(Продолжение.Начало — №4/2001)

Микроконтроллеры? Это же просто!

При публикации статьи Александра Фрунзе «Микроконтроллеры? Это же просто!» по вине редакции была пропущена заключительная часть главы «Регистры микроконтроллера». Мы приносим свои извинения автору и читателям и публикуем пропущенный материал.

Глава 3. Регистры микроконтроллера

Регистр-указатель данных

Последний из рассматриваемых в этой главе регистров — регистр-указатель данных DPTR. В отличие от предыдущих 8-разрядных (однобайтовых) регистров он — 16-разрядный (двухбайтовый). Его основное назначение — хранение адреса ячейки внешней памяти данных при обращении микроконтроллера к этой памяти.

Вообще у x51 есть всего две команды для работы с внешней памятью данных — MOVX A,@DPTR и MOVX @DPTR,A. Первая из них осуществляет чтение данных в аккумулятор из ячейки внешней памяти, адрес которой помещен в DPTR. Вторая же осуществляет обратную операцию — запись данных из аккумулятора в ячейку внешней памяти, адрес которой хранится в DPTR. Замечу, что буква X в конце команды MOVX подчеркивает, что команда предназначена для работы с внешней памятью данных (eXternal — внешний). Знак @ означает, что данные пересылаются в ячейку, адрес которой хранится в регистре, записанном после этого знака (в данном случае DPTR).

Ну а занести адрес интересующей нас ячейки в регистр DPTR можно командой MOV DPTR,#data16, где #data16 — 16-разрядное число, которое может принимать значение от 0 до 65535 (0FFFFH). Вспомним наш пример из предыдущего подраздела («Схемотехника», №10/2001), когда мы говорили о том, что микроконтроллер записывает число 145 в 84-ю ячейку внешнего ОЗУ. Сделать это можно при помощи следующего фрагмента:

```
;
MOV   A,#145      ;ЗАНОСИМ В АККУМУЛЯТОР ЧИСЛО 145
MOV   DPTR,#84    ;ЗАНОСИМ В DPTR ЧИСЛО 84 (АДРЕС
ЯЧЕЙКИ)
MOVX  @DPTR,A ;ЗАПИСЫВАЕМ 145 В 84-Ю ЯЧЕЙКУ
;
```

Попробуйте самостоятельно модифицировать этот фрагмент таким образом, чтобы он предписал МК прочитать в аккумулятор число из 84-й ячейки внешнего ОЗУ.

Пример: подпрограмма, использующая регистры МК

Как уже было замечено ранее, нельзя научиться плавать в бассейне без воды. Точно также нельзя научиться работать с микроконтроллерами, не освоив основных навыков написания программ. Поэтому с целью закрепления материала, касающегося регистров, я предлагаю вам как пример одну довольно важную для ваших дальнейших изысканий подпрограмму, которая использует почти все рассмотренные в этой главе регистры. Заодно я покажу вам, как составляется блок-схема алгоритма. Замечу, что на самом деле составление блок-схем — это самое главное в программировании. Когда блок-схема составлена, написать текст программы — дело техники, причем почти неважно, на каком языке писать и для какого микроконтроллера.

Теперь о подпрограмме, которую нам предстоит рассмотреть. Это подпрограмма преобразования хранящегося в регистрах R3 и R2 МК двухбайтового числа (т. е. лежащего в диапазоне от 0000H до 0FFFFH) в так называемое двоично-десятичное.

Что представляет из себя двоично-десятичное представление числа? Проще всего это объяснить на следующем примере. Воспользуйтесь, как было описано в главе 1, Windows-калькулятором и убедитесь, что 0FF00H = 1111111100000000B = 65280D. Это шестнадцатеричное, двоичное и обычное десятичное представление числа 65280. Двоично-десятичное его представление выглядит следующим образом: 0110 0101 0010 1000 0000. Если вы внимательно всмотритесь в него, то обнаружите, что старшие 4 бита (0110) кодируют в двоичном представлении цифру 6, идущие вслед за ними 4 бита (0101) — цифру 5, следующие 4 бита (0010) — цифру 2, потом идет восьмерка (1000) и нуль (0000). Таким образом, каждый десятичный разряд нашего числа кодируется четырехбитным двоичным числом. Напомню, каковы эти двоичные числа: 0000 — это 0, 0001 — 1, 0010 — 2, 0011 — 3, 0100 — 4, 0101 — 5, 0110 — 6, 0111 — 7, 1000 — 8 и 1001 — 9. Если понятно, то какому числу соответствует следующее двоично-десятичное представление: 0010 0001 1001 0101 0111? Правильно, 21957. Кто этого не понял, еще раз внимательно прочитайте текущий абзац.

Для чего нужно двоично-десятичное представление чисел? Для подпрограмм, работающих с цифровыми знакосинтезирующими индикаторами. Прежде, чем отобразить тот или иной результат на семисегментном индикаторе, жидкокристаллическом или светодиодном, отображаемое число нужно преобразовать в двоично-десятичное представление.

Теперь поговорим более конкретно, о том, что, откуда и куда мы будем преобразовывать. Исходное число будет храниться в регистрах R3 и R2 — старшие 8 бит в R3, младшие — в R2. Результат преобразования будет располагаться в регистрах R6, R5, и R4. При этом в младших 4 битах R6 будет располагаться старший десятичный разряд (десятки тысяч), в старших 4 битах R5 — разряд единиц тысяч, в младших 4 битах R5 — разряд сотен, а в старших и младших 4 битах R4 – разряды десятков и единиц соответственно. Например, уже упомянутое 0FF00H должно оказаться в регистрах R6, R5, и R4 в следующем виде: R6 = 00000110B, R5 = 01010010B, R4 = 10000000B. Сравните с тем, как 0FF00H было представлено двумя абзацами выше.

А теперь самое главное, — каким образом из числа 0FF00H (или любого другого) выделить десятки тысяч, единицы тысяч, сотни, десятки и единицы? Как без помощи Windows-калькулятора определить, что 0FF00H состоит из 6 десятков тысяч, 5 тысяч, 2 сотен, 8 десятков и 0 единиц? На самом деле очень просто, хотя пока у вас нет никаких навыков вы даже и не представляете, как это делается.

Ответьте пожалуйста, сколько раз подряд из числа, принадлежащего к диапазону от 60000 до 69999, можно вычитать 10000, чтобы при этом остаток был больше или равен нулю? Не правда ли, ровно 6 раз. Когда вы попытаетесь это сделать в седьмой раз, у вас получится отрицательный результат. Соответственно, из числа, лежащего в диапазоне 20000-29999 можно до получения нулевого или положительного остатка вычесть 10000 не более 2 раз, из числа, лежащего в диапазоне 40000-49999, — не более 4 раз и т. д. Понимаете, к чему я клоню?

Для тех, кто еще не догадался, поясню. Вычитая раз за разом из преобразуемого числа 10000, определим, сколько раз эти 10000 содержаться в нем — это будет разряд десятков тысяч. Далее будем из остатка точно также вычитать 1000, и таким образом определим, сколько раз в нем содержится

1000 — это будет разряд тысяч, и так далее.

Отсюда алгоритм преобразования должен выглядеть следующим образом. Вначале разряд десятков тысяч преобразуемого числа принимаем равным 0. После этого из преобразуемого числа нужно вычесть 10000 и проверить, остаток больше или равен 0 или нет. Если да, то разряд десятков тысяч увеличиваем на 1 (т. е. после первого удачного вычитания он будет равен 1), и снова повторяем вышеописанные действия). После второго удачного вычитания разряд десятков тысяч будет равен 2, после третьего — 3 и т. д.

Рано или поздно, но после какого-то вычитания остаток станет отрицательным. (Кстати, если преобразуемое число не более 9999, он отрицательным будет уже после первого вычитания.) В этом случае следует оставить неизменным разряд десятков тысяч, прибавить к полученному отрицательному остатку 10000 и запомнить его и найденный разряд для дальнейшего преобразования. В рассматриваемом нами примере с преобразованием числа 0FF00H перед седьмым вычитанием остаток будет равен 5280, а разряд десятков тысяч — 6. После седьмого вычитания в остатке остается (простите за повторение) минус 4720. Отрицательный остаток сигнализирует нам, что с вычитаниями мы слегка переборщили, и что надо прибавить к этому остатку 10000, получив при этом исходные 5280, запомнить это число и то, что разряд десятков тысяч в 0FF00H равен 6. А далее, как нетрудно догадаться, следует перейти к нахождению разряда единиц тысяч.

Определять разряд тысяч будем по тому же алгоритму. Вначале разряд единиц тысяч преобразуемого числа принимаем равным 0. После этого из преобразуемого числа нужно вычесть 1000 и проверить, остаток больше или равен 0 или нет. Если да, то разряд единиц тысяч увеличиваем на 1 (т. е. после первого удачного вычитания он будет равен 1), и снова повторяем вышеописанные действия. После второго удачного вычитания разряд единиц тысяч будет равен 2, после третьего — 3 и т. д.

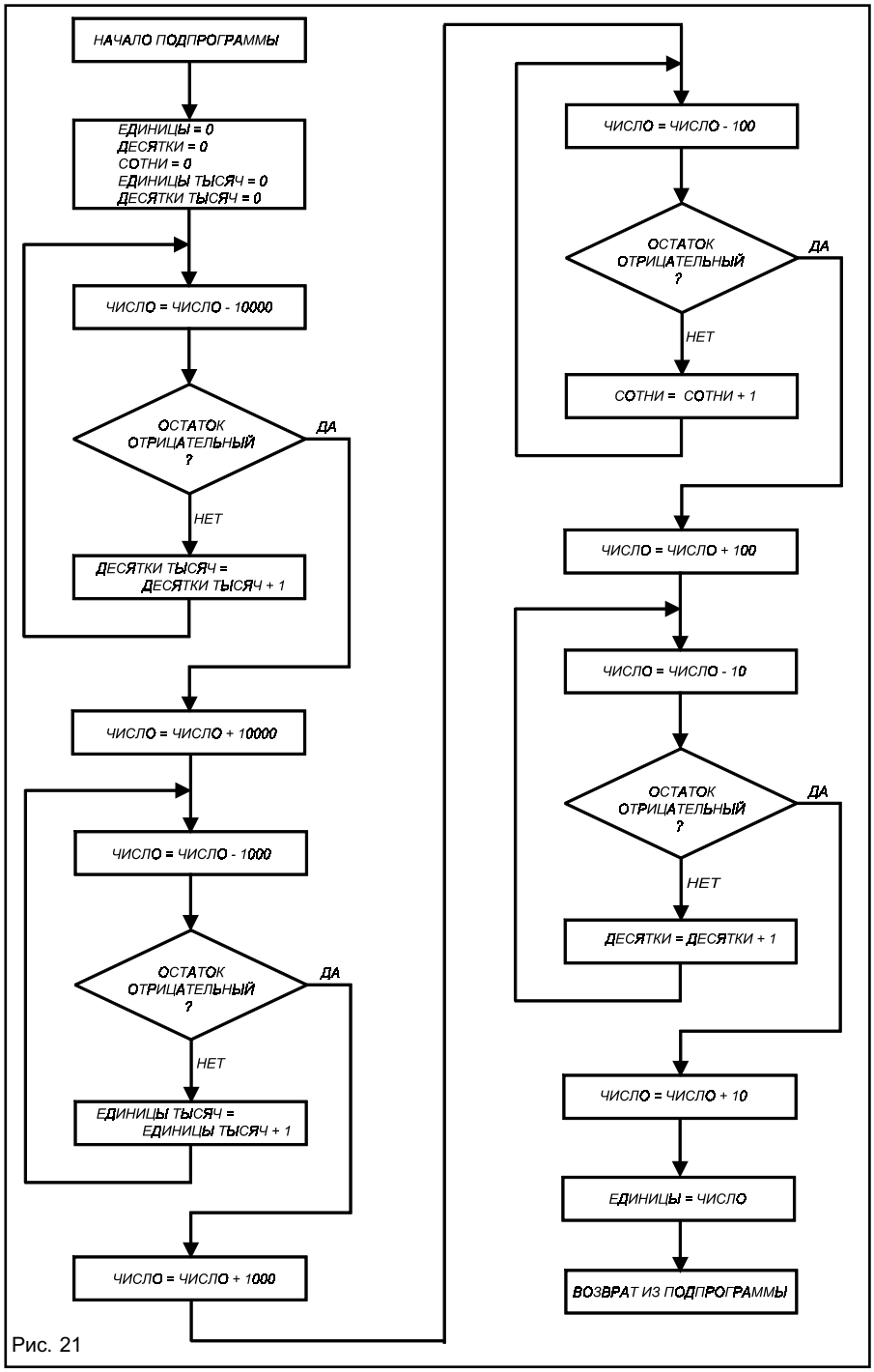

Рис. 21

Когда после какого-то вычитания остаток станет отрицательным, нужно будет оставить неизменным разряд единиц тысяч, прибавить к остатку 1000 и запомнить его и полученный разряд для дальнейшего преобразования. В рассматриваемом нами примере с 0FF00H мы начали вычитать тысячи из 5280, доставшегося нам в наследство после этапа определения разряда десятков тысяч. Перед шестым вычитанием остаток будет равен 280, а разряд единиц тысяч — 5. После шестого вычитания в остатке будет минус 720. Отрицательный остаток опять сигнализирует нам, что с вычитаниями мы переборщили, и что надо прибавить к этому отрицательному остатку 1000, получив при этом исходные 280, запомнить это число и то, что разряд единиц тысяч в 0FF00H равен 5. А далее, как очевидно, перейти к нахождению разряда сотен.

Описывать, как мы должны из 280 раз за разом вычитать 100, я не буду — это должно быть уже понятным для всех. Если что-то непонятно, попробуйте еще раз внимательно прочесть содержимое последних восьми абзацев. Если же и это не поможет, не опускайте руки, у вас в распоряжении мой электронный адрес, попытайтесь сформулировать, что же вы не поняли, и киньте мне письмо с этой формулировкой.

После определения разряда сотен определим разряд десятков, а остаток в результате определения десятков и будет разрядом единиц. Вот, собственно, и весь алгоритм.

Сказанное иллюстрируется блок-схемой алгоритма, приведенной на рис. 21.

Блок-схема представляет из себя последовательность идущих сверху вниз прямоугольников и ромбов, внутри которых кратко описаны действия, которым они соответствуют, а стрелки, их соединяющие, показывают порядок выполнения этих действий. Действия, описанные в ромбах, являются проверкой тех или иных условий (в нашем случае проверка результата вычитания, отрицателен он или нет). Поскольку при невыполнении проверяемого условия мы должны совершить одну последовательность действий, а при выполнении — другую, то из каждого ромба выходит

две стрелки (одна с надписью «да», т. е. соответствующая выполнению, другая с надписью «нет» — невыполнению). Собственно, больше и говорить нечего — сравните то, что говорилось в семи последних абзацах с тем, что изображено на рис. 21 и убедитесь, что алгоритм, изображенный графически, соответствует алгоритму, сформулированному словами. Но графическое представление проще и нагляднее.

Приведенная на рис. 21 блок-схема, если так можно выразиться, машинно-независима, т. е. никак не учитывает особенностей того или иного микроконтроллера, и, следовательно, применима к любому из них. Но нам нужно составить блок-схему, ориентированную на регистры и систему команд x51. Такая схема приведена на рис. 22.

Как видите, структура алгоритма осталась неизменной. Просто действия описаны поконкретнее — вместо «вычесть из результата 10000» стоит «R3R2-10000», вместо «результат вычитания отрицательный?» стоит «CY=1?», и т. д.

На что я хотел бы обратить выше внимание? Во-первых, в приведенном ниже тексте подпрограммы, соответствующей этому алгоритму, вы обнаружите, что вместо того, чтобы вычитать из преобразуемого числа вначале 10000, затем 1000, затем 100 и затем 10 (это четыре различных фрагмента программы) я вначале заношу 10000 (1000, 100 или 10) в регистр DPTR, а затем вычитаю его из R3R2. В этом случае все четыре вычитания выполняются идентично, поэтому они могут быть оформлены в виде подпрограммы (я назвал ее R32MNDPT), которая вызывается по мере необходимости. Аналогичная подпрограмма (R32PLDPT) осуществляет прибавление по мере надобности к числу в R3R2 10000, 1000, 100 и 10.

Во-вторых, разряды десятков тысяч, единиц тысяч, сотен, десятков и единиц я занулял не по мере необходимости, как в описанном словами алгоритме, а в самом начале программы (записью нулевого значения в регистры R6, R5 и R4, где будет хранится результат преобразования).

В-третьих, поскольку разряд единиц тысяч и разряд сотен должны в результате преобразования храниться в одном и том же регистре (R5), то единицы тысяч я подсчитываю в дополнительно используемом регистре R7, а сотни — в R5, и затем младшие 4 бита R7 я переношу в старшие, суммируя его с R5. В результате в старшей половине R5 окажутся единицы тысяч, в младшей — сотни, что нам и требовалось. Аналогично я действую и в отношении десятков и единиц, которые я собираю в регистре R4.

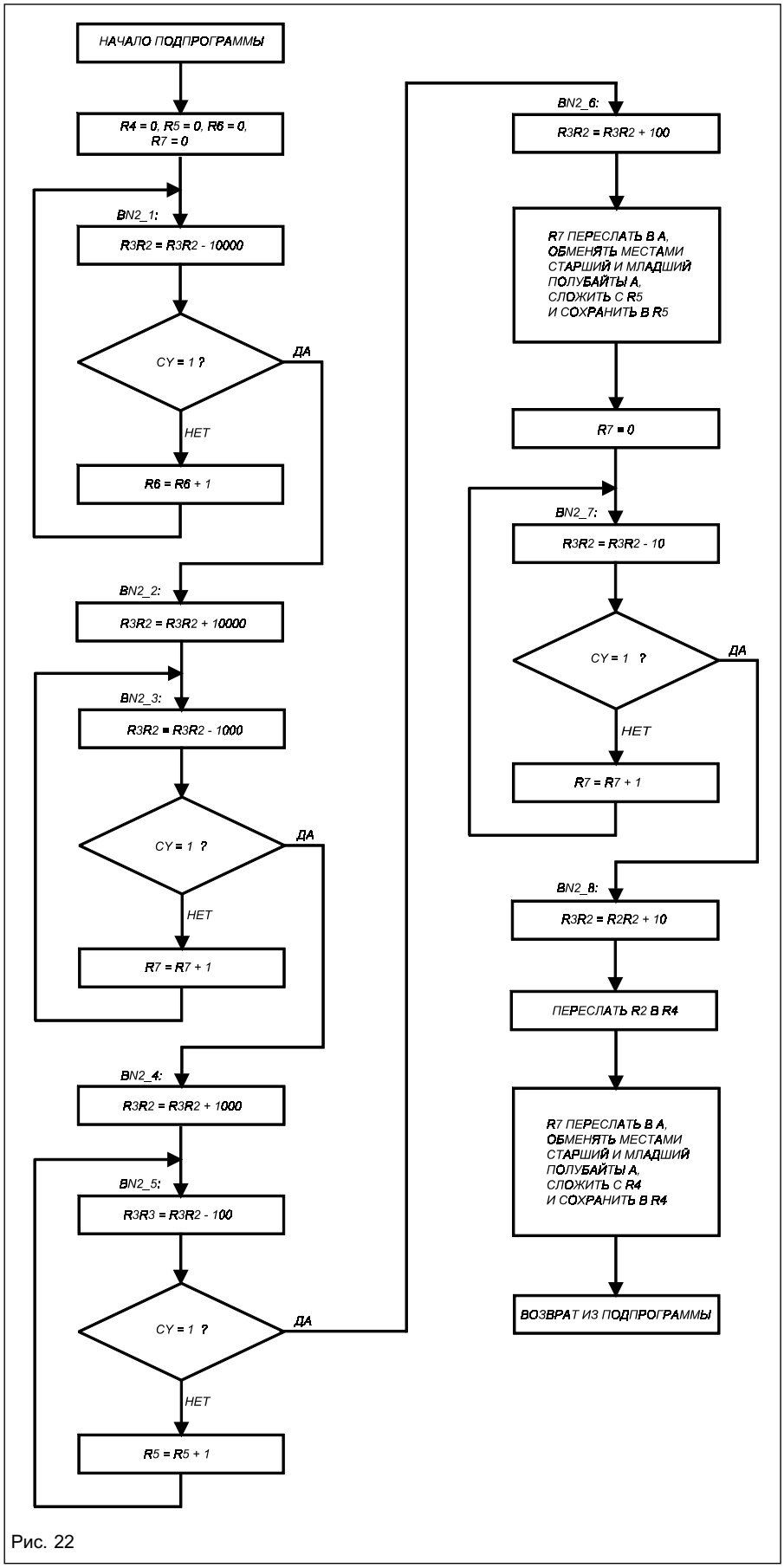

Рис. 22

В-четвертых, когда стрелка, показывающая переход к следующей команде, указывает не на ближайший снизу ромб или прямоугольник блок-схемы, это означает, что следующей должна выполняться не нижеидущая за текущей команда, а другая, расположенная несколькими командами выше или ниже. Для правильного перехода на эту (расположенную выше или ниже) команду она должна быть снабжена меткой (до 8 букв и цифр с двоеточием на конце). Эти метки я также внес в блок-схему. По стандартным прави-

лам делать это не обязательно, но практика показывает, что если не пренебрегать этим, то количество ошибок, возникающих при написании программы, заметно сокращается.

Ну а теперь — текст самой программы с соответствующими комментариями.

```
;      ПРОГРАММА, ДЕМОНСТРИРУЮЩАЯ РАБОТУ
;      ПОДПРОГРАММЫ ПРЕОБРАЗОВАНИЯ ДВУХБАЙТОВЫХ
;      ЧИСЕЛ ИЗ ШЕСТНАДЦАТИРИЧНОГО В ДВОИЧНО-ДЕ-
СЯТИЧНЫЙ
;      ФОРМАТ
;                              *****
;
R7      .EQU   7     ;АДРЕСА РЕГИСТРОВ R0-R7
R6      .EQU   6
R5      .EQU   5
R4      .EQU   4
R3      .EQU   3
R2      .EQU   2
R1      .EQU   1
R0      .EQU   0
ACC      .EQU   0E0H   ;АДРЕС АККУМУЛЯТОРА
B       .EQU   0F0H   ;АДРЕС РЕГИСТРА B
PSW     .EQU   0D0H   ;АДРЕС РЕГИСТРА (СЛОВА) СОСТО-
ЯНИЯ
SP      .EQU   81H    ;АДРЕС УКАЗАТЕЛЯ СТЕКА
DPL     .EQU   82H    ;АДРЕС МЛАДШЕЙ ПОЛОВИНЫ DPTR
DPH      .EQU   83H    ;АДРЕС СТАРШЕЙ ПОЛОВИНЫ DPTR
P0      .EQU   80H    ;АДРЕС РЕГИСТРА ПОРТА P0
P1      .EQU   90H    ;АДРЕС РЕГИСТРА ПОРТА P1
P2      .EQU   0A0H   ;АДРЕС РЕГИСТРА ПОРТА P2
P3      .EQU   0B0H   ;АДРЕС РЕГИСТРА ПОРТА P3
B.0     .EQU   0F0H   ;АДРЕСА ОТДЕЛЬНЫХ БИТОВ РЕГИ-
СТРА B
B.1     .EQU   0F1H
B.2     .EQU   0F2H
B.3     .EQU   0F3H
B.4     .EQU   0F4H
B.5     .EQU   0F5H
B.6     .EQU   0F6H
B.7     .EQU   0F7H
ACC.0        .EQU    0E0H    ;АДРЕСА ОТДЕЛЬНЫХ БИТОВ
АККУМУЛЯТОРА
ACC.1       .EQU   0E1H
ACC.2       .EQU   0E2H
ACC.3       .EQU   0E3H
ACC.4       .EQU   0E4H
ACC.5       .EQU   0E5H
ACC.6       .EQU   0E6H
ACC.7       .EQU   0E7H
PSW.0        .EQU    0D0H    ;АДРЕСА ОТДЕЛЬНЫХ БИТОВ
РЕГИСТРА PSW
PSW.1       .EQU   0D1H
PSW.2       .EQU   0D2H
PSW.3       .EQU   0D3H
PSW.4       .EQU   0D4H
PSW.5       .EQU   0D5H
PSW.6       .EQU   0D6H
PSW.7       .EQU   0D7H
P0.0      .EQU   080H   ;АДРЕСА ОТДЕЛЬНЫХ ЛИНИЙ ПОРТА
P0
P0.1      .EQU   081H
P0.2      .EQU   082H
P0.3      .EQU   083H
P0.4      .EQU   084H
P0.5      .EQU   085H
P0.6      .EQU   086H
P0.7      .EQU   087H
P1.0      .EQU   090H   ;АДРЕСА ОТДЕЛЬНЫХ ЛИНИЙ ПОРТА
P1
P1.1      .EQU   091H
P1.2      .EQU   092H
P1.3      .EQU   093H
P1.4      .EQU   094H
P1.5      .EQU   095H
P1.6      .EQU   096H
P1.7      .EQU   097H
P2.0      .EQU   0A0H   ;АДРЕСА ОТДЕЛЬНЫХ ЛИНИЙ ПОРТА
P2
P2.1      .EQU   0A1H
P2.2      .EQU   0A2H
P2.3      .EQU   0A3H
P2.4      .EQU   0A4H
P2.5      .EQU   0A5H
P2.6      .EQU   0A6H
P2.7      .EQU   0A7H
P3.0      .EQU   0B0H   ;АДРЕСА ОТДЕЛЬНЫХ ЛИНИЙ ПОРТА
P3
P3.1      .EQU   0B1H
P3.2      .EQU   0B2H
P3.3      .EQU   0B3H
P3.4      .EQU   0B4H
P3.5      .EQU   0B5H
P3.6      .EQU   0B6H
P3.7      .EQU   0B7H
;
;
    .ORG 0          ;НИЖЕСЛЕДУЮЩАЯ КОМАНДА С АДРЕ-
СА 0
;
    LJMP   START       ;НА КОМАНДУ ПОСЛЕ МЕТКИ START
;
     .ORG   100H          ;НИЖЕСЛЕДУЮЩАЯ КОМАНДА С
АДРЕСА 100H
;
START:                ;ПРОГРАММА, ЗАГРУЖАЮЩАЯ ДВУХ-
БАЙТОВОЕ
    MOV   DPTR,#0FF00H   ;ЧИСЛО В R3 И R2 И ВЫЗЫВАЮ-
ЩАЯ
    MOV   R2,DPL       ;ЗАТЕМ ПОДПРОГРАММУ ПРЕОБРА-
ЗОВАНИЯ
    MOV   R3,DPH       ;ЭТОГО ЧИСЛА В ДВОИЧНО-ДЕСЯ-
ТИЧНЫЙ
    LCALL  BN2BCD      ;ФОРМАТ
    SJMP   START
;
BN2BCD:               ;ПРЕОБРАЗУЕМОЕ ЧИСЛО В R3R2
                ;(МЛ.БАЙТ В R2)
    MOV   R4,#0        ;НАЧАЛЬНАЯ УСТАНОВКА
    MOV   R5,#0
    MOV   R6,#0
    MOV   R7,#0
;
    MOV   DPTR,#10000
BN2_1:
    LCALL  R32MNDPT     ;R3R2 - 10000
    JC    BN2_2        ;ПЕР. НА BN2_2, ЕСЛИ CY=1
    INC   R6          ;R6=R6 + 1 - ЭТО ДЕЛАЕМ, ЕСЛИ CY=0
    SJMP   BN2_1        ;ПЕРЕХОД НА BN2_1
BN2_2:
    LCALL  R32PLDPT     ;R3R2 + 10000
;
    MOV   DPTR,#1000
BN2_3:
    LCALL  R32MNDPT     ;R3R2 - 1000
    JC    BN2_4        ;ПЕР. НА BN2_4, ЕСЛИ CY=1
    INC   R7          ;R7=R7 + 1 - ЭТО ДЕЛАЕМ, ЕСЛИ CY=0
    SJMP   BN2_3        ;ПЕРЕХОД НА BN2_3
BN2_4:
    LCALL  R32PLDPT     ;R3R2 + 1000
;
    MOV   DPTR,#100
BN2_5:
    LCALL  R32MNDPT     ;R3R2 - 100
    JC    BN2_6        ;ПЕР. НА BN2_6, ЕСЛИ CY=1
    INC   R5          ;R5=R5 + 1 - ЭТО ДЕЛАЕМ, ЕСЛИ CY=0
    SJMP   BN2_5        ;ПЕРЕХОД НА BN2_5
BN2_6:
    LCALL  R32PLDPT
;
    MOV   A,R7         ;ПЕРЕСЛАЛИ ИЗ R7 В A
     SWAP   A           ;ОБМЕНЯЛИ МЕСТАМИ СТАРШИЙ И
МЛАДШИЙ
                ;ПОЛУБАЙТЫ A
    ADD   A,R5         ;ПРИБАВИЛИ К A R5
    MOV   R5,A         ;ПЕРЕСЛАЛИ СУММУ В R5
    MOV   R7,#0        ;R7 = 0
;
    MOV   DPTR,#10
BN2_7:
    LCALL  R32MNDPT     ;R3R2 - 10
    JC    BN2_8        ;ПЕР. НА BN2_8, ЕСЛИ CY=1
    INC   R7          ;R7=R7 + 1 - ЭТО ДЕЛАЕМ, ЕСЛИ CY=0
    SJMP   BN2_7        ;ПЕРЕХОД НА BN2_7
BN2_8:
    LCALL  R32PLDPT     ;R3R2 + 10
;
    MOV   A,R2
    MOV   R4,A         ;R2 ПЕРЕСЛАЛИ ЧЕРЕЗ АККУМУЛЯТОР
В R4
```

```
;
      MOV    A,R7          ;ПЕРЕСЛАЛИ ИЗ R7 В A
       SWAP   A             ;ОБМЕНЯЛИ МЕСТАМИ СТАРШИЙ И
МЛАДШИЙ
                    ;ПОЛУБАЙТЫ A
      ADD    A,R4          ;ПРИБАВИЛИ К A R4
      MOV    R4,A          ;ПЕРЕСЛАЛИ СУММУ В R4
      RET                  ;ВОЗВРАТ ИЗ ПОДПРОГРАММЫ
;
;ПОДПРОГРАММЫ ДЛЯ BN2BCD
;
R32PLDPT:
      MOV    A,R2          ;R2 В АККУМУЛЯТОР
       ADD    A,DPL         ;СКЛАДЫВАЕМ ЕГО С МЛ.БАЙТОМ
DPTR
      MOV    R2,A          ;ВОЗВРАЩАЕМ СУММУ В R2
      MOV    A,R3          ;R3 В АККУМУЛЯТОР
      ADDC   A,DPH         ;СКЛАДЫВАЕМ ЕГО С БИТОМ ПЕРЕ-
НОСА
                    ;И С МЛ.БАЙТОМ DPTR
      MOV    R3,A          ;ВОЗВРАЩАЕМ СУММУ В R3
      RET
;
R32MNDPT:
      CLR    C             ;ОЧИЩАЕМ CY
      MOV    A,R2          ;R2 В АККУМУЛЯТОР
      SUBB   A,DPL         ;ВЫЧИТАЕМ ИЗ НЕГО МЛ.БАЙТ DPTR
И CY
      MOV    R2,A          ;ВОЗВРАЩАЕМ РАЗНОСТЬ В R2
      MOV    A,R3          ;R3 В АККУМУЛЯТОР
      SUBB   A,DPH         ;ВЫЧМТАЕМ ИЗ НЕГО БИТ ПЕРЕНОСА
                    ;И СТ.БАЙТ DPTR
      MOV    R3,A          ;ВОЗВРАЩАЕМ РАЗНОСТЬ В R3
      RET
;
;
.END
```

Долго комментировать текст программы после обсуждения блок-схемы программы нет необходимости. Собственно программа осуществляет занесение в регистры R2 и R3 числа 65280D = 0FF00H и вызов подпрограммы преобразования BN2BCD. Последняя написана в соответствии с блок-схемой на рис. 22 и комментариями к ней.

Теперь о новых для вас командах, с которыми вы столкнулись при знакомстве с текстом подпрограммы. Самая интересная из них — JC (JC BN2_2, JC BN2_4, JC BN2_6 и JC BN2_8). Это так называемая команда условного перехода. В ходе ее выполнения МК проверяет состояние флага переноса CY, и если он установлен (т. е. CY=1), то МК переходит к выполнению команды, отмеченной меткой, имя которой идет вслед за JC (соответственно BN2_2, BN2_4, BN2_6 или BN2_8). Естественно, если флаг переноса сброшен, то никакого перехода нет, и выполняется команда, идущая вслед за JC xxx (xxx — помещенная в скобки в предыдущем предложении метка адреса перехода).

Кстати, команд условного перехода довольно много: переход, если флаг CY установлен (JC xxx), если флаг CY сброшен (JNC xxx), если содержимое аккумулятора равно 0 (JZ xxx), если оно не равно 0 (JNZ xxx) и ряд других. Их разумное использование позволяет организовывать простые, но в то же время весьма эффективные алгоритмы.

INC Rn (n=0-7) — команда инкрементирования (т. е. увеличения на 1) содержимого соответствующего регистра. Она проста и удобна, мы будем пользоваться ей довольно часто.

SWAP A — команда обмена местами старших и младших 4 бит аккумулятора. В основном она используется для того, чтобы собрать в аккумуляторе содержимое двух регистров — одного в младшей половине аккумулятора, другого — в старшей. Для этого содержимое одного регистра пересылается в аккумулятор, старший и младший полубайты меняются местами, а затем к содержимому аккумулятора прибавляется содержимое второго регистра. Если старшие 4 бита обоих регистров изначально были нулями, то после описанной операции в старших 4 битах аккумулятора будет храниться содержимое первого из упомянутых регистров, а в младших 4 битах — второго.

Следующие две команды часто употребляются — это сложение аккумулятора с одним из регистров и вычитание из аккумулятора содержимого регистра. Команда ADD A,reg (reg — любой регистр, в данном случае R5: ADD A,R5) вынуждает МК сложить с аккумулятором содержимое R5 и результат оставить в аккумуляторе. Еще одна разновидность команды сложения — ADDC A,reg (reg — опять-таки любой регистр, в данном случае DPH: ADDC A,DPH) — складывает с аккумулятором содержимое старшей половины регистра DPTR (8-разрядного регистра DPH) и вдобавок прибавляет к нему бит переноса. (Кстати, младшая половина DPTR, также 8-разрядная, называется DPL.)

Рассмотренные выше две команды (ADD и ADDC) складывают 8-разрядные числа (от 00000000B до 11111111B). А как быть, если нужно сложить, например, два шестнадцатиразрядных числа? С этим мы познакомимся на примере подпрограммы R32PLDPT. Она складывает два шестнадцатиразрядных числа, одно из которых хранится в регистрах R2 (младшие биты) и R3 (старшие), а другое — в DPTR. Вначале содержимое R2 пересылается в аккумулятор, складывается с DPL и возвращается обратно в R2. Затем то же самое совершается с регистрами R3 и DPH, но используемая команда ADDC складывает с содержимым регистров еще и бит переноса.

Для чего при втором сложении нужно прибавлять бит переноса? Вспомните, как нас учили в школе складывать «в столбик». Записываем одно число под другим, и вначале складываем межу собой две правые цифры (т. е. младшие разряды, единицы). Если их сумма больше 10, то «единичка идет на ум», и при сложении следующих двух цифр (вторых справа, т. е. десятков) к ним нужно прибавить еще и эту единицу.

Собственно, команда ADDC именно это и делает — если при первом сложении сумма оказалась больше 16 (т. е. больше, чем 10H), то в аккумуляторе окажется число, на которое сумма превышает 16, и установится флаг переноса — та самая единичка. Поэтому когда мы осуществим при помощи команды ADDC сложение следующих цифр, эта единичка учтется правильным образом. Кстати, если сумма младших двух цифр будет меньше 16, флаг переноса не установится, и при следующем сложении с помощью команды ADDC никакой единицы к ним не прибавится. Таким образом, команда ADDC специально предназначена именно для этого случая — сложения второй, третьей и т. д. пары байт в многобайтных числах.

Подведем небольшой итог. Когда мы складываем многобайтные числа длиной два, три и более байт, то первое сложение нужно осуществлять при помощи команды ADD, которая не принимает во внимание никаких переносов, а все последующие — с помощью команды ADDC. Подпрограмма R32PLDPT является примером такого сложения многобайтных (здесь двухбайтных) чисел.

Аналогична и подпрограмма вычитания одного двухбайтового числа из другого (R32MNDPT), но вместо команд сложения в нее, естественно, входят команды вычитания. Здесь также есть своя хитрость. Вспомним опять школьные годы, вычитание «в столбик». По-прежнему первыми вычитаются самые правые, младшие разряды. При этом, если нам нужно вычесть из меньшего числа большее, мы «занимаем единичку» (осуществляем «заем») от более старшего разряда числа, из которого вычитаем (т. е. из той цифры уменьшаемого, которая стоит левее используемой в текущий момент). А далее, когда мы будем осуществлять вычитание следующей цифры из той цифры уменьшаемого, где мы что-то заняли, последнюю сначала нужно уменьшить на 1, а затем что-то из нее вычитать.

Для осуществления описанных действий в МК x51 есть команда вычитания с заемом SUBB. Она вначале вычитает из уменьшаемого содержимое бита переноса (т. е. 1, если флаг CY установлен, и 0, если сброшен; кстати, обратите внимание на то, что в этом случае бит переноса фактически является битом заема), а затем — и само вычитаемое. Собственно, именно это нам и нужно для организации программ вычитания. Но здесь есть один нюанс: нужно помнить, что вычитание младших цифр — первое, ему не предшествовал никакой заем, и для получения правильного результата нужно либо использовать команду вычитания без заема (та-

кие есть у многих МК, но не у x51), либо использовать команду вычитания с заемом SUBB, но перед этим принудительно установить в 0 флаг CY. Именно это и делается при помощи знакомой вам команды CLR C.

Вернемся теперь к самой программе. Я рекомендую загрузить ее в симулятор EMF52L и осуществить ее выполнение по шагам с внимательным анализом чисел в регистрах и состояния флага CY. Если вы не поленитесь и благополучно справитесь с этим заданием, то вы заложите прочный фундамент в формирование у вас навыков написания программ. А для закрепления этого первого навыка я рекомендую вам написать по образу и подобию с рассмотренной подпрограмму преобразования в двоично-десятичный формат трехбайтового числа (т. е. лежащего в диапазоне от 000000H до 0FFFFFFH). При этом преобразуемое число должно изначально находиться в R3R2R1, а результат после преобразования — в R7R6R5R4.

Краткие выводы

Итак, уважаемые читатели, мы познакомились с главными регистрами микроконтроллеров семейства x51. Параллельно с этим мы рассмотрели, как наш МК сопрягается с внешней памятью данных, какие микросхемы памяти для этого наиболее применимы, и какими командами осуществляется обмен данными с ней. А для закрепления знакомства с регистрами мы с вами рассмотрели программу преобразования двухбайтового числа в двоично-десятичный формат. В ней использованы почти все регистры, рассмотренные в этой главе. Кстати, обратите внимание на то, что при написании этой программы я попытался все используемые ей данные размещать исключительно в регистрах, а не в памяти, что, как упоминалось выше, необходимо для быстрого ее выполнения микроконтроллером. В данном случае это, как видите, удалось.

Также, как и в предыдущих главах, мы с вами рассмотрели фрагменты программ и познакомились с еще несколькими командами МК. Помимо этого, вы впервые столкнулись с блок-схемой программы — совершенно неотъемлемым элементом программирования. Если вы научитесь составлять подобные блок-схемы, писать программы для МК для вас не составит труда. Правильно составленная блок-схема — это по сути дела 80…90% программы. Так что прошу вас очень серьезно отнестись к тем нескольким примерам, где я буду довольно подробно описывать составление блок-схем — их в обязательном порядке надо научиться составлять. Кстати, не так уж это и сложно. Все, кто пишут программы на любом из языков, вполне справляются с этим, так что и вам это, наверняка, по силам — вопрос только в желании и во времени. Поэтому еще раз призываю тех, кто поленился разобраться с предложенным материалом, потратьте еще некоторое количество времени и разберитесь с ним, ибо если этого не сделать, вряд ли вам удастся освоить микроконтроллеры.

И в заключение, еще одно замечание. Обратите внимание на то, какого ограниченного набора команд хватило нам для написания подпрограммы преобразования числа из одного формата в другой. Наш МК не умеет ни делить толком, ни умножать, лишь пересылает данные туда-сюда, что-то суммирует и вычитает, да выполняет условные и безусловные переходы. И этого оказывается достаточно для решения поставленной задачи! Ну не удивительно ли? Более того, далее мы с вами еще не однажды и не дважды убедимся в том, что в основе всех тех умных интеллектуальных машин и приборов, поражающих нас иной раз своими «мозгами», стоят простые и вполне заурядные команды. Ну и конечно мозги (уже без кавычек) программистов, придумавших эти поражающие нас алгоритмы и реализовавших их в своих программах.

Александр Фрунзе
alex.fru@dian.ru

Продолжение следует