AVRForth

1. AVRForth - что это?

AVRForth - это кросс-компилятор языка Форт для 8-битных микроконтроллеров ATMEL. В нем нет возможности добавлять новые слова прямо в микроконтроллере, для этого можете взять AMForth.

Написан на SP-Forth.

- Гарвардская модель – код и данные разделены
- Подпрограммный шитый код
- Стек параметров во внутреннем ОЗУ, стек возвратов аппаратный (во внутреннем ОЗУ)
- Использование внутреннего ОЗУ:

- Область банков регистров общего назначения.

- Область переменных, векторов, массивов – между верхним адресом области регистров и стеком параметров.

- Стек параметров — 16 разрядный

Дно размещается в конце стека возвратов. Стек растет в сторону уменьшения адресов. В качестве верхней ячейки стека параметров используется xh:xl. Указателем на вторую ячейку стека параметров является yh:yl.

- Стек возвратов

Имеет размер в 80 байт (40 адресов). Дно стека возвратов размещается в конце ОЗУ и растет в сторону уменьшения адресов. Указателем вершины стека возвратов является SP.

2. Как компилировать

Для получения файла прошивки .hex вам нужно скомпилировать вашу программу. **В Linux** делается это так:

```
./spf4 S\" program.spf\" S\" deivces/atmega8.spf\" avrforth.spf BYE
```

В Windows аналогично:

```
spf4.exe UNIX-LINES S" %1" S" devices/atmega8.spf" AVRFORTH.spf BYE
```

В результате вы получите файл прошивки program.spf.hex, которым можно прошить контроллер, и файл листинга program.spf.lst

Для облегчения компиляции можно воспользоваться файлом compile, указав в нем нужный контроллер и вызывая так:

```
./compile program.spf
```

или соответственно

```
compile program.spf
```

Кроме того, в каталогах с примерами есть удобные файлы **Makefile**, помогающие компилировать в Linux одной командой make, прошивать командой make flash и удалять ненужные файлы — make clean.

Для конфигурирования проекта в начале файла Makefile задайте:

```
project = tst_buttons
```

файл проекта без расширения

```
device = devices/atmega168.spf
```

файл с определением контроллера

```
compiler = ../../spf4
```

местонахождение SP-Forth

```
avrdude_programmer = usbasp
```

тип программатора для программы avrdude

`avrdude_device = m168`
тип контроллера для программы avrdude

Для всех примеров в каталоге examples есть общий Makefile, который позволяет быстро перекомпилировать все примеры:

```
make clean && make
```

3. Как писать

- Подключение нужных слов

В самом начале вам доступны только управляющие конструкции и определяющие слова, чтобы использовать некоторое слово форта, нужно подключить его, например

```
REQ_T DUP
```

Сделайте это для всех слов вашей программы, что необходимо для минимизации размера прошивки.

- Перечень доступных слов

При разработке я старался придерживаться стандарта, поэтому доступные слова, соответствующие стандарту, будут без комментариев.

Слово	**Стековый комментарий**	**Описание**	**Где определено**
Стековые операции:			
DUP			inline.spf
DROP			inline.spf
NIP			inline.spf
OVER			kernel/over.spf
SWAP			kernel/swap.spf
PICK			kernel/pick.spf
?DUP			kernel/if_dup.spf
2DUP			kernel/2dup.spf
2DROP			kernel/2drop.spf
2OVER			kernel/2over.spf
2SWAP			kernel/2swap.spf
ROT			kernel/rot.spf
-ROT			kernel/minus_rot.spf
TUCK			kernel/tuck.spf
SP@			kernel/sp_fetch.spf
DEPTH			kernel/depth.spf
Арифметика			
0			kernel/zero.spf
1			kernel/one.spf
2			kernel/two.spf
-1			kernel/minus_one.spf

1-			inline.spf
1+			inline.spf
2-			inline.spf
2+			inline.spf
CELL-			inline.spf
CELL+			inline.spf
CELLS			inline.spf
CHAR+			inline.spf
CHARS			inline.spf
+			kernel/plus.spf
-			kernel/minus.spf
LSHIFT	(x1 u -- x2)	Выполняет логический левый сдвиг x1 на u двоичных разрядов и возвращает x2. Помещает нули в самые младшие биты, освобожденные сдвигом.	kernel/lshift.spf
RSHIFT	(x1 u -- x2)	Выполняет правый логический сдвиг x1 на u двоичных разрядов, возвращает x2. Помещает нули в самые старшие биты, освобожденные сдвигом.	kernel/rshift.spf
EXP2	(n --)	Возвести 2 в степень n	kernel/exp2.spf
INVERT			inline.spf
HI	(n -- high)	Возвращает старший байт числа	inline.spf
LO	(n -- low)	Возвращает младший байт числа	inline.spf
NIBBLE-SPLIT	(c -- low high)	Разделяет младший байт числа на две тетрады	kernel/nibble-split.spf
NIBBLE-SWAP	(c1 – c2)	Меняет местами нибблы младшего байта числа	kernel/nibble-swap.spf
NIBBLE-JOIN	(low high -- c)	Объединяет две тетрады в байт	kernel/nibble-join.spf
BYTE-SPLIT	(n -- low high)	Разделяет число на младший и старший байт	kernel/byte-split.spf
BYTE-JOIN	(low high -- c)	Объединяет байты в число	kernel/byte-join.spf
NEGATE			kernel/negate.spf
?NEGATE	(n1 n2 -- n3)	Меняет знак n1, если n2<0	kernel/if_negate.spf
ABS			kernel/abs.spf
MAX			kernel/max.spf
MIN			kernel/min.spf
0MAX	(n1 -- n2)	Эквивалентно 0 MAX	kernel/0max.spf
C>N	(c -- n)	Расширить байт до числа со знаком	kernel/c_to_n.spf
U2/	(u1 -- u2)	Беззнаковое деление на 2	inline.spf
2/			inline.spf
*			kernel/star.spf
2*			inline.spf
*/MOD			kernel/star_slash_mod.spf
*/			kernel/star_slash.spf
/MOD			kernel/slash_mod.spf

MOD			kernel/mod.spf
/			kernel/slash.spf
UM+	(u1 u2 -- ud)	Беззнаковое сложение, результат - двойное без знака	kernel/um_plus.spf
D+			kernel/d_plus.spf
D-			kernel/d_minus.spf
DNEGATE			kernel/dnegate.spf
DABS			kernel/d_abs.spf
S>D			kernel/s_to_d.spf
UD2/	(ud1 -- ud2)	Беззнаковое деление числа двойной длины на 2	kernel/ud2_div.spf
D2/			kernel/d2_div.spf
D2*			kernel/d2_star.spf
UM*			kernel/um_star.spf
UM/MOD			kernel/um_slash_mod.spf
MU/MOD	(ud1 u1 -- ud2 u2)	Делит ud1 на u1, возвращает частное ud2 и остаток u2. Все значения и арифметика без знака.	kernel/mu_slash_mod.spf
UD*	(ud1 u -- ud2)	Беззнаковое умножение числа двойной длины на одинарное, результат двойной	kernel/ud_star.spf
M*			kernel/m_star.spf
FM/MOD			kernel/m_slash_mod.spf
Стек возвратов			
R@			kernel/rfetch.spf
>R			kernel/to_r.spf
DUP>R	(n -- n)	Эквивалентно DUP >R	kernel/dup_to_r.spf
R>			kernel/from_r.spf
RDROP	(--)	Эквивалентно R> DROP	kernel/rdrop.spf
Работа с памятью			
C!			kernel/c_store.spf
C@			kernel/c_fetch.spf
C@P	(c-addr -- char)	Выбирает байт из памяти программ	kernel/c_fetchp.spf
!			kernel/store.spf
+!			kernel/plus_store.spf
@			kernel/fetch.spf
@P	(addr -- n)	Выбирает число из памяти программ	kernel/fetchp.spf
INCR	(addr --)	Увеличивает значение по адресу на 1	kernel/incr.spf
DECR	(addr --)	Уменьшает значение по адресу на 1	kernel/decr.spf
ON	(addr --)	Записывает -1 по адресу	kernel/on.spf

OFF	(addr --)	Записывает 0 по адресу	kernel/off.spf
2@			kernel/2_fetch.spf
2!			kernel/2_store.spf
FILL			kernel/fill.spf
ERASE			kernel/erase.spf
CMOVE			kernel/cmove.spf
CMOVEP	(c-addr-rom c-addr-ram u --)	копировать из ROM в RAM u байт	kernel/cmovep.spf
C@E	(c-addr -- char)	Выбирает байт из EEPROM	kernel/c_fetch_e.spf
@E	(addr -- n)	Выбирает число из EEPROM	kernel/fetch_e.spf
C!E	(c c-addr --)	Записывает байт в EEPROM	kernel/c_store_e.spf
!E	(n addr --)	Записывает число в EEPROM	kernel/store_e.spf
Логические операции			
AND			kernel/and.spf
OR			kernel/or.spf
XOR			kernel/xor.spf
>			kernel/more.spf
>=			kernel/more_or_equal.spf
<			kernel/less.spf
U<			kernel/u_less.spf
0<			kernel/zero_less.spf
0=			kernel/zero_eq.spf
=			kernel/equal.spf
<>			kernel/not_equal.spf
WITHIN			kernel/within.spf
D0<>			kernel/d_zero_not_eq.spf
D<			kernel/d_less.spf
D0<			kernel/d_zero_less.spf
Строки			
C”	Компиляция: ("ccc<quote>" --) Время-выполнения: (-- c-addr)	Выделяет ccc, ограниченное " (двойная кавычка), и добавляет семантику времени-выполнения данную ниже к текущему определению. Возвращает c-addr строку со счетчиком, состоящую из символов ccc. Строка находится в памяти программ.	xfbase.f
COUNT			kernel/count.spf
COUNTP		То же, что и COUNT для строки, находящейся в памяти программ	kernel/countp.spf
SP”		То же, что и S” в обычном форте, но дает адрес в памяти программ	kernel/sp_quote.spf

."			kernel/dot_quote.spf
EMIT		Вектор, который нужно переопределить, по умолчанию не указывает ни на какое слово	kernel/emit.spf
TYPE			kernel/type.spf
TYPEP		То же, что и TYPE для строки, находящейся в памяти программ	kernel/typep.spf
SPACE			kernel/space.spf
SPACES			kernel/spaces.spf
CR			kernel/cr.spf
Форматный вывод			
>DIGIT	(n -- char)	Преобразовать цифру в символ	kernel/digit.spf
BASE			kernel/num_conv.spf
HLD		Переменная - текущее место в буфере для преобразования	kernel/num_conv.spf
PAD-SIZE		Константа - размер буфера для преобразования	kernel/num_conv.spf
PAD		Буфер для преобразования	kernel/num_conv.spf
NUMPAD		Константа — конец буфера для преобразования	kernel/num_conv.spf
DECIMAL			kernel/num_conv.spf
HEX			kernel/num_conv.spf
<#			kernel/num_conv.spf
HOLD			kernel/num_conv.spf
#			kernel/num_conv.spf
#S			kernel/num_conv.spf
SIGN			kernel/num_conv.spf
#>			kernel/num_conv.spf
(UD.)	(ud — addr len)	Преобразовать двойное беззнаковое	kernel/u_dot.spf
(U.)	(u — addr len)	Преобразовать беззнаковое	kernel/u_dot.spf
U.			kernel/u_dot.spf
(D.)	(d — addr len)	Преобразовать двойное со знаком	kernel/_dot.spf
(.)	(n — addr len)	Преобразовать число со знаком	kernel/_dot.spf
.			kernel/dot.spf
(0.R)	(n width -- addr len)	Преобразовать число со знаком, выровнять по правому краю в поле из width знаков, остальное заполнить нулями	kernel/_zero_dot_r.spf
(.R)	(n width -- addr len)	Преобразовать число со знаком, выровнять по правому краю в поле из width знаков, остальное заполнить пробелами	kernel/_dot_r.spf
(U.R)	(u width -- addr len)	Преобразовать число без знака, выровнять по правому краю в поле из width знаков, остальное заполнить пробелами	kernel/_u_dot_r.spf
.S			kernel/dot_s.spf

.SN	(n --)	Распечатать n верхних элементов стека	kernel/dot_sn.spf
Работа с портами		***Здесь только слова низкого уровня, см. также соответствующий раздел документации***	
C@	(addr -- c)	Прочитать байт из порта Использование: DDRC C@	
C!	(c addr --)	Записать байт в порт 0x7 DDRC C!	
SET	(mask addr --)	Прочитать байт из порта, выполнить логическое «или» с маской и записать обратно в порт. Значение маски и порта должны быть известны во время компиляции. Использование: 7 BIT DDRC SET	inline.spf
MASK	(mask addr --)	Прочитать байт из порта, выполнить логическое «и» с маской и записать обратно в порт. Значение и порт должны быть известны во время компиляции. Использование: 7 BIT DDRC SET	inline.spf
TOGGLE	(mask addr --)	Прочитать байт из порта, выполнить исключающее «или» с маской и записать обратно в порт. Значение и порт должны быть известны во время компиляции. Использование: 7 BIT PORTB TOGGLE	inline.spf
CLEAR	(mask addr --)	Прочитать байт из порта, выполнить логическое «или» с инвертированной маской и записать обратно в порт (очистить биты). Значение и порт должны быть известны во время компиляции. Использование: 7 BIT PORTB CLEAR	inline.spf
BITS{	(-- -1)		Макрос, xfbase.f
}BITS	(-1 b1 b2 … bx -- n)	Получить маску из номеров установленных битов. Скомпилировать как литерал, если в режиме целевой компиляции, иначе оставить маску на стеке Использование: BITS{ WGM01 WGM00 COM0A1 }BITS	Макрос, xfbase.f
BIT	(n -- mask)	Получить маску из номера установленного бита (должен быть известен во время компиляции). Скомпилировать как литерал, если в режиме целевой компиляции. Иначе — положить маску на стек. Использование: WGM01 BIT	Макрос, xfbase.f
Разное			
EXECUTE			kernel/execute.spf
[']			Макрос, xfbase.f
‘			Макрос, xfbase.f

LITERAL			inline.spf
2LITERAL			inline.spf
REBOOT	(--)	Перезагрузить контроллер	inline.spf
EXIT	(--)	Выйти из определения	inline.spf
\	(–)	Комментарий до конца строки	xfbase.spf
\\\	(–)	Комментарий до конца строки, дублируется в листинге	avrdis.f
Управляющие структуры			
IF			inline.spf
ELSE			inline.spf
THEN			inline.spf
BEGIN			inline.spf
AGAIN			inline.spf
REPEAT			inline.spf
WHILE			inline.spf
UNTIL			inline.spf
[UNTIL<>0]	(flag — flag)	Как обычный 0 <> UNTIL, не разрушает вершину стека.	inline.spf
DO			inline.spf
?DO			inline.spf
LOOP			inline.spf
+LOOP			inline.spf
I			inline.spf
J			kernel/j.spf
LEAVE			inline.spf
UNLOOP			inline.spf
Компиляция			
ALLOT	(n --)	Зарезервировать место для n байт в памяти RAM	xfbase.f
TALLOT	(n --)	Зарезервировать место для n байт в памяти программ	xfbase.f
EALLOT	(n --)	Зарезервировать место для n байт в EEPROM	xfbase.f
[	(--)	Выключить режим целевой компиляции	xfbase.f
]	(--)	Включить режим целевой компиляции	xfbase.f
[[		Слово [из словаря FORTH	xfbase.f
]]		Слово] из словаря FORTH	xfbase.f
[T]		На стек словарей положить TARGET	xfbase.f
[P]		Эквивалентно PREVIOUS	xfbase.f
[F]		На стек словарей положить FORTH	xfbase.f
T,	(n --)	Скомпилировать число в память программ	flash.spf

TC,	(c --)	Скомпилировать байт в память программ	flash.spf
THERE	(-- a)	Текущий адрес компиляции	flash.spf
TALIGN	(--)	Выровнять указатель компиляции	flash.spf
E-HERE	(-- a)	Текущий адрес в EEPROM	xfbase.f
D-HERE	(-- a)	Текущий адрес в RAM	xfbase.f
Определяющие слова			
:			xfbase.f
CALL-ONLY:		Начать определение, которое исключается из процедуры оптимизации хвостовой рекурсии (то есть может вызываться только через rcall/call)	xfbase.f
;			xfbase.f
MACRO:		Начать макроопределение	xfbase.f
;MACRO		Закончить макроопределение	xfbase.f
INT:		Начать обработчик прерываний	isr.f
;INT		Закончить обработчик прерываний	isr.f
ASMINT:		Начать обработчик прерываний на ассемблере	isr.f
;ASMINT		Закончить обработчик прерываний на ассемблере	isr.f
CREATE		См. Использование определяющих слов	xfbase.f
TCREATE		См. Использование определяющих слов	xfbase.f
PCREATE		См. Использование определяющих слов	xfbase.f
RCREATE		См. Использование определяющих слов	xfbase.f
ECREATE		См. Использование определяющих слов	xfbase.f
VARIABLE		Переменная в RAM	xfbase.f
CVARIABLE		Байтовая переменная в RAM	xfbase.f
2VARIABLE		4-х байтовая переменная в RAM	xfbase.f
EVARIABLE		Переменная в EEPROM	xfbase.f
CEVARIABLE		Байтовая переменная в EEPROM	xfbase.f
2EVARIABLE		4-х байтовая переменная в EEPROM	xfbase.f
CONSTANT ==		Константа, компилируется как литерал	xfbase.f
2CONSTANT 2==		Двойная константа, компилируется как двойной литерал	xfbase.f
FCONST		Обычная константа форта, не компилируется в определение	xfbase.f
PDOES>		См. Использование определяющих слов	xfbase.f
DOES>		См. Использование определяющих слов	xfbase.f
[VECT]		Определение статического вектора	xfbase.f
[->]		Назначить статический вектор. Только для режима интерпретации. Использование: ' WORD [->] VECTOR	xfbase.f

->INT		Назначить обработчик прерывания. Только для режима интерпретации. Использование: ' HANDLER ->INT VECTOR	isr.f
VECT		Определение вектора, изменяемого во время выполнения программы	kernel/dovect.spf
IS		Назначить вектор. Только для режима компиляции. Использование: ['] WORD IS VECTOR	kernel/dovect.spf
Переменные, константы			
mega8515, mega168, mega8, mega48, mega88, mega128		Константы, соответствующие моделям контроллеров	devices/devices.spf
DEVICE		Константа, содержит текущую модель контроллера	devices/xxxxxx.spf
ROM-SIZE		VALUE, Размер ПЗУ	devices/xxxxxx.spf
RAM-BOTTOM		VALUE, Начало ОЗУ	devices/xxxxxx.spf
RAM-SIZE		VALUE, Размер ОЗУ	devices/xxxxxx.spf
RAM-TOP		VALUE, Конец ОЗУ	devices/xxxxxx.spf
RSTACK-SIZE		VALUE, Размер стека возвратов	devices/xxxxxx.spf
SPTR0		VALUE, Дно стека данных	devices/xxxxxx.spf
VECTOR-SIZE		VALUE, Размер вектора прерываний в таблице прерываний (байт)	devices/xxxxxx.spf
HAS_MUL?		VALUE, Флаг — истина, если в контроллере есть аппаратный умножитель	devices/xxxxxx.spf
FALSE	(-- 0)	Константа	xfbase.f
TRUE	(-- -1)	Константа	xfbase.f
BL	(-- 0x20)	Константа	xfbase.f

- Скелет программы

В простейшем случае ваша программа должна выглядеть так:

```
S" kernel/dovar.spf" INCLUDED  \ если нужны определяющие слова
\ включаем необходимые слова
REQ_T DUP
REQ_T DROP

: MAIN
      DUP DROP
;
\ задаем точку входа
MAIN [->] {MAIN}
```

- Использование определяющих слов

Так как в микроконтроллерах AVR есть целых три адресных пространства (Flash, RAM, EEPROM), пришлось создать три набора определяющих слов для них.

Flash

PCREATE, PDOES> - аналоги фортовских CREATE, DOES> используются для создания определяющих слов с параметрами в Flash. Пример:

```
\ константа
:: ===
      PCREATE T,
      PDOES> @P ;
;;
0x1 === ONE
: TEST
  ONE +
;
```

Заметьте, нужно использовать «::» , «;» и «;;» обязательно! И конечно, после PDOES> используйте выборку из Flash: @P или C@P.

Для создания массива констант в Flash используйте TCREATE:

```
TCREATE ROM
0x1 T, 2 T, 3 T, 4 T,
: TEST
  ROM 1 CELLS + P@
;
```

RAM

Используйте стандартные CREATE, DOES>, ALLOT например:

```
:: VECTOR
  CREATE [F] 2* [P] ALLOT
  DOES> SWAP CELLS + ;
;;

10 VECTOR X

: TEST
  1 3 VECTOR !
  3 VECTOR @
;
```

Для создания массива в RAM можно использовать RCREATE:

```
RCREATE RAM5 0x5 ALLOT
: TEST
  RAM5 2 CHARS + C@
;
```

EEPROM

Для этого типа памяти используется ECREATE, EALLOT:

```
ECREATE eeVAR
10 EALLOT

: TEST
  eeVAR 1+ C@E
;
```

- Константы, переменные

Для определения константы используйте слова «==», «2==»:

```
32 == BL
```

Константы компилируются в определения как литералы. Для создания обычной константы форта используйте FCONST.

Для переменных в RAM и EEPROM используются соответственно VARIABLE, CVARIABLE и EVARIABLE, CEVARIABLE:

```
EVARIABLE eeVAR
CVARIABLE VAR
: TEST
  eeVAR E@ VAR C!
;
```

- Использование макросов

Макросы позволяют включать код напрямую в слово, не используя вызов слов-подпрограмм:

```
: z
  DUP
  DUP ;

MACRO: M1
  z
  DROP
;MACRO

: MACRO-TEST
  M1 M1
;
: MACRO-TEST1
  z DROP z DROP
;
\ оба слова в листинге будут выглядеть одинаково
```

Вместо [и] в макросах следует использовать T[и]T.

- Доступ к портам ввода/вывода

Для этого существует три способа:

1. Низкий уровень

В файлах устройств devices/xxx.spf определены константы (со значениями регистров) PORTX, PINX, DDRX для каждого порта ввода/вывода.

Для чтения и записи регистров используйте C@ и C! соответственно.

Кроме того, существуют несколько полезных слов: SET, MASK, CLEAR, TOGGLE:

```
PORTB == BTN_PORT \ обычная константа
DDRB  == BTN_DDR
PINB  == BTN_PIN
0x1 == BUTTON           \ PB1 - сюда подключена кнопка

PORTD == LED_PORT
DDRD  == LED_DDR
0x4 == LED              \ PD4 – сюда подключен светодиод

: MAIN
  BITS{ CS01 CS00 }BITS TCCR0 C!  \ установить биты CS01 и CS00 в регистре  TCCR0,
остальные очистить
  LED BIT     LED_DDR SET               \ нога с светодиодом на выход
  BUTTON BIT  BTN_DDR CLEAR        \ нога с кнопкой на вход
  BUTTON BIT  BTN_DDR SET               \ подтяжка
  BEGIN
    PINB C@
    PORTD C!
  AGAIN
;
```

2. С помощью библиотеки lib/ports.spf:

В ней определены слова {PORTX}, которые оставляют на стеке адрес структуры порта ввода/вывода:

```
\ структура порта
0x0
CELL -- port
CELL -- ddr
CELL -- pin
DROP
```

Кроме того, для каждой линии порта есть слова PXX, например PB0, PD7 и т. д. Они оставляют на стеке свою маску и порт (адрес структуры).

Слово WIRE позволяет поименовать линию порта, а слова GROUP{ }GROUP создать группу линий **одного порта**.

```
PD7 WIRE POWER_LED      \ светодиод
PB0 WIRE BTN_POWER         \ кнопки
PB1 WIRE BTN_UP
PB2 WIRE BTN_DOWN
PB3 WIRE BTN_OK
PB4 WIRE BTN_CANCEL
GROUP{ BTN_POWER BTN_UP BTN_DOWN BTN_OK BTN_CANCEL }GROUP BUTTONS
```

Линиями и группами линий можно манипулировать с помощью слов {SET}, {TOGGLE}, {CLEAR}, {OUTPUT}, {INPUT}, {PULL_UP}, например

```
    BUTTONS {INPUT}     \\ настроить группу линий на вход
    BUTTONS {PULL_UP}   \\ включить подтяжку
    POWER_LED {OUTPUT}  \\ на выход
    LED {TOGGLE}     \\ переключть
    PD7 {TOGGLE}     \\ или так
    LED {SET}        \\ установить высокий уровень на выходе
```

Слова {READ} и {WRITE} читают из входного регистра порта группы и записывают в выходной регистр соответственно.

PORT.READ и PORT.WRITE читают и пишут конкретный порт, например:

```
{PORTB} PORT.READ \ можно так
BUTTONS {READ}          \ или так
```

Все эти слова сведены в таблицу:

Слово	**Стековый комментарий (h: - стек форт-системы хоста)**	**Описание**
{PORTA} .. {PORTF}	(h: -- port_addr) (--)	Порты ввода/вывода. Оставляют на стеке адрес структуры порта ввода/вывода
PXX	(h: -- mask port_addr) (--)	Линии портов ввода/вывода. Оставляют на стеке маску и адрес структуры порта ввода/вывода
GROUP{	(h: -- -1)	Начало определения группы линий.
}GROUP name	(h: -1 mask0 port0 mask1 port1 .. maskx portx --) исполнение (h: -- mask port)	Определить группу линий порта ввода/вывода
+GROUP	(h: mask0 port0 mask1 port1 -- mask port)	Объединить две группы линий в одну
-GROUP	(h: mask0 port0 mask1 port1 -- mask port)	Изъять линии второй группы линий из первой
WIRE name	компиляция (h: mask port --) исполнение (h: -- mask port)	Определить линию порта ввода/вывода (задать имя)
{GROUP.PORT}	(h: mask port -- port)	Компилировать номер регистра PORT группы как литерал

	(-- port_reg)	(в режиме целевой компиляции) или оставить адрес структуры порта на стеке хоста
{GROUP.MASK}	(h: mask port -- port) (-- mask)	Компилировать маску группы как литерал (в режиме целевой компиляции) или оставить на стеке хоста
{SET}	(h: mask port --) (--)	Установить на выходе линий группы «1»
{TOGGLE}	(h: mask port --) (--)	Переключить выходы линий группы
{CLEAR}	(h: mask port --) (--)	Установить на выходе линий группы «0»
{OUTPUT}	(h: mask port --) (--)	Настроить линии группы на выход
{INPUT}	(h: mask port --) (--)	Настроить линии группы на вход
{PULL_UP}	(h: mask port --) (--)	Подключить подтягивающие резисторы
{READ}	(h: mask port --) (– c)	Прочитать значение из регистра PIN группы
{WRITE}	(h: mask port --) (c --)	Записать значение в регистр PORT группы
PORT.READ	(h: port --) (-- c)	Прочитать значение из регистра PIN порта ввода/вывода
PORT.WRITE	(h: port --) (c --)	Записать значение в регистр PORT порта ввода/вывода

3. С помощью библиотеки lib/bus.spf

У предыдущего способа есть существенное ограничение: в группе можут быть линии только одного порта. Тут мы вводим понятие шины как совокупности линий любых портов ввода/вывода. Максимальная разрядность шины — 16 бит. Шина определяется так же, как и группа, например:

```
\     15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
BUS{                        PC3 PD7 PD6 PD5 PD4 PD3 PC4 PB3 PB1 PB0 }BUS DATA1
BUS{                        PC3 PD7 PD6 PD5 PD4 PD3 PD2 PD1 PD0 PB2 }BUS CONTROL
BUS{     PB3 PB2 PB1 PB0 PB7 PB6 PB5 PB4 PC7 PC6 PC5 PC4 PC3 PC2 PC1 }BUS DATA2
BUS{                                PC0 PC1 PC2 PC3 PC4 PC5 PC6 PC7 }BUS DATA3
BUS{                                                            PD7 }BUS DATA4
BUS{ PB7 PB6 PB5 PB4 PB3 PB2 PB1 PB0 PC7 PC6 PC5 PC4 PC3 PC2 PC1 PC0 }BUS DATA5
BUS{                                PC7 PC6 PC5 PC4 PC3 PC2 PC1 PC0 }BUS DATA6
BUS{                        PD2 PC1 PC0 PB6 PB7 PD1 PD0 PB5 PB4 PB2 }BUS BUTTONS
```

При записи значения в шину биты этого значения запишутся в соответствующие им линии портов (например для шины CONTROL младший бит значения попадет на выход линии PB2, 1-й бит — на линию PD0 и т. д.). При чтении шины произойдут обратные преобразования. Компилятор постарается сгенерировать оптимальный код, хотя обычно он получается достаточно объемным. Поэтому запись в шину/чтение из шины лучше оформить отдельным словом. Если в шину записывается константа, код будет гораздо меньше по размеру.

Примеры:

```
DATA1 BUS.OUTPUT        \\ настроить шину на выход
BUTTONS BUS.INPUT       \\ на вход
BUTTONS BUS.PULL_UP     \\ подтягивающие резисторы
0x00 DATA1 BUS.WRITE    \\ запись константы
DATA6 BUS.READ DATA3 BUS.WRITE     \\ чтение шины и запись в другую шину
```

Слово	**Стековый комментарий (h: - стек форт-системы хоста)**	**Описание**

BUS{	(h: -- -1)	Начало определения шины
}BUS name	(h: -1 mask0 port0 mask1 port1 .. maskx portx --) исполнение (h: --)	Определить шину
BUS.WRITE	(h: --) (w --)	Записать значение в линии шины
BUS.DDR.WRITE	(h: --) (w --)	Записать значение в регистры DDR портов шины
BUS.INPUT	(h: --) (--)	Настроить линии шины на вход
BUS.OUTPUT	(h: --) (--)	Настроить линии шины на выход
BUS.PULL_UP	(h: --) (--)	Подключить подтягивающие резисторы
BUS.SET	(h: --) (--)	Вывести на линии шины «1»
BUS.CLEAR	(h: --) (--)	Вывести на линии шины «0»
BUS.READ	(h: --) (-- w)	Чтение шины

- Встроенный ассемблер

В форт-словах можно использовать встроенный ассемблер. Он постфиксный, кроме того, константы должны обрамляться квадратными скобками. Порядок операндов:

```
источник приемник команда
```

Для некоторых регистров контроллера введены мнемоники для лучшей переносимости:

```
wl CONSTANT cntl
wh CONSTANT cnth
\ wh:wl - временный регистр, счетчик цикла DO
r4 CONSTANT cbl
r5 CONSTANT cbh
\ r4:r5 - верхний предел в цикле DO
r0 CONSTANT tmpl
r1 CONSTANT tmph
\ r0:r1 - временный регистр
xl CONSTANT tosl
xh CONSTANT tosh
\ xh:xl - число на вершине стека данных
```

Кроме этих, используются такие регистры

```
\ yh:yl - указатель на второе число в стеке данных
\ zh:zl - временный регистр
\ sph:spl – указатель стека возвратов
\ r16:r17 - временный
\ r18:r19 - временный
Все временные регистры можно свободно изменять.
```

Есть полезные макросы:

Макрос	**Описание**	**Пример**
(CALL)	Вызов слова, независимо от того, длинный адрес или короткий	`' OVER (CALL)`
(JMP)	Переход на слово, независимо от того, длинный адрес или короткий	`' OVER (JMP)`
pushd	Поместить регистровую пару в стек данных,	`tmpl pushd`

	вершина остается неизменной	
popd	Снять регистровую пару со стека данных, вершина остается неизменной	`tmpl popd`
pushtmp	Поместить временный регистр в стек данных, вершина остается неизменной	`pushtmp`
poptmp	Снять временный регистр со стека данных, вершина остается неизменной	`poptmp`
pusht	Эквивалентно DUP	`pusht`
popt	Эквивалентно DROP	`popt`
pushw	Сохранить пару регистров в стеке возвратов	`r16 pushw`
popw	Восстановить пару регистров из стека возвратов	`r16 popw`
_in	Чтение порта, в зависимости от порта используются инструкции in или lds	`PIND r16 _in`
_out	Запись в порт, в зависимости от порта используются инструкции out или sts	`r16 DDRD _out`

В ассемблере реализованы некоторые управляющие структуры:

if_z if_< if_u< if_>= if_<0 if_<>0 if_c then else	Используются флаги контроллера, вершина стека не изменяется Можно комбинировать с ELSE и THEN форта

begin
again
while_<>0
repeat
until_<>0
until_=0
	Используются флаги контроллера, вершина стека не изменяется

for
<register> next
	Цикл по некоторому регистру. Задайте нужное значение регистра, в конце цикла он будет декрементироваться и сравниваться с 0, например
	`0x4 r16 ldi`
	`clc`
	`for`
	`  -[x] tmpl ld`
	`  -[z] tmph ld`
	`  tmph tmpl adc`
	`  tmpl [z] st`
	`r16 next  \ 4 раза`

Обработка прерываний

Сначала создайте обработчик прерываний словами: INT:, ASMINT:

INT: создает обрабочик прерывания на форте, сохраняются все описанные выше регистры и регистр состояния, например:

```
INT: TIMER
    SHOW_DIGIT
```

```
    RTOS_TIMER_SERVICE
;INT
```
ASMINT: создает обрабочкик прерывания на ассемблере, сохраняется только вершина стека и регистр состояния, например:

```
ASMINT: TIMER
    [ r18 inc
    ]
;ASMINT
```
Задать обработчик прерывания можно словом ->INT во время компиляции.

```
' TIMER ->INT TIMER0_OVF     \ задать обработчик прерывания по таймеру
```

Оптимизация

На данный момент в системе оптимизируется использование двух последних литералов некоторыми словами:

- C@ - один литерал. Если адрес известен во время компиляции, генерируется более оптимальный код.
- C! - один или два литерала. Возможна компиляция с использованием инструкции out.
- AND, OR, XOR — один литерал.
- «+», «-» - один литерал, используются adiw, sbiw.
- SET, CLEAR, TOGGLE, MASK — требуют два литерала, если возможно, используются sbi, cbi.
- BIT — требует один литерал, номер бита заменяется его маской и компилируется опять как литерал.
- DROP + один литерал, генерируется более оптимальный код
- DO, ?DO - один или два литерала
- +LOOP - один литерал
- Слова из lib/ports.spf, lib/bus.spf.

Кроме того, оптимизируется хвостовая рекурсия, удаляются лишние DUP/DROP.

Отладочные сообщения

Для облегчения отладки при компиляции и при работе программы введены так называемые уровни отладки:

- уровень 0: отладочные сообщения отключены
- уровень 1: отладочные сообщения уровня 1 включены
- уровень 2: отладочные сообщения уровней 1 и 2 включены
- уровень 3: отладочные сообщения уровней 1, 2 и 3 включены

На уровне 3 выводятся сообщения оптимизатора (их может быть много).

Уровни отладки задаются словами -DEBUG, +DEBUG (эквивалентно +DEBUG1), +DEBUG1, +DEBUG2, +DEBUG3, которые отключают отладку или включают соответствующие уровни вывода отладочных сообщений.

Использование в двоеточечных определениях для целевой системы

Используется конструкция [IFDEBUG] | [IFDEBUG1] | [IFDEBUG2] | [IFDEBUG3] … [ELSE] … [THEN]. Поскольку в определениях используется режим интерпретации, ненужный код не будет включен в определение, например

```
: TST
  [IFDEBUG]
    1200 BAUD USART_INIT
```

```
  [THEN]
  …
;
```
Здесь при включении отладки инициализируется USART, при выключенной отладке соответствующий код не будет скомпилирован. [IFDEBUG] можно использовать и вне определений

Использование в макросах

Поскольку макросы — это обычные форт-слова, при их компиляции используется режим компиляции. Конструкция почти аналогична: [IFDEBUG] | [IFDEBUG1] | [IFDEBUG2] | [IFDEBUG3] … **ELSE** … [THEN]. Обратите внимание на ELSE. [IFDEBUG] скомпилируется как обычный IF, поэтому уровень отладки будет проверяться каждый раз при выполнении макроса. Удобно использовать слова DEBUG", DEBUG1", DEBUG2", DEBUG3" сразу для вывода сообщений. Например:

```
: CELLS HAS CELLS  ( x1 -- x2 )           \   ( multiply by cell size)
[F]
  QLIT1? IF
    DEBUG3" CELLS"
    2* (LIT)
  ELSE
    tosl lsl
    tosh rol
    EOPT
  THEN
[P]
;
```
Здесь текущий адрес компиляции и текст «CELLS» выводится только на уровне отладки 3.

Замечания

1. Так как AVRForth — это кросс-транслятор, то внутри всех ваших определений (высокоуровневых и ассемблерных) не используется режим компиляции, а только интерпретации, то есть STATE всегда 0. В то же время переменная T-STATE показывает, будут ли компилироваться слова в целевую память. Некоторые слова зависят от состояния T-STATE. Использование определений целевой системы в режиме интерпретации приводит к ошибке.
2. Можно подключить слова -1, 0, 1, 2, которые экономят ПЗУ, так как они оформлены не литералами, а отдельными словами. В связи с этим использовать не в режиме целевой компиляции эти слова нельзя:
   `: TEST   0 1 + ;`
   сработает, а
   `DEVICE 0 = [IF] … [THEN]`
   нет, так как 0 выдаст ошибку. Для устранения этого эффекта применяйте вместо 0 — 0x0 или 00. Также можете использовать 00, если знаете, что следующее слово оптимизировано для использования литералов.
3. Числа компилируются как литералы (в режиме целевой компиляции), поэтому в ассемблерных вставках обрамляйте числа (или всю вставку) квадратными скобками:
   `[ 0x1 ] tosl adiw`
   или
   `[  0x1 tosl adiw ]`
4. Полезное слово \\ позволяет включать комментарии в листинг.
5. Во время компиляции доступен только словарь TARGET. Для использования слов из словаря FORTH делайте так:
   `[F] bla bla [P]`
   Слова [F] и [P] — немедленного исполнения.
   Некоторые слова из словаря форта переопределены в TARGET: это
   `[[ (вместо [),`
   `]] (вместо ]),`
   `:: (вместо :),`
   `;; (вместо ;),`
   `INCLUDED, REQUIRE, \EOF, CHAR,`

```
FCONST (вместо CONSTANT),
.S, ORDER, \, (, TO, S», [IF], [ELSE], [THEN]
```

6.

4. Библиотеки

Библиотеки подключаются обычным REQUIRE

delay.spf

Программные задержки

Подключение (обязательно задайте частоту контроллера):

```
8000000 ( Hz ) == F_CPU
REQUIRE MS lib/delay.spf
```

Использование:

MS	(n --)	Задержка на n миллисекунд
US	(us --)	Задержка на us микросекунд, должно быть известно во время компиляции.

i2c.spf

Реализация аппаратного интерфейса I2C/TWI

Подключение:

```
REQUIRE I2C_START lib/i2c.spf
```

Использование:

I2C_INIT	(--)	Инициализация интерфейса
I2C_START	(address -- err)	Послать стартовую посылку устройству с выбранным адресом. Возвращает 0 — все ок, -1 — ошибка
I2C_STOP	(--)	Послать стоповую посылку
I2C_WRITE	(data -- err)	Послать байт. Возвращает 0 — все ок, 1 — ошибка
I2C_READ_ACK	(-- byte)	Прочитать байт, запросить следующую порцию данных.
I2C_READ_NAK	(-- byte)	Прочитать байт, окончить передачу.

ds1307.spf

Общение с контроллером ds1307 по аппаратному I2C

Подключение:

```
REQUIRE DS1307      lib/ds1307.spf
```

Использование:

DS1307_INIT	(-- 0 | -1)	Проверка наличия подключенной микросхемы, инициализация. Возвращает 0 — есть, -1 - нет
DS1307_GET	(reg -- value)	Получить значение регистра, например DS1307_Weekday DS1307_GET
DS1307_SET	(reg value --)	Установить значение регистра
DS1307_GET_TIME	(-- s m h)	Прочитать время
DS1307_SET_TIME	(h m s --)	Записать время
DS1307_GET_DATE	(-- d m y)	Прочитать дату
DS1307_SET_DATE	(y m d --)	Записать дату

eertos.spf

Простой диспетчер задач. Основан на диспетчере DiHALTa (easyelectronics.ru)

Суть в том, чтобы иметь возможность запускать задания по таймеру через некоторое количество тиков. В качестве задания может выступать любое двоеточечное определение форта. По умолчанию один тик равен 1 мс. В прерывании таймера уменьшаются счетчики каждой задачи. В основном цикле менеджер задач проверяет, есть ли задачи, у которых счетчик достиг 0 и запускает их.

Подключение (обязательно задайте частоту контроллера):

```
8000000 ( Hz ) == F_CPU
REQUIRE RTOS_INIT   lib/eertos.spf
```

Использование:

RTOS_INIT	(--)	Инициализация диспетчера
RTOS_RUN	(--)	Запуск диспетчера
RTOS_SET_TIMER_TASK	(task time --)	Поставить задачу в очередь на выполнение через time тиков. Пример: ['] GET_TIME 250 RTOS_SET_TIMER_TASK \ через 250 мс
RTOS_SET_TASK	(task --)	Поставить задачу в очередь на выполнение немедленно
RTOS_REMOVE_TASK	(task --)	Удалить задачу из очереди
RTOS_TASK_MANAGER	(--)	Менеджер задач. Необходимо вызывать в цикле главной процедуры. Именно из него задачи вызываются на выполнение.
RTOS_TIMER_SERVICE	(--)	Служба таймеров диспетчера. Вызывается из прерывания по таймеру.
RTOS_INT	(--)	Обработчик прерывания по таймеру.

Пример использования:

```
8000000 ( Hz ) == F_CPU

\ определения портов
PORTD == LEDS_PORT
    07  == LED
DDRD  == LEDS_DDR

REQUIRE RTOS_INIT   lib/eertos.spf

: LED_ON
    LED BIT  LEDS_PORT CLEAR
;

: LED_OFF
    LED BIT  LEDS_PORT SET
;

: BLINK         \ мигнуть светодиодом на 100 мс
    LED_ON
    ['] LED_OFF 100 RTOS_SET_TIMER_TASK \ выключить светодиод через 100 мс
    ['] BLINK   200 RTOS_SET_TIMER_TASK \ включить через 200 мс и повторить
;

: INIT
    LED BIT  LEDS_DDR SET

    RTOS_INIT
```

```
    ['] BLINK RTOS_SET_TASK \ поставить задачу в очередь на выполнение
    RTOS_RUN \ поехали!
;

: MAIN
    INIT
    BEGIN
        RTOS_TASK_MANAGER \ здесь и будут выполнятся задания
    AGAIN
;

MAIN [->] {MAIN}
```

buttons.spf

Опрос кнопок, подключенных к одному порту, с подавлением дребезга и автоповтором

Подключение:

```
REQUIRE DEBOUNCE    lib/buttons.spf
```

Использование:

1. Перед подключением нужно определить группу BUTTONS:

```
PB0 WIRE BTN_SET              \  кнопки
PB1 WIRE BTN_+
PB2 WIRE BTN_-
GROUP{ BTN_SET BTN_+ BTN_- }GROUP BUTTONS
```

2. Затем определить слова, которые будут выполняться при нажатии каждой кнопки:

```
: BTN0_TASK … ;
: BTN1_TASK … ;
: BTN2_TASK … ;
```

3. Выделить область в памяти для адресов этих задач. Первая задача — для кнопки, подключенной к 0-й линии порта, вторая — к 1-й и т.д.

```
TCREATE BTN_TASKS           \ задачи для кнопок
' BTN0_TASK T,    ' BTN1_TASK T,    ' BTN2_TASK T,
```

4. В инициализирующем коде нужно включить нужные кнопки:

```
BUTTONS {GROUP.MASK} BUTTONS_ENABLED C!                      \ все кнопки вкл
```

5. задать кнопки с автоповтором:

```
[ BUTTONS BTN_SET -GROUP {GROUP.MASK} ] LITERAL \ эта кнопка не повторяет нажатия
  BUTTONS_REPEATABLE C!
```

6. установить задачи, выполняющиеся при нажатии кнопок:

```
BTN_TASKS BUTTONS_TASKS !                         \ задачи для кнопок
```

7. инициализировать порт, к которому подключены кнопки

```
BUTTONS_INIT                                      \ инициализация порта
```

8. и поставить на выполнение задачу-обработчик нажатий:

```
['] BUTTONS_TASK RTOS_SET_TASK                    \ обработчик кнопок
```

См.код примера tst_buttons.spf

lcd.spf

Подключение LCD-дисплея с контроллером HD44780 по 4-битному интерфейсу.

Подключение:

```
REQUIRE LCD_CLRSCR      lib/lcd/lcd.spf          \ библиотеки LCD
REQUIRE LCD_CHAR        lib/lcd/definechar.spf   \ определение пользовательских
символов
:: !BIG_LETTERS ;;
REQUIRE BIG_SETUP       lib/lcd/big.spf          \ большие символы
```

Использование:

Как подключается дисплей физически — указывается в коде библиотеки:

```
PD3 WIRE D3
PD2 WIRE D2
PD1 WIRE D1
PD0 WIRE D0
BUS{ D3 D2 D1 D0 }BUS DATA
PD4 WIRE RS           \ RS WIRE
PD5 WIRE RW           \ RW WIRE
PD6 WIRE E            \ E WIRE
```

После подключения библиотеки дисплей нужно инициализировать словом LCD_INIT, после чего весь вывод через EMIT и TYPE будет идти на дисплей.

Если вы хотите использовать большие символы, подключите lib/lcd/big.spf, если перед этим было определено слово !BIG_LETTERS, то в код кроме цифр включаются еще и большие английские литеры.

Необходимые слова вынесены в таблицу.

LCD_INIT	(dispAttr --)	Инициализация дисплея dispAttr може быть таким: LCD_DISP_OFF — дисплей выключен LCD_DISP_ON — дисплей включен, курсор выключен LCD_DISP_ON_CURSOR — дисплей включен, курсор включен LCD_DISP_CURSOR_BLINK — дисплей включен, курсор включен и мигает
LCD_COMMAND	(c --)	Послать команду
LCD_DATA	(c --)	Послать данные (вывести символ с кодом c)
LCD_GOTOXY	(x y --)	Установить позицию курсора
LCD_GETXY	(-- addr)	Получить текущий адрес

LCD_CHAR name	(b0 .. b7 --)
	исполнение:
	(c --)
		Определить символ с кодом c
		Пример использования:
		`0x00 0x10 0x08 0x04 0x02 0x01 0x00 0x00  LCD_CHAR BACKSLASH`
		после LCD_INIT
		`0x1 BACKSLASH \ определить символ с кодом 1`
		`0x1 EMIT      \ вывод символа`

BIG_SETUP	(--)	Инициализировать символы для больших цифр/букв
BIG_LETTERS	(--)	Переключиться на вывод больших символов
NORMAL_LETTERS	(--)	Переключиться на вывод обычных символов

Также см. код примеров tst_lcd.spf и big_test.spf

wdt.spf, не закончена

TODO:

полное описание

usart.spf

Простой вывод данных в USART без использования прерываний.

Подключение:

```
S" lib/usart.spf" INCLUDED
```

Использование:

USART_INIT	(ubrr --)	Инициализировать передатчик, установить EMIT на вывод в USART. Скорость задается при помощи слова BAUD
BAUD	(speed -- ubrr)	Преобразовать скорость в константу, только для компиляции. См. пример

USART_SEND	(c --)	Передает символ
USART1_INIT	(ubrr --)	Инициализирует второй USART (если есть), установить EMIT на вывод в USART
USART1_SEND	(c --)	Передает символ во второй USART (если есть)

Перед подключением обязательно задайте частоту контроллера F_CPU. По умолчанию при подключении библиотеки вектор EMIT настраивается на USART_SEND.

Пример:

```
1000000 ( Hz ) == F_CPU
S" lib/usart.spf" INCLUDED
: MAIN
  1200 BAUD USART_INIT
  ." Test" CR
;
```

5. Описание каталогов и файлов

devices/devices.spf	Файл с константами типов микроконтроллеров
devices/*.spf	Файлы с определениями микроконтроллеров
kernel/*.spf	Ядро
lib/*.spf	Библиотеки
~mak/*	Служебные
avrasm.f	Ассемблер
avrdis.f	Дизассемблер
avrforth.spf	Собственно форт
avrlist.f	Генерация листинга
avrmac.f	Оптимизатор
compile, compile.bat	Скрипты для компиляции
flash.spf	Целевая компиляция
init.spf	Инициализация контроллера
inline.spf	Управляющие структуры и макросы
isr.spf	Прерывания
manual/Manual.odt, Manual.pdf, Manual.doc	Это руководство
require.cfg	Файл конфигурации для подключения нужных слов
require.spf	Полуавтоматическое подключение нужных слов
xfbase.f	Определяющие слова
util.spf	Разные слова
examples/clock	Пример: Часы с динамической индикацией
examples/tst	Пример: Тестовая программа
examples/tst_min	Пример: Скелет программы
examples/tester	Пример: Программа тестирования правильности выполнения слов
examples/buttons	Пример: Тест обработчика кнопок
examples/lcd	Пример: Тест подключения LCD и DS1307
examples/lcd_big	Пример: Вывод больших букв на LCD
examples/spi	Пример: Управление регистрами 74HC595 по интерфейсу SPI
examples/usart	Пример: Тестирование вывода на терминал

6. Файлы определений микроконтроллеров

См. любой из файлов с комментариями