеет ядро ARM с набором инструкций ARMv5TE. PXA255 также имеет в своем составе несколько блоков периферии. Некоторая периферия будет описана в этом курсе далее.

10.2.2 Инсталляция Qemu на *Debian GNU/Linux*

Этот учебник требует **Qemu** версии не ниже 0.9.1. Пакет **Qemu** доступный для современных дистрибутивов *Debian GNU/Linux*, вполне удовлетворяет этим условиям, и собирать свежий **Qemu** из исходников совсем не требуется[5]. Установим пакет командой:

```
$ sudo apt install qemu
```

10.2.3 Установка кросс-компилятора GNU Toolchain для ARM

Если вы предпочитаете простые пути, установите пакет кросс-компилятора командной

```
sudo apt install gcc-arm-none-eabi
```

или

1. Годные чуваки из CodeSourcery[6] уже давно запилили несколько вариантов **GNU Toolchain**ов для разных ходовых архитектур. Скачайте готовую бинарную бесплатную lite-сборку **GNU Toolchain-ARM**

[4] Точнее SoC: система-на-кристалле

[5] хотя может быть и очень хочется

[6] подразделение Mentor Graphics

2. Распакуйте tar-архив в каталог */toolchains*:

```
$ mkdir ~/toolchains
$ cd ~/toolchains
$ tar -jxf ~/downloads/arm-2008q1-126-arm-none-eabi-i686-pc-linux-gnu.tar.bz2
```

3. Добавьте bin-каталог тулчейна в переменную среды `PATH`.

```
$ PATH=$HOME/toolchains/arm-2008q1/bin:$PATH
```

4. Чтобы каждый раз не выполнять предыдущую команду, вы можете прописать ее в дот-файл **.bashrc**.

Для совсем упертых подойдет рецепт сборки полного комплекта кросс-компилятора из исходных текстов, описанный в 12.

10.3 Hello ARM

В этом разделе вы научитесь пользоваться arm-ассемблером, и тестировать вашу программу на голом железе — эмуляторе платы **connex** в **Qemu**.

Файл исходника ассемблерной программы состоит из последовательности инструкций, по одной на каждую строку. Каждая инструкция имеет формат (каждый компонент не обязателен):

```
<метка>:    <инструкция>          @ <комментарий>
```

метка — типичный способ пометить адрес инструкции в памяти. Метка может быть использована там, где требуется указать адрес, например как операнд в команде перехода. Метка может состоять из латинских букв, цифр[7], символов `_` и `$`.

[7] не может быть первым символом метки

комментарий начинается с символа `@` — все последующие символы игнорируются до конца строки

инструкция может быть инструкцией процессора или директивой ассемблера, начинающейся с точки "."

Вот пример простой ассемблерной программы 1 для процессора ARM, складывающей два числа:

Листинг 1: Сложение двух чисел

```
        .text
start:                          @ метка необязательна
        mov   r0, #5            @ загрузить в регистр r0 значение 5
        mov   r1, #4            @ загрузить в регистр r1 значение 4
        add   r2, r1, r0        @ сложить r0+r1 и сохранить в r2 (справа налево)

stop:   b stop                  @ пустой бесконечный цикл для останова выполнения
```

`.text` ассемблерная директива, указывающая что последующий код должен быть *ассемблирован* в ***секцию кода*** *`.text`* а не в секцию `.data`. *Секции* будут подробно описаны далее.

10.3.1 Сборка бинарника

Сохраните программу в файл **add.s** [8]. Для ассемблирования файла вызовите ассемблер **as**:

```
$ arm-none-eabi-as -o add.o add.s
```

[8] .s или .S стандарное расширение в мире OpenSource, указывает что это файл с программной на ассемблере

Опция `-o` указывает выходной файл с ***объектным кодом***, имеющий стандартное расширение `.o`[9].

Команды кросс-тулчейна всегда имеют префикс целевой архитектуры (target triplet), для которой они были собраны, чтобы предотвратить конфликт имен с хост-тулчейном для вашего рабочего компьютера. Далее утилиты **GNU Toolchain** будут использоваться без префикса для лучшей читаемости. **не забывайте добавлять arm-none-eabi-, иначе получите множество странных ошибок типа "unexpected command"**.

```
$ (arm-none-eabi-)as -o add.o add.s
$ (arm-none-eabi-)objdump -x add.o
```

вывод команды **arm-none-eabi-objdump -x**: ELF-заголовки в файле объектного кода

```
1  add.o:     file format elf32-littlearm
2  add.o
3  architecture: armv4, flags 0x00000010:
4  HAS_SYMS
5  start address 0x00000000
6  private flags = 5000000: [Version5 EABI]
7
8  Sections:
9  Idx Name          Size      VMA       LMA       File off  Algn
10   0 .text         00000010  00000000  00000000  00000034  2**2
11                   CONTENTS, ALLOC, LOAD, READONLY, CODE
12   1 .data         00000000  00000000  00000000  00000044  2**0
```

[9] и внутренний формат ELF (как завещал великий *Linux*)

```
13                      CONTENTS, ALLOC, LOAD, DATA
14   2 .bss             00000000  00000000  00000000  00000044  2**0
15                      ALLOC
16   3 .ARM.attributes  00000014  00000000  00000000  00000044  2**0
17                      CONTENTS, READONLY
18 SYMBOL TABLE:
19 00000000 l    d  .text  00000000 .text
20 00000000 l    d  .data  00000000 .data
21 00000000 l    d  .bss   00000000 .bss
22 00000000 l       .text  00000000 start
23 0000000c l       .text  00000000 stop
24 00000000 l    d  .ARM.attributes      00000000 .ARM.attributes
```

Секция `.text` имеет размер `Size=0x0010` =16 байт, и содержит **машинный код**:

машинный код из секции `.text`: **objdump -d**

```
1 add.o:     file format elf32-littlearm
2
3 Disassembly of section .text:
4
5 00000000 <start>:
6    0:   e3a00005    mov r0, #5
7    4:   e3a01004    mov r1, #4
8    8:   e0812000    add r2, r1, r0
9
10 0000000c <stop>:
11   c:   eafffffe    b   c <stop>
```

Для генерации **исполняемого файла**[10] вызовем ***линкер*** **ld**:

```
$ arm-none-eabi-ld -Ttext=0x0 -o add.elf add.o
```

Опять, опция `-o` задает выходной файл. `-Ttext=0x0` явно указывает адрес, от которого будут отсчитываться все метки, т.е. секция инструкций начинается с адреса `0x0000`. Для просмотра адресов, назначенных меткам, можно использовать команду `(arm-none-eabi-)nm` [11]:

```
ponyatov@bs:/tmp$ arm-none-eabi-nm add.elf
...
00000000 t start
0000000c t stop
```

* если вы забудете опцию -T, вы получите этот вывод с адресами `00008xxx` — эти адреса были заданы при компиляции **GNU Toolchain**-ARM, и могут не совпадать с необходимыми вам. Проверяйте ваши .elfы с помощью **nm** или **objdump**, если программы не запускаются, или **Qemu** ругается на ошибки (защиты) памяти.

Обратите внимание на ***назначение адресов*** для меток `start` и `stop`: адреса начанаются с 0x0. Это адрес первой инструкции. Метка `stop` находится после третьей инструкции. Каждая инструкция занимает 4 байта[12], так что `stop` находится по адресу $0xC_{hex} = 12_{dec}$. ***Линковка*** с другим ***базовым адресом*** `-Ttext=nnnn` приведет к сдвигу адресов, назначенных меткам.

---

[10] обычно тот же формат ELF.o, слепленный из одного или нескольких объектных файлов, с некоторыми модификациями см. опцию -T далее

[11] NaMes

[12] в множестве команд ARM-32, если вы компилируете код для микроконтроллера Cortex-Mx в режиме команд Thumb или Thumb2, команды 16-битные, т.е. 2 байта

```
$ arm-none-eabi-ld -Ttext=0x20000000 -o add.elf add.o
$ arm-none-eabi-nm add.elf
... clip ...
20000000 t start
2000000c t stop
```

Выходной файл, созданный **ld** имеет формат, который называется ***ELF***. Существует множество форматов, предназначенных для хранения выполняемого и объектного кода[13]. Формат ELF применяется для хранения машинного кода, если вы запускаете его в базовой ОС[14], но поскольку мы собираемся запускать нашу программу на bare metal[15], мы должны сконвертировать полученный .elf файл в более простой ***бинарный формат***.

Файл в *бинарном формате* содержит последовательность байт, начинающуюся с определенного адреса памяти, поэтому бинарный файл еще называют *образом памяти*. Этот формат типичен для утилит программирования флеш-памяти микроконтроллеров, так как все что требуется сделать — последовательно скопировать каждый байт из файла в FlashROM-память микроконтроллера, начиная с определенного начального адреса.[16]

Команда **GNU Toolchain objcopy** используется для конвертирования машинного кода между разными объектными форматами. Типичное использование:

```
$ objcopy -O <выходной_формат> <входной_файл> <выходной_файл>
```

Конвертируем **add.elf** в бинарный формат:

[13] можно отдельно отметить Microsoft COFF (объектные файлы .obj) и PE (.exe)cutable

[14] прежде всего “большой” или встраиваемый *Linux*

[15] голом железе

[16] Та же операция выполняется и для SoC-систем с NAND-флешем: записать бинарный образ начиная с некоторого аппаратно фиксированного адреса.

```
$ objcopy -O binary add.elf add.bin
```

Проверим размер полученного бинарного файла, он должен быть равен тем же 16 байтам[17]:

```
$ ls -al add.bin
-rw-r--r-- 1 vijaykumar vijaykumar 16 2008-10-03 23:56 add.bin
```

Если вы не доверяете **ls**, можно дизассемблировать бинарный файл:

```
ponyatov@bs:/tmp$ arm-none-eabi-objdump -b binary -m arm -D add.bin

add.bin:     file format binary

Disassembly of section .data:

00000000 <.data>:
   0:   e3a00005        mov     r0, #5
   4:   e3a01004        mov     r1, #4
   8:   e0812000        add     r2, r1, r0
   c:   eafffffe        b       0xc
ponyatov@bs:/tmp$
```

Опция `-b` задает формат файла, опция `-m` (machine) архитектуру процессора, получить полный список сочетаний `-b/-m` можно командной `arm-none-eabi-objdump -i`.

[17] 4 инструкции по 4 байта каждая

10.3.2 Выполнение в Qemu

Когда ARM-процессор сбрасывается, он начинает выполнять команды с адрсе 0x0. На плате Commnex установлен флеш на 16 мегабайт, начинающийся с адрес 0x0. Таким образом, при сбросе будут выполняться инструкции с начала флеша.

Когда **Qemu** эмулирует плату connex, в командной строке должен быть указан файл, который будет считаться образом флеш-памяти. Формат флеша очень прост — это побайтный образ флеша без каких-либо полей или заголовков, т.е. это тот же самый *бинарный формат*, описанный выше.

Для тестирования программы в эмуляторе Gumstix connex, сначала мы создаем 16-мегабайтный файл флеша, копируя 16M нулей из файла **/dev/zero** с помощью команды **dd**. Данные копируются 4Кбайтными блоками[18] (4096 x 4K):

```
$ dd if=/dev/zero of=flash.bin bs=4K count=4K
4096+0 записей получено
4096+0 записей отправлено
 скопировано 16777216 байт (17 MB), 0,0153502 c, 1,1 GB/c

$ du -h flash.bin
16M     flash.bin
```

Затем переписываем начало **flash.bin** копируя в него содержимое **add.bin**:

```
$ dd if=add.bin of=flash.bin bs=4K conv=notrunc
0+1 записей получено
0+1 записей отправлено
 скопировано 16 байт (16 B), 0,000173038 c, 92,5 kB/c
```

[18] опция bs= (blocksize)

После сброса процессор выполняет код с адреса 0x0, и будут выполняться инструкции нашей программы. Команда запуска **Qemu**:

```
$ qemu-system-arm -M connex -pflash flash.bin -nographic -serial /dev/null

QEMU 2.1.2 monitor - type 'help' for more information
(qemu)
```

Опция `-M connex` выбирает режим эмуляции: **Qemu** поддерживает эмуляцию нескольких десятков вариантов железа на базе ARM процессоров. Опция `-pflash` указывает файл образа флеша, который должен иметь определенный размер (16M). `-nographic` отключает эмуляцию графического дисплея (в отдельном окне). Самая важная опция `-serial /dev/null` подключает последовательный порт платы на **/dev/null**, при этом в терминале после запуска **Qemu** вы получите **консоль монитора**.

Qemu выполняет инструкции, и останавливается в бесконечном цикле на `stop`, выполняя команду `stop: b stop`. Для просмотра содержимого регистров процессора воспользуемся ***монитором***. Монитор имеет интерфейс командной строки, который вы можете использовать для контроля работы эмулируемой системы. Если вы запустите **Qemu** как указано выше, монитор будет доступен через stdio.

Для просмотра регистров выполним команду `info registers`:

```
(qemu) info registers
R00=00000005 R01=00000004 R02=00000009 R03=00000000
R04=00000000 R05=00000000 R06=00000000 R07=00000000
R08=00000000 R09=00000000 R10=00000000 R11=00000000
R12=00000000 R13=00000000 R14=00000000 R15=0000000c
PSR=400001d3 -Z-- A svc32
FPSCR: 00000000
```

Обратите внимание на значения в регистрах r00..r02: 4, 5 и ожидаемый результат 9. Особое значение для ARM имеет регистр r15: он является указателем команд, и содержит адрес текущей выполняемой машинной команды, т.е. `0x000c: b stop`.

10.3.3 Другие команды монитора

Несколько полезных команд монитора:

help	список доступных команд
quit	выход из эмулятора
xp /fmt addr	вывод содержимого физической памяти с адреса addr
system_reset	перезапуск

Команда `xp` требует некоторых пояснений. Аргумент `/fmt` указывает как будет выводиться содержимое памяти, и имеет синтаксис `<счетчик><формат><размер>`:

счетчик число элементов данных

size размер одного элемента в битах: b=8 бит, h=16, w=32, g=64

format определяет формат вывода:

- **x** hex
- **d** десятичные целые со знаком
- **u** десятичные без знака
- **o** 8ричные
- **c** символ (char)
- **i** инструкции ассемблера

Команда `xp` в формате `i` будет дизассемблировать инструкции из памяти. Выведем дамп с адреса 0x0 указав fmt=4iw: 4 — 4 , i — инструкции размером w=32 бита:

```
(qemu) xp /4wi 0x0
0x00000000:  e3a00005      mov  r0, #5  ; 0x5
0x00000004:  e3a01004      mov  r1, #4  ; 0x4
0x00000008:  e0812000      add  r2, r1, r0
0x0000000c:  eafffffe      b    0xc
```

10.4 Директивы ассемблера

В этом разделе мы посмотрим несколько часто используемых директив ассемблера, используя в качестве примера пару простых программ.

10.4.1 Суммирование массива

Следующий код 2 вычисляет сумму массива байт и сохраняет результат в r3:

Листинг 2: Сумма массива

```
        .text
entry:  b start                 @ перепрыгиваем данные
arr:    .byte 10, 20, 25        @ read-only массив байт
eoa:                            @ адрес конца массива + 1

        .align
start:
        ldr   r0, =eoa          @ r0 = &eoa
        ldr   r1, =arr          @ r1 = &arr
        mov   r3, #0            @ r3 = 0
```

```
loop:   ldrb  r2, [r1], #1      @ r2 = *r1++
        add   r3, r2, r3        @ r3 += r2
        cmp   r1, r0            @ if (r1 != r2)
        bne   loop              @ goto loop
stop:   b stop
```

В коде используются две новых ассемблерных директивы, описанных далее: `.byte` и `.align`.

`.byte`

Аргументы директивы `.byte` ассемблируются в последовательность байт в памяти. Также существуют аналогичные директивы `.2byte` и `.4byte` для ассемблирования 16- и 32-битных констант:

```
.byte   exp1, exp2, ...
.2byte  exp1, exp2, ...
.4byte  exp1, exp2, ...
```

Аргументом может быть целый числовой литерал: двоичный с префиксом 0b, 8-ричный префикс 0, десятичный и hex 0x. Также может использоваться строковая константа в одиночных кавычках, ассемблируемая в ASCII значения.

Также аргументом может быть Си-выражение из литералов и других символов, примеры:

```
pattern:  .byte 0b01010101, 0b00110011, 0b00001111
npattern: .byte npattern - pattern
halpha:   .byte 'A', 'B', 'C', 'D', 'E', 'F'
dummy:    .4byte 0xDEADBEEF
nalpha:   .byte 'Z' - 'A' + 1
```

`.align`

Архитектура ARM требует чтобы инструкции находились в адресах памяти, выровненных по границам 32-битного слова, т.е. в адресах с нулями в 2х младших разрядах. Другими словами, адрес каждого первого байта из 4-байтной команды, должен быть кратен 4. Для обеспечения этого предназначена директива `.align`, которая вставляет выравнивающие байты до следующего выровненного адреса. Ее нужно использовать только если в код вставляются байты или неполные 32-битные слова.

10.4.2 Вычисление длины строки

Этот код вычисляет длину строки и помещает ее в `r1`:

Листинг 3: Длина строки

```
        .text
        b start

str:    .asciz "Hello World"

        .equ   nul, 0

        .align
start:  ldr   r0, =str          @ r0 = &str
        mov   r1, #0

loop:   ldrb  r2, [r0], #1      @ r2 = *(r0++)
        add   r1, r1, #1        @ r1 += 1
        cmp   r2, #nul          @ if (r1 != nul)
```

```
        bne   loop              @ goto loop

        sub   r1, r1, #1        @ r1 -= 1
stop:   b stop
```

Код иллюстрирует применение директив `.asciz` и `.equ`.

`.asciz`

Директива `.asciz` принимает аргумент: строковый литерал, последовательность символов в двойных кавычках. Строковые литералы ассемблируются в память последовательно, добавляя в конец нулевой символ `\0` (признак конца строки в языке Си и стандарте POSIX).

Директива `.ascii` аналогична `.asciz`, но конец строки не добавляется. Все символы — 8-битные, кириллица может не поддерживаться.

`.equ`

Ассемблер при свой работе использует ***таблицу символов***: она хранит соответствия меток и их адресов. Когда ассемблер встречает очередное определение метки, он добавляет в таблицу новую запись. Когда встречается упоминание метки, оно заменяется соответствующим адресом из таблицы.

Использование директивы `.equ` позволяет добавлять записи в таблицу символов вручную, для привязки к именам любых числовых значений, не обязательно адресов. Когда ассемблер встречает эти имена, они заменяются на их значения. Эти имена-константы, и имена меток, называются ***сиволами***, а таблицы записанные в объектные файлы, или в отдельные .sym файлы — ***таблицами символов***[19].

Синтаксис директивы .equ:

[19] также используются отладчиком, чтобы показывать адреса переходов в виде понятных программисту сиволов, а не мутных числовых констант

```
.equ <имя>, <выражение>
```

Имя символа имеет те же ограничения по используемым символам, что и метка. Выражение может быть одиночным литералом или выражением как и в директиве `.byte`.

> В отличие от .byte, директива .equ не выделяет никакой памяти под аргумент. Она только добавляет значение в таблицу символов.

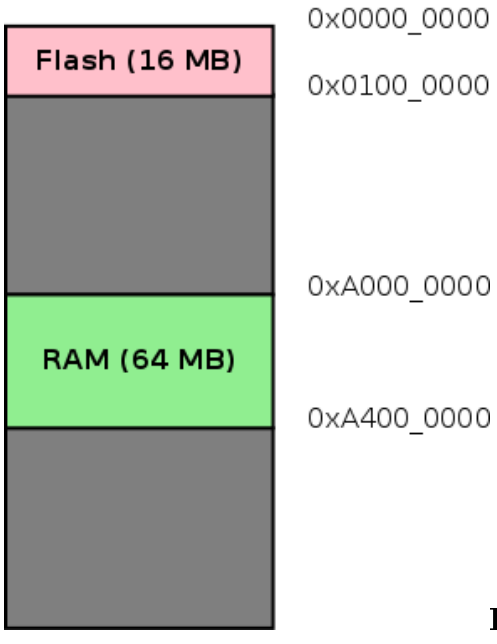
10.5 Использование ОЗУ (адресного пространства процессора)

Flash-память описанная ранее, в которой хранится машинный код программы, один из видов EEPROM[20]. Это вторичный носитель данных, как например жесткий диск, но в любом случае хранить данные и значения переменных во флеше неудобно как с точки зрения возможности перезаписи, так и прежде всего со скоростью чтения флеша, и кешированием.

В предыдущем примере мы использовали флеш как EEPROM для хранения константного массива байт, но вообще переменные должны храниться в максимально быстрой и неограниченно перезаписываемой RAM.

Плата connex имеет 64Mb ОЗУ начиная с адреса `0xA0000000`, для хранения данных программы. Карта памяти может быть представлена в виде диаграммы:

[20] Electrical Eraseable Programmable Read-Only Memory, электрически стираемая перепрограммируемая память только-для-чтения



Карта памяти Gumstix connex

[21]

Для настройки размещения переменных по нужным физическим адресам **нужна** некоторая **настройка адресного пространства**, особенно **если вы используете внешнюю память и переферийные устройства, подключемые к внешней шине**[22].

Для этого нужно понять, какую роль в распределении памяти играют ассемблер и линкер.

[21] здесь адреса считаются сверху вниз, что нетипично, обычно на диаграммах памяти адреса увеличиваются снизу вверх.

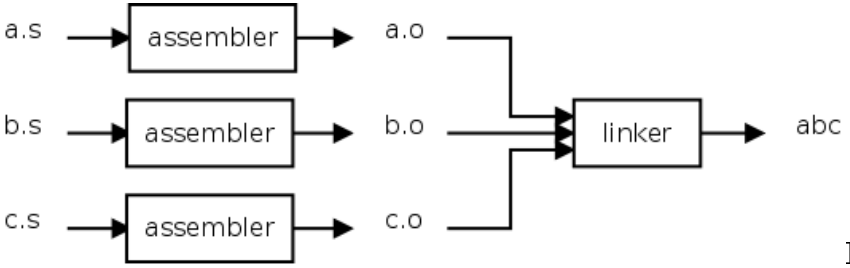
[22] или используете малораспространенные клоны ARM-процессоров, типа Миландровского 1986BE9x “чернобыль”

10.6 Линкер

Линкер позволяет ***скомпоновать*** исполняемый файл программы из нескольких объектных файлов, поделив ее на части. Чаще всего это нужно при использовании нескольких компиляторов для разных языков программирования: ассемблер, компиляторы C_+^+, Фортрана и Паскакаля.

Например, очень известная библиотека численных вычислений на базе матриц BLAS/LAPACK написана на Фортране, и для ее использования с сишной программой нужно слинковать program.o, blas.a и lapack.a[23] в один исполняемый файл.

При написании многофайловой программы (еще это называют ***инкрементной компоновкой***) каждый файл исходного кода ассемблируется в индивидуальный файл объектного кода. Линкер[24] собирает объектные файлы в финальный исполняемый бинарник.



Роль линкера

При сборке объектных файлов, линкер выполняет следующие операции:

- symbol resolution (разрешение символов)
- relocation (релокация)

В этой секции мы детально рассмотрим эти операции.

[23] .a — файлы архивов из пары сотен отдельных .o файлов каждый, по одному .o файлу на каждый возможный вариант функции линейной алгебры

[24] или компоновщик

10.6.1 Разрешение символов

В программе из одного файла при создании объектного файла все ссылки на метки заменяются их адресами непосредственно ассемблером. Но в программе из нескольких файлов существует множество ссылок на метки в других файлах, неизвестные на момент ассемблирования/компиляции, и ассемблер помечает их “unresolved” (неразреше́нные). Когда эти объектные файлы обрабатываются линкером, он определяет адреса этих меток по информации из других объектных файлов, и корректирует код. Этот процесс называется ***разрешением символов***.

Пример суммирования массива разделен на два файла для демонстрации разрешения символов, выполняемых линкером. Эти файлы ассемблируются отдельно, чтобы их таблицы символов содержали неразрешенные ссылки.

Файл **sumsub.s** содержит процедуру суммирования, а **summain.s** вызывает процедуру с требуемыми аргументами:

Листинг 4: summain.s вызов внешней процедуры

```
        .text
        b start                 @ пропустить данные
arr:    .byte 10, 20, 25        @ константный массив байт
eoa:                            @ адрес массива + 1
        .align
start:
        ldr   r0, =arr          @ r0 = &arr
        ldr   r1, =eoa          @ r1 = &eoa
        bl    sum               @ (вложенный) вызов процедуры
stop:   b stop
```

Листинг 5: sumsub.s код процедуры

```
        @ Аргументы (в регистрах)
        @ r0: начальный адрес массива
        @ r1: конечный адрес массива
        @
        @ Возврат результата
        @ r3: сумма массива

        .global sum

sum:    mov   r3, #0            @ r3 = 0
loop:   ldrb  r2, [r0], #1      @ r2 = *r0++    ; получить следующий элемент
        add   r3, r2, r3        @ r3 += r2      ; суммирование
        cmp   r0, r1            @ if (r0 != r1) ; проверка на конец массива
        bne   loop              @ goto loop     ; цикл
        mov   pc, lr            @ pc = lr       ; возврат результата по готовности
```

[25]

Применение директивы `.global` обязательно. В Си все функции и переменные, определенные вне функций, считаются видимыми из других объектных файлов, если они не определены с модификатором `static`. В ассемблере наоборот все метки считаются *статическими*[26], если с помощью директивы `.global` специально не указано, что они должны быть видимы извне.

Ассемблируйте файлы, и посмотрите дамп их таблицы символов используя комунду **nm**:

```
$ arm-none-eabi-as -o main.o main.s
```

[25] в архитектуре ARM нет специальной команды возврата ret, ее функцию выполняет прямое присваивание регистра указателя команд mov pc,lr

[26] или локальными на уровне файла

```
$ arm-none-eabi-as -o sum-sub.o sum-sub.s
$ arm-none-eabi-nm main.o
00000004 t arr
00000007 t eoa
00000008 t start
00000018 t stop
         U sum
$ arm-none-eabi-nm sum-sub.o
00000004 t loop
00000000 T sum
```

Теперь сфокусируемся на букве во втором столбце, который указывает тип символа: `t` указывает что символ определен в секции кода `.text`, `u` указывает что символ не определен. Буква в верхнем регистре указывает что символ глобальный и был указан в директиве `.global`.

Очевидно, что символ `sum` определ в **sum-sub.o** и еще не разрешен в **main.o**. Вызов линкера разрешает символьные ссылки, и создает исполняемый файл.

10.6.2 Релокация

Релокация — процесс изменения уже назначенных меткам адресов. Он также выполняет коррекцию всех ссылок, чтобы отразить заново назначенные адреса меток. В общем, релокация выполняется по двум основным причинам:

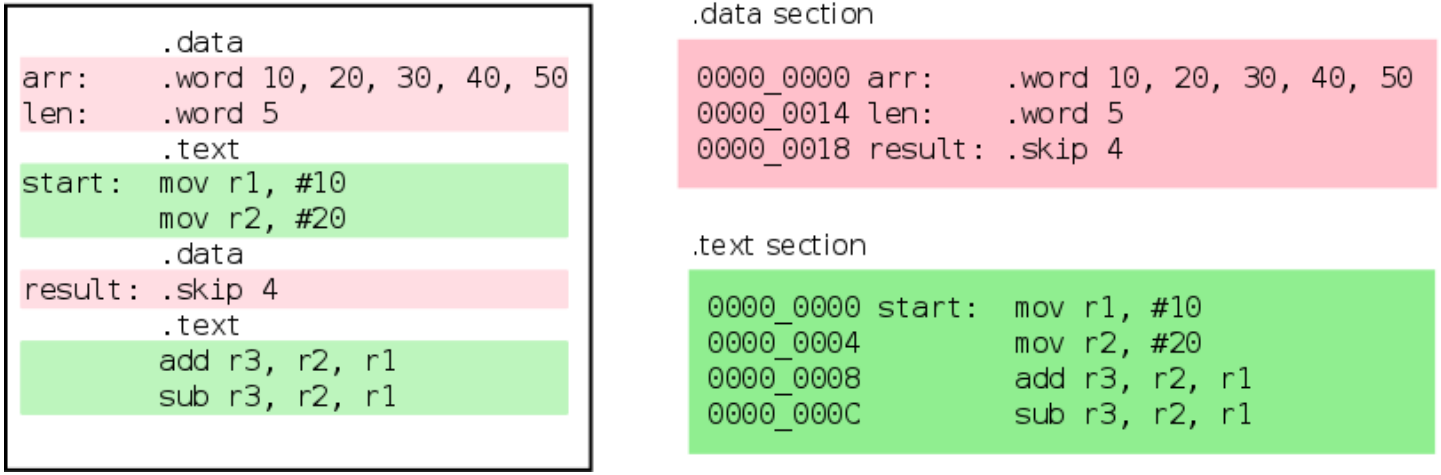
1. Объединение секций
2. Размещение секций

Для понимания процеса релокации, нужно понимание самой концепции секций.

Код и данные отличаются по требованиям при исполнении. Например код может размещаться в ROM-памяти, а данные требуют память открытую на запись. Очень хорошо, если **области кода и данных**

не пересекаются. Для этого программы делятся на секции. Большиство программ имеют как минимум две секции: `.text` для кода и `.data` для данных. Ассемблерные директивы `.text` и `.data` ожидаемо используются для переключения между этими секциями.

Хорошо представить каждую секцию как ведро. Когда ассемблер натыкается на директиву секции, он начинает сливать код/данные в соответствующее ведро, так что они размещаются в смежных адресах. Эта диаграмма показывает как ассемблер упорядочивает данные в секциях:



Секции

Теперь, когда у нас есть общее понимание ***секционирования*** кода и данных, давайте посмотрим по каким причинам выполняется релокация.

Объединение секций

Когда вы имеете дело с многофайловыми программами, секции в каждом объектном файле имеют одинаковые имена (‘.text‘,. . .), линкер отвечает за их объединение в выполняемом файле. По умолчанию секции

с одинаковыми именами из каждого **.o** файла объединяются последовательно, и метки корректируются на новые адреса.

Эффекты объединения секций можно посмотреть, анализируя таблицы символов отдельно в объектных и исполняемом файлах. Многофайловая программа суммирования может иллюстрировать объединение секций. Дампы таблиц символов:

```
$ arm-none-eabi-nm main.o
00000004 t arr
00000007 t eoa
00000008 t start
00000018 t stop
         U sum
$ arm-none-eabi-nm sum-sub.o
00000004 t loop <1>
00000000 T sum
$ arm-none-eabi-ld -Ttext=0x0 -o sum.elf main.o sum-sub.o
$ arm-none-eabi-nm sum.elf
...
00000004 t arr
00000007 t eoa
00000008 t start
00000018 t stop
00000028 t loop <1>
00000024 T sum
```

1. символ `loop` имеет адрес `0x4` в **sum-sub.o**, и `0x28` в **sum.elf**, так как секция `.text` из **sum-sub.o** размещена сразу за секцией `.text` из **main.o**.

Размещение секций

Когда программа ассемблируется, каждой секции назначается стартовый адрес `0x0`. Поэтому всем переменным назначаются адреса относительно начала секции. Когда создается финальный исполнямый файл, секция размещаются по некоторому адресу `X`, и все адреса меток, назначенные в секции, увеличиваются на `X`, так что они указывают на новые адреса.

Размещение каждой секции по определенному месту в памяти и коррекцию всех ссылок на метки в секции, выполняет линкер.

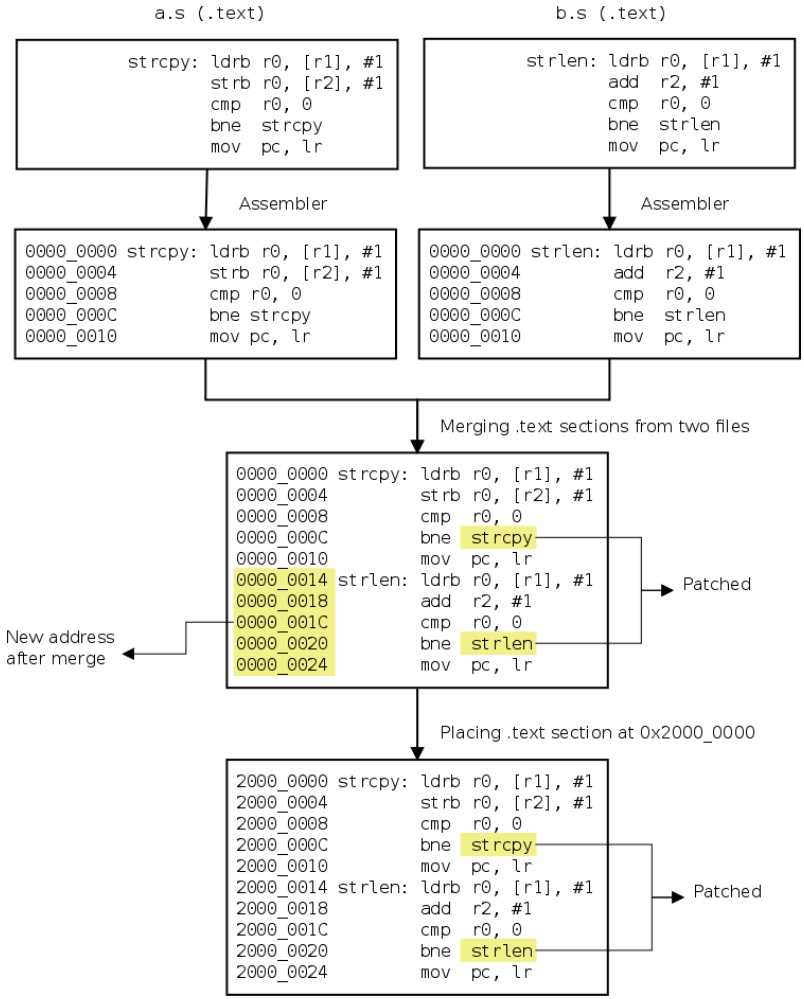
Эффект размещения секций можно увидеть по тому же дампу символов, описанному выше. Для простоы используем объектный файл однофайловой программы суммирования **sum.o**. Для увеличения заметности искуственно разместим секцию `.text` по адресу `0x100`:

```
$ arm-none-eabi-as -o sum.o sum.s
$ arm-none-eabi-nm -n sum.o
00000000 t entry <1>
00000004 t arr
00000007 t eoa
00000008 t start
00000014 t loop
00000024 t stop
$ arm-none-eabi-ld -Ttext=0x100 -o sum.elf sum.o <2>
$ arm-none-eabi-nm -n sum.elf
00000100 t entry <3>
00000104 t arr
00000107 t eoa
00000108 t start
00000114 t loop
00000124 t stop
```

...

1. Адреса меток назначаются с `0` от начала секции.
2. Когда создается выполняемый файл, линкеру указано разместить секцию кода с адреса `0x100`.
3. Адреса меток в `.text` переназначаются начиная с `0x100`, и все ссылки на метки корректируются.

Процесс объединения и размещения секций в общем показаны на диаграмме:



Объединение и размещение секций

10.7 Скрипт линкера

Как было описано в предыдущем разделе, объединение и размещение секций выполняется линкером. Программист может контролировать этот процесс через ***скрипт линкера***. Очень простой пример скрипта линкера:

Листинг 6: Простой скрипт линкера

```
SECTIONS { <1>
. = 0x00000000; <2>
.text : { <3>
abc.o (.text);
def.o (.text);
} <3>
}
```

1. Команда `SECTIONS` наиболее важная команда, она указывает как секции объединяются, и по каким адресам они размещаются.

2. Внутри блока `SECTIONS` команда `.` (точка) представляет ***указатель адреса размещения***. Указатель адреса всегда инициализируется `0x0`. Его можно модифицировать явно присваивая новое значение. Показанная явная установка на `0x0` на самом деле не нужна.

3. Этот блок скрипта определяет что секция `.text` выходного файла составляется из секций `.text` в файлах **abc.o** и **def.o**, причем именно в этом порядке.

Скрипт линкера может быть значительно упрощен и универсализирован с помощью использования символа шаблона `*` вместо индивидуального указания имен файлов:

Листинг 7: Шаблоны в скриптах линкера

```
SECTIONS {
. = 0x00000000;
.text : { * (.text); }
}
```

Если программа одновременно содержит секции ‘.text‘ и ‘.data‘, объединение и размещение секций можно прописать вот так:

Листинг 8: Несколько секций

```
SECTIONS {
 . = 0x00000000;
 .text : { * (.text); }

 . = 0x00000400;
 .data : { * (.data); }
}
```

Здесь секция `.text` размещается по адресу `0x0`, а секция `.data` — `0x400`. Отметим, что если указателю размещения не приваивать значения, секции `.text` и `.data` будут размещены в смежных областях памяти.

10.7.1 Пример скрипта линкера

Для демонстрации использования скриптов линкера рассмотрим применение скрипта **??** для размещения секций `.text` и `.data`. Для этого используем немного измененный пример программы суммирования массива, разделив код и данные в отдельные секции:

Листинг 9: Программа суммирования массива

```
.data
arr: .byte 10, 20, 25 @ Read-only array of bytes
eoa: @ Address of end of array + 1

.text
start:
ldr   r0, =eoa   @ r0 = &eoa
ldr   r1, =arr @ r1 = &arr
mov   r3, #0 @ r3 = 0
loop: ldrb  r2, [r1], #1 @ r2 = *r1++
add   r3, r2, r3 @ r3 += r2
cmp   r1, r0 @ if (r1 != r2)
bne   loop @    goto loop
stop: b stop
```

1. Изменения заключаются в выделении массива в секцию `.data` и удалении директивы выравнивания `.align`.

2. Также не требуется инструкция перехода на метку `start` для обхода данных, так как линкер разместит секции отдельно. В результате команды программы размещаются любым удобным способом, а скрипт линкера позаботится о правильном размещении сегментов в памяти.

При линковке программы в командной строке нужно указать использования скрипта:

```
$ arm-none-eabi-as -o sum-data.o sum-data.s
$ arm-none-eabi-ld -T sum-data.lds -o sum-data.elf sum-data.o
```

Опция `-T sum-data.lds` указывает что используется скрипт **sum-data.lds**. Выводим таблицу символов и видим размещение сегментов в памяти:

```
$ arm-none-eabi-nm -n sum-data.elf
00000000 t start
0000000c t loop
0000001c t stop
00000400 d arr
00000403 d eoa
```

Из таблицы символолов видно что секция `.text` размещена с адреса `0x0`, а секция `.data` с `0x400`.

10.7.2 Анализ объектного/исполняемого файла утилитой objdump

Более подробную информацию даст утилита **objdump**:

```
$ arm-none-eabi-as -o sum-data.o sum-data.s
$ arm-none-eabi-ld -T sum-data.lds -o sum-data.elf sum-data.o
$ arm-none-eabi-objdump sum-data.elf
```

Листинг 10: sum-data.objdump

1. указание на архитектуру,
2. для которой предназначен исполняемый файл
3. стартовый адрес в секции `.text`, по умолчанию `0x0`[27]

[27] обязателен и фиксирован для прошивок микроконтроллеров, т.к. на него перескакивает аппаратный сброс

4. ***ABI*** — соглашения о передаче
5. параметров в регистрах/стеке (для Си кода)
6. приведена подробная информация о секциях
7. `.text` секция кода
8. `.data` секция данных
9. служебная информация
10. столбец `Size` указывает размер секции в байтах (hex)
11. `VMA`[28] указывает адрес размещения сегмента
12. `Algn` (Align) автоматическое выравнивание содержимого сегмента в памяти, в степени двойки `2**n`: код выравнивается кратно `2**2=4` байтам, данные не выравниваются `2**0=1`
13. Флаг `ALLOC` (Allocate) указывает что при загрузке программы под этот сегмент должна быть выделена память.
14. `LOAD` указывает что содержимое сегмента должно загружаться из исполняемого файла в память при использовании ОС, а для микроконтроллеров указывает программатору что сегмент нужно прошивать.
15. `READONLY` сегмент с константными неизменяемыми данными, которые могут быть размещены в ROM, а при использовании ОС область памяти должна быть помечена в таблице системы защиты памяти как R/O. Отсутствие флага `READONLY` + наличие `LOAD` указывает что данные должны загружаться **только в ОЗУ**.

[28] Virtual Memory Address

16. сегмент кода
17. сегмент данных
18. таблица символов
19. дизассемблированный код из секций, помеченных флагом `CODE`: `.text`

10.8 Данные в RAM, пример

Теперь мы знаем как писать скрипты линкера, и можем попытаться написать программу, разместив данные в секции `.data` в ОЗУ.

Программа сложения модифицирована для загрузки значений из ОЗУ, и записи результата обратно в ОЗУ: память для операндов и результат резмещена в секции `.data`.

Листинг 11: Данные в ОЗУ

```
.data
val1: .4byte 10 @ First number
val2: .4byte 30 @ Second number
result: .4byte 0 @ 4 byte space for result

.text
.align
start:
ldr   r0, =val1   @ r0 = &val1
ldr   r1, =val2 @ r1 = &val2
```

```
ldr   r2, [r0] @ r2 = *r0
ldr   r3, [r1] @ r3 = *r1

add   r4, r2, r3 @ r4 = r2 + r3

ldr   r0, =result @ r0 = &result
str   r4, [r0] @ *r0 = r4

stop: b stop
```

Листинг 12: Скрипт для линковки

```
SECTIONS {
. = 0x00000000;
.text : { * (.text); }

. = 0xA0000000;
.data : { * (.data); }
}
```

Дамп таблицы символов:

```
$ arm-none-eabi-nm -n add-mem.elf
00000000 t start
0000001c t stop
a0000000 d val1
a0000001 d val2
a0000002 d result
```

Срипт линкера решил проблему с размещением данных, но **проблема с использованием ОЗУ еще не решена !**

10.8.1 RAM энергозависима (volatile) !

ОЗУ стирается при отключении питания, поэтому для использования ОЗУ недостаточно разместить сегменты.

> **Во флеше должнен храниться** не только код, но **и данные**, чтобы при подаче питания специальный ***startup код*** выполнил **инициалиацию ОЗУ**, копируя данные из флеша. Затем управление передается основной программе.

Поэтому секция `.data` имеет **два адреса размещения**: ***адрес хранения*** во флеше ***LMA*** и ***адрес размещения*** в ОЗУ ***VMA***.

TIP: как видно из раздела **??**, в терминах **ld** адрес хранения (загрузки) называется **LMA** (Load Memory Address), а адрес размещения (времени выполнения) **VMA** (Virtual Memory Address).

Нужно сделать следующие две мсодификации, чтобы программа работала корректно:

1. модифицировать .lds чтобы для секции `.data` в нем учитывались оба адреса: LMA и VMA.
2. написать небольшой кусочек кода, который будет **инициализировать память данных**, копируя образ секции `.data` из флеша (из адреса хранения LMA) в ОЗУ (по адресу исполнения, VMA).

10.8.2 Спецификация адреса загрузки LMA

VMA это адрес, который должен быть использован для вычисления адресов всех меток при исполнении программы. В предыдущем линк-скрипте мы задали VMA секции `.data`. LDA не указан, и по умолчанию

равен VMA. Это нормально для сегментов, размещаемых в ROM. Но если используются инициализируемые сегменты в ОЗУ, нужно задать отдельно VMA и LMA.

Адрес загрузки LMA, отличающийся от адреса выполнения VMA, задается с помощью команды `AT`. Модифицированный скрипт показан ниже:

```
SECTIONS {
. = 0x00000000;
.text : { * (.text); }
etext = .; <1>

. = 0xA0000000;
.data : AT (etext) { * (.data); } <2>
}
```

1. В блоках описания секций можно создавать символы, назначая им значения: числовой адрес или текущую позицию с помощью точки ".". Символу `etext` назначается адрес флеша, следующий сразу за концом кода. Отметим что `etext` сам по себе не выделяет никакой памяти, а только помечает адрес LMA сегмента .data в таблице символов.

2. При настройке сегмента `.data` с помощью ключевого слова `AT (etext)` назначается LMA для хранения содержимого сегмента данных. Команде `AT` может быть передан любой адрес или символ[29]. Так что в результате мы настроили адрес хранения `.data` на область флеша, помеченную символов `etext`.

10.8.3 Копирование ‘.data‘ в ОЗУ

Для копирования данных инициализации из флеша в ОЗУ требуется следующая информация:

[29] значением которого является валидный адрес

1. Адрес данных во флеше (`flash_sdata`)
2. Адрес данных в ОЗУ (`ram_sdata`)
3. Размер секции `.data` (`data_size`)

Имея эту информацию, сегмент `.data` может быть инициализирован может быть скопирован следующим стартовым кодом:

```
ldr   r0, =flash_sdata
ldr   r1, =ram_sdata
ldr   r2, =data_size
copy:
ldrb  r4, [r0], #1
strb  r4, [r1], #1
subs  r2, r2, #1
bne   copy
```

Для получения такой информации скрипт линкера нужно немного модифицировать:

Листинг 13: Скрипт линкера с символами для копирования секции `.data`

```
SECTIONS {
. = 0x00000000;
.text : { * (.text); }
flash_sdata = .; <1>

. = 0xA0000000;
ram_sdata = .; <2>
.data : AT (flash_sdata) { * (.data); }
ram_edata = .; <3>
```

```
data_size = ram_edata - ram_sdata; <3>
}
```

1. Начало данных во флеше сразу за секцией кода.

2. Начало данных — базовый адрес ОЗУ в адресном пространстве процессора.

3. Получение размера непросто: размер вычисляется вычитанием адресов метод начала и конца данных. Да, простые выражения тоже можно использовать в скрипте линкера.

Полный листинг программы с добавленной инициализацией данных:

Листинг 14: Инициализация ОЗУ

```
.data
val1: .4byte 10 @ First number
val2: .4byte 30 @ Second number
result: .space 4 @ 1 byte space for result

.text

;; Copy data to RAM.
start:
ldr   r0, =flash_sdata
ldr   r1, =ram_sdata
ldr   r2, =data_size

copy:
```

```
ldrb  r4, [r0], #1
strb  r4, [r1], #1
subs  r2, r2, #1
bne   copy

;; Add and store result.
ldr   r0, =val1   @ r0 = &val1
ldr   r1, =val2 @ r1 = &val2

ldr   r2, [r0] @ r2 = *r0
ldr   r3, [r1] @ r3 = *r1

add   r4, r2, r3 @ r4 = r2 + r3

ldr   r0, =result @ r0 = &result
str   r4, [r0] @ *r0 = r4

stop: b stop
```

Листинг 15: add-ram.objdump Программа была ассемблирована и слинкована используя .lds в **??**.

Запуск и тестирование программы в Qemu:

```
qemu-system-arm -M connex -pflash flash.bin -nographic -serial /dev/null
(qemu) xp /4dw 0xA0000000
a0000000:         10         30         40          0
```

На реальной физической системе с SDRAM, память не может использована сразу. Сначала нужно инициализировать контроллер памяти, и только затем обращаться к ОЗУ. Наш код работает потому, что симулятор не требует инициализации контроллера.

10.9 Обработка аппаратных исключений

Все примеры программ, приведенные выше, содержат гигантский баг: **первые 8 машинных слов в адресном пространстве зарезервированы для векторов исключений**. Когда возникает исключение, выполняется аппаратный переход на один из этих жестко заданных меток. Исключения и их адреса приведены в следующей таблице:

Адреса векторов исключений

Исключение		Адрес
Сброс	Reset	0x00
Неопределенная инструкция	Undefined Instruction	0x04
Программное прерывание (SWI)	Software Interrupt (SWI)	0x08
Ошибка предвыборки	Prefetch Abort	0x0C
Ошибка данных	Data Abort	0x10
Резерв, не используется	Reserved, not used	0x14
Аппаратное прерывание	IRQ	0x18
Быстрое прерывание	FIQ	0x1C

Предполагается что по этим адресам находятся команды перехода, которые передадут управление на соответствующий произвольный адрес обработчика исключения. Во всех примерах ранее бы не вставляли таблицу обработчиков исключений, так как мы предполагали что эти исключения не случатся. Все эти программы можно скорректировать, слинковав их со следующим кодом:

```
.section "vectors"
reset: b     start
undef:  b     undef
swi: b     swi
pabt: b     pabt
dabt: b     dabt
nop
irq: b     irq
fiq: b     fiq
```

Только обработчик `reset` векторизован на отдельный адрес `start`. Все остальные исключения векторизованы сами на себя. Таким образом если случится любое исключение, процессор зациклится на соответствущем векторе. В этом случае возникшее исключение может быть идентифицирвовано в отладчике (мониторе Qemu, в нашем случае) по адресу указателя команд `pc=r15`.

В ассемблерном коде видно применение директивы `.section` которая позволяет создавать секции с произвольными именами, чтобы прописать для них отдельную обработку в скрипте линкера.

Чтобы обеспечить правильное размещение таблицы обработчиков, нужно скорректировать скрипт линкера:

```
SECTIONS {
. = 0x00000000;
.text : {
* (vectors);
* (.text);
...
}
...
}
```

Обратите внимание что секция `vectors` размещена сразу за инициализацией указателя размещения на первом месте, до всего остального кода, что гарантирует что таблица векторов будет находится по жесткому адресу `0x0`.

10.10 Стартап-код на Си

Если процесс только что был сброшен, невозможно напрямую выполниь Си-код, так как в отличие от ассемблера, программы на Си требуют для себя некоторой предварительной настройки. В этом разделе описаны эти предварительные требования, и как их выполнить.

Мы возьмем пример Си-программы которая вычисляет сумму массива. И к концу раздела мы уже будем способны, сделав некоторые низкоуровневые настройки, передать управление и выполнить ее.

Листинг 16: Сумма массива на Си

```
static int arr[] = { 1, 10, 4, 5, 6, 7 };
static int sum;
static const int n = sizeof(arr) / sizeof(arr[0]);

int main()
{
int i;

for (i = 0; i < n; i++)
sum += arr[i];
}
```

Перед передачей управления Си-коду, нужно выполнить следующие настройки:

1. Стек
2. Глобальные переменные
   (a) Инициализированные
   (b) Неинициализированные
3. Константные данные

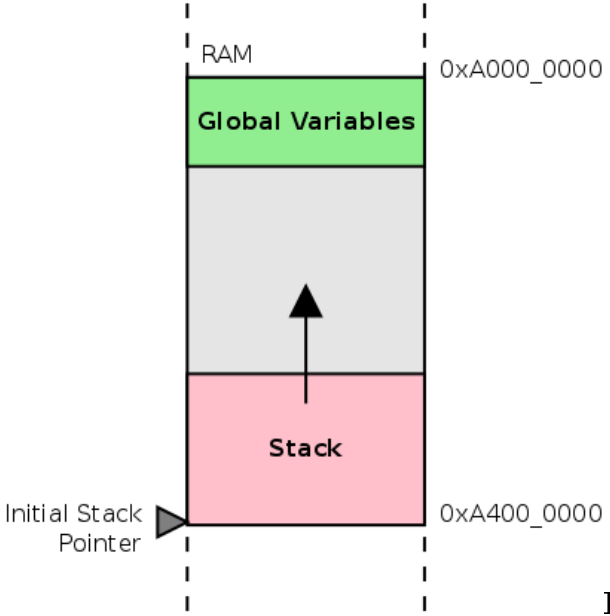
10.10.1 Стек

Си использует стек для хранения локальных (авто) переменных, передачи аргументов и результата функций, хранения адресов возврата из функций и т.д. Так что необходимо чтобы стек был настроен корректно перед передачей управление Си-коду.

На архитектуре ARM стеки очень гибкие, поэтому их реализация польностью ложиться на программное обеспечение. Для людей не знакомых с ARM, некоторый обзор приведен в **??**.

Чтобы быть уверенным, что разные части кода, сгенерированного **разными** компиляторами, работали друг с другом, ARM создал Стандарт вызова процедур для архитектуры ARM (AAPCS). В нем описаны регистры которые должны быть использованы для работы со стеком и направление в котором растет стек. Согласно AAPCS, **регистр** `r13` должен быть использован для указателя стека. Также стек должен быть для указателя стека. Также стек должен быть **full-descending** (нисходящим).

Один из способов размещения глобальных переменных на стеке показан в диаграмме:



Размещение стека

Так что все, что нужно сделать в стартовом коде для стека — выставить `r13` на старший адрес ОЗУ, так что стек может расти вниз (в сторону младших адресов). Для платы `connex` это можно сделать командой

```
ldr sp, =0xA4000000
```

Обратите внимание что ассемблер предоставляет алиас `sp` для регистра `r13`.

Адрес `0xA4000000` сам по себе не указывает на ОЗУ. ОЗУ кончается адресом `0xA3FFFFFF`. Но это нормально, так как стек **full**-descending, т.е. во время первого `push` в стек указатель **сначала уменьшится**, и только потом значение будет записано уже в ОЗУ.

10.10.2 Глобальные переменные

Когда компилируется Си-код, компилятор размещает инициализированные глобальные переменные в секцию `.data`. Как и для ассемблера, сегмент `.data` должен быть скопирован стартовым кодом в ОЗУ из флеша.

Язык Си гарантирует что все неинициализированные глобальные переменные будут инициализированы нулем[30]. Когда Си-программа компилируется, создается отдельный сегмент `.bss` для неинициализированных переменных. Так как для всего сегмента должно быть выполнено обнуление, его не нужно хранить во флеше. Перед передачей управления на Си-код, содержимое `.bss` дожно быть зачищено startup-кодом.

10.10.3 Константные данные

GCC размещает переменные, помеченные модификатором `const`, в отдельный сегмент `.rodata`. Также `.rodata` используется для хранения всех "строковых констант".

Так как содержимое `.rodata` не модифицируется, оно может быть размещено в Flash/ROM. Для этого нужно модифицировать .lds.

10.10.4 Секция `.eeprom` (AVR8)

При написании прошивок для Atmel AVR8, существует модификатор `EEMEM` определенный в **avr/eeprom.h**:

```
#define EEMEM __attribute__((section(".eeprom")))
```

который использует модификатор GCC `__attribute__((section("...")))`, который приписывает объект данных к любой указанной секции. В частности, секция `.eeprom` выделяется из финального объектного файла, и программируется в Atmel ATmega отдельным вызовом `avrdude` (ПО программатора).

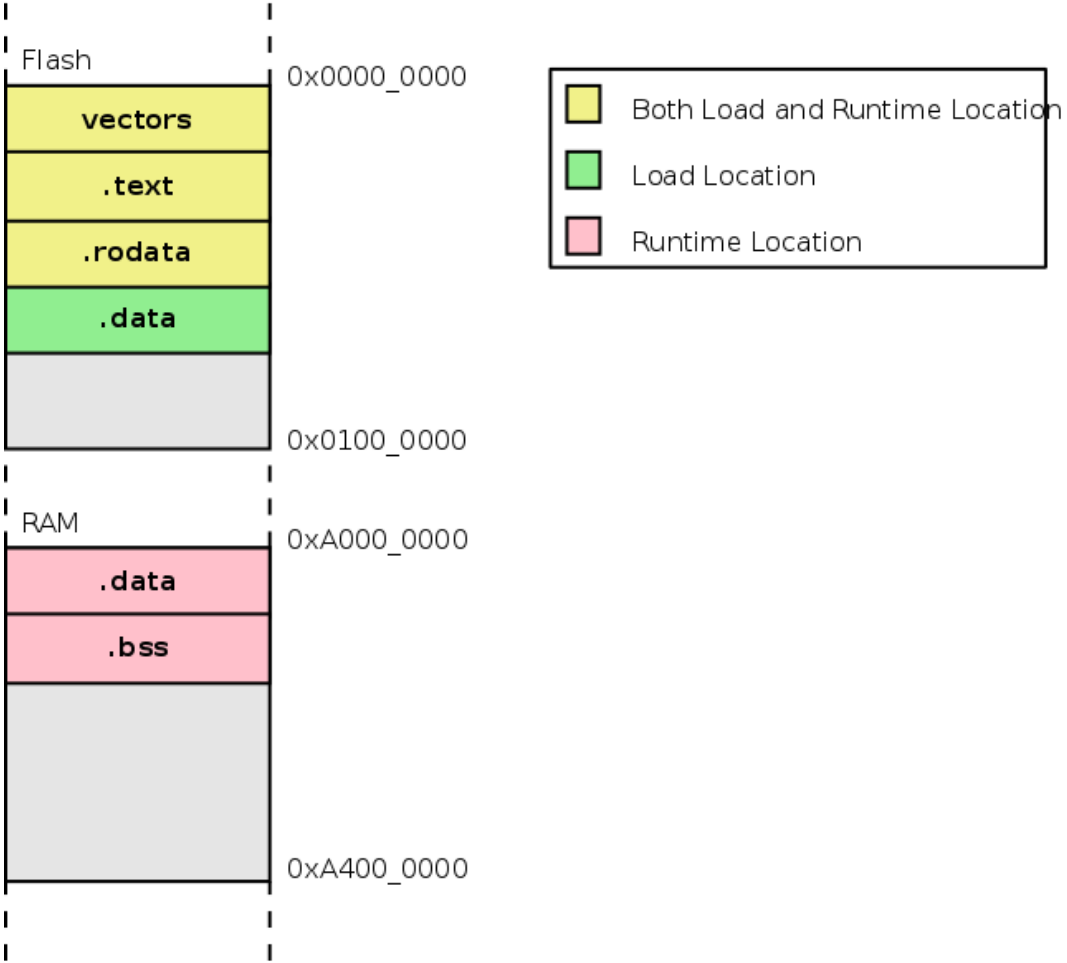
[30] старый стандарт Си не гарантировал

10.10.5 Стартовый код

Теперь все готово к написанию скрипта линкера и стартового кода. Скрипт **??** модифицируется с добавлением размещения секций:

1. `.bss`
2. `vectors`
3. `.rodata`

Секция `.bss` размещается сразу за секцией `.data` в ОЗУ. Также создаются символы маркирующие начало и конец секции `.bss`, которые будут использованы в startup-коде при ее очистке. `.rodata` размещается сразу за `.text` во флеше:



Размещение секций

Листинг 17: Скрипт линкера для Си кода

```
SECTIONS {
. = 0x00000000;
.text : {
      * (vectors);
      * (.text);
}
.rodata : {
      * (.rodata);
}
flash_sdata = .;

. = 0xA0000000;
ram_sdata = .;
.data : AT (flash_sdata) {
      * (.data);
}
ram_edata = .;
data_size = ram_edata - ram_sdata;

sbss = .;
.bss : {
     * (.bss);
}
ebss = .;
bss_size = ebss - sbss;
```

```
}
```

Startup-код включает следующие части:

1. вектора исключений
2. код копирования `.data` из Flash в RAM
3. код обнуления `.bss`
4. код установки указателя стека
5. переход на `_main`

Листинг 18: Стартовый код для Си программы на ассемблере

```
.section "vectors"
reset: b     start
undef:  b     undef
swi: b     swi
pabt: b     pabt
dabt: b     dabt
nop
irq: b     irq
fiq: b     fiq

.text
start:
@@ Copy data to RAM.
ldr   r0, =flash_sdata
ldr   r1, =ram_sdata
ldr   r2, =data_size
```

```
@@ Handle data_size == 0
cmp   r2, #0
beq   init_bss
copy:
ldrb   r4, [r0], #1
strb   r4, [r1], #1
subs   r2, r2, #1
bne    copy

init_bss:
@@ Initialize .bss
ldr   r0, =sbss
ldr   r1, =ebss
ldr   r2, =bss_size

@@ Handle bss_size == 0
cmp   r2, #0
beq   init_stack

mov   r4, #0
zero:
strb  r4, [r0], #1
subs  r2, r2, #1
bne   zero

init_stack:
@@ Initialize the stack pointer
```

```
ldr   sp, =0xA4000000

bl    main

stop: b     stop
```

Для компиляции кода не требуется отдельно вызывать ассемблер, линкер и компилятор Си: программа **gcc** является оберткой, которая умеет делать это сама, автоматически вызывая ассемблер, компилятор и линкер в зависимости от типов файлов. Поэтому мы можем скомпилировать весь наш код одной командой:

```
$ arm-none-eabi-gcc -nostdlib -o csum.elf -T csum.lds csum.c startup.s
```

Опция `-nostdlib` используется для указания, что нам при компиляции не нужно подключать стандартную библиотеку Си (**newlib**). Эта библиотека крайне полезна, но для ее использования нужно выполнить некоторые дополнительные действия, описанные в разделе **??**.

Дамп таблицы символов даст больше информации о расположении объектов в памяти:

```
$ arm-none-eabi-nm -n csum.elf
00000000 t reset <1>
00000004 A bss_size
00000004 t undef
00000008 t swi
0000000c t pabt
00000010 t dabt
00000018 A data_size
00000018 t irq
0000001c t fiq
00000020 T main
```

```
00000090 t start <2>
000000a0 t copy
000000b0 t init_bss
000000c4 t zero
000000d0 t init_stack
000000d8 t stop
000000f4 r n <3>
000000f8 A flash_sdata
a0000000 d arr <4>
a0000000 A ram_sdata
a0000018 A ram_edata
a0000018 A sbss
a0000018 b sum <5>
a000001c A ebss
```

1. `reset` и остальные вектора исключений размещаются с `0x0`.
2. ассемблерный код находится сразу после 8 векторов исключений `( 8 * 4 = 32 = 0x20 )`.
3. константные данные `n`, размещены во флеше после кода.
4. инициализированные данные `arr`, массив из 6 целых, размещен с начала ОЗУ `0xA0000000`.
5. неинициализированные данные `sum` размещен после массива из 6 целых. `( 6 * 4 = 24 = 0x18 )`

Для выполнения программы преобразуем ее в `.bin` формат, запустим в **Qemu**, и выведем дамп переменной `sum` по адресу `0xA0000018`:

```
$ arm-none-eabi-objcopy -O binary csum.elf csum.bin
$ dd if=/dev/zero of=flash.bin bs=4K count=4K
$ dd if=csum.bin of=flash.bin bs=4096 conv=notrunc
$ qemu-system-arm -M connex -pflash flash.bin -nographic -serial /dev/null
(qemu) xp /6dw 0xa0000000
a0000000:           1         10          4          5
a0000010:           6          7
(qemu) xp /1dw 0xa0000018
a0000018:          33
```

10.11 Использование библиотеки Си

FIXME: Эту секцию еще нужно написать.

10.12 Inline-ассемблер

FIXME: Эту секцию еще нужно написать.

10.13 Использование ‘make‘ для автоматизации компиляции

Если вам надоело каждый раз вводить длинные команды, компилируя примеры из этого учебника, пришло время научиться пользоваться утилитой **make**. **make** это утилита, отслеживающая зависимости файлов, описанные в файле **Makefile**.

Умение читать и писать makefайлы **must have** навык для программиста, особенно для больших многофайловых проектов, содержащих сотни и тысячи файлов, которые должны быть ассемблированы,

откомплированы или оттранслированы в различные форматы.

Каждая зависимости между двумя или более файлами прописывается в ***make-правиле***:

```
<цель> : <источник>
<tab><команда компиляции 1>
<tab><команда компиляции 2>
...
```

цель одно имя файла, или несколько имен, разделенных пробелами. Этот файл(ы) будут созданы или обновлены этим правилом.

источник 0+ имен файлов разделенных пробелами. Эти файл(ы) будут ‘проверяться на изменения‘ используя метку времени последней модификации.

tab символ табуляции с ascii кодом 0x09, вы должны использовать текстовый редактор, который умеет работать с табуляциями, не заменяя их последовательностями пробелов.

команда компиляции любая команда, такая как вызов ассемблера или линкера, которая обновляет *цель*, выполняя некоторую полезную работу. **make-правило может не иметь команд компиляции**, если вам нужно прописать только зависимость файлов.

> основной принцип каждого make-правила: если один из файлов-*источников* **новее** чем один из *целевых* файлов, будет выполнено тело правила, которое обновит *цели*.

Давайте напишем простой **Makefile** для простейшей программы, описанной в разделе **??**:

Листинг 19: Makefile

```
emulation: add.flash
    qemu-system-arm -M connex -pflash add.flash \
        -nographic -serial /dev/null
flash.bin: add.bin
    dd if=/dev/zero of=flash.bin bs=4K count=4K
    dd if=add.bin of=flash.bin bs=4K conv=notrunc
add.bin: add.elf
    arm-none-eabi-objcopy -O binary add.elf add.bin
add.elf: add.o
    arm-none-eabi-ld -o add.elf add.o
add.o: add.s
    arm-none-eabi-as -o add.o add.s
```

- обратите внимание на обратный слэш и следующую табулированную строку: вы можете делить длинные команды на несколько строк; каждая строка должна быть табулирована для следования синтаксису make-правила.

Введине в командной строке команду **make** без параметров, находясь в каталоге проета, в котором находится **Makefile** и исходные тексты программы, и вы сразу получите автоматически скомпилированые бинарные файлы и запущенный **Qemu**:

```
$ make
...
QEMU 2.1.2 monitor - type 'help' for more information
(qemu)
```

Если вы запускаете **make** без параметров, **первое правило** в **Makefile** будет обработано как *главная цель*, с обходом всех зависимостей в других правилах.

10.13.1 Выбор конкретной *цели*

Если вам нужно обновить только определенный файл-*цель*, поместите необходимое имя файла после команды **make**:

```
$ make add.o
make: 'add.o' is up to date.
```

Эта команда будет перекомпилировать только файл **add.o**, в том и только в том случае, если вы перед запуском команды изменяли **add.s**. Если вы видите сообщение типа `make: add.o is up to date.`, **исходные файлы не менялись**, и **make не будет запускать правило ассемблирования**.

Это очень полезно если у вас очень много файлов исходников[31], и вы изменили несколько символов в одном файле. Без **make**[32] каждое микроскопическое изменение потребует перекомпиляции всего проекта, которое может длиться **несколько часов (!)**. Использование **make** позволяет выполнить всего несколько вызовов компилятора и линкера, что будет намного намного быстрее.

Возаращаясь к нашему **add.o**, вы можете заставить ассемблер выполниться не изменяя файл **add.s**, через команду **touch**:

```
$ touch add.s
$ make add.o
arm-none-eabi-as -o add.o add.s
```

[31] например тысячи файлов, как у ядра *Linux*

[32] используя простой .rc shell-скрипт или .batник

Команда **touch** изменяет только дату модификации исходного файла **add.s**, не меняя его содержимое, так что **make** увидит что этот файл обновился, и запустит ассемблер для указанной цели **add.o**.

По умолчанию **make** выводит каждую команду и ее вывод. Если у вас есть какие-то причины для "тихой" работы **make**, вы можете добавить префикс "-" (минус) к командам компиляции.

10.13.2 Переменные

10.14 13. Contributing

10.15 14. Credits

10.15.1 14.1. People

10.15.2 14.2. Tools

10.16 15. Tutorial Copyright

10.17 A. ARM Programmer's Model

10.18 B. ARM Instruction Set

10.19 C. ARM Stacks

Глава 11

Embedded Systems Programming in C_+^+ [20]

[1]

[1] © http://www.bogotobogo.com/cplusplus/embeddedSystemsProgramming.php

Глава 12

Сборка кросс-компилятора GNU Toolchain из исходных текстов

Если вам по каким-то причинам не подходит одна из типовых сборок кросс-компиляторов, поставляемых в виде готовых бинарных пакетов из репозитория вашего дистрибутива *Linux*, **GNU Toolchain** можно легко скопилировать **из исходных текстов** и установить в систему, даже имея только пользовательские права доступа.

Сборка **GNU Toolchain** из исходников может понадобиться, если вы хотите:

- самую свежую или какую-то конкретную версию **GNU Toolchain**
- опции компиляции: малораспространенный `target`-процессор, **нетиповой формат файлов объектного кода**[1] или экспериментальные оптимизаторы, не включенные в бинарные пакеты из дистрибутива *Linux*

[1] например для i386 может понадобится сборка кросс-компилятора с `-target=i486-none-elf` VI или `i686-linux-uclibc` вместо типовой компиляции для *Linux* типа `i486-linux-gnu`

- полпроцента ускорения работы компилятора благодаря жесткой оптимизации его машинного кода точно под ваш рабочий компьютер (`-march=native -mtune=native -O3`)

При сборке используется утилита **make** 10.13, которой можно передать набор переменных конфигурирования. В таблице перечислен набор переменных конфигурирования сборки с указанием их значения по умолчанию[2] и имя mk-файла, где оно задано:

`APP`	`cross`	`Makefile`	приложение: условное имя проекта (только латиница, буквы a-z)

`HW`	`x86`	`Makefile`	`qemu vmware virtualpc`
			`x86 pc686 amd64`
			`cortexM avr8`

`CPU`	`i386`	`hw/$(HW).mk`	
`ARCH`	`i386`	`cpu/$(CPU).mk`	
`TARGET`	`$(CPU)-pc-elf`	`hw/$(HW).mk`	`i686-linux-uclibc x86_64-linux-gnu` `i386-pc-elf arm-none-eabi avr-none-elf`

APP/HW: приложение/платформа

Для сборки необходимо выбрать имя проекта[3] и аппаратную платформу, для которой будет настраиваться пакет кросс-компилятора.

Особенно это важно для варианта сборки, когда собирается не только кросс-компилятор, но и базовая ОС — минимальная *Linux*-система из ядра, libc и дополнительных прикладных библиотек. В этом случае **APP/HW** используются для формирования имен файлов ядра `$(APP)$(HW).kernel`, названия и состава загрузочного образа `$(APP)$(HW).rootfs`, и внутренних настроек.

[2] также приведены часто используемые варианты значения

[3] только латиница, буквы a-z

Подготовка BUILD-системы: необходимое ПО

Для сборки необходимо установить следующие пакеты:

```
sudo apt install gcc g++ make flex bison m4 bc bzip2 xz-utils libncurses-dev
```

dirs: создание структуры каталогов

```
user@bs:~/boox/cross$ make dirs
mkdir -p
    /home/user/boox/cross/gz /home/user/boox/cross/src /home/user/boox/cross/tmp
    /home/user/boox/cross/toolchain /home/user/boox/cross/root
```

Командной `make dirs` создается набор вспомогательных каталогов:

TC	$(CWD)/$(APP)$(ROOT).cross	каталог установки кросс-компилятора
ROOT	$(CWD)/$(APP)$(ROOT)	каталог файловой системы для целевого em*Linux*

CWD	$(CURDIR)	текущий каталог
GZ	$(CWD)/gz	архивы исходных текстов GNU Toolchain, загрузчика, и библиотек
SRC	$(CWD)/src	каталог для распаковки исходников
TMP	$(CWD)/tmp	каталог для out-of-tree сборки GNU toolchain

mk/dirs.mk

```
1 CWD = $(CURDIR)
2
3 GZ = $(CWD)/gz
4 SRC = $(CWD)/src
```

```
5  TMP = $(CWD)/tmp
6
7  ROOT = $(CWD)/$(APP)$(HW)
8  TC = $(CWD)/$(APP)$(HW).cross
9
10 DIRS = $(GZ) $(SRC) $(TMP) $(TC) $(ROOT)
11 .PHONY: dirs
12 dirs:
13     mkdir -p $(DIRS)
```

Сборка в ОЗУ на ramdiske

Если у вас есть админские права и достаточный объем RAM, после выполнения `make dirs` рекомендуется примонировать на каталоги `SRC` и `TMP` файловую систему tmpfs — это значительно ускорит компиляцию, т.к. все временные файлы будут хранится только в ОЗУ:

/etc/fstab

```
1 tmpfs /home/user/src tmpfs auto,uid=yourlogin,gid=yourgroup 0 0
2 tmpfs /home/user/tmp tmpfs auto,uid=yourlogin,gid=yourgroup 0 0
```

Если вы прописали монтирование *ramdisk*ов в **/etc/fstab**, или сделали `mount -t tmpfs` вручную, может оказаться нужным запускать **make** с явным указанием значений переменных SRC/TMP:

```
make blablabla SRC=/home/user/src TMP=/home/user/tmp
```

Пакеты системы кросс-компиляции

GNU Toolchain

mk/pack_cross.mk

```
# bintuils: assembler, linker, objfile tools
BINUTILS_VER= 2.24
# 2.25 build error

# gcc: C/C++ cross-compiler
GCC_VER      = 4.9.2
# 4.9.2 used: bug arm/62098 fixed

# gcc support libraries
## required for GCC build
GMP_VER      = 5.1.3
MPFR_VER     = 3.1.3
MPC_VER      = 1.0.2
## loop optimisation
ISL_VER      = 0.11.1
# 0.11 need for binutils build
CLOOG_VER    = 0.18.1

# standard C/POSIX library libc (newlib)
NEWLIB_VER  = 2.3.0.20160226

# loader for i386 target
SYSLINUX_VER = 6.03
```

```
24
25 # packages
26 BINUTILS      = binutils-$(BINUTILS_VER)
27 GCC           = gcc-$(GCC_VER)
28 GMP           = gmp-$(GMP_VER)
29 MPFR          = mpfr-$(MPFR_VER)
30 MPC           = mpc-$(MPC_VER)
31 ISL           = isl-$(ISL_VER)
32 CLOOG         = cloog-$(CLOOG_VER)
33 NEWLIB        = newlib-$(NEWLIB_VER)
34 SYSLINUX      = syslinux-$(SYSLINUX_VER)
```

make

newlib стандартная библиотека **libc**

gz: загрузка исходного кода для пакетов

```
user@bs$ make APP=cross HW=x86 GZ=/home/user/gz gz
```

В примере команды показано два обязательных параметра `APP/HW`[4] и необязательный `GZ`: поскольку я собираю кросс-компиляторы для нескольких целевых платформ, я создал каталог `$(HOME)/gz` и загружаю туда архивы исходников **для всех проектов сразу**[5]. Более простой способ – просто сделать симлинк `ln -s ~/gz project/gz` и не переопределять переменную `GZ` явно.

[4] по ним могут закачиваться дополнительные файлы исходников, зависящие от платформы — например исходник загрузчика или бинарные файлы (блобы) драйверов от производителя железки

[5] а не в /gz каждого проекта, нет смысла дублировать исходники **GNU Toolchain** одной и той же версии

mk/gz.mk

```
1  WGET = -wget -N -P $(GZ) -t2 -T2
2
3  .PHONY: gz
4  gz: gz_cross gz_libs gz_$(ARCH)
5
6  .PHONY: gz_cross
7  gz_cross:
8      $(WGET) ftp://ftp.gmplib.org/pub/gmp/$(GMP).tar.bz2
9      $(WGET) http://www.mpfr.org/mpfr-current/$(MPFR).tar.bz2
10     $(WGET) http://www.multiprecision.org/mpc/download/$(MPC).tar.gz
11     $(WGET) ftp://gcc.gnu.org/pub/gcc/infrastructure/$(ISL).tar.bz2
12     $(WGET) ftp://gcc.gnu.org/pub/gcc/infrastructure/$(CLOOG).tar.gz
13     $(WGET) http://ftp.gnu.org/gnu/binutils/$(BINUTILS).tar.bz2
14     $(WGET) http://gcc.skazkaforyou.com/releases/$(GCC)/$(GCC).tar.bz2
15
16 .PHONY: gz_libs
17 gz_libs:
18     $(WGET) ftp://sourceware.org/pub/newlib/$(NEWLIB).tar.gz
19
20 .PHONY: gz_i386
21 gz_i386:
22     $(WGET) https://www.kernel.org/pub/linux/utils/boot/syslinux/$(SYSLINUX).tar.xz
```

Макро-правила для автоматической распаковки исходников

mk/src.mk

```
$(SRC)/%/README: $(GZ)/%.tar.gz
	cd $(SRC) && zcat $< | tar x && touch $@
$(SRC)/%/README: $(GZ)/%.tar.bz2
	cd $(SRC) && bzcat $< | tar x && touch $@
$(SRC)/%/README: $(GZ)/%.tar.xz
	cd $(SRC) && xzcat $< | tar x && touch $@
```

Общие параметры для ./configure

mk/cfg.mk

```
# configure parameters for all packages
CFG_ALL = --disable-nls --disable-werror \
    --docdir=$(TMP)/doc --mandir=$(TMP)/man --infodir=$(TMP)/info

# [B]uild host configure
BCFG = configure $(CFG_ALL) --prefix=$(TC)

XPATH = PATH=$(TC)/bin:$(PATH)

# [T]arget configure
TCFG = configure $(CFG_ALL) --prefix=$(ROOT) CC=$(TARGET)-gcc

# get cpu cores
CPU_CORES ?= $(shell grep processor /proc/cpuinfo |wc -l)

```

```
16 # run make with -j flag or make CPU_CORES=<none> for one thread build
17 MAKE = make -j$(CPU_CORES)
18 INSTALL = make install
```

12.1 Сборка кросс-компилятора

Для пакетов кросс-компилятора существуют два варианта сборки пакетов:

Пакеты с 0 в конце имени задают сборку программ, которые будут выполняться на BUILD-компьютере, и компилировать код для TARGET-системы, т.е. это простейший вариант кросс-компиляции.

Пакеты без 0 , которые могут появиться в будущем — **относятся только к сборке em***Linux*, собирают кросс-компилятор ***канадским крестом***:

- пакет собирается на BUILD-системе — ваш рабочий компьютер,
- выполняется на HOST-системе — например PC104 или роутер с em*Linux*,
- и компилирует код для TARGET-микропроцессора — модуль ввода/вывода на USB, подключенный к PC104)

12.1.1 cclibs0: библиотеки поддержки gcc

Для сборки **GNU Toolchain** необходим набор нескольких библиотек, причем **успешность сборки сильно зависит от их версий**, поэтому библиотеки **нужно собрать из исходников**, а не использовать девелоперские пакеты из дистрибутива BUILD-*Linux*.

Библиотеки чисел произвольной точности:
gmp0 целых

gmfr0 с плавающей точкой
gmc0 комплексных

Библиотеки для работы с графами (нужны для компилятора оптимизатора **Graphite**)
cloog0 polyhedral оптимизации
isl0 манипуляция наборами целочисленных точек

mk/cclibs.mk

```
1  WITH_CCLIBS0 = --with-gmp=$(TC) --with-mpfr=$(TC) --with-mpc=$(TC) \
2      --without-ppl --without-cloog
3  #   --with-isl=$(TC) --with-cloog=$(TC)
4
5  CFG_CCLIBS0 = $(WITH_CCLIBS0) --disable-shared
6  .PHONY: cclibs0
7  cclibs0: gmp0 mpfr0 mpc0
8  # cloog0 isl0
9
10 CFG_GMP0 = $(CFG_CCLIBS0)
11 .PHONY: gmp0
12 gmp0: $(SRC)/$(GMP)/README
13     rm -rf $(TMP)/$(GMP) && mkdir -p $(TMP)/$(GMP) && cd $(TMP)/$(GMP) &&\
14     $(SRC)/$(GMP)/$(BCFG) $(CFG_GMP0) && $(MAKE) && $(INSTALL)-strip
15
16 CFG_MPFR0 = $(CFG_CCLIBS0)
17 .PHONY: mpfr0
18 mpfr0: $(SRC)/$(MPFR)/README
19     rm -rf $(TMP)/$(MPFR) && mkdir -p $(TMP)/$(MPFR) && cd $(TMP)/$(MPFR) &&\
20     $(SRC)/$(MPFR)/$(BCFG) $(CFG_MPFR0) && $(MAKE) && $(INSTALL)-strip
21
```

```
22 CFG_MPC0 = $(CFG_CCLIBS0)
23 .PHONY: mpc0
24 mpc0: $(SRC)/$(MPC)/README
25     rm -rf $(TMP)/$(MPC) && mkdir -p $(TMP)/$(MPC) && cd $(TMP)/$(MPC) &&\
26     $(SRC)/$(MPC)/$(BCFG) $(CFG_MPC0) && $(MAKE) && $(INSTALL)-strip
27
28 CFG_CLOOG0 = --with-gmp-prefix=$(TC) $(CFG_CCLIBS00)
29 .PHONY: cloog0
30 cloog0: $(SRC)/$(CLOOG)/README
31     rm -rf $(TMP)/$(CLOOG) && mkdir $(TMP)/$(CLOOG) && cd $(TMP)/$(CLOOG) &&\
32     $(SRC)/$(CLOOG)/$(BCFG) $(CFG_CLOOG0) && $(MAKE) && $(INSTALL)-strip
33
34 CFG_ISL0 = --with-gmp-prefix=$(TC) $(CFG_CCLIBS00)
35 .PHONY: isl0
36 isl0: $(SRC)/$(ISL)/README
37     rm -rf $(TMP)/$(ISL) && mkdir $(TMP)/$(ISL) && cd $(TMP)/$(ISL) &&\
38     $(SRC)/$(ISL)/$(BCFG) $(CFG_ISL0) && $(MAKE) && $(INSTALL)-strip
```

12.1.2 binutils0: ассемблер и линкер

Чтобы побыстрее получить результат, который можно сразу потестировать, соберем сначала кросс-**bintuils**, а потом все что относится к Сишному компилятору[6].

`-target` триплет целевой системы, например `i386-pc-elf`

[6] на самом деле **binutils0** надо собирать после **cclibs0**, так как есть зависимость от библиотек **isl0** и **cloog0**

`CFG_ARCH CFG_CPU` задаются в файлах **arch/$(ARCH).mk** и **cpu/$(CPU).mk**, и определяют опции сборки **binutils/gcc** для конкретного процессора[7]

`-with-sysroot` каталог где должны храниться файлы для целевой системы: откомпилированные библиотеки и каталог **include**

`-with-native-system-header-dir` имя каталога с **include**-файлами, относительно **sysroot**

arch/i386.mk

```
1 CFG_ARCH =
```

cpu/i386.mk

```
1 ARCH = i386
2 CFG_CPU = --with-cpu=i386 --with-tune=i386
```

mk/bintuils.mk

```
1 CFG_BINUTILS0 = --target=$(TARGET) $(CFG_ARCH) $(CFG_CPU) \
2     --with-sysroot=$(ROOT) --with-native-system-header-dir=/include \
3     --enable-lto --disable-multilib $(WITH_CCLIBS0) \
4     --disable-target-libiberty --disable-target-zlib \
5     --disable-bootstrap --disable-decimal-float \
6     --disable-libmudflap --disable-libssp \
7     --disable-libgomp --disable-libquadmath
8
9 .PHONY: binutils0
```

[7] например `-without-fpu` для **cpu/i486sx.mk**

```
10 binutils0: $(SRC)/$(BINUTILS)/README
11     rm -rf $(TMP)/$(BINUTILS) && mkdir -p $(TMP)/$(BINUTILS) && cd $(TMP)/$(BINUTILS) &&\
12     $(SRC)/$(BINUTILS)/$(BCFG) $(CFG_BINUTILS0) && $(MAKE) && $(INSTALL)-strip
```

Файлы **binutils0** с TARGET-префиксами и типовые скрипты линкера

```
1  crossx86.cross/bin/i386-pc-elf-readelf
2  crossx86.cross/bin/i386-pc-elf-addr2line
3  crossx86.cross/bin/i386-pc-elf-size
4  crossx86.cross/bin/i386-pc-elf-objdump
5  crossx86.cross/bin/i386-pc-elf-objcopy
6  crossx86.cross/bin/i386-pc-elf-nm
7  crossx86.cross/bin/i386-pc-elf-ld.bfd
8  crossx86.cross/bin/i386-pc-elf-elfedit
9  crossx86.cross/bin/i386-pc-elf-as
10 crossx86.cross/bin/i386-pc-elf-ranlib
11 crossx86.cross/bin/i386-pc-elf-c++filt
12 crossx86.cross/bin/i386-pc-elf-gprof
13 crossx86.cross/bin/i386-pc-elf-ar
14 crossx86.cross/bin/i386-pc-elf-strip
15 crossx86.cross/bin/i386-pc-elf-strings
16
17 crossx86.cross/i386-pc-elf/lib/ldscripts/elf_i386.xr
18 crossx86.cross/i386-pc-elf/lib/ldscripts/elf_i386.xsc
19 crossx86.cross/i386-pc-elf/lib/ldscripts/elf_i386.xdc
20 crossx86.cross/i386-pc-elf/lib/ldscripts/elf_i386.xu
21 crossx86.cross/i386-pc-elf/lib/ldscripts/elf_i386.xc
22 crossx86.cross/i386-pc-elf/lib/ldscripts/elf_i386.x
23 crossx86.cross/i386-pc-elf/lib/ldscripts/elf_i386.xbn
```

```
crossx86.cross/i386-pc-elf/lib/ldscripts/elf_i386.xsw
crossx86.cross/i386-pc-elf/lib/ldscripts/elf_i386.xs
crossx86.cross/i386-pc-elf/lib/ldscripts/elf_i386.xw
crossx86.cross/i386-pc-elf/lib/ldscripts/elf_i386.xn
crossx86.cross/i386-pc-elf/lib/ldscripts/elf_i386.xdw
crossx86.cross/i386-pc-elf/lib/ldscripts/elf_i386.xd
```

12.1.3 gcc00: сборка stand-alone компилятора Си

Сборка кросс-компилятора Си выполняется в два этапа

gcc00 минимальный **gcc** необходимый для сборки libc **??**

newlib сборка станадртной библиотеки Си

gcc0 пересборка полного кросс-компилятора Си/C_+^+

mk/gcc.mk

```
CFG_GCC_DISABLE =

CFG_GCC00 = $(CFG_BINUTILS0) $(CFG_GCC_DISABLE) \
    --disable-threads --disable-shared --without-headers --with-newlib \
    --enable-languages="c"

CFG_GCC0 = $(CFG_BINUTILS0) $(CFG_GCC_DISABLE) \
    --with-newlib \
    --enable-languages="c,c++"
```

```
10
11 .PHONY: gcc00
12 gcc00: $(SRC)/$(GCC)/README
13     rm -rf $(TMP)/$(GCC) && mkdir -p $(TMP)/$(GCC) && cd $(TMP)/$(GCC) &&\
14     $(SRC)/$(GCC)/$(BCFG) $(CFG_GCC00)
15     cd $(TMP)/$(GCC) && $(MAKE) all-gcc && $(INSTALL)-gcc
16     cd $(TMP)/$(GCC) && $(MAKE) all-target-libgcc && $(INSTALL)-target-libgcc
```

12.1.4 newlib: саборка стандартной библиотеки libc

Стандартная библиотека **libc**[8] обеспечивает слой совместимости со стандартом POSIX для ваших программ. Это удобно при адаптации чужих программ под вашу ОС, и при написании собственного **мультиплатформенного** кода.

mk/libc.mk

```
1 CFG_NEWLIB = --host=$(TARGET)
2 .PHONY: newlib
3 newlib: $(SRC)/$(NEWLIB)/README
4     rm -rf $(TMP)/$(NEWLIB) && mkdir -p $(TMP)/$(NEWLIB) && cd $(TMP)/$(NEWLIB) &&\
5     $(XPATH) $(SRC)/$(NEWLIB)/$(TCFG) $(CFG_NEWLIB)
6 #     && $(MAKE) && $(INSTALL)-strip
```

[8] для микроконтроллерных систем — обрезанная версия, **newlib**

12.1.5 gcc0: пересборка компилятора Си/C_+^+

12.2 Поддерживаемые платформы

12.2.1 i386: ПК и промышленные PC104

arch/i386.mk

```
1 CFG_ARCH =
```

12.2.2 x86_64: серверные системы

arch/x86_64.mk

12.2.3 AVR: Atmel AVR Mega

arch/avr.mk

12.2.4 arm: процессоры ARM Cortex-Mx

arch/arm.mk

12.2.5 armhf: SoCи Cortex-A, PXA270,..

arch/armhf.mk

12.3 Целевые аппаратные системы

12.3.1 x86: типовой компьютер на процессоре i386+

hw/x86.mk

```
1 CPU = i386
2 TARGET = $(CPU)-pc-elf
```

Глава 13

Porting The GNU Tools To Embedded Systems

Embed With GNU

Porting The GNU Tools To Embedded Systems

Spring 1995

Very *Rough* Draft

Rob Savoye - Cygnus Support

http://ieee.uwaterloo.ca/coldfire/gcc-doc/docs/porting_toc.html

Глава 14

Оптимизация кода

14.1 PGO опитимизация

[1]

[1] © https://habrahabr.ru/post/138132/

Часть V

Микроконтроллеры Cortex-Mx

Часть VI

os86: низкоуровневое программирование i386

Если вам по каким-то причинам не подходит одна из типовых распространенных ОС, например требуется сделать систему управления жесткого реального времени[2], информация в этом разделе поможет сделать ОС-поделку для типового Wintel ПК.

Специализированный GNU Toolchain для i386-pc-gnu

Для компиляции кода вам потребуется специально собранный из исходников кросс-**GNU Toolchain** для целевой архитектуры i386 — *триплет* `TARGET=i386-pc-elf`. Процесс сборки подробно описан в отдельном разделе 12.

Для упрощения не будем завязываться на особенности конкретного ПК или эмулятора **Qemu**[3], все они вполне аппаратно совместимы с любым i386 компьютером в базовой конфигурации, для которого мы и будем рассматривать примеры кода:

- `APP=bare` metal программирование, без базовой ОС
- `HW=x86` типовой минимальный i386 компьютер

os86/Makefile

```
1 APP = bare
2 HW = x86
3 TARGET = i386-pc-elf
4
5 TODO = gz dirs cclibs0 binutils0 gcc00 newlib
6 .PHONY: toolchain
7 toolchain: $(APP)$(HW).cross/bin/$(TARGET)-g++
8 $(APP)$(HW).cross/bin/$(TARGET)-g++:
```

[2] или вы любитель гадить из прикладного ПО в аппаратные порты в обход всех соглашений и средств защиты ОС

[3] VMWare, VirtualPC

```
9       cd ../cross; $(MAKE) $(TODO) \
10          CWD=$(CURDIR) GZ=$(HOME)/L/gz SRC=$(HOME)/L/src TMP=$(HOME)/L/tmp \
11          APP=$(APP) HW=$(HW)
```

MultiBoot-загрузчик

Благодаря усилиям сообщества разработчиков OpenSource была успешно решена одна из проблем начинающего системного программиста — было создано несколько универсальных ***загрузчиков***, берущих на себя заботу о чтении ядра ОС или bare metal программы, начальную инициализацию оборудования, включении защищенного режима, и передачу управления вашей ОС.

Чтобы ваша bare metal программа была успешно загружена, она должна удовлетворять требованиям ***спецификации MultiBoot*** VII быть слинкована в формат ELF и включать заголовок multiboot.

Часть VII

Спецификация MultiBoot

http://www.gnu.org/software/grub/manual/multiboot/multiboot.html

Этот файл документирует ***Спецификацию Multiboot***, проект стандарта на последовательность загрузки. Этот документ имеет редакцию 0.6.96.

Глава 15

Introduction to Multiboot Specification

This chapter describes some rough information on the Multiboot Specification. Note that this is not a part of the specification itself.

15.1 The background of Multiboot Specification

Every operating system ever created tends to have its own boot loader. Installing a new operating system on a machine generally involves installing a whole new set of boot mechanisms, each with completely different install-time and boot-time user interfaces. Getting multiple operating systems to coexist reliably on one machine through typical chaining mechanisms can be a nightmare. There is little or no choice of boot loaders for a particular operating system — if the one that comes with the operating system doesn't do exactly what you want, or doesn't work on your machine, you're screwed.

While we may not be able to fix this problem in existing proprietary operating systems, it shouldn't be too difficult for a few people in the free operating system communities to put their heads together and solve this problem for the popular free operating systems. That's what this specification aims for. Basically, it specifies an

interface between a boot loader and a operating system, such that any complying boot loader should be able to load any complying operating system. This specification does not specify how boot loaders should work — only how they must interface with the operating system being loaded.

15.2 The target architecture

This specification is primarily targeted at i386 PC, since they are the most common and have the largest variety of operating systems and boot loaders. However, to the extent that certain other architectures may need a boot specification and do not have one already, a variation of this specification, stripped of the x86-specific details, could be adopted for them as well.

15.3 The target operating systems

This specification is targeted toward free 32-bit operating systems that can be fairly easily modified to support the specification without going through lots of bureaucratic rigmarole. The particular free operating systems that this specification is being primarily designed for are Linux, the kernels of FreeBSD and NetBSD, Mach, and VSTa. It is hoped that other emerging free operating systems will adopt it from the start, and thus immediately be able to take advantage of existing boot loaders. It would be nice if proprietary operating system vendors eventually adopted this specification as well, but that's probably a pipe dream.

15.4 Boot sources

It should be possible to write compliant boot loaders that load the OS image from a variety of sources, including floppy disk, hard disk, and across a network.

Disk-based boot loaders may use a variety of techniques to find the relevant OS image and boot module data on disk, such as by interpretation of specific file systems[1], using precalculated *blocklists*[2], loading from a special *boot partition*[3], or even loading from within another operating system[4]. Similarly, network-based boot loaders could use a variety of network hardware and protocols.

It is hoped that boot loaders will be created that support multiple loading mechanisms, increasing their portability, robustness, and user-friendliness.

15.5 Configure an operating system at boot-time

It is often necessary for one reason or another for the user to be able to provide some configuration information to an operating system dynamically at boot time. While this specification should not dictate how this configuration information is obtained by the boot loader, it should provide a standard means for the boot loader to pass such information to the operating system.

15.6 How to make OS development easier

OS images should be easy to generate. Ideally, an OS image should simply be an ordinary 32-bit executable file in whatever file format the operating system normally uses. It should be possible to **nm** or disassemble OS images just like normal executables. Specialized tools should not be required to create OS images in a **special** file format. If this means shifting some work from the operating system to a boot loader, that is probably appropriate, because all the memory consumed by the boot loader will typically be made available again after the boot process is created, whereas every bit of code in the OS image typically has to remain in memory forever.

---

[1] e.g. the BSD/Mach boot loader
[2] e.g. LILO
[3] e.g. OS/2
[4] e.g. the VSTa boot code, which loads from DOS

The operating system should not have to worry about getting into 32-bit mode initially, because mode switching code generally needs to be in the boot loader anyway in order to load operating system data above the 1MB boundary, and forcing the operating system to do this makes creation of OS images much more difficult.

Unfortunately, there is a horrendous variety of executable file formats even among free Unix-like pc-based operating systems — generally a different format for each operating system. Most of the relevant free operating systems use some variant of a.out format, but some are moving to elf. It is highly desirable for boot loaders not to have to be able to interpret all the different types of executable file formats in existence in order to load the OS image — otherwise the boot loader effectively becomes operating system specific again.

This specification adopts a compromise solution to this problem. Multiboot-compliant OS images always contain a magic *Multiboot header* (see OS image format **??**), which allows the boot loader to load the image without having to understand numerous a.out variants or other executable formats. This magic header does not need to be at the very beginning of the executable file, so kernel images can still conform to the local a.out format variant in addition to being Multiboot-compliant.

15.7 Boot modules

Many modern operating system kernels, such as Mach and the microkernel in VSTa, do not by themselves contain enough mechanism to get the system fully operational: they require the presence of additional software modules at boot time in order to access devices, mount file systems, etc. While these additional modules could be embedded in the main OS image along with the kernel itself, and the resulting image be split apart manually by the operating system when it receives control, it is often more flexible, more space-efficient, and more convenient to the operating system and user if the boot loader can load these additional modules independently in the first place.

Thus, this specification should provide a standard method for a boot loader to indicate to the operating system what auxiliary boot modules were loaded, and where they can be found. Boot loaders don't have to support multiple boot modules, but they are strongly encouraged to, because some operating systems will be

unable to boot without them.

The definitions of terms used through the specification

must We use the term must, when any boot loader or OS image needs to follow a rule — otherwise, the boot loader or OS image is not Multiboot-compliant.

should We use the term should, when any boot loader or OS image is recommended to follow a rule, but it doesn't need to follow the rule.

may We use the term may, when any boot loader or OS image is allowed to follow a rule.

boot loader Whatever program or set of programs loads the image of the final operating system to be run on the machine. The boot loader may itself consist of several stages, but that is an implementation detail not relevant to this specification. Only the final stage of the boot loader — the stage that eventually transfers control to an operating system — must follow the rules specified in this document in order to be Multiboot-compliant; earlier boot loader stages may be designed in whatever way is most convenient.

OS image The initial binary image that a boot loader loads into memory and transfers control to start an operating system. The OS image is typically an executable containing the operating system kernel.

boot module Other auxiliary files that a boot loader loads into memory along with an OS image, but does not interpret in any way other than passing their locations to the operating system when it is invoked.

Multiboot-compliant A boot loader or an OS image which follows the rules defined as must is Multiboot-compliant. When this specification specifies a rule as should or may, a Multiboot-complaint boot loader/OS image doesn't need to follow the rule.

u8 The type of unsigned 8-bit data.

u16 The type of unsigned 16-bit data. Because the target architecture is little-endian, *u16* is coded in **little-endian**.

u32 The type of unsigned 32-bit data. Because the target architecture is little-endian, *u32* is coded in **little-endian**.

u64 The type of unsigned 64-bit data. Because the target architecture is little-endian, *u64* is coded in little-endian.

Глава 16

The exact definitions of Multiboot Specification

There are three main aspects of a boot loader/OS image interface:

1. The format of an OS image as seen by a boot loader.
2. The state of a machine when a boot loader starts an operating system.
3. The format of information passed by a boot loader to an operating system.

16.1 OS image format

An OS image may be an ordinary 32-bit executable file in the standard format for that particular operating system, except that it may be linked at a non-default load address to avoid loading on top of the pc's I/O region or other reserved areas, and of course it should not use shared libraries or other fancy features.

An OS image must contain an additional header called *Multiboot header*, besides the headers of the format used by the OS image. The Multiboot header must be contained completely within the first 8192 bytes of the OS image, and must be longword (32-bit) aligned. In general, it should come **as early as possible**, and may be embedded in the beginning of the text segment after the real executable header.

16.1.1 The layout of Multiboot header

The layout of the Multiboot header must be as follows:

Offset	Type	Field Name	Note
0	u32	magic	required
4	u32	flags	required
8	u32	checksum	required
12	u32	header_addr	if flags[16] is set
16	u32	load_addr	if flags[16] is set
20	u32	load_end_addr	if flags[16] is set
24	u32	bss_end_addr	if flags[16] is set
28	u32	entry_addr	if flags[16] is set
32	u32	mode_type	if flags[2] is set
36	u32	width	if flags[2] is set
40	u32	height	if flags[2] is set
44	u32	depth	if flags[2] is set

The fields 'magic', 'flags' and 'checksum' are defined in Header magic fields 16.1.2, the fields 'header_addr', 'load_addr', 'load_end_addr', 'bss_end_addr' and 'entry_addr' are defined in Header address fields 16.1.1, and the fields 'mode_type', 'width', 'height' and 'depth' are defined in Header graphics fields 16.1.4.

16.1.2 The magic fields of Multiboot header

'magic' The field 'magic' is the magic number identifying the header, which must be the hexadecimal value 0x1BADB002.

'flags' The field 'flags' specifies features that the OS image requests or requires of an boot loader. Bits 0-15 indicate requirements; if the boot loader sees any of these bits set but doesn't understand the flag or

can't fulfill the requirements it indicates for some reason, it must notify the user and fail to load the OS image. Bits 16-31 indicate optional features; if any bits in this range are set but the boot loader doesn't understand them, it may simply ignore them and proceed as usual. Naturally, all as-yet-undefined bits in the 'flags' word must be set to zero in OS images. This way, the 'flags' fields serves for version control as well as simple feature selection.

If bit 0 in the 'flags' word is set, then all boot modules loaded along with the operating system must be aligned on page (4KB) boundaries. Some operating systems expect to be able to map the pages containing boot modules directly into a paged address space during startup, and thus need the boot modules to be page-aligned.

If bit 1 in the 'flags' word is set, then information on available memory via at least the 'mem_*' fields of the Multiboot information structure (see Boot information format **??**) must be included. If the boot loader is capable of passing a memory map (the 'mmap_*' fields) and one exists, then it may be included as well.

If bit 2 in the 'flags' word is set, information about the video mode table (see Boot information format **??**) must be available to the kernel.

If bit 16 in the 'flags' word is set, then the fields at offsets 12-28 in the Multiboot header are valid, and the boot loader should use them instead of the fields in the actual executable header to calculate where to load the OS image. This information does not need to be provided if the kernel image is in elf format, but it must be provided if the images is in a.out format or in some other format. Compliant boot loaders must be able to load images that either are in elf format or contain the load address information embedded in the Multiboot header; they may also directly support other executable formats, such as particular a.out variants, but are not required to.

'checksum' The field 'checksum' is a 32-bit unsigned value which, when added to the other magic fields (i.e. 'magic' and 'flags'), must have a 32-bit unsigned sum of zero.

16.1.3 The address fields of Multiboot header

All of the address fields enabled by flag bit 16 are physical addresses. The meaning of each is as follows:

header_addr Contains the address corresponding to the beginning of the Multiboot header — the physical memory location at which the magic value is supposed to be loaded. This field serves to **synchronize** the mapping between OS image offsets and physical memory addresses.

load_addr Contains the physical address of the beginning of the text segment. The offset in the OS image file at which to start loading is defined by the offset at which the header was found, minus (header_addr - load_addr). load_addr must be less than or equal to header_addr.

load_end_addr Contains the physical address of the end of the data segment. (load_end_addr - load_addr) specifies how much data to load. This implies that the text and data segments must be consecutive in the OS image; this is true for existing a.out executable formats. If this field is zero, the boot loader assumes that the text and data segments occupy the whole OS image file.

bss_end_addr Contains the physical address of the end of the bss segment. The boot loader initializes this area to zero, and reserves the memory it occupies to avoid placing boot modules and other data relevant to the operating system in that area. If this field is zero, the boot loader assumes that no bss segment is present.

entry_addr The physical address to which the boot loader should jump in order to start running the operating system.

16.1.4 The graphics fields of Multiboot header

All of the graphics fields are enabled by flag bit 2. They specify the preferred graphics mode. Note that that is only a recommended mode by the OS image. If the mode exists, the boot loader should set it, when the user doesn't specify a mode explicitly. Otherwise, the boot loader should fall back to a similar mode, if available.

The meaning of each is as follows:

mode_type Contains '0' for linear graphics mode or '1' for EGA-standard text mode. Everything else is reserved for future expansion. Note that the boot loader may set a text mode, even if this field contains '0'.

width Contains the number of the columns. This is specified in pixels in a graphics mode, and in characters in a text mode. The value zero indicates that the OS image has no preference.

height Contains the number of the lines. This is specified in pixels in a graphics mode, and in characters in a text mode. The value zero indicates that the OS image has no preference.

depth Contains the number of bits per pixel in a graphics mode, and zero in a text mode. The value zero indicates that the OS image has no preference.

16.2 Machine state

When the boot loader invokes the 32-bit operating system, the machine must have the following state:

'EAX' Must contain the magic value '0x2BADB002'; the presence of this value indicates to the operating system that it was loaded by a Multiboot-compliant boot loader (e.g. as opposed to another type of boot loader that the operating system can also be loaded from).

'EBX' Must contain the 32-bit physical address of the Multiboot information structure provided by the boot loader (see Boot information format).

'CS' Must be a 32-bit read/execute code segment with an offset of '0' and a limit of '0xFFFFFFFF'. The exact value is undefined.

'DS'

'ES'

'FS'

'GS'

'SS' Must be a 32-bit read/write data segment with an offset of '0' and a limit of '0xFFFFFFFF'. The exact values are all undefined.

'A20 gate' Must be enabled.

'CR0' Bit 31 (PG) must be cleared. Bit 0 (PE) must be set. Other bits are all undefined.

'EFLAGS' Bit 17 (VM) must be cleared. Bit 9 (IF) must be cleared. Other bits are all undefined.

All other processor registers and flag bits are undefined. This includes, in particular:

'ESP' The OS image must create its own stack as soon as it needs one.

'GDTR' Even though the segment registers are set up as described above, the 'GDTR' may be invalid, so the OS image must not load any segment registers (even just reloading the same values!) until it sets up its own 'GDT'.

'IDTR' The OS image must leave interrupts disabled until it sets up its own IDT.

However, other machine state should be left by the boot loader in normal working order, i.e. as initialized by the bios (or DOS, if that's what the boot loader runs from). In other words, the operating system should be able to make bios calls and such after being loaded, as long as it does not overwrite the bios data structures before doing so. Also, the boot loader must leave the pic programmed with the normal bios/DOS values, even if it changed them during the switch to 32-bit mode.

16.3 Boot information format

FIXME: Split this chapter like the chapter "OS image format".

Upon entry to the operating system, the EBX register contains the physical address of a Multiboot information data structure, through which the boot loader communicates vital information to the operating system. The operating system can use or ignore any parts of the structure as it chooses; all information passed by the boot loader is advisory only.

The Multiboot information structure and its related substructures may be placed anywhere in memory by the boot loader (with the exception of the memory reserved for the kernel and boot modules, of course). It is the operating system's responsibility to avoid overwriting this memory until it is done using it.

The format of the Multiboot information structure (as defined so far) follows:

```
        +-------------------+
0       | flags             |    (required)
        +-------------------+
4       | mem_lower         |    (present if flags[0] is set)
8       | mem_upper         |    (present if flags[0] is set)
        +-------------------+
12      | boot_device       |    (present if flags[1] is set)
        +-------------------+
16      | cmdline           |    (present if flags[2] is set)
        +-------------------+
20      | mods_count        |    (present if flags[3] is set)
24      | mods_addr         |    (present if flags[3] is set)
        +-------------------+
28 - 40 | syms              |    (present if flags[4] or
        |                   |                flags[5] is set)
```

```
        +-------------------+
44      | mmap_length       |    (present if flags[6] is set)
48      | mmap_addr         |    (present if flags[6] is set)
        +-------------------+
52      | drives_length     |    (present if flags[7] is set)
56      | drives_addr       |    (present if flags[7] is set)
        +-------------------+
60      | config_table      |    (present if flags[8] is set)
        +-------------------+
64      | boot_loader_name  |    (present if flags[9] is set)
        +-------------------+
68      | apm_table         |    (present if flags[10] is set)
        +-------------------+
72      | vbe_control_info  |    (present if flags[11] is set)
76      | vbe_mode_info     |
80      | vbe_mode          |
82      | vbe_interface_seg |
84      | vbe_interface_off |
86      | vbe_interface_len |
        +-------------------+
```

The first longword indicates the presence and validity of other fields in the Multiboot information structure. All as-yet-undefined bits must be set to zero by the boot loader. Any set bits that the operating system does not understand should be ignored. Thus, the 'flags' field also functions as a version indicator, allowing the Multiboot information structure to be expanded in the future without breaking anything.

If bit 0 in the 'flags' word is set, then the 'mem_*' fields are valid. 'mem_lower' and 'mem_upper' indicate the amount of lower and upper memory, respectively, in kilobytes. Lower memory starts at address 0, and upper

memory starts at address 1 megabyte. The maximum possible value for lower memory is 640 kilobytes. The value returned for upper memory is maximally the address of the first upper memory hole minus 1 megabyte. It is not guaranteed to be this value.

If bit 1 in the 'flags' word is set, then the 'boot_device' field is valid, and indicates which bios disk device the boot loader loaded the OS image from. If the OS image was not loaded from a bios disk, then this field must not be present (bit 3 must be clear). The operating system may use this field as a hint for determining its own root device, but is not required to. The 'boot_device' field is laid out in four one-byte subfields as follows:

```
+-------+-------+-------+-------+
| part3 | part2 | part1 | drive |
+-------+-------+-------+-------+
```

The first byte contains the bios drive number as understood by the bios INT 0x13 low-level disk interface: e.g. 0x00 for the first floppy disk or 0x80 for the first hard disk.

The three remaining bytes specify the boot partition. 'part1' specifies the top-level partition number, 'part2' specifies a sub-partition in the top-level partition, etc. Partition numbers always start from zero. Unused partition bytes must be set to 0xFF. For example, if the disk is partitioned using a simple one-level DOS partitioning scheme, then 'part1' contains the DOS partition number, and 'part2' and 'part3' are both 0xFF. As another example, if a disk is partitioned first into DOS partitions, and then one of those DOS partitions is subdivided into several BSD partitions using BSD's disklabel strategy, then 'part1' contains the DOS partition number, 'part2' contains the BSD sub-partition within that DOS partition, and 'part3' is 0xFF.

DOS extended partitions are indicated as partition numbers starting from 4 and increasing, rather than as nested sub-partitions, even though the underlying disk layout of extended partitions is hierarchical in nature. For example, if the boot loader boots from the second extended partition on a disk partitioned in conventional DOS style, then 'part1' will be 5, and 'part2' and 'part3' will both be 0xFF.

If bit 2 of the 'flags' longword is set, the 'cmdline' field is valid, and contains the physical address of the command line to be passed to the kernel. The command line is a normal C-style zero-terminated string.

If bit 3 of the 'flags' is set, then the 'mods' fields indicate to the kernel what boot modules were loaded along with the kernel image, and where they can be found. 'mods_count' contains the number of modules loaded; 'mods_addr' contains the physical address of the first module structure. 'mods_count' may be zero, indicating no boot modules were loaded, even if bit 1 of 'flags' is set. Each module structure is formatted as follows:

```
        +-------------------+
0       | mod_start         |
4       | mod_end           |
        +-------------------+
8       | string            |
        +-------------------+
12      | reserved (0)      |
        +-------------------+
```

The first two fields contain the start and end addresses of the boot module itself. The 'string' field provides an arbitrary string to be associated with that particular boot module; it is a zero-terminated ASCII string, just like the kernel command line. The 'string' field may be 0 if there is no string associated with the module. Typically the string might be a command line (e.g. if the operating system treats boot modules as executable programs), or a pathname (e.g. if the operating system treats boot modules as files in a file system), but its exact use is specific to the operating system. The 'reserved' field must be set to 0 by the boot loader and ignored by the operating system.

Caution: Bits 4 & 5 are mutually exclusive.

If bit 4 in the 'flags' word is set, then the following fields in the Multiboot information structure starting at byte 28 are valid:

```
        +-------------------+
28      | tabsize           |
32      | strsize           |
```

```
36      | addr               |
40      | reserved (0)       |
        +--------------------+
```

These indicate where the symbol table from an a.out kernel image can be found. 'addr' is the physical address of the size (4-byte unsigned long) of an array of a.out format nlist structures, followed immediately by the array itself, then the size (4-byte unsigned long) of a set of zero-terminated ascii strings (plus sizeof(unsigned long) in this case), and finally the set of strings itself. 'tabsize' is equal to its size parameter (found at the beginning of the symbol section), and 'strsize' is equal to its size parameter (found at the beginning of the string section) of the following string table to which the symbol table refers. Note that 'tabsize' may be 0, indicating no symbols, even if bit 4 in the 'flags' word is set.

If bit 5 in the 'flags' word is set, then the following fields in the Multiboot information structure starting at byte 28 are valid:

```
        +--------------------+
28      | num                |
32      | size               |
36      | addr               |
40      | shndx              |
        +--------------------+
```

These indicate where the section header table from an ELF kernel is, the size of each entry, number of entries, and the string table used as the index of names. They correspond to the 'shdr_*' entries ('shdr_num', etc.) in the Executable and Linkable Format (elf) specification in the program header. All sections are loaded, and the physical address fields of the elf section header then refer to where the sections are in memory (refer to the i386 elf documentation for details as to how to read the section header(s)). Note that 'shdr_num' may be 0, indicating no symbols, even if bit 5 in the 'flags' word is set.

If bit 6 in the 'flags' word is set, then the 'mmap_*' fields are valid, and indicate the address and length of a buffer containing a memory map of the machine provided by the bios. 'mmap_addr' is the address, and 'mmap_length' is the total size of the buffer. The buffer consists of one or more of the following size/structure pairs ('size' is really used for skipping to the next pair):

```
        +-------------------+
-4      | size              |
        +-------------------+
0       | base_addr         |
8       | length            |
16      | type              |
        +-------------------+
```

where 'size' is the size of the associated structure in bytes, which can be greater than the minimum of 20 bytes. 'base_addr' is the starting address. 'length' is the size of the memory region in bytes. 'type' is the variety of address range represented, where a value of 1 indicates available ram, and all other values currently indicated a reserved area.

The map provided is guaranteed to list all standard ram that should be available for normal use.

If bit 7 in the 'flags' is set, then the 'drives_*' fields are valid, and indicate the address of the physical address of the first drive structure and the size of drive structures. 'drives_addr' is the address, and 'drives_length' is the total size of drive structures. Note that 'drives_length' may be zero. Each drive structure is formatted as follows:

```
        +-------------------+
0       | size              |
        +-------------------+
4       | drive_number      |
        +-------------------+
```

```
5       | drive_mode        |
        +-------------------+
6       | drive_cylinders   |
8       | drive_heads       |
9       | drive_sectors     |
        +-------------------+
10 - xx | drive_ports       |
        +-------------------+
```

The 'size' field specifies the size of this structure. The size varies, depending on the number of ports. Note that the size may not be equal to (10 + 2 * the number of ports), because of an alignment.

The 'drive_number' field contains the BIOS drive number. The 'drive_mode' field represents the access mode used by the boot loader. Currently, the following modes are defined:

'0' CHS mode (traditional cylinder/head/sector addressing mode).

'1' LBA mode (Logical Block Addressing mode).

The three fields, 'drive_cylinders', 'drive_heads' and 'drive_sectors', indicate the geometry of the drive detected by the bios. 'drive_cylinders' contains the number of the cylinders. 'drive_heads' contains the number of the heads. 'drive_sectors' contains the number of the sectors per track.

The 'drive_ports' field contains the array of the I/O ports used for the drive in the bios code. The array consists of zero or more unsigned two-bytes integers, and is terminated with zero. Note that the array may contain any number of I/O ports that are not related to the drive actually (such as dma controller's ports).

If bit 8 in the 'flags' is set, then the 'config_table' field is valid, and indicates the address of the rom configuration table returned by the GET CONFIGURATION bios call. If the bios call fails, then the size of the table must be zero.

If bit 9 in the ‘flags’ is set, the ‘boot_loader_name’ field is valid, and contains the physical address of the name of a boot loader booting the kernel. The name is a normal C-style zero-terminated string.

If bit 10 in the ‘flags’ is set, the ‘apm_table’ field is valid, and contains the physical address of an apm table defined as below:

Examples

History

Index

Часть VIII

Технологии

Часть IX

Сетевое обучение

Часть X

Базовая теоретическая подготовка

Глава 17

Математика

17.1 Высшая математика в упражнениях и задачах [59]

В этом разделе будут размещены решения некторых задач из [59] в "техническом" стиле: главное быстрый результат, а не точное аналитическое решение, поэтому будем использовать системы компьютерной математики. Будут рассмотрены приемы применения OpenSource пакетов:

Maxima [18] символьная математика, аналог **MathCAD**

Octave [19] численная математика, аналог **MATLAB**

GNUPLOT [?] простейшее средство построения 3D/3D графиков

WolframAlpha http://www.wolframalpha.com/ бесплатная on-line система символьной математики и база знаний

Python скриптовый язык программирования, в последнее время получил широкое применение в области численных методов, анализа данных и автоматизации, чаще всего применяется в комплекте с библиотеками:

NumPy поддержка многомерных массивов (включая матрицы) и высокоуровневых математиче-

ских функций, предназначенных для работы с ними

SciPy библиотека предназначенная для выполнения научных и инженерных расчётов: поиск минимумов и максимумов функций, вычисление интегралов функций, поддержка специальных функций, обработка сигналов, обработка изображений, работа с генетическими алгоритмами, решение обыкновенных дифференциальных уравнений,...

SymPy библиотека символьной математики https://en.wikipedia.org/wiki/SymPy

Matplotlib библиотека на языке программирования Python для 2D/3D визуализации данных. Получаемые изображения могут быть использованы в качестве иллюстраций в публикациях.

Подробно с применением *Python* при обработке данных можно ознакомиться в http://scipy-cookbook.readthedocs.org/

Также этот раздел можно использовать как пример использования системы верстки LaTeX для научных публикаций — смотрите **исходные тексты** файла https://github.com/ponyatov/boox/tree/master/math/danko/**danko.tex**.

17.1.1 Аналитическая геометрия на плоскости

Прямоугольные и полярные координаты

1. Координаты на прямой. Деление отрезка в данном отношении Точку M координатной оси Ox, имеющую ***абсциссу*** x, обозначают через $M(x)$.

Расстояние d между точками $M_1(x_1)$ и $M_2(x_2)$ оси при любом расположении точек на оси находятся по формуле:

$$d = |x_2 - x_1| \tag{17.1}$$

Пусть на произвольной прямой задан отрезок AB (A — начало отрезка, B — конец), тогда всякая третья точка C этой прямой делить отрезок AB в некотором отношении λ, где $\lambda = \pm AC : CB$. Если

отрезки AC и CB направлены в одну сторону, то λ приписывают знак “плюс”; если же отрезки AC и CB направлены в противоположные стороны, то λ приписывающт знак “минус”. Иными словами, $\lambda > 0$ если точка C лежит между точками A и B; $\lambda < 0$ если точка C лежит вне отрезка AB.

Пусть точки A и B лежит на оси Ox, тогда ***координата точки*** $C(\bar{x})$, делящей отрезок между точками $A(x_1)$ и $B(x_2)$ в отношении λ, находится по формуле:

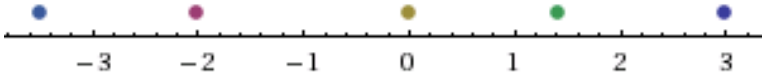
$$\bar{x} = \frac{x_1 + \lambda x_2}{1 + \lambda} \tag{17.2}$$

В частности, при $\lambda = 1$ получается формула для координаты середины отрезка:

$$\bar{x} = \frac{x_1 + x_2}{2} \tag{17.3}$$

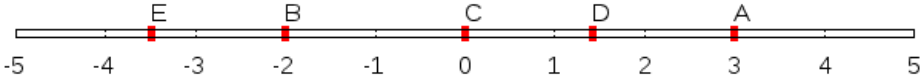
1. Построить на прямой точки $A(3)$, $B(-2)$, $C(0)$, $D(\sqrt{2})$, $E(-3.5)$.

WolframAlpha



`number line -3,2,0,sqrt(2),-3.5`

Листинг 20: GNUPLOT



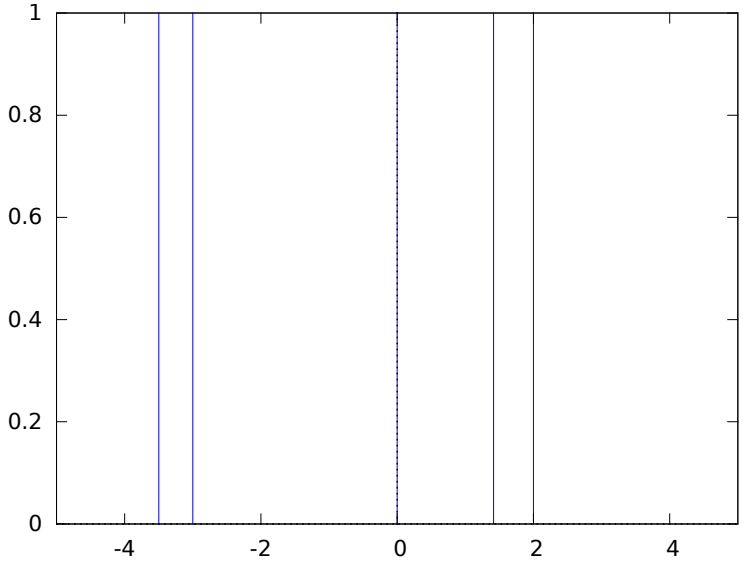

```
set terminal png size 640,64
set output 'g_1_1_1.pdf'
unset key
unset ytics
set xtics 1
set label "A" at 3,3
set label "B" at -2,3
set label "C" at 0,3
set label "D" at sqrt(2),3
set label "E" at -3.5,3
plot [-5:+5][0:1] '-' u 1:2 w i lw 5
3 1
-2 1
0 1
1.4142 1
-3.5 1
e
```

1

[1] $\sqrt{2}$ пришлось указать численно, значение функции не подставилось

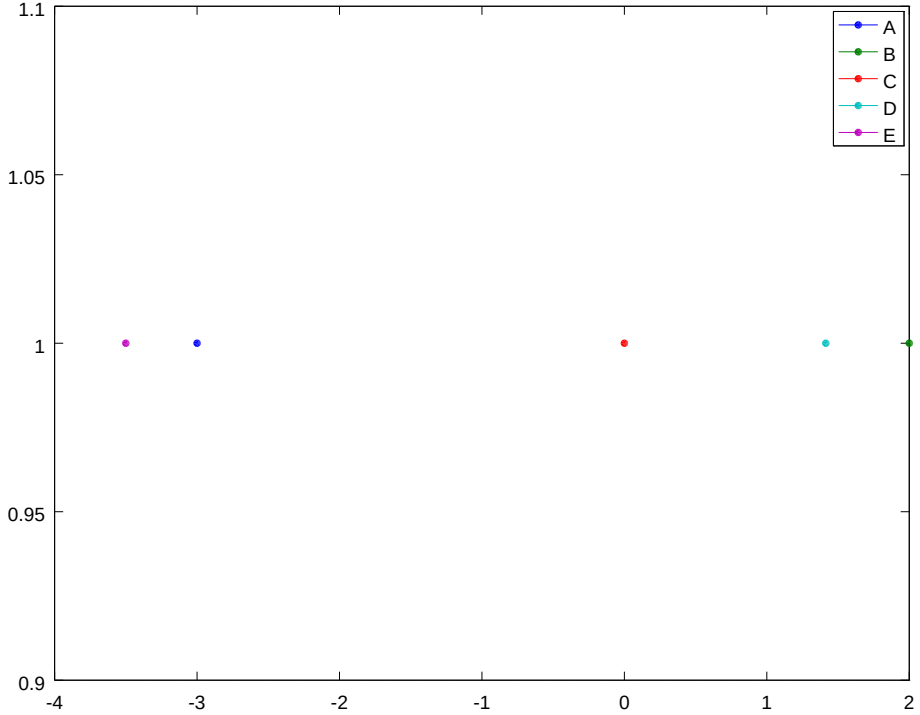
Листинг 21: Maxima

```
A:-3;
B:2;
C:0;
D:sqrt(2);
E:-3.5;

dat:[[A,1],[B,1],[C,1],[D,1],[E,1]];

plot2d([discrete,dat],\
  [x,-5,+5],[y,0,1],\
  [style,impulses],\
  [xlabel,false],[ylabel,false],\
  [gnuplot_term,pdf],\
  [gnuplot_out_file,"./m_1_1_1.pdf"]);
```



Листинг 22: Octave

```
A=-3;
B=2;
C=0;
D=sqrt(2);
E=-3.5;

plot(A,1,B,1,C,1,D,1,E,1)
legend('A','B','C','D','E');
print o_1_1_1.pdf
```



2. Отрезок AB четырьмя точками разделен на пять равных частей. Найти координату ближайшей к A точки деления, если $A(-3)$, $B(7)$.

Пусть $C(\bar{x})$ — искомая точка, тогда $\lambda = \frac{AC}{CB} = \frac{1}{4}$. Следовательно, по формуле 17.2 находим

$$C(\bar{x}) = \frac{x_1 + \lambda x_2}{1 + \lambda} = \frac{-3 + \frac{1}{4} \cdot 7}{1 + \frac{1}{4}} = C(-1)$$

Maxima

```
m_1_1_2 (x1,x2,lambda) := (x1+lambda*x2)/(1+lambda);
A : -3 ;
B : 7 ;
lambda : 1/4 ;

C = m_1_1_2(A,B,lambda);
```

Определяем функцию `m(axima)_<глава>_<параграф>_<задача>` (по нумерации задач в [59]), и вычисляем функцию с подстановкой числовых значений.

```

Maxima 5.34.1 http://maxima.sourceforge.net
using Lisp GNU Common Lisp (GCL) GCL 2.6.12 (a.k.a. GCL)
Distributed under the GNU Public License. See the file COPYING.
Dedicated to the memory of William Schelter.
The function bug_report() provides bug reporting information.
(%i1)                                                    x1 + lambda x2
(%o1)                m_1_1_2(x1, x2, lambda) := ----------------
                                                      1 + lambda
(%i2) (%o2)                                        - 3
(%i3) (%o3)                                          7
(%i4)                                                1
(%o4)                                          -
```

```
14                                   4
15 (%i5) (%o5)                      C = - 1
16 (%i6)
```

В **Octave файлы с расширением .m** могут содержать не только последовательность команд, но и **выполнять роль определения библиотечной функции**. В этом случае имя функции должно совпадать с именем файла, где прописано ее определение.

Листинг 23: шаблон определения функции

```
function [<результат_1>,<результат_2>,..] = <имя> [<параметр_1>,<параметр_2>,..]
    оператор_1;
    ...
    оператор_N;
end;
```

Octave — o_1_1_2.m

```
1 function [xn] = o_1_1_2 (x1,x2,lambda)
2     xn = (x1+lambda*x2)/(1+lambda);
3 end
4 A = -3
5 B = 7
6 lambda = 1/4
7
8 o_1_1_2(A,B,lambda)
```

Определяем функцию `o(ctave)_<глава>_<параграф>_<задача>` (по нумерации задач в [59]), и вычисляем функцию с подстановкой числовых значений.

```
GNU Octave, version 3.8.2
Copyright (C) 2014 John W. Eaton and others.
This is free software; see the source code for copying conditions.
There is ABSOLUTELY NO WARRANTY; not even for MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE.  For details, type 'warranty'.

Octave was configured for "x86_64-pc-linux-gnu".

Additional information about Octave is available at http://www.octave.org.

Please contribute if you find this software useful.
For more information, visit http://www.octave.org/get-involved.html

Read http://www.octave.org/bugs.html to learn how to submit bug reports.
For information about changes from previous versions, type 'news'.

A = -3
B =  7
lambda =  0.25000
ans = -1
```

Часть XI

Прочее

Ф.И.Атауллаханов об учебниках США и России

© Доктор биологических наук Фазли Иноятович Атауллаханов.
МГУ им. М. В. Ломоносова, Университет Пенсильвании, США

http://www.nkj.ru/archive/articles/19054/

...

У необходимости рекламировать науку есть важная обратная сторона: каждый американский учёный непрерывно, с первых шагов и всегда, учится излагать свои мысли внятно и популярно. В России традиции быть понятными у учёных нет. Как пример я люблю приводить двух великих физиков: русского Ландау и американца Фейнмана. Каждый написал многотомный учебник по физике. Первый — знаменитый "Ландау-Лифшиц", второй — "Лекции по физике". Так вот, "Ландау-Лифшиц" прекрасный справочник, но представляет собой полное издевательство над читателем. Это типичный памятник автору, который был, мягко говоря, малоприятным человеком. Он излагает то, что излагает, абсолютно пренебрегая своим читателем и даже издеваясь над ним. А у нас целые поколения выросли на этой книге, и считается, что всё нормально, кто справился, тот молодец. Когда я столкнулся с "Лекциями по физике" Фейнмана, я просто обалдел: оказывается, можно по-человечески разговаривать со своими коллегами, со студентами, с аспирантами. Учебник Ландау — пример того, как устроена у нас вся наука. Берёшь текст русской статьи, читаешь с самого начала и ничего не можешь понять, а иногда сомневаешься, понимает ли автор сам себя. Конечно, крупицы осмысленного и разумного и оттуда можно вынуть. Но автор явно считает, что это твоя работа — их оттуда извлечь. Не потому, что он не хочет быть понятым, а потому, что его не научили правильно писать. Не учат у нас человека ни писать, ни говорить понятно, это считается неважным.

...

Думаю, американская наука в целом устроена именно так: она продаёт не просто себя, а всю свою страну. Сегодня американцы дороги не метут, сапоги не тачают, даже телевизоры не собирают, за них это делает весь остальной мир. А что же делают американцы ? Самая богатая страна в мире ? Они объяснили,

в первую очередь самим себе, а заодно и всему миру, что они — мозг планеты. Они изобретают. “Мы придумываем продукты, а вы их делайте. В том числе и для нас”. Это прекрасно работает, поэтому они очень ценят науку.

...

Глава 18

Настройка редактора/IDE (g)Vim

При использовании редактора/IDE **(g)Vim** удобно настроить сочетания клавиш и подсветку синтаксиса языков, которые вы используете так, как вам удобно.

18.1 для вашего собственного скриптового языка

Через какое-то время практики FSP у вас выработается один диалект скриптов для всех программ, соответсвующий именно вашим вкусам в синтаксисе, и в этом случае его нужно будет описать только в файлах /.vim/(ftdetect|syntax).vim, и привязать их к расширениям через dot-файлы **(g)Vim** в вашем домашнем каталоге:

filetype.vim	**(g)Vim**	привязка расширений файлов (.src .log) к настройкам **(g)Vim**
syntax.vim	**(g)Vim**	синтаксическая подсветка для скриптов
/.vimrc	*Linux*	настройки для пользователя
/vimrc	*Windows*	
/.vim/ftdetect/src.vim	*Linux*	привязка команд к расширению .src
/vimfiles/ftdetect/src.vim	*Windows*	
/.vim/syntax/src.vim	*Linux*	синтаксис к расширению .src
/vimfiles/syntax/src.vim	*Windows*	

Книги must have любому техническому специалисту

Математика, физика, химия

- Бермант **Математический анализ** [26]
- Тихонов, Самарский **Математическая физика** [35, 60]
- Демидович, Марон **Численные методы** [40, 41]
- Кремер **Теория вероятностей и матстатистика** [31]
- Ван дер Варден **Математическая статистика** [27]
- Кострикин **Введение в алгебру** [29, 30]
- Ван дер Варден **Алгебра** [28]

- Демидович **Сборник задач по математике для втузов. В 4 частях** [61, ?, ?, ?]
- Будак, Самарский, Тихонов **Сборник задач по математической физике** [60]

Фейнмановские лекции по физике

1. Современная наука о природе. Законы механики. [46]
2. Пространство. Время. Движение. [47]

3. Излучение. Волны. Кванты. [48]
4. Кинетика. Теплота. Звук. [49]
5. Электричество и магнетизм [50]
6. Электродинамика. [51]
7. Физика сплошных сред. [52]
8. Квантовая механика 1. [53]
9. Квантовая механика 2. [54]

- Цирельсон **Квантовая химия** [56]
- Розенброк **Вычислительные методы для инженеров-химиков** [57]
- Шрайвер Эткинс **Неорганическая химия** [58]

Обработка экспериментальных данных и метрология

- Смит **Цифровая обработка сигналов** [32]
- Князев, Черкасский **Начала обработки экспериментальных данных** [33]

Программирование

- **Система контроля версий Git и git-хостинга GitHub**
  хранение наработок с полной историей редактирования, правок, релизов для разных заказчиков или вариантов использования
- **Язык Python** [24]
  написание скриптов обработки данных, автоматизации, графических оболочек и т.п. утилит
- **JavaScript [22] + HTML**
  генерация отчетов и ввод исходных данных, интерфейс к сетевым расчетным серверам на *Python*,

простые браузерные граф.интерфейсы и расчетки

- **Реляционные (и объектные) базы данных** /MySQL, Postgres (,ZODB,GOODS)/ хранение и простая черновая обработка табличных (объектных) данных экспериментов, справочников, настроек, пользователей.
- **Язык C_+^+, утилиты GNU toolchain** [20, 21] (gcc/g++, make, ld)
  базовый Си, ООП очень кратко[1], без излишеств профессионального программирования[2], чисто вспомогательная роль для написания вычислительных блоков и критичных к скорости/памяти секций, использовать в связке с Python.
  Знание базового Си **критично при использовании микроконтроллеров**, из C_+^+ необходимо владение особенностями использования ООП и управления крайне ограниченной памятью: пользовательские менеджеры памяти, статические классы.
- Использование утилит **flex/bison**
  обработка текстовых форматов данных, часто необходимая вещь.

САПР, пакеты математики, моделирования, визуализации

- **Maxima** символьная математика [18]
- **Octave** численные методы [19]
- **GNUPLOT** простой вывод графиков
- **ParaView/VTK** навороченнейший пакет/библиотека визуализации всех видов
- LaTeX верстка научных публикаций и генерация отчетов
- **KiCAD + ng-spice** электроника: расчет схем и проектирование печатных плат
- **FreeCAD** САПР общего назначения

[1] наследование, полиморфизм, операторы для пользовательских типов, использование библиотеки STL
[2] мегабиблиотека Boost, написание своих библиотек шаблонов и т.п.

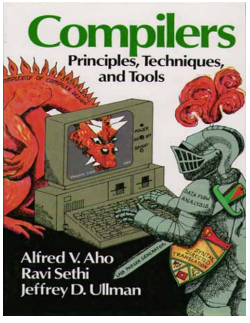
- **Elmer, OpenFOAM** расчетные пакеты метода конечных элементов (мультифизика, сопротивление материалов, конструкционная устойчивость, газовые и жидкостные потоки, теплопроводность)
- **CodeAster + Salome** пакет МКЭ, особо заточенный под сопромат и расчет конструкций
- **OpenModelica** симуляция моделей со средоточенными параметрами[3] (электроника, электротехника, механика, гидропневмоавтоматика и системы управления)
- **V-REP** робототехнический симулятор
- **SimChemistry**[4] интересный демонстрационный симулятор химической кинетики молекул на микроуровне (обсчитывается движение и столкновение отдельных молекул)
- **Avogadro** 3D редактор молекул

[3] для описания моделей элементов использует ООП-язык Modelica

[4] *Windows*

Литература

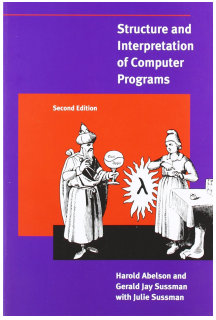
Разработка языков программирования и компиляторов

[1] 

Dragon Book
Компиляторы. Принципы, технологии, инструменты.
Альфред Ахо, Рави Сети, Джеффри Ульман.
Издательство Вильямс, 2003.
ISBN 5-8459-0189-8

[2] **Compilers: Principles, Techniques, and Tools**
Aho, Sethi, Ullman

Addison-Wesley, 1986.
ISBN 0-201-10088-6



[3] **SICP**
Структура и интерпретация компьютерных программ
Харольд Абельсон, Джеральд Сассман
ISBN 5-98227-191-8
EN: web.mit.edu/alexmv/6.037/sicp.pdf

[4] **Функциональное программирование**
Филд А., Харрисон П.

М.: Мир, 1993
ISBN 5-03-001870-0

[5] Функциональное программирование: применение и реализация
П.Хендерсон
М.: Мир, 1983

[6] **LLVM. Инфраструктура для разработки компиляторов**
Бруно Кардос Лопес, Рафаэль Аулер

Lisp/Sheme

Haskell

ML

[7] http://homepages.inf.ed.ac.uk/mfourman/teaching/mlCourse/notes/L01.pdf
Basics of Standard ML
© Michael P. Fourman
перевод 1

[8] http://www.soc.napier.ac.uk/course-notes/sml/manual.html
A Gentle Introduction to ML
© Andrew Cumming, Computer Studies, Napier University, Edinburgh

[9] http://www.cs.cmu.edu/~rwh/smlbook/book.pdf
Programming in Standard ML
© Robert Harper, Carnegie Mellon University

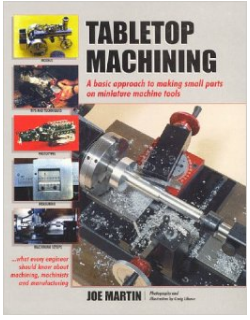
Электроника и цифровая техника

[10] **An Introduction to Practical Electronics, Microcontrollers and Software Design**
Bill Collis
2 edition, May 2014
http://www.techideas.co.nz/

Конструирование и технология

Приемы ручной обработки материалов

Механообработка

[11] 

Tabletop Machining
Martin, Joe and Libuse, Craig
Sherline Products, 2000

[12] **Home Machinists Handbook**
Briney, Doug, 2000

[13] **Маленькие станки**
Евгений Васильев
Псков, 2007
http://www.coilgun.ru/stanki/index.htm

Использование OpenSource программного обеспечения

LaTeX

[14] **Набор и вёрстка в системе LaTeX**
С.М. Львовский
3-е издание, исправленное и дополненное, 2003
http://www.mccme.ru/free-books/llang/newllang.pdf

[15] **e-Readers and LaTeX**
Alan Wetmore
https://www.tug.org/TUGboat/tb32-3/tb102wetmore.pdf

[16] **How to cite a standard (ISO, etc.) in BibLaTeX ?**
http://tex.stackexchange.com/questions/65637/

Математическое ПО: Maxima, Octave, GNUPLOT,..

[17] **Система аналитических вычислений Maxima для физиков-теоретиков**
В.А. Ильина, П.К.Силаев
http://tex.bog.msu.ru/numtask/max07.ps

[18]
Компьютерная математика с Maxima
Евгений Чичкарев

[19]
Введение в Octave для инженеров и математиков
Е.Р. Алексеев, О.В. Чеснокова

САПР, электроника, проектирование печатных плат

Программирование

GNU Toolchain

[20] **Embedded Systems Programming in C_+^+**
© http://www.bogotobogo.com/
http://www.bogotobogo.com/cplusplus/embeddedSystemsProgramming.php

[21] **Embedded Programming with the GNU Toolchain**
Vijay Kumar B.
http://bravegnu.org/gnu-eprog/

JavaScript, HTML, CSS, Web-технологии:

[22] **On-line пошаговый учебник** *JavaScript* на английском, поддерживает множество языков и ИТ-технологий, курс очень удобен и прост для совсем начинающих https://www.codecademy.com

[23] On-line учебник *JavaScript* на русском http://learn.javascript.ru/

Python

[24] **Язык программирования Python**
Россум, Г., Дрейк, Ф.Л.Дж., Откидач, Д.С., Задка, М., Левис, М., Монтаро, С., Реймонд, Э.С., Кучлинг, А.М., Лембург, М.-А., Йи, К.-П., Ксиллаг, Д., Петрилли, Х.Г., Варсав, Б.А., Ахлстром, Дж.К., Роскинд, Дж., Шеменор, Н., Мулендер, С.
© Stichting Mathematisch Centrum, 1990–1995 and Corporation for National Research Initiatives, 1995–2000 and BeOpen.com, 2000 and Откидач, Д.С., 2001
http://rus-linux.net/MyLDP/BOOKS/python.pdf

Python является простым и, в то же время, мощным интерпретируемым объектно-ориентированным языком программирования. Он предоставляет структуры данных высокого уровня, имеет изящный синтаксис и использует динамический контроль типов, что делает его идеальным языком для быстрого написания различных приложений, работающих на большинстве распространенных платформ. Книга содержит вводное руководство, которое может служить учебником для начинающих, и справочный материал с подробным описанием грамматики языка, встроенных возможностей и возможностей, предоставляемых модулями стандартной библиотеки. Описание охватывает наиболее распространенные версии Python: от 1.5.2 до 2.0.

Разработка операционных систем и низкоуровневого ПО

[25] **OSDev Wiki**
http://wiki.osdev.org

Базовые науки

Математика

[26] **Краткий курс математического анализа для ВТУЗов**
Бермант А.Ф, Араманович И.Г.
М.: Наука, 1967
https://drive.google.com/file/d/0B0u4WeMjO894U1Y1dEJ6cncxU28/view?usp=sharing

Пятое издание известного учебника, охватывает большинство вопросов программы по высшей математике для инженерно-технических специальностей вузов, в том числе дифференциальное исчисление функций одной переменной и его применение к исследованию функций; дифференциальное исчисление функций нескольких переменных; интегральное исчисление; двойные, тройные и криволинейные интегралы; теорию поля; дифференциальные уравнения; степенные ряды и ряды Фурье. Разобрано много примеров и задач из различных разделов механики и физики. **Отличается крайней доходчивостью и отсутвием филонианов и "легко догадаться".**

[27] **Математическая статистика** Б.Л. Ван дер Варден

[28] **Алгебра** Б.Л. Ван дер Варден

[29] **Введение в алгебру. В 3 частях. Часть 1. Основы алгебры** А.И. Кострикин

[30] **Введение в алгебру. В 3 частях. Линейная алгебра. Часть 2** А.И. Кострикин

[31] 

Теория вероятностей и математическая статистика
Наум Кремер
М.: Юнити, 2010

[32] 

Цифровая обработка сигналов. Практическое руководство для инженеров и научных работников

Стивен Смит
Додэка XXI, 2008
ISBN 978-5-94120-145-7
В книге изложены основы теории цифровой обработки сигналов. Акцент сделан на доступности изложения материала и объяснении методов и алгоритмов так, как они понимаются при практическом использовании. Цель книги - практический подход к цифровой обработке сигналов, позволяющий преодолеть барьер сложной математики и абстрактной теории, характерных для традиционных учебников. Изложение материала сопровождается большим количеством примеров, иллюстраций и текстов программ

[33] **Начала обработки экспериментальных данных**
Б.А.Князев, В.С.Черкасский
Новосибирский государственный университет, кафедра общей физики, Новосибирск, 1996
http://www.phys.nsu.ru/cherk/Metodizm_old.PDF
Учебное пособие предназначено для студентов естественно-научных специальностей, выполня- ющих лабораторные работы в учебных практикумах. Для его чтения достаточно знаний математики в объеме средней школы, но оно может быть полезно и тем, кто уже изучил математическую статистику, поскольку исходным моментом в нем является не математика, а эксперимент. Во второй части пособия подробно описан реальный эксперимент — от появления идеи и проблем постановки эксперимента до получения результатов и обработки данных, что позволяет получить менее формализованное представление о применении математической статистики. Посо- бие дополнено обучающей программой, которая позволяет как углубить и уточнить знания, полученные в методическом пособии, так и проводить собственно обработ- ку результатов лабораторных работ. Приведен список литературы для желающих углубить свои знания в области математической статистики и обработки данных.

[34] **Принципы современной математической физики** Р. Рихтмайер

[35] **Уравнения математической физики** А.Н. Тихонов, А.А. Самарский

Символьная алгебра

[36] **Компьютерная алгебра**
Панкратьев Евгений Васильевич
МГУ, 2007
Настоящее пособие составлено на основе спецкурсов, читавшихся автором на механико-математическом факультете в течение более 10 лет. Выбор материала в значительной мере определялся пристрастиями автора. Наряду с классическими результатами компьютерной алгебры в этих спецкурсах (и в настоящем пособии) нашли отражение исследования нашего коллектива. Прежде всего, это относится к теории дифференциальной размерности.

[37]

Элементы компьютерной алгебры

Евгений Панкратьев

Год выпуска 2007

ISBN 978-5-94774-655-6, 978-5-9556-0099-4

Учебник посвящен описанию основных структур данных и алгоритмов, применяемых в символьных вычислениях на ЭВМ. В книге затрагивается широкий круг вопросов, связанных с вычислениями в кольцах целых чисел, многочленов и дифференциальных многочленов.

[38]

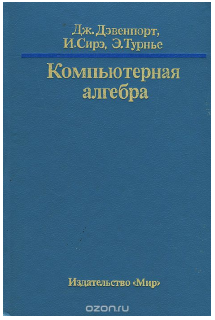
Элементы абстрактной и компьютерной алгебры

Дмитрий Матрос, Галина Поднебесова

2004

ISBN 5-7695-1601-1

В книгу включены следующие главы: алгебры, введение в системы компьютерной алгебры, кольцо целых чисел, полиномы от одной переменной, полиномы от нескольких переменных, формальное интегрирование, кодирование. Разбор доказательств утверждений и выполнение упражнений, приведенных в учебном пособии, позволят студентам овладеть методами решения практических задач, навыками конструирования алгоритмов.



[39] **Компьютерная алгебра**

Дж.Дэвенпорт, И.Сирэ, Э.Турнье

Книга французских специалистов, охватывающая различные вопросы компьютерной алгебры: проблему представления данных, полиномиальное упрощение, современные алгоритмы вычисления НОД полиномов и разложения полиномов на множители, формальное интегрирование, применение систем компьютерной алгебры. Первый автор знаком читателю по переводу его книги "Интегрирование алгебраических функций"
(М.: Мир, 1985).

Численные методы

[40] **Основы вычислительной математики**
Борис Демидович, Исаак Марон
Книга посвящена изложению важнейших методов и приемов вычислительной математики на базе общего втузовского курса высшей математики. Основная часть книги является учебным пособием по курсу приближенных вычислений для вузов.

[41] **Численные методы анализа. Приближение функций, дифференциальные и интегральные уравнения**
Б. П. Демидович, И. А. Марон, Э. З. Шувалова
В книге излагаются избранные вопросы вычислительной математики, и по содержанию она является продолжением учебного пособия [40]. Настоящее, третье издание отличается от предыдущего более доходчивым изложением. Добавлены новые примеры.

Теория игр

[42] **Теория игр**
Петросян Л. А. Зенкевич Н.А., Семина Е.А.
Учеб. пособие для ун-тов. — М.: Высш. шк., Книжный дом «Университет», 1998.
ISBN 5-06-001005-8, 5-8013-0007-4.

[43] **Математическая теория игр и приложения**
Мазалов В.В.
Санкт-Петербург - Москва - Краснодар: Лань, 2010.
ISBN 978-5-8114-1025-5.

[44] **Теория игр**

Оуэн Г.

Книга представляет собой краткое и сравнительно элементарное учебное пособие, пригодное как для первоначального, так н для углубленного изучения теории игр. Для ее чтения достаточно знания элементов математического анализа и теории вероятностей.

Книга естественно делится на две части, первая из которых посвящена играм двух лиц, а вторая — играм N лиц. Она охватывает большинство направлений теории игр, включая наиболее современные. В частности, рассмотрены антагонистические игры, игры двух лиц с ненулевой суммой и основы классической кооперативной теории. Часть материала в монографическом изложении появляется впервые. Каждая глава снабжена задачами разной степени сложности.

Физика

[45] Савельев И.В.

Фейнмановские лекции по физике
Ричард Фейнман, Роберт Лейтон, Мэттью Сэндс

[46] **Современная наука о природе. Законы механики.**

[47] **Пространство. Время. Движение.**

[48] **Излучение. Волны. Кванты.**

[49] **Кинетика. Теплота. Звук.**

[50] **Электричество и магнетизм.**

[51] **Электродинамика.**

[52] **Физика сплошных сред.**

[53] **Квантовая механика 1.**

[54] **Квантовая механика 2.**

[55] **Основы квантовой механики** Д.И. Блохинцев

Химия

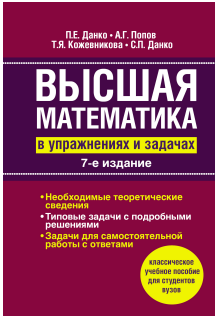
[56] **Квантовая химия. Молекулы, молекулярные системы и твердые тела. Учебное пособие** Владимир Цирельсон

[57] **Вычислительные методы для инженеров-химиков** Х. Розенброк, С. Стори

[58] **Неорганическая химия** В 2 томах
Д. Шрайвер, П. Эткинс

Задачники

Математика



[59] **Высшая математика в упражнениях и задачах**

П.Е. Данко, А.Г.Попов, Т.Я. Кожевникова, С.П. Данко

[60] **Сборник задач по математической физике** Будак Б.М., Самарский А.А., Тихонов А.Н.

[61] **Сборник задач по математике для втузов. В 4 частях. Часть 1. Линейная алгебра и основы математического анализа**
Демидович

Стандарты и ГОСТы

[62] 2.701-2008 **Схемы. Виды и типы. Общие требования к выполнению**
http://rtu.samgtu.ru/sites/rtu.samgtu.ru/files/GOST_ESKD_2.701-2008.pdf

Предметный указатель