Инструментальная система GM3P
Руководство пользователя

ОГЛАВЛЕНИЕ

Назначение

Инструментальная система (G)M3P предназначена для решения следующего ряда задач:

1. Отладки, тестирование и внутрисистемного программирования встроенных систем;
2. Интеграции инструментальных средств в единую систему;
3. Связывания разнородных инструментальных средств посредством языка сценариев.

В качестве языка сценариев в M3P используется FORTH. В качестве базового стандарта языка использован стандарт FORTH83.

Язык FORTH

Ниже приведен ряд глав из [1].

Работа в диалоговом режиме

Программирование на языке Форт является существенно диалоговым. Работая за терминалом, программист вводит слова-команды, а загруженная в память ЭВМ форт-система, т.е. реализация языка Форт на данной ЭВМ, немедленно выполняет обозначаемые этими словами действия. О своей готовности к обработке очередной строки текста форт-система обычно сообщает программисту специальным приглашением (например, знаком < , который печатается на терминале). Получив такое приглашение, программист набирает на терминале очередную порцию форт-текста, заканчивая ее специальным управляющим символом (например, нажимая клавишу "Ввод" или "Перевод строки"). Получив сигнал о завершении ввода, форт-система начинает обработку введенного текста (он размещается в буфере для ввода с терминала), выделяя в нем слова-команды и исполняя их. Успешно обработав весь введенный текст, форт-система вновь приглашает программиста к вводу, и описанный цикл диалога повторяется. 8Многие форт-системы после успешного завершения обработки выводят на терминал подтверждающее сообщение (обычно OK - сокращение от английского o'kay - все в порядке). Если во время обработки введенного текста выявляется какая-либо ошибка (например, встретилось неизвестное форт-системе слово), то на терминал выводится поясняющее сообщение, обработка введенного текста прекращается и форт-система приглашает программиста к вводу нового текста.

Для завершения работы обычно предусматриваются специальные слова-команды.

Непосредственно вводить с терминала большие форт-тексты неудобно, поэтому их хранят во внешней памяти на дисках, а с терминала программист вводит только слова-команды, отсылающие форт-систему к обработке тех или иных блоков из внешней памяти Для создания и исправления форт-текстов во внешней памяти используется текстовый редактор, который является специализированным расширением форт-системы.

Стек данных и вычисления

Как уже говорилось, в основе вычислительной модели для языка Форт лежит стековая машина. Ее команды (слова в языке Форт) обычно используют в качестве своих операндов верхние элементы стека, убирая их со стека и возвращая результаты (если они есть) на место операндов Как правило, слова используют одно-два верхних значения на стеке. Для их описания будем применять следующую диаграмму:

```
имя     вершина стека до  --->  вершина стека после
слова   исполнения слова        исполнения слова
```

При этом считаем, что самое верхнее значение в стеке (последнее добавленное) находится справа.

Для работы с собственно вершиной стека имеются следующие слова:

```
DUP     A -> A,A
DROP    A ->
OVER    A,B --> A,B,A
ROT     A,B,C --> B,C,A
SWAP    A,B --> B,A
```

Слово DUP (от DUPLICATE - дублировать) дублирует вершину стека, добавляя

в стек еще одно значение, равное тому, которое было до этого верхним. Слово DROP (сбросить) убирает верхнее значение. Слово OVER (через) дублирует значение, лежащее на стеке непосредственно под верхним. Слово ROT (от ROTATE - вращать) циклически переставляет по часовой стрелке три верхних значения в стеке. Наконец, слово SWAP (обменять) меняет местами два верхних значения.

Можно работать с любым элементом стека с помощью слов

```
PICK  An,An-1,...Ao,n  --->  An,An-1,...Ao,An
ROLL  An,An-1,...Ao,n  --->  An-1,...Ao,An
```

Слово PICK (взять) дублирует n-й элемент стека (считая от нуля), так что 0 PICK равносильно DUP, а 1 PICK равносильно OVER. Слово ROLL (повернуть) циклически переставляет n верхних элементов стека (тоже считая от нуля) по часовой стрелке, так что 2 ROLL равносильно ROT, 1 ROLL равносильно SWAP, а 0 ROLL является пустой операцией.

Чтобы "увидеть" верхнее значение на стеке, используется слово . (точка) A -->, которое снимает значение с вершины стека и печатает его на терминале как целое число в свободном формате (т.е. без ведущих нулей и со знаком минус, если число отрицательно). Вслед за последней цифрой числа слово-точка выводит один пробел, чтобы выводимые подряд числа не сливались в сплошной ряд цифр. Если программист хочет, чтобы напечатанное значение осталось на стеке, он должен исполнить текст DUP . . Слово DUP создаст копию верхнего значения, а точка ее распечатает и уберет со стека.

Перечисленные выше слова работают со значениями, уже находящимися в стеке. А как занести значение в стек? Язык Форт имеет следующее замечательное правило умолчания: если введенное слово форт-системе не известно, то прежде чем сообщать программисту об ошибке, форт-система пытается понять это слово как запись числа. Если слово состоит из одних цифр с возможным начальным знаком минус, то ошибки нет: слово считается известным и его действие состоит в том, что данное число кладется на вершину стека.

Теперь у нас достаточно средств, чтобы привести примеры диалога. Рассмотрим следующий протокол работы:

```
>   5  6  7
 OK
> SWAP . . .
6 7 3 OK
```

В ответ на приглашение к вводу (знак > , печатаемый системой) программист вводит три числа: 5, 6 и 7. Обрабатывая введенный текст, форт-система кладет эти числа в указанном порядке на стек и по завершении обработки выводит подтверждающее сообщение OK и вновь приглашает программиста к вводу. Далее программист вводит текст из четырех слов: SWAP и три точки. Исполняя эти слова-команды, форт-система меняет местами два верхних элемента стека (5, 6, 7 -> 5, 7, 6) и затем поочередно три раза снимает верхнее значение со стека и печатает его. В результате на терминале появляется текст 6 7 5 и сообщение OK, указывающее на завершение обработки, после чего система вновь выдает программисту приглашение на ввод.

Для внешнего представления чисел используется система счисления, задаваемая программистом. Стандарт языка предусматривает следующие слова для переключения в наиболее общеупотребительные системы:

```
DECIMAL    --->   десятичная
HEX        --->   шестнадцатеричная
OCTAL      --->   восьмиричная
```

Первоначально устанавливается десятичная система. Если в процессе работы

будет исполнено, например, слово HEX (от HEXADECIMAL - шестнадцатиричная), то при дальнейшем вводе и выводе чисел будет использоваться шестнадцатиричная система с цифрами от 0 до 9 и от A до F до тех пор, пока основание системы счисления не будет вновь изменено. Внутренним же представлением чисел является обычный двоичный дополнительный код, применяемый в большинстве существующих ЭВМ.

Слова-команды, выполняющие арифметические операции над числами, являются общепринятыми математическими обозначениями:

```
+       A,B  --->  сумма A+B
-       A,B  --->  разность A-B
*       A,B  --->  произведение A*B
/       A,B  --->  частное от A/B
MOD     A,B  --->  остаток от A/B
/MOD    A,B  --->  остаток от A/B, частное от A/B
ABS       A  --->  абсолютная величина A
NEGATE    A  --->  значение с обратным знаком -A
1+        A  --->  A+1
1-        A  --->  A-1
2+        A  --->  A+2
2-        A  --->  A-2
2/        A  --->  частное от A/2
```

При сложении, вычитании и умножении не учитывается возможность переполнения, в случае его возникновения используются младшие 16 разрядов результата. Такая арифметика называется арифметикой по модулю 65536 (2 в степени 16); ее основное достоинство состоит в том, что она дает одинаковые в двоичном представлении результаты независимо от того, как понимаются операнды: как числа со знаком в диапазоне от -32768 до +32767 или как числа без знака в диапазоне от 0 до 65535.

Операции деления / , MOD и /MOD рассматривают свои операнды как числа со знаком. Из нескольких известных математических определений деления с остатком (которые по-разному трактуют случаи, когда операнды имеют разные знаки или оба отрицательны) язык Форт использует так называемое деление с нижней границей: остаток имеет знак делителя или равен нулю, а частное округляется до его арифметической нижней границы ("пола") [11]. Во многих ЭВМ, имеющих аппаратную реализацию деления, применяются другие правила для определения частного и остатка, однако это обычно не вызывает трудностей при программировании, поскольку самый важный случай с неотрицательными операндами все определения трактуют одинаково

При выполнении деления возможно возникновение ошибочной ситуации, если делитель - нуль или частное не умещается в 16 разрядов (переполнение, возникающее при делении -32768 на -1).

Одноместные операции ABS и NEGATE игнорируют переполнение, возникающее в том единственном случае, когда операндом является число -32768, возвращая в качестве результата 0.

Наконец, одноместные операции 1+, 1-, 2+, 2- выполняют действие, отраженное в их мнемонике: увеличение или уменьшение значения на вершине стека на 1 или 2; аналогично слово 2/ возвращает частное от деления своего параметра на 2. Эти слова включены в стандарт ввиду частного использования соответствующих действий.

Использование стека для хранения промежуточных значений естественным образом приводит к так называемой "обратной польской форме" - одному из способов бесскобочной записи арифметических выражений, подразумевающему постановку знака операции после операндов. Например, выражение (A/B+C)*(D*E-F*(G-H)) записывается следующим образом: A B / C + D E * F G H - * - * . Очевидно, что этот текст выполним для Форта, если A, B и т.д.- слова, которые кладут на стек по одному числу. Таким образом, форт-систему

можно использовать как калькулятор. Чтобы вычислить, например, значение (25+18+32)*5, достаточно ввести такой текст: 25 18 + 32 + 5 * . . В ответ система напечатает (исполняя точку) ответ -375.

Чтобы повысить точность вычислений в последовательности умножение-деление, стандарт предусматривает два необычных слова:

```
*/     A,B,C ---> частное от (A*B/C)
*/MOD  A,B,C ---> остаток, частное от (A*B/C)
```

Особенность этих слов состоит в том, что промежуточный результат A*B вычисляется с двойной точностью, и в качестве делимого для C используются все его 32 разряда. Окончательные же результаты являются 16-разрядными числами.

Наряду с описанной выше 16-разрядной арифметикой, язык Форт имеет полный набор средств для работы с 32-разрядными целыми числами через стандартное расширение двойной точности. Внутренним представлением таких чисел является 32-разрядный двоичный дополнительный код, представляющий их как числа со знаком в диапазоне от -2147483648 до +2147483647 или как числа без знака в диапазоне от 0 до 4294967295. При размещении в стеке числа двойной точности занимает два элемента: верхний - старшая половина, предыдущий - младшая. Такое расположение делает простым переход от двойной точности к обычной через слово DROP. Расширение двойных чисел включает следующие слова:

```
2DROP    AA --->
2DUP     AA ---> AA,AA
2OVER    AA,BB ---> AA,BB,AA
2ROT     AA,BB,CC --> BB,CC,AA
2SWAP    AA,BB ---> BB,AA
D.       AA --->
D+       AA,BB ---> сумма AA+BB
D-       AA,BB ---> разность AA-BB
DABS     AA ---> абсолютная величина AA
DNEGATE  AA ---> число с обратный знаком -AA
```

Здесь обозначение типа AA подчеркивает, что значение занимает два элемента стека. Хотя стандартное расширение этого не предусматривает, во многих реализациях языка Форт этот ряд продолжают операции умножения и деления:

```
D*      AA,BB ---> произведенме AA*BB
D/      AA,BB ---> частное от AA/BB
DMOD    AA,BB ---> остаток от AA/BB
```

Операндами и результатами перечисленных слов являются значения двойной длины, занимающие по два элемента стека каждый; это обстоятельство отражено в мнемонике, начинающейся с цифры 2, когда два элемента стека выступают как одно целое, или с буквы D (от слова DOUBLE-двойной), когда речь идет арифметическом значении двойной длины.

Следующие два слова являются переходными между арифметическими операциями одинарной и двойной точности; их мнемоника включает букву M слова MIX - смесь) и U (от слова UNSIGNED - беззнаковый):

```
UM*      A,B ---> CC
UM/MOD   AA,B ---> C,D
```

Слово UM* перемножает операнды A и B как 16-разрядные числа без знака, возвращая все 32 разряда получившегося произведения. Слово UM/MOD рассматривает 32-разрядное делимое AA и 16-разрядный делитель B как числа без знака и возвращает получающиеся 16-разрядные остаток C и частное D. Если делитель нуль или частное превышает 65535, то это рассматривается как ошибка. Для перехода к двойной точности с учетом знака многие реализации

имеют слово S>D A --> AA, которое расширяет исходное число A до числа двойной точности распространением знакового разряда.

Чтобы при вводе различать числа двойной и одинарной точности, в сформулированное выше правило умолчания внесена оговорка: если не найденное в словаре слово состоит только из цифр с возможным начальным знаком минус, то это число одинарной точности, а если в слово входят точки (в любом месте), то это число двойной длины. Пример использования форт-системы как калькулятора для работы с числами двойной длины представляет следующий протокол работы:

```
> 1234567.  7654321.  D+  D.
8888888 OK
```

В ответ на приглашение (знак >) программист вводит два числа двойной длины, операцию сложения этих чисел и операцию печати результата. Выполняя указанные действия, форт-система печатает ответ (заметьте, что он уже не содержит точки) и подтверждающее сообщение OK.

Введение новых слов

Замечательное свойство языка Форт - это возможность вводить в него новые слова, расширяя тем самым набор его команд в нужном программисту направлении. Для введения новых слов чаще всего используется определение через двоеточие - определение нового слова через уже известные. Такое определение начинается словом : (двоеточие) и заканчивается словом ; (точка с запятой). Сразу после двоеточия идет определяемое слово, а за ним - последовательность слов, через которые оно определяется. Например, текст : S2 DUP * SWAP DUP * + ; о пределяет слово S2, вычисляющее сумму квадратов двух чисел, снимаемых с вершины стека S2 A,B --> A**2+B**2. После ввода данного описания слово S2 можно исполнять и включать в описания других слов. При создании таких определений рекомендуется тщательно комментировать все изменения стека. Слово ((открывающая круглая скобка) отмечает начало комментария; все следующие литеры до первой) (закрывающей скобки) считаются комментарием и при обработке вводимого форт-текста пропускаются.

Перепишем приведенное выше определение слова S2, показывая состояние вершины стека после исполнения каждой строки:

```
: S2 ( A,B ---> A**2+B**2 сумма квадратов)
  DUP       ( A,B,B)
  * SWAP    ( B**2,B)
  DUP *     ( B**2,A**2)
  +         ( A**2+B**2)
```

По-видимому, минимальным требованием к документированности определения следует считать задание начального и конечного состояний вершины стека при работе слова.

Рассмотрим подробнее работу форт-системы во время определения новых слов. Мы уже знаем, что получив от программиста очередную порцию входного текста, форт-система выделяет в ней отдельные слова и ищет их в своем словаре. Эту работу выполняет текстовый интерпретатор форт-системы. Если слово в словаре не найдено, то текстовый интерпретатор пытается понять его как число, используя описанное выше правило умолчания. Если слово найдено или оказалось записью числа, то дальнейшие действия интерпретатора зависят от его. текущего состояния. В каждый момент времени текстовый интерпретатор находится в одном из двух состояний: в состоянии исполнения или в состоянии компиляции. В состоянии, исполнения найденное слово исполняется (т.е. выполняется действие, составляющее его семантику), а число кладется на стек. Если же интерпретатор находится в состоянии компиляции, то найденное слово не исполняется, а компилируется, т.е. включается в создаваемую последовательность действий для определяемого в данный момент

слова. Найденное и скомпилированное таким образом слово будет исполнено наряду с другими такими словами во время исполнения определенного через них слова. Если требуется скомпилировать число, то текстовый интерпретатор компилирует особый литеральный код, который во время исполнения положит значение данного числа на стек.

Проследим за работой текстового интерпретатора по обработке уже рассмотренного определения слова : S2 DUP * SWAP DUP * + ; . Предположим, что перед началом обработки введенной строки интерпретатор находится в состоянии исполнения. Первым словом является : (двоеточие), которое исполняется. Его семантика состоит в том, что из входной строки выбирается очередное слово и запоминается в качестве определяемого, а интерпретатор переключается в состояние компиляции. Следующие слова, которые интерпретатор будет извлекать из входной строки (DUP, *, SWAP и т.д.), будут компилироваться, а не исполняться, так как интерпретатор находится в состоянии компиляции. В результате с определяемым словом S2 связывается последовательность действий, отвечающая этим словам. Процесс выделения и компиляции слов будет продолжаться до тех пор, пока не встретится ; (точка с запятой). Это слово особенное, оно имеет так называемый "признак немедленного исполнения". Слова с таким признаком исполняются независимо от текущего состояния текстового интерпретатора, поэтому точка с запятой будет вторым исполненным словом после двоеточия. Семантика точки с запятой заключается в том, что построение определения, начатого двоеточием, завершается и интерпретатор вновь переключается в состояние исполнения Поэтому после ввода определения слова S2 мы тут же можем проверить, как оно работает на конкретных значениях:

```
> 5  4  S2  .
41 OK
```

Слова с признаком немедленного исполнения (другим примером такого слова помимо точки с запятой является открывающая круглая скобка - начало комментария) выполняются по-разному в зависимости от состояния текстового интерпретатора. Некоторые из них даже используются только в одном из этих состояний. Поэтому на диаграмме для таких слов будем отмечать эти случаи, специально указывая состояние компиляции

```
(   --->
    --->   (компиляция)
```

Слово (в обоих состояниях действует одинаково: пропускает все следующие вводимые литеры до закрывающей круглой скобки включительно или до конца введенной строки, если закрывающая скобка отсутствует.

Слово ; допустимо применять только в состоянии компиляции: ; --> (компиляция). Оно завершает построение нового определения и переключает текстовый интерпретатор в состояние исполнения.

Слово IMMEDIATE (немедленный) --> устанавливает признак немедленного исполнения для последнего определенного слова. (Подробнее использование этого признака рассматривается в п.1.7.)

Введенные слова можно исключить из словаря с помощью слова FORGET (забыть) -->, которое выбирает из входной строки следующее слово и исключает его из словаря вместе со всеми словами, определенными позже.

Разберем следующий протокол диалога:

```
> 2  2  *   .
4 OK
> :  2  3  ;
 OK
> 2  2  *  .
9 OK
```

```
> FORGET  2
 OK
> 2  2  *  .
4 OK
```

Сначала программист вычисляет произведение от умножения 2 на 2 и получает ответ 4. Введя затем определение слова 2 как числа 3, он в дальнейшем получает уже другой ответ. Исключив это определение слова 2 через FORGET, он возвращается к прежней семантике слова 2.

В процессе работы текстового интерпретатора программист может переключать его из состояния компиляции в состояние исполнения и обратно с помощью слов [(открывающая квадратная скобка) --> и] (закрывающая квадратная скобка) -->. Слово [имеет признак немедленного исполнения и переключает интерпретатор в состояние исполнения, а слово] переключает его в состояние компиляции. Обычно эти слова используются внутри определения через двоеточие, чтобы вызвать исполнение слова или группы слов, не имеющих признака немедленного исполнения. Например, если в тексте определения понадобилась константа FF00 (в шестнадцатиричной системе), а текущей используемой системой является десятичная, то было бы неправильно включить в текст определения фрагмент HEX FF00 DECIMAL, поскольку слово HEX будет не выполнено, а скомпилировано и число не будет воспринято правильно. Вместо этого следует писать: [HEX] FF00 [DECIMAL] . В языке есть и другие способы, чтобы выразить это же более точно и красиво.

Еще один пример дает использование слова LITERAL (литерал), имеющего признак немедленного исполнения, внутри определения через двоеточие. Оно используется в виде: [<значение>] LITERAL, где <значение> - слова, вычисляющие на вершине стека значение, которое словом LITERAL будет скомпилировано в код как число. Во время исполнения определения это значение будет положено на стек. Таким образом, текст [2 2 *] LITERAL внутри определения через двоеточие эквивалентен употреблению слова-числа 4. Это дает большие возможности для использования констант, "вычисляемых" во время компиляции определения. Аналогичное соответствие имеет место и вне определения через двоеточие, поскольку в состоянии исполнения слово LITERAL не выполняет никаких действий, и поэтому на стеке остается вычисленное перед этим значение.

Константы и переменные, работа с памятью

Программисту часто бывает удобно работать не с "анонимными" значениями, а с именованными. По аналогии со средствами других языков эти средства языка Форт называются константами и переменными. Впоследствии мы увидим, что они являются не "изначальными", а (наряду с определениями через двоеточие) частными случаями более общего понятия "определяющие слова".

Слово CONSTANT (константа) A --> работает следующим образом. Со стека снимается верхнее значение, а из входного текста выбирается очередное слово и запоминается в словаре как новая команда. Ее действие состоит в следующем: поместить на стек значение A, снятое со стека в момент ее определения. Например, 4 CONSTANT XOP . В дальнейшем при исполнении слова XOP число 4 будет положено на стек.

Слово VARIABLE (переменная) A --> резервирует в словаре два байта, а из входного потока выбирает очередное слово и вносит его в словарь как новую команду, которая кладет на стек адрес зарезервированной двухбайтной области. Можно сказать, что переменная работает, как константа, значением которой является адрес зарезервированной двухбайтной области.

Работа с переменной помимо получения ее адреса состоит в получении ее текущего значения и присваивании нового. Для этого язык Форт имеет следующие слова:

```
@    A ---> B
!  B,A --->
```

Слово @ (читается "разыменовать") снимает со стека значение и, рассматривая его как адрес области оперативной памяти, кладет на стек двухбайтное значение, находящееся по этому адресу. Обратное действие выполняет слово ! (восклицательный знак, читается "присвоить"), которое снимает со стека два значения и, рассматривая верхнее как адрес области оперативной памяти, засылает по нему второе снятое значение. Эти слова можно использовать не только для переменных, но и для любых адресов оперативной памяти. Следующий протокол работы показывает порядок использования переменной в сочетании с этими словами:

```
> VARIABLE  X  1  X  !
 OK
> X  @  .
1  OK
> X @ NEGATE  X  !  X  .
-1  OK
```

В первой строке определяется переменная X, и ей присваивается начальное значение 1. Затем текущее значение переменной X распечатывается. После этого текущее значение меняется на противоположное по знаку и вновь распечатывается.

Полезным вариантом слова ! является слово +! (плюс-присвоить) N,A --> , которое увеличивает на N значение, находящееся в памяти по адресу A. Несмотря на то, что это слово легко выразить через + и !:

```
: +!  ( N,A ---> )  DUP @ ROT + SWAP ! ;
```

оно включено в обязательный набор слов.

Слова, определенные через CONSTANT и VARIABLE,- такие же равноправные слова форт-системы, как и определенные через двоеточие. Их также можно использовать в определениях других слов и исключать из словаря словом FORGET.

Для работы со значениями двойной длины имеются слова

```
2CONSTANT    AA --->
2VARIABLE       --->
2@            A ---> BB
2!         BB,A --->
```

из стандартного расширения двойных чисел. При размещении в памяти такие значения рассматриваются как пары смежных двухбайтных значений одинарной длины: сперва (в меньших адресах) располагается старшая половина, затем (в больших адресах) младшая. Адресом значения двойной длины считается адрес его старшей половины.

Константы и переменные позволяют программисту использовать память в словаре вполне определенным образом. А как быть, если требуется что-то иное? Общий принцип языка Форт состоит в том, чтобы не закрывать от программиста даже самые элементарные единицы, из которых строятся его более сложные слова, а предоставлять их наравне с другими словами. Элементарными словами для работы с памятью в словаре, помимо приведенных выше @ и ! , являются следующие:

```
HERE      ---> A
ALLOT   A --->
,       A --->
```

Предполагается, что словарь занимает некоторую начальную часть адресного

форт-пространства и у него имеется указатель текущей вершины (словарь растет в сторону увеличения адресов). Слово HERE (здесь) возвращает текущее значение указателя (адрес первого свободного байта, следующего за последним занятым байтом словаря).

Слово ALLOT (распределить) резервирует в словаре область памяти, размер которой (в байтах) снимает со стека. Резервирование состоит в том, что текущее значение указателя изменяется на заданную величину, поэтому при положительном значении запроса память отводится вплотную от вершины словаря, а при отрицательном значении запроса происходит освобождение соответствующего участка памяти. Наконец, слово , (запятая) выполняет так называемую "компиляцию" значения, снимаемого со стека: значение переносится на вершину словаря, после чего указатель вершины продвигается на 2 (размер в байтах скомпилированного значения). Некоторые из приведенных слов легко выражаются через другие:

```
: , ( A --> )  HERE ! 2 ALLOT ;
: 2@ ( A --> BB )  DUP 2 + @ SWAP @ ;
```

Такая избыточность позволяет программисту быстрее находить нужные ему решения и делает программы более удобочитаемыми.

А как создать в словаре поименованную область памяти? Можно завести область и связать ее адрес с именем через описание константы: HERE 10 ALLOT CONSTANT X10. Слово HERE оставляет на стеке адрес текущей вершины словаря, затем при исполнении текста 10 ALLOT от этой вершины резервируется 10 байт, после чего слово CONSTANT связывает адрес зарезервированной области с именем X10. В дальнейшем при исполнении слова X10 этот адрес будет положен на стек.

Другой возможный путь состоит в использовании слова CREATE (создать) --> в таком контексте: CREATE X10 10 ALLOT . Слово CREATE, подобно слову VARIABLE, выбирает из входной строки очередное слово и определяет его как новую команду с таким действием: положить на стек адрес вершины словаря на момент создания этого слова. Поскольку следующие действия в приведенном примере резервируют память, то слово X10 будет класть на стек адрес зарезервированной области. Очевидно, что слово VARIABLE можно выразить через CREATE

```
: VARIABLE  ( ---> )  CREATE 2 ALLOT ;
```

или иначе (если мы хотим инициализировать нулем значение создаваемой переменной):

```
: VARIABLE  ( ---> )  CREATE 0 , ;
```

(Другой аспект использования слова CREATE рассматривается в п. 1.10).

В стандарте определен ряд системных переменных, к которым программист может свободно обращаться, в том числе STATE (состояние) -> A и BASE (основание) --> A. Исполнение каждого из этих слов заключается в том, что на стеке оставляется адрес ячейки, в которой хранится значение данной переменной.

Переменная STATE представляет текущее состояние текстового интерпретатора: нуль для исполнения и не нуль (обычно -1) для компиляции. Поэтому вся реализация слов [и] , переключающих интерпретатор из одного состояния в другое, сводится к одному присваиванию:

```
: [ ( ---> )  0 STATE ! ; IMMEDIATE
: ] ( ---> )  -1 STATE ! ;
```

Переменная BASE хранит текущее основание системы счисления для ввода - вывода чисел, поэтому реализация слов для установки стандартных систем

выглядит так:

```
: DECIMAL  ( ---> )  10 BASE ! ;
: HEX      ( ---> )  16 BASE ! ;
```

Отсюда следует более изящный способ кратковременной смены системы счисления во время компиляции определения: [BASE @ HEX [FF00 [BASE !] . Сначала на стеке запоминается текущее основание, и система счисления переключается на основание 16, в котором и воспринимается следующее число FF00, после чего восстанавливается прежнее основание. А как узнать текущее основание системы счисления? Исполнение текста BASE @ не поможет, поскольку ответом всегда будет 10 (почему?). Правильный ответ даст исполнение текста BASE @ DECIMAL . , в результате чего значение основания будет напечатано как число в десятичной системе. Еще более правильным было бы использовать текст BASE @ DUP DECIMAL . BASE ! , который после печати основания в десятичной системе восстанавливает его прежнее значение

Логические операции

В языке Форт имеется только один тип значений - 16-разрядные двоичные числа, которые, как мы видели, рассматриваются в зависимости от ситуации как целые числа со знаком или как адреса и т. д. Точно так же подходят и к проблеме представления логических значений ИСТИНА и ЛОЖЬ: число 0, в двоичном представлении которого все разряды нули, представляет значение ЛОЖЬ, а любое другое 16-разрядное значение понимается как ИСТИНА. Вместе с тем стандартные слова, которые должны возвращать в качетве результата логическое значение, из всех возможных представлений значения ИСТИНА используют только одно: число -1 (или, что то же самое, 65535), в двоичном представлении которого все разряды единицы. Такое соглашение связано с тем, что традиционные логические операции конъюнкции, дизъюнкции и отрицания выполняются в Форте поразрядно над всеми шестнадцатью разрядами операндов:

```
AND   A,B ---> A  B   логическое И
OR    A,B ---> A  B   логическое ИЛИ
XOR   A,B ---> A  B   исключающее ИЛИ
NOT     A --->  ^ A   логическое НЕ
```

Как и в предыдущих случаях, эти операции не являются независимыми: операция отрицания (поразрядное инвертирование) легко выражается через исключающее ИЛИ (поразрядное сложение по модулю два):

```
: NOT ( A --> ^A )  -1 XOR ;
```

Нетрудно увидеть, что для принятого в Форте стандартного представления значений ИСТИНА и ЛОЖЬ все эти слова работают, как обычные логические операции.

Логические значения возникают в операциях сравнения, которые входят в обязательный набор слов и имеют общепринятую программистскую мнемонику:

```
<  A,B ---> A < B   меньше
=  A,B ---> A > B   равно
>  A,B ---> A > B   больше
```

Эти операции снимают со стека два врехних значения, сравнивают их как числа со знаком (операция "равно" выполняет поразрядное сравнение) и возвращают результат сравнения как значение ИСТИНА и ЛОЖЬ в описанном выше стандартном представлении. Из-за стремления к минимизации обязательного набора операций в него не включены слова для операций смешанного сравнения, поскольку их легко выразить через уже имеющиеся:

```
: <=  ( A,B ---> A <= B )  SWAP < NOT ;
: >=  ( A,B ---> A >= B >  SWAP > NOT ;
```

```
: <>  ( A,B ---> A <> B )  = NOT ;
```

Для сравнения 16-разрядных чисел без знака имеется слово U< A,B --> A < B. Эта операция обычно используется для сравнения адресов, которые лежат в диапазоне от 0 до 65535. Буква и (от UNSIGNED - беззнаковый) в ее мнемонике говорит о том, что операнды рассматриваются как числа без знака.

Ввиду частого использования и возможности непосредственной реализации на многих существующих ЭВМ в обязательный набор слов включены одноместные операции сравнения с нулем:

```
0<   A ---> A < 0
0=   A ---> A = 0
0>   A ---> A > 0
```

При этом слово 0= можно использовать вместо NOT как операцию логического отрицания, и в отличие от NOT оно будет правильно работать при любых представлениях логического значения ИСТИНА.

Описанные выше двухместные операции сравнения естественным образом выражаются через сравнения с нулем:

```
: <  ( A,B ---> A < B )  - 0< ;
: =  ( A,B ---> A = B )  - 0= ;
: >  ( A,B ---> A > B )  - 0> ;
```

Стандартное расширение двойных чисел имеет аналогичные слова для сравнения 32-разрядных значений:

```
D0=  AA ---> AA = 0
D<   AA,BB ---> AA < BB
D=   AA,BB ---> AA = BB
DU<  AA,BB ---> AA < BB
```

Слова D< и DU< различаются тем, что первое рассматривает свои операнды как числа со знаком, а второе - как числа без знака. Для слов D0= и D= такое различие несущественно, их, например, можно определить так:

```
: D0=  ( AA ---> AA = 0 )   OR 0= ;
: D=   ( AA,BB ---> AA = BB )  D- D0= ;
```

Слово OR (логическое ИЛИ) в определении слова D0= логически складывает старшую и младшие половины исходного 32-разрядного значения. Нулевой результат будет получен тогда и только тогда, когда исходное значение было нулевым. Следующее слово 0= преобразует этот результат к логическому значению в стандартном представлении. Исполнение слова D= состоит в вычислении разности его операндов в сравнении этой разности с нулем.

Структуры управления

Во всех приводившихся выше определениях слов тело определения записывалось как последовательность уже известных слов-команд; семантика определяемого таким образом слова состоит в последовательном выполнении слов-команд тела. Помимо последовательного исполнения традиционными приемами в программировании стали ветвление (выбор между разными последовательностями действий) и цикл (многократное повторение одной последовательности действий).

В языке Форт тоже имеются условные операторы и циклы, реализованные с помощью специальных слов-команд, которые легко выражаются через другие, более элементарные слова (см. гл. 2). Это открывает путь к реализации таких вариантов структур управления, которые больше всего подходят для данной задачи, что дает программисту широкие возможности для создания новых

структур управления.

Стандарт языка предусматривает ряд слов для построения условных операторов и циклов в некотором конкретном виде. Эти слова используются внутри определений через двоеточие и разделяют тело определения на отрезки. Действия, соответствующие словам из этих отрезков, выполняются, не выполняются или выполняются многократно в зависимости от условий, проверяемых во время выполнения данного определения. Условные операторы и циклы могут свободно вкладываться друг в друга.

Условный оператор строится с помощью слов:

```
IF     A --->    (исполнение)
ELSE     --->    (исполнение)
THEN     --->    (исполнение)
```

Внутри определения через двоеточие отрезок текста IF <часть-то> ELSE <часть-иначе> THEN задает следующую последовательность действий. Слово IF (если) снимает значение с вершины стека и рассматривает его как логическое. Если это ИСТИНА (любое нулевое значение), то выполняется часть "то" - слова, находящиеся между IF и THEN, а если ЛОЖЬ (равно нулю), то исполняется часть "иначе" - слова между ELSE и THEN . Сами слова ELSE (иначе) и THEN (то) играют роль ограничителей для слова IF и самостоятельной семантики не имеют. Часть <иначе> вместе со словом ELSE может отсутствовать, и тогда условный оператор имеет сокращенную форму IF <часть-то> THEN . Если логическое значение, снимаемое со стека словом IF, ИСТИНА, то выполняются слова, составляющие часть "то", а если ЛОЖЬ, то данный оператор не выполняет никаких действий. Обратите внимание, что условие для слова IF вычисляется предшествующими словами.

Для примера рассмотрим определение слова ABS, вычисляющего абсолютное значение числа:

```
: ABS ( A --->абс A) DUP 0< IF NEGATE THEN ;
```

Слово DUP дублирует исходное значение, следующее слово 0< проверяет его знак, заменяя копию на логическое значение - результат проверки. Слово IF снимает со стека этот результат, и если это ИСТИНА, то лежащее на стеке исходное отрицательное значение словом NEGATE заменяется на противоположное.

Еще пример: стандартное слово ?DUP дублирует верхнее значение, если это не ноль, и оставляет стек в исходном состоянии, если на вершине ноль:

```
: ?DUP ( A -> A,A/0 )  DUP  IF DUP THEN ;
```

В спецификации данного слова косая черта разделяет два варианта результата, который это слово может оставить на стеке. Примером использования полного условного оператора может служить определение слова S>D, расширяющего исходное значение до значения двойной длины распространением знакового разряда:

```
: S>D   ( A -> AA )  DUP 0< IF -1 ELSE 0 THEH ;
```

Это слово добавляет в стек -1, если число на вершине стека отрицательно, или 0 в противном случае. Таким образом, выполняется распространение знакового раз- ряда на старшую половину значения двойной точности.

В стандарт языка Форт включены циклы с условием и циклы со счетчиком. Циклы первой группы образуются с помощью слов

```
BEGIN     --->  (исполнение)
UNTIL   A --->  (исполнение)
WHILE   A --->  (исполнение)
REPEAT    --->  (исполнение)
```

и имеют две формы:

```
BEGIN <тело> UNTIL
BEGIN <тело-1> WHILE <тело-2>  REPEAT
```

В обоих случаях цикл начинается словом BEGIN (начать), которое служит открывающей скобкой, отмечающей начало цикла, и во время счета не выполняет никаких действий.

Цикл BEGIN-UNTIL называется циклом с проверкой в конце. После исполнения слов, составляющих его тело, на стеке остается логическое значение - условие завершения цикла. Слово UNTIL (пока не) снимает это значение со стека и анализирует его. Если это ИСТИНА (не нуль), то исполнение цикла завершается, т. е. далее будут исполняться слова, следующие за UNTIL, а если это ЛОЖЬ (нуль), то управление возвращается к началу цикла от слова BEGIN. Например, определение слова, вычисляющего факториал, может выглядеть так:

```
: ФАКТОРИАЛ ( N ---> N!       ВЫЧИСЛЕНИЕ N ФАКТОРИАЛ)
    DUP 2 < IF DROP 1       ( 1    ЕСЛИ N<2, ТО N!=1)
           ELSE             ( N    ИНАЧЕ             )
      DUP                   ( S,K     S=N,K=N        )
    BEGIN                   ( S,K                    )
         1-                 ( S,K'    K'=K-1         )
          SWAP  OWER        ( K',S,K'                )
          *  SWAP           ( S',K'   S'=S*K'        )
         DUP  1  =          ( S',K',K'=1             )
    UNTIL                   ( S',1    S'=N!          )
  DROP  THEN  ;             ( N!                     )
```

Как и ранее, в комментариях, сопровождающих каждую строчку текста, мы указываем значения, которые остаются на вершине стека после ее исполнения. Такое документирование облегчает понимание программы и помогает при ее отладке.

Цикл с проверкой в начале BEGIN-WHILE-REPEAT используется, когда в цикле есть действия, которые не надо выполнять в заключительной итерации. Исполнение слов, составляющих его тело-1, оставляет на стеке логическое значение - условие продолжения цикла. Слово WHILE (пока) снимает это значение со стека и анализирует его. Если это ИСТИНА (не нуль), то исполняются слова, составляющие тело-2 данного цикла до ограничивающего слова REPEAT (повторять), после чего вновь исполняется тело-1 от слова BEGIN. Если же значение условия ЛОЖЬ (нуль), то исполнение данного цикла завершается и начинают выполняться слова, следующие за REPEAT. Заметьте, что в отличие от цикла BEGIN-UNTIL, значение ИСТИНА соответствует продолжению цикла. Для примера рассмотрим программу вычисления наибольшего общего делителя по алгоритму Евклида:

```
: НОД   (  A,B ---> C:НАИБОЛЬШИЙ ОБЩИЙ ДЕЛИТЕЛЬ)
  2DUP  <  IF  SWAP  THEN   ( ТЕПЕРЬ  A>=B     )
     BEGIN  DUP             ( A,B,B            )
     WHILE                  ( ПОКА B НЕ НОЛЬ   )
            2DUP  MOD       ( A,B,C:ОСТАТОК    )
          ROT               ( B,C,A            )
          DROP              ( A',B'  A'=B,B'=C )
     REPEAT DROP ;          ( НОД              )
```

Для организации циклов с целочисленной переменной - счетчиком цикла - используются слова

```
DO     A,B --->     (исполнение)
LOOP       --->     (исполнение)
+LOOP    A --->     (исполнение)
```

```
I          ---> A    (исполнение)
J          ---> A    (исполнение)
LEAVE      --->      (исполнение)
```

Такие циклы записываются в одной из следующих двух форм: DO <тело> LOOP или DO <тело> +LOOP. В обоих случаях цикл начинается словом DO (делать), которое снимает со стека два значения: начальное (на вершине стека) и конечное (второе сверху) - и запоминает их. Текущее значение счетчика полагается равным начальному значению, после чего исполняются слова, составляющие тело цикла. Слово LOOP увеличивает текущее значение счетчика на единицу и проверяет условие завершения цикла. В отличие от него, слово +LOOP прибавляет к текущему значению счетчика значение шага, которое вычисляется на стеке телом цикла и рассматривается как число со знаком. В обоих случаях условием завершения цикла является пересечение границы между A-1 и A при переходе от прежнего значения счетчика к новому (направление перехода определяется знаком шага), где A-конечное значение, снятое со стека словом DO.

Если пересечения не произошло, то тело цикла исполняется вновь с новым значением счетчика в качестве текущего.

Такое определение позволяет рассматривать исходные параметры цикла (начальное и конечное значения) и как числа со знаком, и как числа без знака (адреса). Например, текст 10 0 DO (тело) LOOP предписывает выполнять тело цикла 10 раз со значениями счетчика 0, 1, 2, ..., 9, а в случае 0 10 DO <тело) LOOP тело цикла будет исполнено 65 526 раз со значением счетчика 10, 11, ..., 32767, -32768, -32767, ... -1 или (что то же самое) со значениями счетчика 10, 11, ..., 65535.

В то же время цикл 0 10 DO <тело> -1 + LOOP будет исполняться 11 раз (а не 10) со значениями счетчика 10, 9, ..., 0, поскольку пересечение границы между -1 и 0 произойдет при переходе от значения счетчика 0 к следующему значению - 1. Цикл 10 0 DO <тело> -1 +LOOP будет исполняться 65 527 раз со значениями счетчика 0, -1, -2, ..., -32768, 32767, ..., 10 или (что то же самое) 0, 65535, 65534, ..., 10. Таким образом, цикл со счетчиком всегда выполняется хотя бы один раз. Некоторые реализации предусматривают слово ?DO, которое не исполняет тело цикла ни разу, если начальное и граничное значения оказались одинаковыми.

Внутри тела цикла слово I кладет на стек текущее значение счетчика. Например, следующее определение вычисляет сумму квадратов первых n натуральных чисел:

```
: SS2   ( N ---> S:СУММА КВАДРАТОВ ОТ 1 ДО N)
    0  SWAP          ( 0,N         S[0]=0  )
    1+  1            ( S[0],N+1,1          )
          DO   I     ( S[I-1],I            )
            DUP      ( S[I] S[I]=S[I-1]+I*I)
          LOOP  ;    ( S[N]   )
```

Слово LEAVE (уйти), употребленное внутри цикла, вызывает прекращение исполнения его тела; управление переходит к словам следующим за словом LOOP или +LOOP.

В программировании часто применяется конструкция цикл в цикле. Чтобы во внутреннем цикле получить текущее значение счетчика объемлющего цикла, используется слово J: DO ... I ... DO ... I ... J ... LOOP ... I ... LOOP. Первое вхождение слова I дает текущее значение счетчика внешнего цикла. Следующее вхождение I дает. уже значение счетчика внутреннего цикла. Чтобы получить счетчик внешнего цикла, надо использовать слово J.

По выходе из внутреннего цикла доступ к этому значению вновь дает слово I. Этим слова I и J отличаются от переменных цикла в традиционных языках программирования. Некоторые реализации предусматривают еще и слово K для

доступа к счетчику третьего объемлющего цикла.

1.8. Литеры и строки, форматный вывод чисел

В современном программировании важное место занимает обработка текстовых данных. С каждой литерой, которую можно ввести с внешнего устройства или вывести на него, связывается некоторое число - код этой литеры, так что в памяти ЭВМ литеры представлены своими кодами. Стандарт языка Форт предусматривает использование таблицы кодов ASCII, в которой задействованы все числа в диапазоне от 0 до 127. Каждый код занимает один 8-разрядный байт, в котором для представления литеры используются младшие 7 разрядов.

Такая привязка к одной конкретной кодировке не является существенным препятствием к использованию других, если сохраняется условие, что код литеры занимает один байт. В применяемых в нашей стране форт-системах помимо ASCII применяются коды КОИ-7, КОИ-8, ДКОИ и другие.

Для доступа к однобайтным значениям, расположенным в памяти, используются слова C@ A-->B и C! B,A -->, которые аналогичны словам @ и !. Префикс C (от слова CHARACTER - литера) говорит о том, что эти слова работают с литерными кодами (однобайтными значениями).

Слово C@ возвращает содержимое байта по адресу A, дополняя его до 16-разрядного значения нулевыми разрядами. Обратное действие выполняет слово C!, которое засылает младшие 8 разрядов значения B в память по адресу A.

Для политерного обмена с терминалом стандарт предусматривает такие слова:

```
KEY          ---> A
EMIT       A --->
CR           --->
TYPE     A,N --->
EXPECT   A,N --->
```

Слово KEY (клавиша) возвращает в младших разрядах значения A - код очередной литеры, введенной с терминала. В отличие от слова C@ старшие разряды зависят от реализации и могут быть ненулевыми. Обратное действие выполняет слово EMIT (испустить), которое снимает значение со стека и, рассматривая его младшие разряды как код литеры, выводит эту литеру на терминал. Специальное слово CR (сокращение от CARRIAGE RETURN - возврат каретки) вынолняет перевод строки при выводе на терминал.

Расположенные в памяти побайтно коды литер образуют строку, которую можно напечатать на терминале словом TYPE (напечатать). Это слово снимает со стека число - количество литер (оно должно быть неотрицательно) и адрес начала строки (ее первого байта):

```
: TYPE ( A,N ---> )    ?DUP IF 0 DO
       DUP I + C@ EMIT LOOP THEN DROP ;
```

Если число литер равно нулю, то ничего не печатается.

Обратное действие - ввод строки литер - выполняет слово EXPECT (ожидать), которое снимает со стека длину и адрес области памяти для размещения вводимых литер. Коды литер, последовательно вводимых с терминала, помещаются в указанную область до тех пор, пока не будет введено заданное число литер или не будет введена управляющая литера "возврат каретки" (код этой литеры в память не заносится). Фактическое число введенных литер сообщается в стандартной переменной SPAN (размер), эти литеры к тому же отображаются на терминале.

Ввиду его особой важности для кодирования пробела выделена специальная константа BL (от BLANK-пробел), которую для кода ASCII можно задать так: 32 CONSTANT BL. При исполнении слова BL на стеке остается код пробела. Чтобы

вывести пробел на терминал, имеются следующие стандартные слова:

```
: SPACE ( ---> )   BL EMIT ;
: SPACES ( K ---> )  ?DUP IF 0 DO SPACE LOOP THEN ;
```

Слово SPACE (пробел) выводит на терминал один пробел, а слово SPACES (пробелы) - несколько, снимая их количество со стека (это значение, как и длина строки в слове TYPE, должно быть неотрицательным).

Внутри определений через двоеточие можно использовать явно заданные тексты для вывода их на терминал. При исдолнении слова ." (точка и кавычка), употребленного в контексте ." текст ", следующие за ним литеры до закрывающей кавычки исключительно печатаются на терминале. Пробел, отделяющий слово ., в число печатаемых литер не входит. Другое слово .((точка и скобка) отличается от ." тем, что ограничителем текста является закрывающая правая скобка, и это слово можно использовать как внутри определений, так и вне их.

Помимо строки - поля байт, длина которого задается отдельно - язык Форт использует строки со счетчиком. Строка со счетчиком представляется полем байт, причем в первом байте записана длина строки. Стандарт не определяет форму представления счетчика длины, оставляя решение этого вопроса на усмотрение разработчиков конкретной реализации. Для перехода от строки со счетчиком к строке с явно заданной длиной имеется слово COUNT (счетчик) A --> B,N, которое преобразует адрес A строки со счетчиком в адрес B ее первой литеры (обычно это A+1) и значение счетчика. Строки со счетчиком используются при вводе слов из входной строки. Стандартное слово WORD (слово) C--> A снимает со стека код литеры-ограничителя и выделяет из входной строки подстроку, ограниченную этой литерой (начальные вхождения литеры-ограничителя пропускаются). Из выделенной подстроки формируется строка со счетчиком, адрес которой возвращается в качестве результата. Слова-команды языка Форт вводятся исполнением текста BL WORD, а текстовая строка в слове ." - исполнением текста QUOTE WORD, где слово QUOTE - константа, обозначающая код кавычки. Литеры введенной строки обычно располагаются вслед за вершиной словаря, т. е. в незащищенном месте, и поэтому их нужно как-то защитить, если предполагается их дальнейшее использование. Некоторые реализации предусматривают слово " (кавычка), которое используется внутри определения через двоеточие и во время его исполнения кладет на стек адрес следующей строки как строки со счетчиком. Это позволяет работать с явно заданными текстами.

Чтобы программист мог задавать коды литер, не связывая себя конкретной кодировкой, во многие реализации введено слово C" , которое кладет на стек код первой литеры следующего слова и может использоваться как внутри определения через двоеточие, так и вне его:

```
: C" ( ---> C )   BL WORD COUNT DROP
        C@ [COMPILE] LITERAL ; IMMEDIATE
```

Исполнение текста BL WORD COUNT DROP C@ оставляет на стеке код первой литеры следующего слова. Далее этот код нужно либо скомпилировать как число, либо оставить на стеке в зависимости от текущего состояния текстового интерпретатора. Для этого используется уже известное нам слово LITERAL. Однако включить его непосредственно в текст нельзя, так как это слово имеет признак немедленного исполнения и будет исполняться во время компиляции данного определения. Чтобы скомпилировать слово с признаком немедленного исполнения, используется слово [COMPILE] (от COMPILE - компилировать) --> (компиляция), которое само имеет такой признак. Оно принудительным образом компилирует следующее за ним слово независимо от наличия у него признака немедленного исполнения. Таким образом, ввод строки, ограниченной кавычкой, с помощью слова C" можно задать так: C" " WORD. Такой текст более нагляден, чем тот, в котором используется конкретный код или обозначающая его константа.

Важной областью применения строковых значений являются форматные преобразования, позволяющие переводить число из машинного двоичного представления в строку литер. Эти преобразования выполняются над числами двойной длины, результирующая строка размещается во временном буфере PAD (прокладка), который заполняется с конца. Такое название буфера связано с тем, что он располагается в незащищенной части адресного пространства между словарем и стеком. Слово PAD кладет на стек адрес конца этого буфера и обычно определяется так:

```
: PAD ( ---> A )  HERE 100 + ;
```

В данном случае предполагается, что размер буфера не будет превышать 100 байт.

Собственно форматное преобразование начинается словом <# которое устанавливает служебную переменную HLD на конец буфера PAD:

```
: <# ( ---> ) PAD HLD ! ;
```

Занесение очередной литеры в буфер PAD выполняет слово HLD (сохранить):

```
: HLD ( C ---> )  -1 HLD +! HLD @ C! ;
```

Преобразование числа выполняет слово # DD1 --> DD2, которое работает со значениями двойной длины. Параметр делится на текущее значение переменной BASE (основание системы счисления) и заменяется на стеке получившимся частным, а остаток переводится в литеру соответствующую ему как цифра в данной системе счисления, и через слово HOLD эта литера добавляется в буфер PAD. Полный перевод числа выполняет слово #S:

```
: #S ( DD ---> 0,0 )  BEGIN # 2DUP D0= UNTIL ;
```

которое выделяет цифры числа последовательным делением, пока не получится нуль. Наконец, слово #> завершает форматное преобразование, возвращая адрес и длину получившейся текстовой строки:

```
: #> ( DD ---> A,N )  2DROP HLD @ PAD OVER - ;
```

Для вывода знака "минус" имеется слово SIGN (знак):

```
: SIGN ( A ---> )  0< IF C" - HOLD THEN ;
```

которое добавляет в буфер PAD знак "минус", если параметр на вершине стека (число одинарной точности) отрицателен.

С помощью перечисленных средств легко определить стандартные слова D. и . для печати чисел и в свободном (минимальном) формате:

```
: D.  ( DD ---> )
   2DUP DABS      ( DD,DDABS      )
    <#  #S        ( DD,0,0        )
        ROT       ( D-МЛ,0,0,D-CT )
        SIGN      ( D-МЛ,0,0      )
     #>           ( D-ML,A,N      )
   TYPE SPACE DROP ;
: .  ( N ---> ) S>D D. ;
```

Слово D. сначала переводит абсолютное значение исходного числа в строку литер, потом добавляет к ней возможный знак "минус", анализируя для этого старшую половину первоначального значения, и затем печатает получившуюся строку, выводя после нее еще один пробел. Слово . дополняет свой параметр до значения двойной длины распространением знакового разряда и обращается к слову D. для печати получившегося числа. Аналогичным образом реализуются стандартные слова D.R и .R , которые печатают число в поле заданного размера

вплотную к его правому краю (отсюда в их мнемонике присутствует буква R от RIGHT-правый), добавляя арн необходимости начальные пробелы:

```
: D.R  ( DD:ЧИСЛО,F:РАЗМЕР ПОЛЯ ---> )
    OVER  2SWAP  DABS       ( D-CT,F,DD,DABS )
     <#  #S  ROT  SIGN  #>  (  F,A,N )
    ROT  OVER  -            ( A,N,F-A )
    DUP  0>  IF SPACES ELSE DROP THEN  TYPE ;
: .R (N:ЧИСЛО,F:РАЗМЕР ПОЛЯ ---> )
    SWAP S>D D.R ;
```

В заключение рассмотрим программу шестнадцатиричиой распечатки областей памяти словом DUMP (дамп), которое получает на стеке адрес области и ее длину:

```
: DUMP ( A:АДРЕС,N:ДЛИНА ---> )
    OVER  BASE  @  HEX  2SWAP          ( A,B,A,N )
    +  ROT  -2  AND                    ( B,A+N,A )
    DO  I  <#  C" *  HOLD              ( B,AI )
      0 15 DO  DUP  I  +               ( B,AI,AI+J )
       C@  DECODE  HOLD  -1 +LOOP      ( B,AI )
              C" *  HOLD               ( B,AI )
      0 14 DO  BL HOLD  DUP            ( B,AI,AI+J )
       @ 0  # #  2DROP  -2 +LOOP       ( B,AI )
  BL HOLD  BL HOLD  0  # # # # #>      ( B,AT,NT )
        CR TYPE  16 +LOOP   BASE ! ;
```

Внешний цикл с шагом 16 формирует и печатает текстовую строку. Первый внутренний цикл с шагом -1 засылает в буфер PAD литерные значения, соответствующие распечатываемым байтам. Здесь слово DECODE C-->C/C1 заменяет код С на некоторый код С1 (например, код литеры "точка"), если он не является кодом литеры, которую можно напечатать на данном терминале. Второй внутренний цикл с шагом -2 засылает в буфер четыре шестнадцатиричные цифры - представления двухбайтных значений, разделяя их одним пробелом. Далее в буфере PAD (т. е. в начале строки) формируется адрес тоже в виде четырехзначного числа. Перед началом работы устанавливается шестнадцатиричная система счисления, а в конце восстанавливается первоначальная. Следующий протокол работы показывает результат исполнения слова DUMP в конкретном случае:

```
> 1000 40 DUMP
03E8 03C0 03EC 45E0 A220 0003 4520 A390 07FA *...Ж..s..,..TM.3*
03F8 04C5 D4C9 E300 03E4 0402 9180 A331 4780 *.EMIT..U..ЈЦТ..B*
0408 A414 45E0 A230 0004 4520 A37A 9180 A330 *U...S.....T:ЈЦТ.* OK
```

Обратите внимание, что адреса и значения байтов напечатаны в шестнадцатиричной системе. Точки в литерном представлении байтов заменяют литеры, которые не могут быть напечатаны.

1.9. Определяющие слова

Конструкции языка Форт, которые мы рассматривали до сих пор, имеют аналоги в других известных языках программирования. Например, определению через двоеточие очевидным образом соответствует описание процедуры в языках Фортран и Паскаль. Теперь мы рассмотрим свойство языка Форт, которое существенно отличает его от других и в значительной степени определяет его как нечто новое.

Программируя какую-либо задачу, мы моделируем понятия, в которых эта задача формулируется и решается, через понятия данного языка программирования. Традиционные языки имеют определенный набор понятий (процедуры, переменные, массивы, типы данных, исключительные ситуации и т.д.), с помощью которых программист должен выразить решение исходной задачи. Этот набор выразительных средств фиксирован в каждом языке, но

поскольку он содержит такие универсальные средства, как процедуры и типы данных, то опытный программист может смоделировать любые необходимые ему понятия программирования.

Язык Форт предлагает вместо фиксированного набора порождающих понятий единый механизм порождения таких порожающих понятий. Таким образом, приступая к программированию задачи на языке Форт, программист определяет необходимые ему понятия и с их помощью решает поставленную задачу. Такое прямое введение в язык нужных для данной задачи средств вместо их моделирования (часто весьма сложного) через имеющиеся универсальные средства уменьшает разрыв между постановкой задачи и ее программной реализацией. Это упрощает понимание и отладку программы, так как в ней более наглядно проступают объекты и отношения исходной задачи, и вместе с тем повышает ее эффективность, так как вместо сложного моделирования понятий задачи через универсальные используется их непосредственная реализация через элементарные.

Новые понятия вводятся посредством определения через двоеточие, в теле которого используются слова

```
CREATE    --->
DOES>     --->   (компиляция)
          ---> A (исполнение)
```

Рассмотрим уже известное нам слово CONSTANT (константа), которое используется для определения констант. Его определение можно задать так:

```
: CONSTANT (  N ---> )  CREATE , DOES> @ ;
```

Часть определения от слова CREATE до DOES> называется создающей (CREATE - создать), остальная часть от слова DOES > и до конца называется исполняющей (DOES - исполняет). В данном случае создающая часть состоит из одного слова , (запятая), а исполняющая часть - из слова @ (разыменование).

Рассмотрим исполнение данного определения на примере 4 CONSTANT XOP. Слово 4 кладет число 4 на стек. Далее исполняется слово CONSTANT. Слово CREATE, с которого начинается его определение, выбирает из входной строки очередное слово (в данном случае XOP) и добавляет его в словарь как новую команду. Создающая часть, состоящая из слова "запятая", переносит число 4 в память, компилируя его на вершину словаря. Слово DOES>, отмечающее конец создающей части, завершает исполнение данного определения, при этом семантикой созданного слова XOP будет последовательность действий исполняющей части, начиная от слова DOES>. В дальнейшем исполнение слова XOP начнется с того, что слово DOES> положит на стек адрес вершины словаря, какой она была на момент начала работы создающей части, после чего будет работать исполняющая часть определения. Поскольку по данному адресу создающая часть скомпилировала число 4, то исполняющая часть - разыменование - заменит на стеке этот адрес его содержимым, т. е. числом 4, что и требуется по смыслу данного понятия.

Рассмотрим другой пример. Введем понятие вектора, При создании вектора будем указывать размер (число элементов), а при обращении к нему - индекс (номер) элемента, в результате чего получается адрес данного элемента. Этот адрес можно разыменовать и получить значение элемента или можно заслать по этому адресу новое значение. Если желательно контролировать правильность индекса при обращении к вектору, то определение может выглядеть так:

```
: ВЕКТОР ( N:РАЗМЕР--->)  CREATE DUP ,  2* ALLOT
    DOES>  ( I:ИНДЕКС,A--->A[I]:АДРЕС ЭЛ-ТА I)
       OVER  1- OVER @ U<  ( ПРОВЕРКА ИНДЕКСА)
          IF  SWAP 2*  +  EXIT THEN
           ." ОШИБКА В ИНДЕКСЕ"  ABORT ;
```

Разберем, как работает данное определение при создании вектора 10 вектор X.

Создающая часть компилирует размер вектора и вслед за этим отводит память на 10*2, т. е. 20 байт. Таким образом, для вектора X в словаре отводится область размером 22 байта, в первых двух байтах которой хранится число 10 - размер вектора. При обращении к вектору X на стеке должно находиться значение индекса. Слово DOES> кладет сверху адрес области, сформированной создающей частью, после чего работает исполняющая часть определения. Проверив, что индекс I лежит в диапазоне от 1 до 10, она оставляет на стеке адрес, равный начальному адресу области плюс 1*2, т. е. адрес 1-го элемента вектора, если считать, что элементы располагаются в зарезервированной области подряд. Слово EXIT (выход) завершает исполнение определения, что позволяет обойтись без части "иначе" в условном операторе. Если окажется, что индекс не положителен или больше числа элементов, то будет напечатано сообщение "ошибка в индексе" словом ." , и исполнение закончится через слово ABORT (выброс). Если по каким-либо причинам контроль индексов не нужен, можно дать более краткое определение:

```
: ВЕКТОР ( N:РАЗМЕР ---> )  CREATE  2 *  ALLOT
    DOES>  ( I:ИНДЕКС,A ---> A[I]:АДРЕС ЭЛ-ТА I)
                SWAP  1-  2  *  +  ;
```

Если мы условимся считать индексы не от единицы, а то нуля, то исполняющая часть еще более сократится за счет исключения слова 1- для уменьшения значения индекса на единицу.

Таким образом, программист может реализовывать варианты понятий, наиболее подходящие для его задачи. Исходный небольшой набор слов-команд форт-системы он может избирательно наращивать а нужном направлении, постоянно совершенствуя свой инструментарий.

Используемый в языке Форт способ введения определяющих слов связан с очень важным понятием - частичной параметризацией. Определяющее слово задает целый класс слов со сходным действием, которое описывается исполняющей частью определяющего слова. Каждое отдельное слово из данного класса характеризуется результатом исполнения создающей части - тем или иным содержимым связанной с ним области памяти, адрес которой передается исполняющей части как параметр. Таким образом, исполняющая часть - то общее, что характеризует данный класс слов,- во время ее исполнения частично параметризуется результатом исполнения создающей части для данного отдельного представителя этого класса. Как создающая часть, так и частично параметризованная исполняющая часть, могут требовать дополнительных параметров для своего исполнения (в примере для вектора это размер вектора и индекс). Все это представляет программисту практически неограниченную свободу в создании новых понятий и удобных инструментальных средств.

Краткое описания команд общего назначения

Команда	Контекст	Описание команды
.TITLE	[FORTH]	Вывод информации о данной программе. Параметров нет.
//	[FORTH]	Комментарий до конца строки
--	[FORTH]	Комментарий до конца строки
(	[FORTH]	Комментарий до закрывающей круглой скобки
BYE	[FORTH]	Выход из программы
+	[FORTH]	Сложение A и B. A B +
-	[FORTH]	Вычитание (A-B). A B -
*	[FORTH]	Умножнение A на B. A B *

```
/               [FORTH] Деление A на B. A B /
%               [FORTH] Остаток от деления A на B. A B %
AND             [FORTH] Логическое И между A и B: A B AND
OR              [FORTH] Логическое ИЛИ между A и B: A B OR
XOR             [FORTH] Исключающее ИЛИ между A и B: A B XOR
NOT             [FORTH] Инверсия A: A NOT
WORDS           [FORTH] Вывод списка команд и контекстов.
.               [FORTH] Вывод числа с вершины стека в текущей системе счисления
D.              [FORTH] Вывод числа в десятичной системе счисления
H.              [FORTH] Вывод числа в шестнадцатиричной системе счисления
2H.             [FORTH] Вывод числа в шестнадцатиричной системе счисления в формате .2X
4H.             [FORTH] Вывод числа в шестнадцатиричной системе счисления в формате .4X
8H.             [FORTH] Вывод числа в шестнадцатиричной системе счисления в формате .8X
:               [FORTH] Начало определения новой команды
;               [FORTH] Завершение определения новой команды
[               [FORTH] Переход в режим исполнения
]               [FORTH] Переход в режим компиляции
C,              [FORTH] Компиляция байта на вершину словаря
,               [FORTH] Компиляция 16-ти разрядного слова на вершину словаря
DUMP            [FORTH] Вывод шестандцатиричного дампа памяти: addr n dump
HERE            [FORTH] Выдача на вершину стека адреса вершины словаря (адрес свободного места)
BEGIN           [FORTH] Начало цикла BEGIN-AGAIN: begin test again
AGAIN           [FORTH] Конец цикла BEGIN-AGAIN
DO              [FORTH] Начало цикла DO-LOOP (аналог цикла for для языка C): 10 0 do i . loop
LOOP            [FORTH] Конец цикла DO LOOP
EXIT            [FORTH] Завершение цикла (аналог break в языке C)
WHILE           [FORTH] Проверка предусловия для цикла BEGIN-WHILE-REPEAT
REPEAT          [FORTH] Конец цикла BEGIN-WHILE-REPEAT
UNTIL           [FORTH] Конец цикла BEGIN-UNTIL (цикл с проверкой в конце)
I               [FORTH] Внешний счетчик цикла DO-LOOP
J               [FORTH] Средний счетчик цикла DO-LOOP
K               [FORTH] Внутренний счетчик цикла DO-LOOP
IF              [FORTH] Часть условного ветвления IF-THEN-ELSE
THEN            [FORTH] Часть условного ветвления IF-THEN-ELSE
ELSE            [FORTH] Часть условного ветвления IF-THEN-ELSE
>               [FORTH] Кладет на стек ИСТИНУ (не 0) если A > B
<               [FORTH] Кладет на стек ИСТИНУ (не 0) если A < B
>=              [FORTH] Кладет на стек ИСТИНУ (не 0) если A >= B
<=              [FORTH] Кладет на стек ИСТИНУ (не 0) если A <= B
==              [FORTH] Кладет на стек ИСТИНУ (не 0) если A = B
<>              [FORTH] Кладет на стек ИСТИНУ (не 0) если A != B
ABORT           [FORTH] Прерывает выполнение текущей программы
ALLOT           [FORTH] Захватывает N байт памяти в словаре от текущего свободного места
R>              [FORTH] Переносит слово из стека данных в стек возвратов
>R              [FORTH] Переносит слово из стека возвратов в стек данных
R@              [FORTH] Копирует слово с вершины стека возвратов на вершину стек данных
R!              [FORTH] Заменяет слово на вершине стека возвратов
SWAP            [FORTH] Меняет два слова на стеке данных местами
DUP             [FORTH] Дублирует вершину стека данных
DROP            [FORTH] Убирает вершину стека данных
S.              [FORTH] Выдает содержимое стека данных
R.              [FORTH] Выдает содержимое стека возвратов
KEY             [FORTH] Выдает код нажатой клавиши (getch() из языка C)
CR              [FORTH] Выводит на консоль коды CR LF
!               [FORTH] Запоминает слово X в словаре по адресу A: X A !
@               [FORTH] Читает слово из словаря по адресу A: A @
ROT             [FORTH] Производит ротацию трех верхних элементов стека
OVER            [FORTH] Дублирует второй сверху элемент стека данных
"               [FORTH] Завершение текстовой строки
."              [FORTH] Вывод текстовой строки на консоль на этапе исполнения
DISFORTH        [FORTH] Дизассемблирование словаря
VARIABLE        [FORTH] Задание переменной
CONSTANT        [FORTH] Задание константы
ALL             [FORTH] Выключение контекстов
FORTH           [FORTH] Переход к контексту FORTH
INST            [FORTH] Переход к контексту INST
COM             [FORTH] Переход к контексту COM
USER            [FORTH] Переход к контексту USER
'               [FORTH] Кладет на стек адрес компиляции команды. Применяется для disforth: ' test disforth
EMIT            [FORTH] Вывод символа X на консоль. X emit
+TERM           [FORTH] Включение вывода символов на консоль. Параметров нет.
-TERM           [FORTH] Выключение вывода символов на консоль. Параметров нет.
+ECHO           [FORTH] Включение дублирования консольного вывода в файл (см. команду echo). Параметров нет.
-ECHO           [FORTH] Выключение дублирования консольного вывода в файл (см. команду echo). Параметров нет.
ECHO            [FORTH] Создание файла для хранения копии консольного вывода (см. команды +echo и -echo). echo
<имя_файла>
\ECHO           [FORTH] Закрытие файла для хранения копии консольного вывода (см. команды echo, +echo и -
echo). Параметров нет.
>>              [FORTH] Сдвиг числа X вправо на 1 бит. X >>
<<              [FORTH] Сдвиг числа X влево на 1 бит. X <<
?TERMINAL       [FORTH] Возвращает ИСТИНУ (не 0), если нажата кнопка на консоли
LFILE           [FORTH] загружает скрипт: lfile <имя>
@TIME           [FORTH] Кладет на стек дату и время в формате ANSI (4 байта)
.CTIME          [FORTH] Распечатывает дату и время в форме "Tue Sep 03 20:33:17 2002". Исходные данные
необходимо предоставить с помощью @time
HELP            [FORTH] Вывод справки по команде: help <имя_команды>
HELPS           [FORTH] Вывод справки по всем командам текущего контекста
HELPALL         [FORTH] Вывод справки по всем командам
SYSTEM"         [FORTH] Передача строки командному интерпретатору ОС. На стеке остается код возврата
(errorlevel): system" строка "
DIR             [FORTH] Вывод списка файлов на консоль (вызывается системная команда dir или ls). dir filemask
```

```
CLOCK            [FORTH] Время в мс от начала запуска программы. Команда может использоваться совместно с
командой ShowClock.
SHOWCLOCK        [FORTH] Выдает на консоль время в секундах (с точностью до десятых) прошедшее с момента
запуска clock
SLEEP            [FORTH] Подвешивание потока Win32 на заданное в мс время: time_ms Sleep
VERSION_CHECK    [FORTH] Контроль версии (защита от использования новых скриптов старыми интерпретаторами).
Если на стеке лежит версия большая (более старшая) чем версия данной программы, то происходит завершение работы
FORGET           [FORTH] Забыть указанное определения и все определения заданные позже. Имена из словаря
исчезают, а память не освобождается. forget name
.(               [FORTH] Вывод текстовой строки до закрывающей скобки (аналог команды ECHO). Команда
используется только вне определений через двоеточие. .( string )
OPENCHANNEL      [COM]   Открытие последовательного канала: 9600 openchannel com2
CLOSECHANNEL     [COM]   Закрытие последовательного канала
WSIO             [COM]   Вывод символа в последовательный канал
RSIO             [COM]   Чтение символа из последовательного канала
?RSIO            [COM]   Выдает ИСТИНУ, если в буфере приема есть символ
DEBUG            [COM]   Переключение системы в отладочный режим: 1 debug
TERM             [INST]  Включение эмулятора терминала: 0 term - ASCII, 1 - HEX, 3 - DEC, 3 term - 5-канальный
ASCII (переключение каналов производится командой <ESC>n со стороны контроллера)


HB166              (->) file1.hex file2.bin

                        HEX - BIN преобразователь для 64K.

                        пример: hexbin file.hex file.bin

HB32                (->) filename.hex filename.bin

                        HEX - BIN преобразователь.

                        пример: hexbin file.hex file.bin
HB32o           (->) filename.hex filename.bin

                        HEX - BIN преобразователь. В отличии от HB32 отрезает
                        пустое пространство из начала файла.

                        пример: hexbin file.hex file.bin


HB64            addr len (->) filename.hex filename.bin

                        HEX - BIN преобразователь для 64K (аналог hb166)
                        Позволяет получить бинарный образ из фрагмента HEX файла.
                        В отличии от HB166 заполняет пустые места кодом 0xFF.

                        пример: 0 2048 hexbin file.hex file.bin

HB_FRAGMENT32   addr len (->) filename.hex filename.bin

                        HEX - BIN преобразователь для файлов с 32-разрядным Intel HEX.
                        Позволяет получить бинарный образ из фрагмента HEX файла.
                        Заполняет пустые места кодом 0xFF.

                        пример: 0 2048 hb_fragment32 file.hex file.bin
```

Поддержка протокола PM3P

Команда	**Назначение**
T_FLASH	Переключение системы для работы с FLASH-памятью.
T_RAM	Переключение системы для работы с RAM.
erasepart	Стирание сектора.
eraseall	Стирание всей микросхемы FLASH
write	Запись файла в цель.
Read	Чтение файла из цели.
Set_target	Установка номера цели (0.255)
Wait_m	Ожидание готовности резидентной части системы к работе.
Port_mode	Режим открытия последовательного порта. При port_mode = 2 происходит сброс APM3000r2 и блокировка COM0.

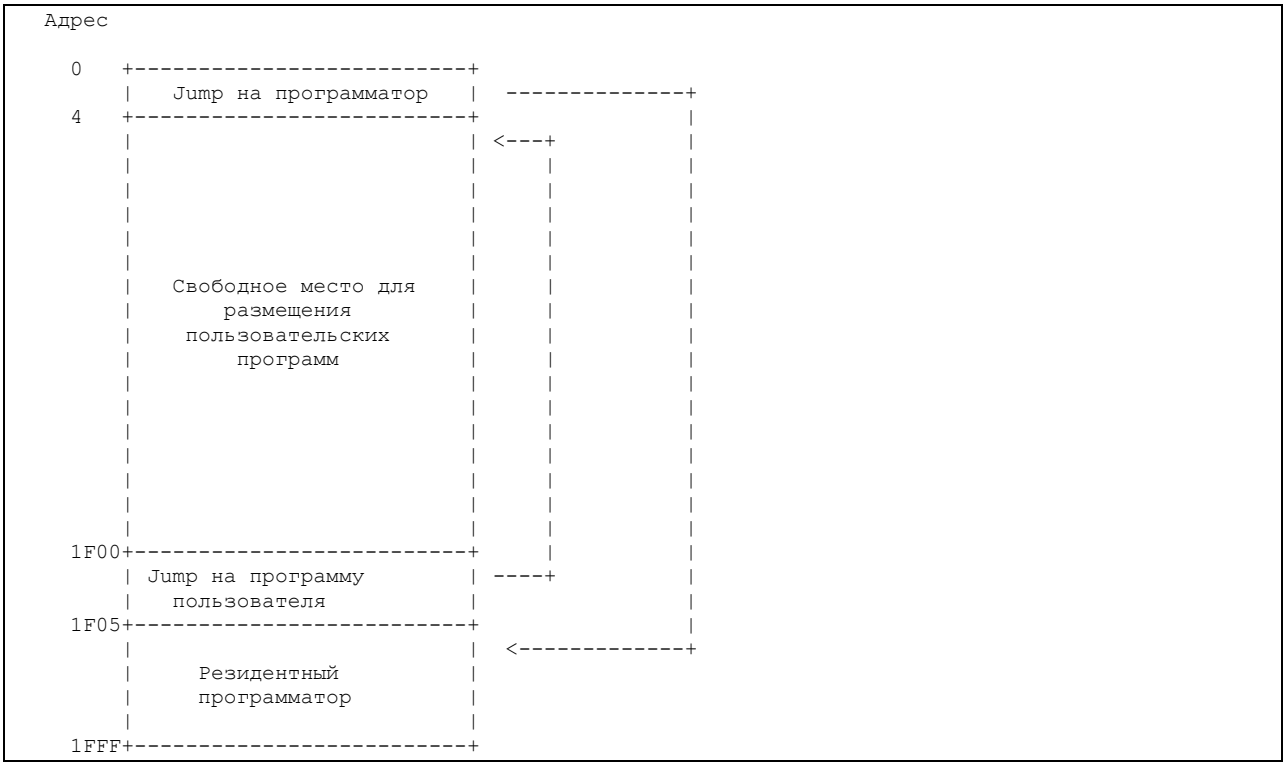
Поддержка загрузчика для PIC и AVR

Команда	Назначение
LOADPIC	Загрузка HEX файла в PIC или AVR. Пример: loadpic ttf50.hex

Резидентный программатор FLASH памяти для PIC 16F87x

Начальная инициализация портов В/В проводится для контроллеров TTF5.0 или TTF4.0 (см. раздел "особенности сборки"). Прерывания не используются.

Карта памяти для PIC 16F877



Алгоритм работы программатора

Сразу после старта программатор выдает символ '*' в последовательный канал.
Далее, в течении некоторого времени он ждет пакета данных от кросс загрузчика для программирования FLASH памяти. Если кросс система не передает никаких данных
в течении одной секунды (1 сек. - для PIC с частотой 4MHz), то контроллер передает
управление пользовательской программе (делает JMP на адрес 1F00h).

Особенности программирования FLASH

Если адрес принимаемых данных попадает в диапазон [0..3], то данные автоматически перенаправляются в ячейки FLASH по адресу 1F00h..1F04h. При компилировании загружаемой программы необходимо указать ключ -ICD.

```
        Описание протокола обмена
        -------------------------

1. После старта резидентный программатор выдает в последовательный канал
   символ '*' (9600,8,n,1).

2. Если в течение некоторого времени (зависит от частоты кварца, для 4 МГц
это
    ~520 мс ) не приходит символ '*' от ПК, то управление передается
   прикладной программе.

3. Пакет данных ПК -> загрузчик


   Формат команды:      <'a'><AddrL><AddrH><Длина><Данные><Контрольная сумма>
                              a - стартовый символ
                              AddrL - младший байт адреса FLASH
                              AddrH - старщий байт адреса FLASH
                              Длина - длина блока данных (до 255)
                              Контрольная сумма - сумма всех байтов
                              посылки включая стартовый символ
   Ответ контроллера:   <'+'> - все нормально
                        <'-'> - ошибка
   Назначение:          Программирование блока данных в память программ PIC

            Кросс средства программирования
            -------------------------------

В качестве кросс средств для программирования PIC можно использовать

Для Win32       - T2    v1.3.16 (и старше) команда loadpic filename.hex
Для DOS         - T167b v3.6    (и старше) команда loadpic filename.hex

            Ососбенности сборки загрузчика
            ------------------------------

Для сборки загрузчика в командной строке компилятора C необходимо указать
опицю -DTTF40 или -DTTF50 в зависимости от типа   контроллера.
```

Команды для работы с Ethernet

Команда	Назначение
tcp_client	Запуск тестового TCP клиента
tcp_server	Запуск тестового TCP сервера

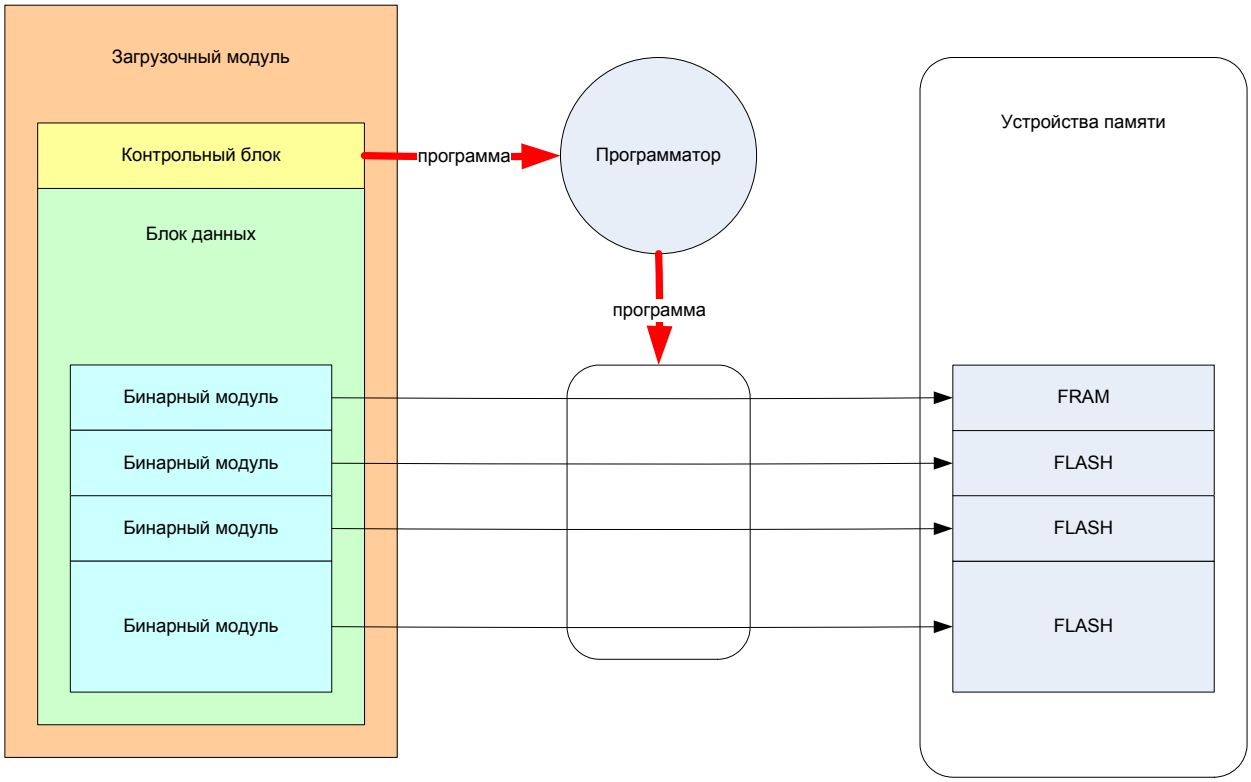
Создание загрузочного модуля

Формат загрузочного модуля

Загрузочный модуль – файл, содержащий всю необходимую информацию для программирования различных устройств памяти контроллера. В контроллере, загрузочный модуль хранится в специально отведенном устройстве памяти.

Контрольный блок предназначен для хранения информации, необходимой для инициализации различных устройств памяти контроллера после старта. В контрольном блоке хранится программа, позволяющая производить операции с целями, в рамках идеологии PM3P. Контрольный блок защищен CRC.

Блок данных содержит несколько бинарных модулей, полученных в процессе компиляции, с помощью различных инструментальных средств. В виде бинарного модуля может быть представлены как программы, так и конфигурационные данные. Блок данных защищен CRC.



Описание загрузчика

Формат контрольного блока

Наименование поля	Длина в байтах	Описание
Контрольный блок		
Сигнатура	4	Байты 0xAA553377
Длина контрольного блока	2	Длина включает в себя сигнатуру и CRC16 (sizeof от структуры).
Версия формата	2	Определяет особенности формата.
Смещение до команд	2	Смещение от начала структуры до начала программы. Позволяет вводить дополнительные поля в структуру.
Длина блока	2	

команд		
Смещение до данных	2	Смещение от начала структуры до начала данных. Позволяет вводить дополнительные поля в структуру.
Длина блока данных	4	Длина блока самих данных без учета полей длины, CRC32 и сигнатуры.
CRC32 блока данных	4	Циклический контрольный код.
Длина блока описания бинарных модулей	2	НЕ ПОДДЕРЖИВАЕТСЯ!
Блок описания бинарных модулей	n	НЕ ПОДДЕРЖИВАЕТСЯ! Target, Addr, Len, CRC32
Резерв	-	Место для дополнительных полей.
Блок команд	n	Программа, по которой производится программирование устройств памяти.
CRC16	2	Циклический контрольный код.
Блок данных		
Блок данных	n	Множество бинарных модулей.

Команды

Мнемоника	Операнды	Код операции	Описание
LD	Reg[2], const[4]	1	Загрузка константы в регистр
ERASE	Addr[4]	2	Стирание цели
COPY	Src[4], Dst[4], Len[4]	3	Копирование из адресного пространства цели SRC в адресное пространство цели DST
CHK	Нет	4	Проверка CRC32 блока данных
COMPARE	Src[4], Dst[4], Len[4]	5	Сравнение блоков данных DST и SRC
JMP_DST	Addr[4]	6	Передача управления прикладной программе, находящейся в адресном пространстве цели dst_target, по адресу dst_addr
VERSION_CON	Const[2]	7	Контроль версии формата.
FN	Const[2]	8	Вызов функции
RONE	Const[1]	9	=1 Перезапускать контроллер при ошибке =0 Игнорировать ошибку
END	Нет	255	Завершение работы программы

Регистры

Наименование	Тип памяти	Адрес	Размер в байтах	Описание
REG0	NVRAM	0	4	Результат выполнения команды
REG1	NVRAM	1	4	Цель источника
REG2	NVRAM	2	4	Цель приемника
REG3	NVRAM	3	4	IP (адрес последней команды) загрузчика
REG4	NVRAM	4	4	Регистр команд ISP
REG5	NVRAM	5	4	Отладочная информация
REG6	NVRAM	6	4	Отладочная информация
REG7	NVRAM	7	4	Отладочная информация
REG8..REG15	RAM	8..15	4	Регистры для использования в различных расширениях

Таблица векторов для команды FN

Наименование поля	Длина в байтах	Описание
Сигнатура	4	0x1234EDCB
Длина блока	2	Длина включает в себя сигнатуру и CRC16 (sizeof от структуры).
Версия формата	2	Определяет особенности формата.
Смещение до таблицы векторов	2	Смещение от начала структуры до начала таблицы векторов.
Количество векторов	2	
CRC32	4	Циклический контрольный код оверлея
Длина оверлея	4	
Смещение до оверлея	2	От начала структуры
Резерв	-	Место для дополнительных полей.
Блок векторов	N * 4	Адреса функций
CRC16	2	Циклический контрольный код.
…		
Оверлей		

Коды ошибок

Код ошибки	Описание
Ошибки PM3P	Обычные ошибки, выдаваемые программатором при работе в рамках протокола PM3P
Некорректная команда	Некорректный код операции или некорректный операнд.
Ошибка CRC блока данных	Испорчены данные в блоке данных.
Ошибка CRC контрольного блока	Испорчены данные в контрольном блоке.

Обработка ошибок

По факту возникновения ошибки действие интерпретатора команд прекращается и вызывается обработчик ошибки.

Описание команд M3P

Описание ассемблера

Мнемоника	Порядок операндов на стеке данных	Описание
LD	Reg const (->)	Загрузка константы в регистр
ERASE	Addr (->)	Стирание цели
COPY	Src Dst Len (->)	Копирование из адресного пространства цели SRC в адресное пространство цели DST
CHK		Проверка CRC32 блока данных
COMPARE	Src Dst Len (->)	Сравнение блоков данных DST и SRC
JMP_DST	Addr (->)	Передача управления
VERSION_CON	Const (->)	Контроль версии формата.

FN	Const (->)	Вызов функции
RONE	Const (->)	=1 Перезапускать контроллер при ошибке =0 Игнорировать ошибку
END		Завершение работы программы

Адреса регистров и номера целей описываются обычными константами языка FORTH.

Служебные функции

Мнемоника	Порядок операндов на стеке данных	Описание
Create_lm filename.lm	(->)	Создает файл загрузочного модуля
Add_file filename.bin	(->) offset, len	Добавляет бинарный модуль с именем filename.bin к загрузочному модулю. Возвращает смещение от начала контрольного блока и длину двоичного модуля.
close_lm	(->)	Закрывает файл загрузочного модуля. Сохраняет контрольный блок и блок данных на жестком диске в файле с именем filename.lm
Dump_lm filename.lm filename.lst	(->)	Дизассемблирует контрольный блок и выводит в листинг, также проверяет CRC контрольного блока и блока данных.

Пример программы

```
0x09 constant TRGT_MONITOR
0x34 constant TRGT_FLASH
0x60 constant TRGT_TGP
0x41 constant TRGT_RAM
0x35 constant TRGT_FLASH_P
0x42 constant TRGT_RAM_P

0x1  constant REG_STRGT
0x2  constant REG_DTRGT

0x00000000  constant    SECTOR0
0x00010000  constant    SECTOR1
0x00020000  constant    SECTOR2
0x00030000  constant    SECTOR3

variable test1_offset
variable test2_offset
variable test3_offset

variable test1_len
variable test2_len
variable test3_len

    create_lm test.lm

    add_file test1.bin  test1_len ! test1_offset !
    add_file test2.bin  test2_len ! test1_offset !
    add_file test3.bin  test3_len ! test1_offset !

-- ----------------------------------------
-- Программа
-- ----------------------------------------

    REG_STRGT   TRGT_FLASH_P    LD
    REG_DTRGT   TRGT_FLASH      LD

    SECTOR0                     ERASE
    SECTOR1                     ERASE
    SECTOR2                     ERASE
    SECTOR3                     ERASE

    0x00000000 SECTOR1 10        COPY

                                CHK

    0x12345678 SECTOR2 11        COMPARE

    0x80000000                  JMP_DST

    0x100                       VERSION_CON

    2                           FN

    1                           RONE

                                END
-- ---------------------------------------

    close_lm

    dump_lm test.lm test.lst
```

Примеры использования команд

```
-- ------------------------------------------------------------------------
--              ЦИКЛЫ
-- ------------------------------------------------------------------------


-- Бесконечный цикл BEGIN AGAIN


: test1 begin
        ." test1 "
    again ;

-- Цикл со счетчиком  DO LOOP, значение счетчика цикла принимает значения от
-- 0 до 9. Аналог на языке C: for(i=0;i<10;i++) { }

: test2 10 0    do
            i .
        loop ;

-- Цикл с проверкой в начале, продолжается если перед while на стеке ИСТИНА
-- (не 0)


: test3 begin   1 while

        ." test3 "
    repeat ;

-- Цикл с проверкой в конце, продолжается пока перед repeat на стеке ЛОЖЬ (0)


: test4 begin

        ." test4 "

    0 until ;
-- ------------------------------------------------------------------------
--              УСЛОВНЫЕ ВЕТВЛЕНИЯ
-- ------------------------------------------------------------------------

-- Операторы IF THEN

: test3
    2 == if
        " это двойка... " type cr
         then ;


-- Операторы IF ELSE THEN

: test4
    2 == if
        " это двойка... " type cr
          else
        " это не двойка" type cr
         then ;

-- 2 test3
-- 3 test4

-- ------------------------------------------------------------------------
--              ПЕРЕМЕННЫЕ и КОНСТАНТЫ
-- ------------------------------------------------------------------------

-- Константы

0x41 constant ADDR1

: test5
    ADDR1 h. cr
    ;


-- Переменные
```

```
variable var1

: test6

    0x15 var1 ! -- Записываем в переменную

    var1 @ h. cr    -- Читаем из переменной

    ;

test5
test6

-- ------------------------------------------------------------------------
--              Работа с файлами
-- ------------------------------------------------------------------------

variable buf
variable handle

here file_len allot buf !

: rfile
    open_file handle !

    handle @ buf @ file_len read_file cr ." Прочитано " . ." байт " cr

    handle @ close_file ;
```

Стек протоколов обмена РМ3Р 1.3[1]

Стек протоколов РМ3Р ориентирован на реализацию в микропроцессорных устройствах с ограниченными вычислительными ресурсами (с небольшим быстродействием и объемом памяти данных, таких как MCS51, PIC micro, Fujitsu $FM^2$16 и т.п.), снабженных радиальными интерфейсами типа точка-точка.

В данном документе рассмотрению подлежат протоколы взаимодействия между двумя активными устройствами (вычислителями) (резидентный монитор – кросс-монитор (на ПК)).

Протоколы, входящие в стек РМ3Р:

1. Канальные протоколы обмена точка-точка
   - 1.1. Пакетный протокол CP_P (Channel Protocol, Packet)
   - 1.2. Протокол CP_B (Channel Protocol, Bitblaster)
2. Прикладные протоколы
   - 2.1. Пакетный протокол AP_P (Application Protocol, Packet)

[1] В версии 1.3 TLA стал 32-разрядным

Базовые концепции протокола

Протокол PM3P поддерживает модель взаимодействия MASTER-SLAVE, при этом роль подчиненного устройства отводится целевой системе. Канальный уровень PM3P обеспечивает надежную доставку данных. Так как PM3P сделан по принципу «ЗАПРОС-ОТВЕТ», каждому запросу сделанному мастером на канальном уровне соответствует ответ целевой системы. При отсутствии ответа, производится ряд перезапросов.
Прикладной уровень обеспечивает работу с так называемыми целями(target). С точки зрения прикладного уровня протокола, целевая система представлена в виде множества целей (не более 256). «Цель» можно трактовать как виртуальную машину с некоторым набором команд, поставляемых вместе с данными через прикладной уровень протокола обмена. Множество команд каждой из целей может отличаться друг от друга, но возможны и пересечения. Например, для целей типа ПАМЯТЬ (RAM, FLASH) естественными являются команды записи данных (write), чтения данных (read), а если память является частью машины Фон-Неймана, то естественной выглядит команда передачи управления (jmp) или сброса (reset). Для специфических видов памяти (например, таких как FLASH), может быть реализована команда стирания (erasepart).
В настоящей спецификации зафиксировано несколько основных целей (FLASH, RAM, LOOPBACK).

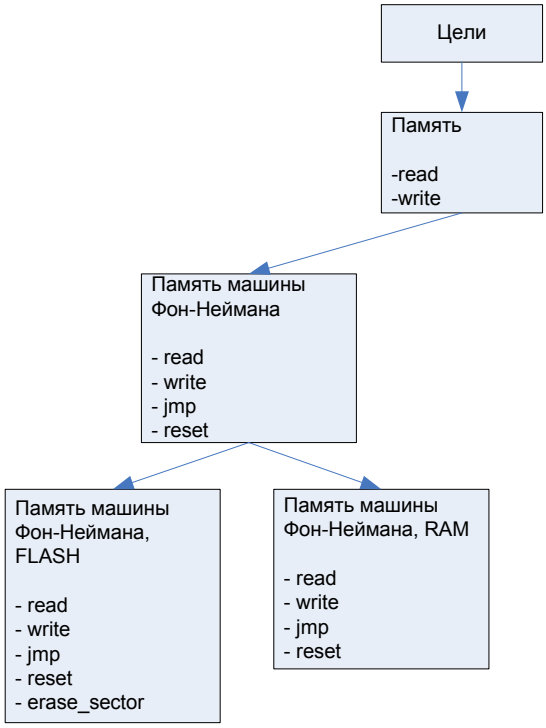


Рисунок 1 Иерархия целей

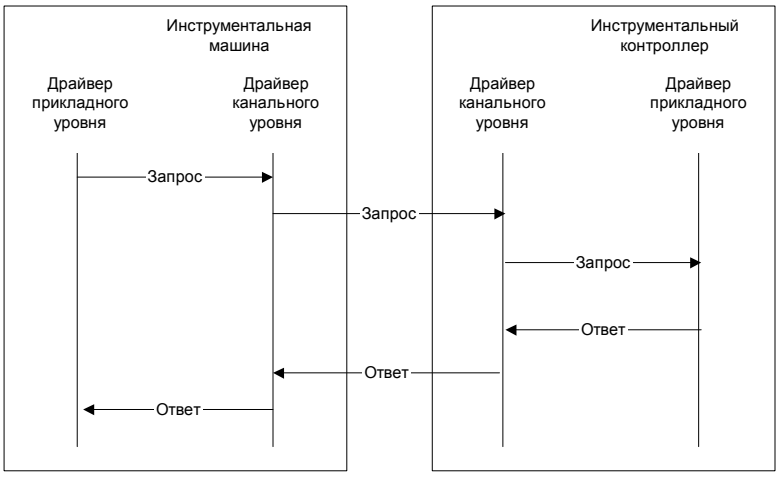
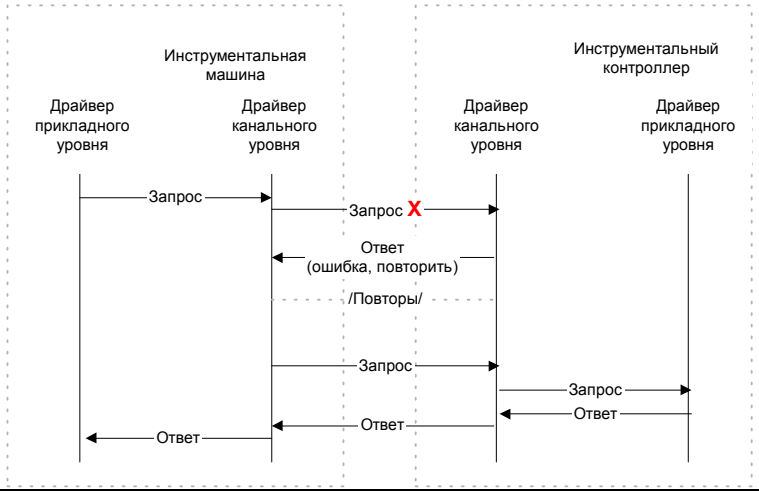
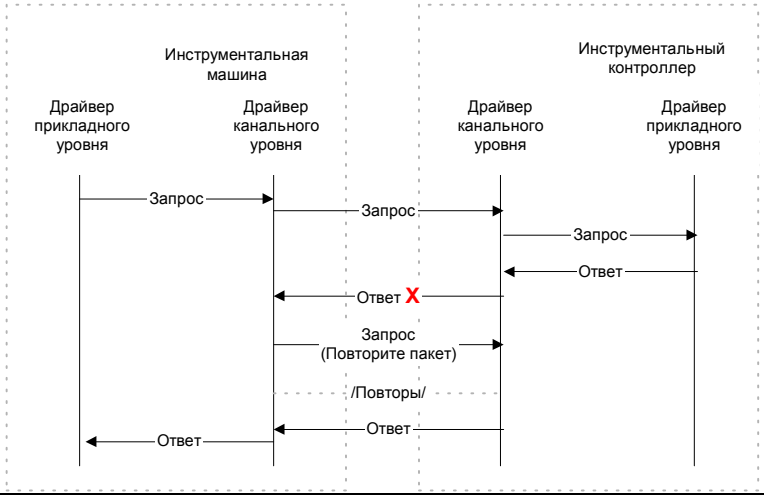
При необходимости, можно расширять и настраивать протокол для различных приложений. Можно предложить две основных стратегии развития PM3P:

- введение новых целей;
- использование адресной селекции, с выделением специальных областей адресного пространства имеющейся цели.

Первый вариант развития предполагает доработку спецификации и может привести к «взрыву» несовместимых версий реализаций. Этот путь развития необходимо использовать при работе с целями, радикально не совпадающими по своей идеологии построения с уже имеющимися в спецификации.
Второй вариант, хотя и не затрагивает спецификацию и системную часть реализации протокола, несколько менее удобен, из-за необходимости реализации адресной селекции на прикладном уровне в целевой системе.

Диаграммы взаимодействий для разных уровней РМ3Р

Таблица 1 Диаграммы взаимодействий РМ3Р

	Обмен данными между инструментальной машиной и инструментальным контроллером без ошибок канального уровня
	Обмен данными между инструментальной машиной и инструментальным контроллером при наличии ошибок канального уровня в пакете ЗАПРОС. Перезапросов может быть не более двух.
	Обмен данными между инструментальной машиной и инструментальным контроллером при наличии ошибок канального уровня в пакете ОТВЕТ. Перезапросов может быть не более двух.

Протоколы канального уровня

Пакетный протокол CP_P

Протокол обмена CP-P представляет собой протокол ЗАПРОС-ОТВЕТ. Инструментальная машина (PC) является master, M3P является slave. Тип кодирования – бинарный, байтовый, помехозащищенный (crc-8). Передаваемая информация сгруппирована в пакеты.

Форматы пакетов CP_P

Формат пакета ЗАПРОС имеет вид:

<СТАРТ><КОМ.><ДЛИНА>[<ПРИКЛАДНОЙ ПАКЕТ>]<CRC-8>

Где,

```
СТАРТ              [1 байт]    Стартовый символ '*'
КОМ.                    [1 байт]    Команда канального уровня
ДЛИНА              [1 байт]    Длина прикладного пакета
                         Если ДЛИНА =0 поле ПРИКЛАДНОЙ
ПАКЕТ - отсутствует
ПРИКЛАДНОЙ ПАКЕТ   [1 байт * ДЛИНА]
CRC-8                   [1 байт]    X8 + X5 + X4 + 1
```

Формат пакета ОТВЕТ имеет вид:

<СТАРТ><КОМ.><ДЛИНА>[<ПРИКЛАДНОЙ ПАКЕТ>]<CRC-8>

Где,

```
СТАРТ              [1 байт]    Стартовый символ '*'
КОМ.                    [1 байт]    Команда канального уровня
ДЛИНА              [1 байт]    Длина прикладного пакета
                         Если ДЛИНА =0 поле ПРИКЛАДНОЙ
                          ПАКЕТ - отсутствует
ПРИКЛАДНОЙ ПАКЕТ   [1 байт * ДЛИНА]
CRC-8                   [1 байт]    X^8 + X^5 + X^4 + 1
```

Таблица 2 Команды канального уровня

Код команды	**Назначение**
Бит 7 = 0	Команды пересылки данных
0x1	Пакет данных
Бит 7 = 1	Служебные команды
0x81	Ошибка CRC, повторите последний пакет еще раз

Протоколы прикладного уровня

Протокол AP_P

Протокол предназначен для:

- Передачи и чтения блоков данных;
- Организации работы с FLASH (чтение, запись, стирание целиком и отдельных секторов);
- Записи конфигурационного файла в Altera FLEX;
- Ретрансляции IEEE 1149.1 (JTAG) (в режиме BitBlaster и BitBlaster+);

Существует два типа пакетов:

- Минимальные (без поля данных)
- Максимальные (с полем данных)

<КОМАНДА><НОМЕР ЦЕЛИ>[<ДАННЫЕ>]

```
КОМАНДА                 [1 байт]    Команда [1..255]
НОМЕР ЦЕЛИ        [1 байт]    Номер цели
ДАННЫЕ                  [n  байт]   Передаваемые данные
```

<КОД ОШ. ПУ><КОМАНДА><НОМЕР ЦЕЛИ> [<ДАННЫЕ>]

```
КОД ОШ. ПУ              [1 байт]    Код ошибки прикладного
                                     уровня [0..255]
КОМАНДА                 [1 байт]    Команда [1..255]
НОМЕР ЦЕЛИ        [1 байт]    Номер цели
ДАННЫЕ                  [n  байт]   Передаваемые данные
```

Таблица 3 Коды ошибок прикладного уровня

Код ошибки/ Приоритет (0 – самый высокий)	**Описание**
ОШИБКИ ПРИКЛАДНОГО УРОВНЯ	
0x00	Нет ошибок
0x11/1	Ошибка кода команды
0x12/1	Ошибка прикладного параметра
0x13/1	Ошибка номера цели
Ошибки целевых устройств	
0x30/3	Ошибка чтения
0x31/3	Ошибка записи
0x32/3	Ошибка стирания
0x33/3	Ошибка доступа
0x34/3	Ошибка верификации записи

Таблица 4 Базовые форматы полей данных

Версия продукта	< v0 >< v1 >< v2 >< v3 > [1 byte][1 byte][1 byte][1 byte] Версия, 4 байта, младший номер версии – вперед
Адрес устройства памяти	< A0>< A1> < A2 >**< A3 >**[2] [1 byte][1 byte][1 byte][1 byte] TLA - младший байт вперед
Массив данных	1) ПРОСТОЕ кодирование <размер>< массив > [1 byte][1 byte * размер] размер = [1..255]
Запрос массива	<размер> [1 byte]
Статус	<тип статуса>< данные > [1 byte] [определяется типом статуса]

[2] Начиная с версии PM3P 1.3, TLA стал 32-разрядным.

Таблица 5 Номера целей

Номер цели	**Имя цели**	**Описание**	**Обязательность реализации**
0x1	LOOPBACK	Тестирование	Желательно
0x09	MONITOR	Монитор	Обязательно
0x34	FLASH	FLASH память с байтовым или пословным (16 разрядов) доступом.	По необходимости
0x41	RAM	ОЗУ	По необходимости
0x35	FLASH_P	Страничная FLASH память (например, такая как в MCU семейства Philips LPC2000). Цель использует _RAM_P для буферизации. По факту k.,jq записи в FLASH_P происходит записи содержимого _RAM_P во FLASH память контроллера по адресу, указанному в команде set_addr. Размер страничного буфера спрашивается у цели T_FLASH_P. Перед запросом размера буфера устанавливается предполагаемый адрес записи. В адресном пространстве цели могут находиться области с разными страничными буферами.	По необходимости
0x42	_RAM_P	Буфер страничной памяти (нужен только для поддержки цели FLASH P).	Обязательно для поддержки FLASH P

Таблица 6 Форматы пакетов прикладного уровня для цели MONITOR

Код команды	**Назначение**	Формат поля данных запроса	Формат поля данных ответа	Обязательность реализации
0x1	Дать версию монитора	Min	Версия продукта	Желательно
0x3	Дать размер канальных буферов	Min	<Вх><Вых> [1 байт][1 байт]	**Обязательно**

Таблица 7 Форматы пакетов прикладного уровня для цели RAM

Код команды	**Назначение**	Формат поля данных запроса	Формат поля данных ответа	Обязательность реализации
0x1	Дать версию монитора	Min	Версия продукта	Желательно
0x3	Дать размер канальных буферов	Min	<Вх><Вых> [1 байт][1 байт]	Желательно
0x4	Установка адреса[3]	Адрес устройства памяти	Min *	**Обязательно**
0x5	Передача массива	Массив данных	Min	**Обязательно**
0x6	Чтение массива	Запрос массива	Массив данных	**Обязательно**
0xC	Передать управление (JMP) на текущий адрес	Min	НЕТ ОТВЕТА	**Обязательно**
0xD	Рестарт	Min	Min	Желательно
0xE	Сброс сквозного счетчика CRC32	Min	Min	Желательно
0xF	Запрос сквозного счетчика CRC32 ***	Min	[b0][b1][b2][b3]	Желательно

* Min – Минимальный формат пакета (не содержит поля данных).
** При работе с массивами происходит автоинкремент адреса
*** Счетчик CRC изменяется при получении и передаче пакетов данных

[3] Начиная с версии PM3P 1.3, TLA стал 32-разрядным.

Таблица 8 Форматы пакетов прикладного уровня для цели FLASH

Код команды	**Назначение**	Формат поля данных запроса	Формат поля данных ответа	Обязательность реализации
0x1	Дать версию монитора	Min	Версия продукта	Желательно
0x3	Дать размер канальных буферов	Min	<Вх><Вых> [1 байт][1 байт]	Желательно
0x4	Установка адреса[4]	Адрес устройства памяти	Min *	**Обязательно**
0x5	Передача массива	Массив данных	Min	**Обязательно**
0x6	Чтение массива	Запрос массива	Массив данных	**Обязательно**
0x7	Стирание всей цели	Min	Min	Желательно
0x8	Стирание фрагмента цели	Min	Min	**Обязательно**
0xC	Передать управление (JMP) на текущий адрес	Min	НЕТ ОТВЕТА	Желательно
0xD	Рестарт	Min	Min	Желательно
0xE	Сброс сквозного счетчика CRC32	Min	Min	Желательно
0xF	Запрос сквозного счетчика CRC32 ***	Min	[b0][b1][b2][b3]	Желательно

* Min - Минимальный формат пакета (не содержит поля данных).
** При работе с массивами происходит автоинкремент адреса
*** Счетчик CRC изменяется при получении и передаче пакетов данных

Таблица 9 Форматы пакетов прикладного уровня для цели _RAM_P

Код команды	**Назначение**	Формат поля данных запроса	Формат поля данных ответа	Обязательность реализации
0x1	Дать версию монитора	Min	Версия продукта	Желательно
0x3	Дать размер канальных буферов	Min	<Вх><Вых> [1 байт][1 байт]	Желательно
0x4	Установка адреса[5]	Адрес устройства памяти	Min *	**Обязательно**
0x5	Передача массива	Массив данных	Min	**Обязательно**
0x6	Чтение массива	Запрос массива	Массив данных	**Обязательно**
0x9	Запрос размера страницы	Min	<SizeL><SizeM> [1 байт][1 байт]	**Обязательно**
0xD	Рестарт	Min	Min	Желательно
0xE	Сброс сквозного счетчика CRC32	Min	Min	Желательно
0xF	Запрос сквозного счетчика CRC32 ***	Min	[b0][b1][b2][b3]	Желательно

* Min - Минимальный формат пакета (не содержит поля данных).
** При работе с массивами происходит автоинкремент адреса
*** Счетчик CRC изменяется при получении и передаче пакетов данных

[4] Начиная с версии PM3P 1.3, TLA стал 32-разрядным.
[5] Начиная с версии PM3P 1.3, TLA стал 32-разрядным.

Таблица 10 Форматы пакетов прикладного уровня для цели FLASH_P

Код команды	**Назначение**	Формат поля данных запроса	Формат поля данных ответа	Обязательность реализации
0x1	Дать версию монитора	Min	Версия продукта	Желательно
0x3	Дать размер канальных буферов	Min	<Вх><Вых> [1 байт][1 байт]	Желательно
0x4	Установка адреса[6]	Адрес устройства памяти	Min *	**Обязательно**
0x5	Передача массива	Массив данных	Min	**Обязательно**
0x6	Чтение массива	Запрос массива	Массив данных	**Обязательно**
0x7	Стирание всей цели	Min	Min	Желательно
0x8	Стирание фрагмента цели	Min	Min	**Обязательно**
0xC	Передать управление (JMP) на текущий адрес	Min	НЕТ ОТВЕТА	Желательно
0xD	Рестарт	Min	Min	Желательно
0xE	Сброс сквозного счетчика CRC32	Min	Min	Желательно
0xF	Запрос сквозного счетчика CRC32 ***	Min	[b0][b1][b2][b3]	Желательно

* Min – Минимальный формат пакета (не содержит поля данных).
** При работе с массивами происходит автоинкремент адреса
*** Счетчик CRC изменяется при получении и передаче пакетов данных

Литература

1. Баранов С.Н. Язык форт и его реализации.

[6] Начиная с версии PM3P 1.3, TLA стал 32-разрядным.