Приведенный ниже материал опубликован в трех статьях:

Черномырдин А. Автоматное программирование для микроконтроллеров //Радиолюбитель. 2005. № 8, с. 45 – 49.

Черномырдин А. Автоматное программирование для микроконтроллеров //Радиолюбитель. 2005. № 9, с. 44 – 49.

Черномырдин А. Автоматное программирование для микроконтроллеров //Радиолюбитель. 2005. № 10, с. 43 – 47.

Журнал издается в Минске. http://www.radioliga.com/

Александр Черномырдин

http://chav1961.narod.ru

АВТОМАТНОЕ ПРОГРАММИРОВАНИЕ ДЛЯ МИКРОКОНТРОЛЛЕРОВ. SWITCH-ТЕХНОЛОГИЯ И ИНТЕРПРЕТАТОРЫ

Лучше день потерять, а потом за пять минут долететь!
Мультфильм «Крылья, ноги и хвосты»

В данной статье речь пойдет об одной из наиболее эффективных техник программирования, которая успешно применяется при работе с микроконтроллерами – автоматном программировании [1]. Знатоков вновь просят не сетовать на терминологию и упрощения.

Начать необходимо с простого вопроса – «а зачем это нужно?». Написав совместно с автором свою первую программу, читатель, вероятно, уже почувствовал, что по большому счету ничего страшного в программировании нет. Всего-то и надо – выучил, какой бит в каком регистре взводить, и пиши себе на здоровье. Радиолюбитель, в конце концов, не какой-то-там программист...

Хочется сразу развеять иллюзии относительно легкости программирования: просто программировать, безусловно, легко (для этого даже особого ума не надо), а вот программировать легко и просто – гораздо труднее. Но в программировании, в отличие от многих других областей деятельности, можно использовать в работе богатый опыт своих предшественников. Программирование – уже достаточно «старая» наука, в ней накоплен солидный материал по различным приемам программирования, и материал этот необходимо знать, чтобы всякий раз не «изобретать велосипед» и не проливать реки пота и крови над каждой конструкцией. Одним из таких приемов (вернее сказать – техник) программирования и является автоматное программирование.

Сначала – немного математики. Автоматом (если говорить точнее – автоматом Мили) в математике называется семерка следующего вида:

$$\{T, N, Z, N_0, N_k, \Omega, \Lambda\},$$

где

T – множество терминальных символов автомата;
N – множество нетерминальных символов автомата;
Z – множество выходных символов автомата;
$N_0 \in N$ – начальное состояние автомата;
$N_k \subset N$ – множество заключительных состояний автомата;
Ω – множество функций перехода, представляющее собой тройки вида $\{N_i, T_i, N_{i+1}\}$;
Λ – множество функций выхода, представляющее собой тройки вида $\{ N_i, T_i, X_{i+1}\}$.

Помимо этой «постоянной части», в любом автомате есть еще и переменная часть – текущее состояние автомата. В начале работы автомата текущее его состояние совпадает с начальным N_0, а затем, под воздействием терминальных символов оно изменяется в соответствии с правилами, заданными во множестве Ω. При этом, в свою очередь, автомат сам воздействует на «окружающую среду» в соответствии с правилами, заданными во множестве Λ. Для описания автомата существует много различных способов, из которых, по мнению автора, наиболее наглядным является граф переходов.

1. Алгоритмизация

Простейший пример автомата, который наверняка попадался на глаза любому радиолюбителю – обыкновенный торшер. Если отвлечься от деталей, торшер представляет собой лампочку, которая может гореть или не гореть, и веревочку, дергая за которую мы заставляем лампочку включаться или выключаться. Граф переходов автомата «торшер» изображен на рис.1.

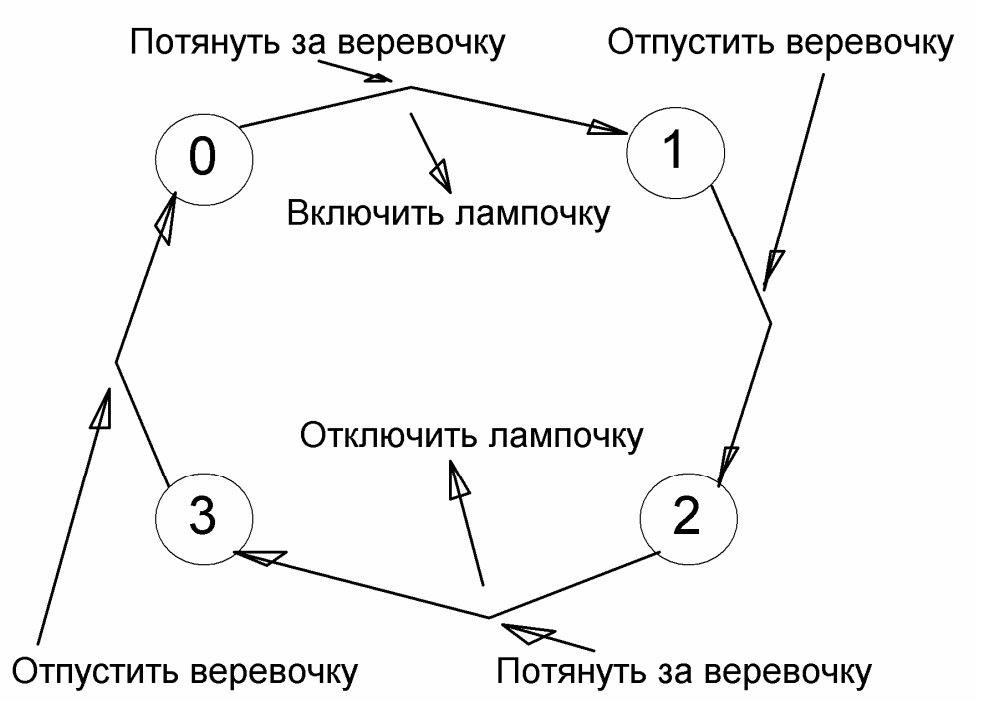

Рис. 1. Граф переходов автомата «торшер»

Ну, а теперь – что обозначают в данном случае все эти T, N и Z на понятном обычному человеку языке:

1. Множество терминальных символов автомата «торшер» – это все внешние воздействия, которые могут быть оказаны на автомат. В данном случае таких внешних воздействий два: {«Потянуть за веревочку», «Отпустить веревочку»}.
2. Множество нетерминальных символов автомата – это все узлы графа переходов. В данном случае их четыре: {0, 1, 2, 3}.
3. Множество выходных символов автомата – это все полезные действия, которые производит автомат. В данном случае таких полезных действий два: {«Включить лампочку», «Отключить лампочку»}.
4. Начальное состояние автомата N_0 в данном случае нулевое. Можно считать, что в этом состоянии автомат находится при включении и сбросе микроконтроллера.
5. Множество заключительных состояний автомата N_k в данном случае пусто: {∅}. Мы можем, однако, сделать его непустым, если введем во множество нетерминальных символов еще один узел, в который автомат попадал бы по терминальному символу «Лампочка сгорела» или «Веревочка оборвалась».
6. Множество функций перехода автомата приведено в табл. 1.

Таблица 1

Функция перехода	Комментарий
{0, «Потянуть за веревочку», 1}	Если автомат находился в состоянии 0, то при получении сигнала «Потянуть за веревочку» он переходит в состояние 1
{1, «Отпустить веревочку», 2}	Если автомат находился в состоянии 1, то при получении сигнала «Отпустить веревочку» он переходит в состояние 2
{2, «Потянуть за веревочку», 3}	Если автомат находился в состоянии 2, то при получении сигнала «Потянуть за веревочку» он переходит в состояние 3
{3, «Отпустить веревочку», 0}	Если автомат находился в состоянии 3, то при получении сигнала «Отпустить веревочку» он переходит в состояние 0

1. Множество функций выхода автомата приведено в табл. 2.

Таблица 2

Функция выхода	Комментарий
{0, «Потянуть за веревочку», «Включить лампочку»}	Если автомат находился в состоянии 0, то при получении сигнала «Потянуть за веревочку» необходимо «Включить лампочку»
{2, «Потянуть за веревочку», «Отключить лампочку»}	Если автомат находился в состоянии 2, то при получении сигнала «Потянуть за веревочку» необходимо «Отключить лампочку»

Очень часто из соображений простоты реализации множества Ω и Λ объединяют. Для этого во множество выходных символов Z вводят еще один выходной символ **«ничего не делать»**, а объединенное множество представляют в виде четверок $\{N_i, T_i, N_{i+1}, Z_{i+1}\}$. При этом автомат «торшер» будет задан следующим образом[1]:

{0, «Потянуть за веревочку», 1, «Включить лампочку»}
{1, «Отпустить веревочку», 2, «Ничего не делать»}
{2, «Потянуть за веревочку», 3, «Отключить лампочку»}
{3, «Отпустить веревочку», 0, «Ничего не делать»}

Следует отметить, что в математике и в прикладном программировании автоматы используются для совершенно различных целей: в математике с помощью автомата определяется допустимость последовательности терминальных символов, а в программировании – задается логика работы программы. Сказывается это отличие на обработке «неожиданных» терминальных символов. Например, если вслед за символом «Потянуть за веревочку» снова получен символ «Потянуть за веревочку» вместо символа «Отпустить веревочку» (как это может случиться – вопрос десятый), то автомат с точки зрения математики просто «отбраковал» бы **всю** последовательность. В прикладном программировании все «неожиданные» терминальные символы автоматом просто игнорируются.

Итак, попытаемся реализовать «торшер» на микроконтроллере КР1878ВЕ1. Схема «торшера» приведена на рис.2. «Лампочкой» будет служить светодиод, подсоединенный на линию 0 порта B. Кнопка, заменяющая «веревочку», подсоединена на линию 0 порта A. Будем считать, что процессор микроконтроллера работает от внутренней RC-цепочки с частотой около 50 кГц.

В программировании для реализации конечных автоматов наиболее часто используются два подхода:

1. Автомат «прошивается» непосредственно в самой программе. В литературе для такого способа реализации в последнее время установился термин «`switch`-технология» [1] (такое название происходит из-за того, что реализация автомата на языках программирования высокого уровня, например *C* или *Pascal*, обычно выполняется по этому способу с помощью оператора выбора `switch`).

[1] Все это так, но целесообразнее сначала выполнить действие, а потом переходить в новое состояние (прим. ред. А. Шалыто).

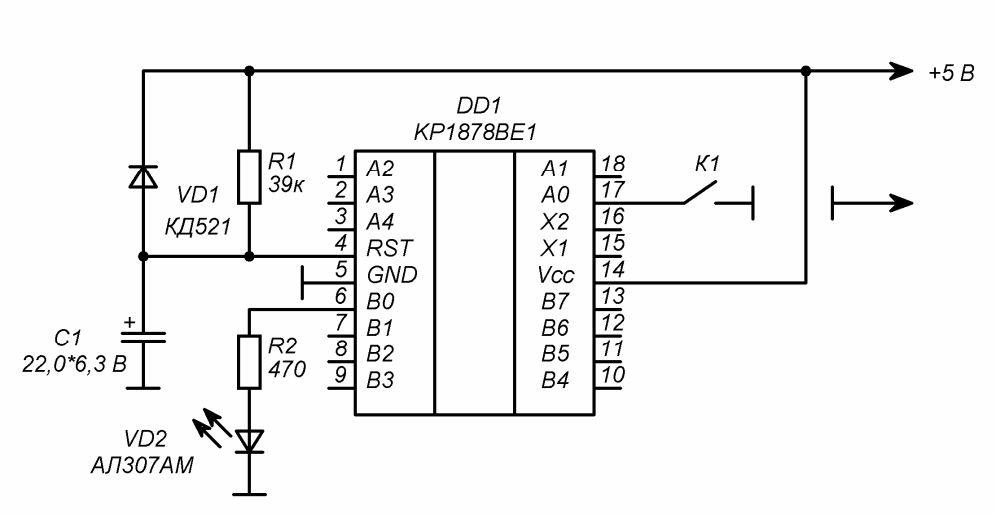

Рис. 2. Схема «торшера»

2. В программе хранится таблица описания автомата, а программа, определенным образом обрабатывая ее, моделирует поведение автомата в соответствии с правилами, заложенными в этой таблице (такие программы обозначаются в программистской литературе общим термином **интерпретаторы**).

Попытаемся в учебных целях сделать наш «торшер» тем и другим способом. Прежде чем начинать писать программу, всегда полезно сначала написать и проработать ее алгоритм. Для представления алгоритма существует множество способов (начиная от блок-схем), но мы воспользуемся для этого простейшим способом – языком, близким к естественному, который, на взгляд автора, должен быть легко понятен читателю. Реализация нашего автомата, выполняемая по графу переходов или по набору «четверок», по `switch`-технологии может быть описана следующим образом:

```
Пусть состояние_автомата = 0
выбор (состояние_автомата)
        {когда равно 0:
                если «Потянуть за веревочку» то
                        {пусть состояние_автомата = 1
                         «Включить лампочку»
                        }
         когда равно 1:
                если «Отпустить веревочку» то
                        {пусть состояние_автомата = 2
                        }
         когда равно 2:
                если «Потянуть за веревочку» то
                        {пусть состояние_автомата = 3
                         «Отключить лампочку»
                        }
         когда равно 3:
                если «Отпустить веревочку» то
                        {пусть состояние_автомата = 0
                        }
        }
повторить сначала с выбора
```

Для описания второго способа реализации (интерпретатора автомата) предположим, что в программе имеется таблица табл. 3.

Таблица 3

Номер текущего состояния автомата	**Ожидаемый терминальный символ**	**Номер нового состояния автомата**	**Действие которое надо выполнить**
0	«Потянуть за веревочку»	**1**	«Включить лампочку»
1	«Отпустить веревочку»	**2**	«Ничего не делать»
2	«Потянуть за веревочку»	**3**	«Отключить лампочку»
3	«Отпустить веревочку»	**0**	«Ничего не делать»

В этом случае интерпретатор автомата реализуется следующим образом:

Пусть состояние_автомата = 0
взять первую строку таблицы
если состояние_автомата совпадает с *номером_текущего_состояния_автомата* из строки таблицы И пришедший терминальный символ совпадает с *ожидаемым_терминальным_символом* из строки таблицы то
{пусть состояние_автомата = *номер_нового_состояния_автомата* из строки таблицы
выполнить *действие_которое_надо_выполнить*, записанное в строке таблицы
начать все сначала с первой строки таблицы
}
иначе взять следующую строку таблицы и повторить обработку
когда таблица закончится, начать перебор снова с первой строки таблицы

2. Реализация автомата на основе `switch`-технологии.

Первый фрагмент программы – инициализация портов A и B:

```
Start:  jmp     begin           ; Переход к началу программы.
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
begin:  ldr     #D,18h          ; Сегмент D – управляющие регистры портов.
        movl    %d1,00011011b   ; Доступ к подрегистру 3 в режиме автоинкремента
        movl    %d1,00000000b   ; Подрегистр 3: все линии - вводные.
        movl    %d1,00000000b   ; Подрегистр 4: режим вывода безразличен.
        movl    %d1,00000001b   ; Подрегистр 5: резистор A[0] подключен.
        movl    %d1,00000000b   ; Подрегистр 6: прерывания нас не интересуют
        movl    %d1,00000000b   ; Подрегистр 7: прерывания нас не интересуют
        movl    %d2,00011011b   ; Доступ к подрегистру 3 в режиме автоинкремента
        movl    %d2,00000001b   ; Подрегистр 3: включить линию B[0] на вывод.
        movl    %d2,00000001b   ; Подрегистр 4: режим вывода – «полноценный».
        movl    %d2,00000000b   ; Подрегистр 5: резисторы нагрузки нам не нужны.
        movl    %d2,00000000b   ; Подрегистр 6: прерывания нас не интересуют
        movl    %d2,00000000b   ; Подрегистр 7: прерывания нас не интересуют
```

Реализация собственно автомата:

```
        ldr     #A,40h          ; <1> Сегмент A – ячейки ОЗУ для работы с автоматом
        ldr     #B,00h          ; Сегмент B – адреса рабочих регистров портов и таймера
        movl    %a0,0           ; В ячейке %a0 будем хранить номер текущего состояния
                                ; автомата. В начале работы он равен нулю
        movl    %b2,0           ; Выключить лампочку (исходное состояние)
again:  mial    %a1,labels      ; <2> В рабочую ячейку %a1 загружаем адрес таблицы
        miah    %a2,labels      ; переходов  (младший байт), в %a2 – старший байт
        add     %a1,%a0         ; <3> Складываем адрес перехода с номером состояния
        adc     %a2             ; Распространяем перенос
        mtpr    #6,%a1          ; <4> Загружаем вычисленный адрес в IR1
        mtpr    #7,%a2
        ijmp                    ; Переходим по вычисленному адресу
labels: jmp     cond_0          ; Таблица перехода на участки, соответствующие
        jmp     cond_1          ; состояниям автомата
        jmp     cond_2
        jmp     cond_3
```

Этот маленький участок программы требует множества пояснений:

1. Поскольку у любого автомата помимо «постоянной части» (описания) есть еще и «переменная часть» (номер текущего состояния), то этот номер необходимо где-то хранить. Воспользуемся для этого ячейкой ОЗУ по адресу 40h. Поэтому с помощью команды **ldr** загрузмм сегментный регистр **#A** адресом 40h – с этого адреса начинается в микроконтроллере ОЗУ. И сразу же – обратите внимание! – ячейку **%a0** необходимо заполнить числом 0 – это начальное состояние N_0 нашего автомата.
2. Далее (начиная с метки again) в программе появляется две странных команды **mial** и **miah**, описания которых нет в системе команд процессора. Это – на самом деле команды **movl**, и записывают они в ячейки **%a1** и **%a2** адрес команды **jmp cond_0**, помеченный в программе меткой label. Адрес команды, помеченной меткой label, можно было бы подсчитать и вручную (посмотреть, за сколько команд она расположена от начала программы), а затем занести в ячейку **%a1** младший байт подсчитанного адреса, а в ячейку **%a2** – старший байт адреса (напоминаем читателю, что все адреса команд программы занимают два байта, потому что емкость памяти программ – 1024 слова, а в одном байте можно представить число величиной не более 255), но компилятор tessa избавляет программиста от такой дурной работы: он сам подсчитает адрес команды, помеченной меткой label, и заменит команду **mial** на команду **movl** с младшим байтом адреса, а команды **miah** – на команду **movl** со старшим байтом адреса. Итак, в результате выполнения команд **mial** и **miah** в ячейках **%a1** и **%a2** окажется адрес команды, помеченной меткой label.
3. Далее следуют команды сложения только что загруженного адреса команды с номером текущего состояния автомата. Обратите внимание – здесь нам впервые пришлось столкнуться с выполнением арифметических операций над **многобайтными** числами. Для сложения многобайтных чисел существует общее правило – сначала сложить с помощью команды **add** младшие байты двух чисел (в данном случае с младшим байтом адреса команды, помеченной меткой label, находящимся в ячейке **%a1**, складывается номер текущего состояния автомата, находящийся в ячейке **%a0**), а затем с помощью команды **adc** распространить флаг переноса на все остальные байты многобайтного числа (вспомните – аналогичным образом устроены все многодекадные счетчики: в них сигнал переноса от младшего счетчика подключен ко входу следующего, более старшего счетчика). Таким образом, после выполнения команд **add** и **adc** в ячейках **%a1** и **%a2** окажется новый адрес: если текущее состояние автомата было нулевым – это по-прежнему будет адрес все той же команды, помеченный меткой label, если текущее состояние было единичным – это будет адрес следующей за ней команды **jmp cond_1**, если состояние было равно двум – адрес команды **jmp cond_2** и т.д.
4. И, наконец, вычисленный таким образом адрес загружается в регистр косвенной адресации IR1 (напоминаем читателю – этот регистр образует регистровая пара, состоящая из регистров **#6** и **#7**), причем младший байт адреса загружается в регистр **#6**, а старший – в регистр **#7**. Делается это с помощью команд **mtpr**. Далее следует команда косвенного

перехода **ijmp**, которой радиолюбители пользуются чрезвычайно редко. Между тем это очень полезная команда – она позволяет передавать управление в то место программы, которое не известно заранее, а вычисляется в процессе работы программы, как это только что проделали мы. Передача управления при этом производится по тому адресу, который находится в регистре IR1: для случая, когда состояние автомата равно нулю, это будет адрес команды, помеченной меткой label, для случая, когда состояние автомата равно единице – адрес следующей команды **jmp cond_1**, для состояния, равного двойке – адрес команды **jmp cond_2** и т.д. Такая конструкция носит в программистской литературе название **вычисляемого перехода** или **вычисляемого goto** (именно так – **goto**, - обозначается оператор перехода в языках высокого уровня). После выполнения команды **ijmp** управление будет передано соответствующей команде из списка, помеченного меткой label, а уж та команда передаст управление на соответствующий участок программы, обрабатывающий одно из четырех состояний автомата.

Теперь реализуем обработку состояний автомата:

```
cond_0: bttl    %b1,00000001b ; <1> Нажата ли кнопка «торшера» (логический 0)
        jnz     again         ; <2> Пока нет – начать работу автомата сначала
        movl    %a0,1         ; <3> Теперь новое состояние автомата = 1
        btsl    %b2,00000001b ; <4> Включить лампочку.
        jmp     again
cond_1: bttl    %b1,00000001b ; Отпущена ли кнопка «торшера» (логическая 1)
        jz      again         ; Пока нет – начать работу автомата сначала
        movl    %a0,2         ; Теперь новое состояние автомата = 2
        jmp     again
cond_2: bttl    %b1,00000001b ; Нажата ли кнопка «торшера» (логический 0)
        jnz     again         ; Пока нет – начать работу автомата сначала
        movl    %a0,3         ; Теперь новое состояние автомата = 3
        btcl    %b2,00000001b ; Отключить лампочку.
        jmp     again
cond_3: bttl    %b1,00000001b ; Отпущена ли кнопка «торшера» (логическая 1)
        jz      again         ; Пока нет – начать работу автомата сначала
        movl    %a0,0         ; Теперь новое состояние автомата = 0
        jmp     again
        .end
```

Как видите, куски программы получились очень похожи, поэтому разберем подробно только один из них – с меткой cond_0:

1. Вначале проверяем, нажата ли кнопка «торшера» (терминальный символ «Потянуть за веревочку»). Обратите внимание на схему «торшера» – когда кнопка будет нажата, на линии A[0] будет присутствовать сигнал логический 0 («шиворот-навыворот»)! Для проверки бита в системе команд процессора есть команда **bttl** (в системе команд процессора существуют по две отдельных команды: с суффиксом **l** – для проверки битов из младшей тетрады байта, и с суффиксом **h** – для проверки старшей тетрады). В принципе, в команде **btt** можно проверять и несколько битов сразу, но так ею практически никогда не пользуются – по результату ее работы устанавливаются флаги Z и S в регистре состояния, и если проверяемых битов будет несколько, непонятно, о каком из проверяемых битов эти флаги будут говорить.
2. Далее следует команда условного перехода **jnz** на метку `again` – новый цикл работы автомата. Это означает, что ожидаемый терминальный символ «Потянуть на веревочку» еще не пришел на вход автомата (на линии A[0] пока присутствует логическая 1). Поэтому никаких действий по обработке терминального символа пока не требуется.
3. Командой **movl** изменяем номер текущего состояния автомата: 0 → 1. На эту команду попадаем, когда кнопка «торшера» нажата, и, следовательно, предыдущая команда условного перехода была выполнена микроконтроллером (терминальный символ «Потянуть за веревочку» все-таки пришел).
4. И, наконец, выполняем полезное действие, предусмотренное на случай прихода данного терминального символа – включаем лампочку командой **btsl**.

Вот, по сути дела, вся реализация автомата методом `switch`-технологии.

3. Реализация автомата интерпретатором

Теперь реализуем автомат вторым способом. Для этого напишем программу-интерпретатор автоматной таблицы. Начальный участок программы – инициализация портов микроконтроллера – остается прежним. Поэтому он здесь не приводится. Начинаем сразу с реализации интерпретатора:

```
        ldr     #A,40h          ; Сегмент A – ячейки ОЗУ для работы с автоматом
        ldr     #B,00h          ; Сегмент B – адреса рабочих регистров портов и таймера
        movl    %a0,0           ; В ячейке %a0 будем хранить номер текущего состояния
                                ; автомата. В начале работы он равен нулю
        movl    %b2,0           ; Выключить лампочку
again:  ldal    #6,table        ; <1> Загружаем адрес начала таблицы автомата в IR1
        ldah    #7,table
        mov     %a3,%d7         ; <2> Занести в ячейку %a3 число строчек в таблице
                                ; автомата  (%a3 у нас будет переменной цикла)
$3:     cmp     %a0,%d7         ; <3> Номер состояния автомата совпадает с табличным?
        jne     $1              ; Нет – перейти к следующей строке таблицы
        mov     %a1,%d7         ; <4> Занести адрес подпрограммы анализа терминального
        mov     %a2,%d7         ; символа во временные ячейки %a1 и %a2
        push    #6              ; <5> Сохранить IR1 в стеке
        push    #7
        mtpr    #6,%a1          ; <6> Занести адрес подпрограммы проверки терминаль-

        mtpr    #7,%a2          ; ного символа в IR1
        ijsr                    ; <7> Вызвать подпрограмму проверки терминального
                                ; символа
        pop     #7              ; <9> Восстановить IR1 из стека
        pop     #6
        jnc     $1              ; <10> Нет, ожидаемого символа пока нет...
        mov     %a0,%d7         ; <11> Да, символ пришел – изменить состояние автомата
        mov     %a1,%d7         ; <12> Занести адрес подпрограммы выхода во временные
        mov     %a2,%d7         ; ячейки %a1 и %a2
        mtpr    #6,%a1          ; Занести адрес подпрограммы выхода в IR1
        mtpr    #7,%a2
        ijsr                    ; Вызвать подпрограмму выхода.
        jmp     again           ; <13> Начать новый цикл работы автомата
$1:     cmpl    %d7,0           ; <14> Фиктивное сравнение – сдвигаем IR1 на следующий
        cmpl    %d7,0           ; байт
$2:     cmpl    %d7,0           ; Фиктивное сравнение – сдвигаем IR1 на следующий байт
        cmpl    %d7,0
        cmpl    %d7,0
        subl    %a3,1           ; Вычесть счетчик цикла (счетчик строк таблицы)
        jnz     $3              ; Строки еще не закончились – продолжить анализ
        jmp     again           ; Таблица закончена – начать новый цикл анализа
```

Кроме этого, нам потребуется в программе еще и автоматная таблица:

```
table:  .byte   4               ; Число строчек в автоматной таблице
;       Первая строка таблицы:
        .byte   0               ; Текущее состояние = 0
        .word   testDown        ; Подпрограмма проверки факта нажатия кнопки
        .byte   1               ; Новое состояние = 1
        .word   turnOn          ; Подпрограмма включения лампочки
;       Вторая строка таблицы:
        .byte   1               ; Текущее состояние = 1
        .word   testUp          ; Подпрограмма проверки факта отпускания кнопки
        .byte   2               ; Новое состояние = 2
        .word   nothing         ; Подпрограмма ничегонеделанья
;       Третья строка таблицы:
```

```
        .byte   2               ; Текущее состояние = 2
        .word   testDown        ; Подпрограмма проверки факта нажатия кнопки
        .byte   3               ; Новое состояние = 3
        .word   turnOff         ; Подпрограмма выключения лампочки
;       Четвертая строка таблицы:
        .byte   3               ; Текущее состояние = 3
        .word   testUp          ; Подпрограмма проверки факта отпускания кнопки
        .byte   0               ; Новое состояние = 0
        .word   nothing
```

Кроме того, понадобятся пять подпрограмм (что это такое – об этом разговор чуть ниже). Вот они:

```
testDown:       ; <8> Подпрограмма проверки факта нажатия кнопки:
        bttl    %b1,00000001b ; Проверить линию A[0]
        jnz     $1              ; Кнопка еще не нажата!
        rtsc    1               ; Выход с установкой флага C=1
$1:     rtsc    0               ; Выход со сбросом флага C=0

testUp:         ; Подпрограмма проверки факта отпускания кнопки:
        bttl    %b1,00000001b ; Проверить линию A[0]
        jnz     $1              ; Кнопка еще не нажата!
        rtsc    1               ; Выход с установкой флага C=1
$1:     rtsc    0               ; Выход со сбросом флага C=0

nothing:        ; Подпрограмма ничего не делать
        rts

turnOn:         ; Подпрограмма включения лампочки
        btsl    %b2,00000001b ; Включить лампочку
        rts

turnOff:        ; Подпрограмма отключения лампочки
        btcl    %b2,00000001b ; Отключить лампочку
        rts
        .end
```

Объем программы, по сравнению со `switch`-технологией, значительно увеличился. Приведем комментарии к этой программе:

1. В программе снова употреблены команды **ldal** и **ldah**, которых нет в описании системы команд микроконтроллера. Это – на самом деле команды **ldr**, и они, по аналогии с командами **mial** и **miah**, также загружают в регистры адрес, отмеченный меткой table. Разница здесь в том, что команды **mial** и **miah** (есть также команды **lial** и **liah**) загружают адрес, выраженный в **словах**, тогда как команды **ldal** и **ldah** (есть также команды **mdal** и **mdah**) загружают адрес, выраженный в **байтах**: напомним читателю, что микроконтроллер содержит память программ объемом 1024 слова, а поскольку слово состоит из двух подряд идущих байт, то объем памяти программ – 2048 байт. В отличие от команд, к которым процессор обращается по словам, при обращении к памяти программ через регистр IR1 можно получить доступ индивидуально к каждому байту, поэтому для работы с ним и используются команды **ldal** и **ldah**. Можно очень грубо считать, что команды **ldal** и **ldah** загрузят в регистр IR1 число, вдвое большее, чем это сделали бы команды **lial** и **liah**.
2. Команда **mov %a3,%d7** – первый случай употребления в программе косвенного регистра IR1 и косвенной адресации. Она пересылает в ячейку **%a3** байт, отмеченный меткой table (в нем находится число строк таблицы автомата), так как только что загрузили регистр IR1 адресом именно этой метки. Обратите внимание на автоматную таблицу: в ней отмечена меткой table конструкция **.byte**. С помощью этой конструкции в памяти программ можно разместить не команду, а именно желаемое значение. В данном случае заносим в очередной байт (не слово!) памяти программ число 4 (оно будет размером один байт). Для занесения в память программ чисел, занимающих два байта (например, для меток) используется другая конструкция – **.word**. Именно с помощью конструкций **.byte** и **.word** заносятся в память программ все данные (например, тексты сообщений, которые необходимо выдать на экран

ЖКИ). Вместо команды **mov** в принципе можно было бы в данном случае просто воспользоваться командой **movl %a3,4**, но команда **mov %a3,%d7** – более правильное решение (почему – будет понятно несколько позже). Напомним читателю, что при употреблении в команде конструкции **%d7** содержимое регистра IR1 после выполнения команды автоматически будет увеличено на единицу (потому что он указывает на память программ, а для нее предусмотрен режим **автоинкремента** – автоматического увеличения содержимого регистра IR1 на единицу). При этом он «сдвинется» на начало первой строчки автоматной таблицы.

3. Команда **cmp** сравнивает текущее состояние автомата с тем, которое записано в очередной строчке таблицы. Хочется сразу же предупредить читателя – **ни в коем случае** нельзя записывать эту команду в виде **cmp %d7,%a0**. Это связано с особенностью реализации ядра микроконтроллера КР1878ВЕ1, и, к великому сожалению, никак не отражено в документации по нему: **если регистр IR1 ссылается на память программ, его ни в коем случае нельзя указывать в качестве «приемника» данных в какой-либо команде! Несмотря на то**, что команда сравнения **ничего никуда не записывает** (кроме флагов регистра состояния), **на нее это правило тоже распространяется**! Если состояние автомата – «не то», следующая команда **jnz** обеспечит переход на новую строчку таблицы автомата. Посмотрите на участок программы, отмеченный меткой $1: там расположена «странная» последовательность из пяти команд сравнения **cmpl**. Этот, на первый взгляд, бессмысленный код, нужен только для того, чтобы просто передвинуть регистр IR1 на начало новой строчки таблицы – а для этого он должен сдвинуться на пять байт (одна строчка автоматной таблицы имеет размер шесть байт, поле **.byte** размером один байт, поле **.word** размером два байта, далее – еще одна такая же «парочка», а на один байт по этой строчке уже сдвинулись при выполнении команды **cmp,** так что остается сдвинуться еще на пять байт). Каждая команда, в которой употреблена конструкция **%d7**, сдвигает регистр IR1 на один байт, поэтому и требуется пять команд, в которых употреблена конструкция **%d7**. Ни для чего иного эти команды не предназначены. Поэтому в качестве команд и выбраны наиболее «безобидные» команды **cmpl**, которые ничего, кроме регистра состояния, не портят.
4. Далее следуют команды **mov %a1,%d7** и mov **%a2,%d7**, которые переписывают в рабочие ячейки адрес подпрограммы анализа терминального символа – для указания адреса достаточно просто употребить в конструкции **.word** соответствующую метку. Отметим сразу же, что после их выполнения регистр IR1 сдвинется на два байта – «доберется» в текущей строчке таблицы до байта с номером нового состояния автомата.

Далее в программе нам встретились новые, ранее не применявшиеся команды **push**. Что это за команды и для чего они применяются? Чтобы ответить на этот вопрос, придется вспомнить, что в микроконтроллере КР1878ВЕ1 существуют две специальные сущности – стек команд и стек данных. Теперь настало время рассмотреть их подробнее.

Стек данных предназначен для временного хранения регистров (**#A**, **#B**, **#C**, **#D**, **#4**, **#5**, **#6**, **#7**), содержимое которых не хочется потерять или испортить. Именно такая ситуация сейчас и сложилась в программе – для того, чтобы выполнить команду **ijsr** (о ней расскажем немного попозже), потребуется загрузить в косвенный регистр IR1 адрес вызываемой подпрограммы (нечто подобное уже делали при написании автомата по `switch`-технологии, только там в регистр IR1 загружался адрес перехода). Однако если его туда загрузить, то испортим прежнее содержимое регистра IR1, и не сможем добраться ни до байта с новым состоянием автомата, ни до адреса подпрограммы выхода. Выход один – прежнее содержимое регистра IR1 надо где-то **сохранить**, затем воспользоваться регистром IR1 для выполнения команды **ijsr**, а затем **восстановить** его прежнее содержимое. Сохранить содержимое регистра в принципе можно в свободных ячейках ОЗУ, например **%a4** и **%a5** (для сохранения содержимого регистра в ячейке ОЗУ в системе команд микроконтроллера есть команда **mfpr**, а для обратной записи ячейки ОЗУ в регистр – команда **mtpr**), но в микроконтроллере для этой цели существует специальный блок – стек данных. Стек представляет собой дополнительный блок памяти размером 16 байт, никак не связанный ни с ОЗУ микроконтроллера, ни с его адресным пространством. Над стеком можно выполнять две команды:

– команда **push** – сохраняет содержимое указанного в команде регистра в стек данных;
– команда **pop** – восстанавливает информацию из стека в указанный в команде регистр.

Принцип хранения данных в стеке – принцип **LIFO** (last in – first out). При этом байт, записанный в стек данных последним, будет извлечен оттуда первым. Стек в программистской

литературе называют еще термином **магазин** – по аналогии с магазином автомата (другого, предназначенного для стрельбы), в котором **самый последний** заряженный в магазин патрон выстрелит **самым первым**. «Верхняя» ячейка стека (последний вставленный патрон) в программистской литературе называется **вершиной** стека. Пользоваться стеком вместо ячеек ОЗУ намного удобнее, потому что не требуется отводить ячейки памяти для временного хранения (ведь свободных ячеек ОЗУ в доступных в данный момент сегментах может просто-напросто не оказаться, и тогда придется перезагружать один из сегментных регистров, чтобы добраться до свободного сегмента, а ведь сегментный регистр терять-то тоже не хочется!) и не требуется задавать никакие адреса, но у стека данных есть ограничение – в него можно записать не более 16 байт! Будьте внимательны – если в программе выполнится слишком много команд **push** без соответствующих команд **pop**, процессор микроконтроллера вырабатывает сигнал немаскируемого прерывания «переполнение стека» (это – чрезвычайно серьезная программная ошибка). Точно так же, если из-за ошибки в программе вы выполните слишком много команд **pop**, процессор микроконтроллера выработает сигнал немаскируемого прерывания «переопустошение стека». Поэтому сразу дадим читателю совет – пользоваться стеком данных в програме нужно аккуратно.

Итак, продолжим комментирование программы:

5. Сохраняем содержимое регистра IR1 (т.е. регистров **#6** и **#7**) в стек данных. Теперь их можно как угодно портить.
6. Загружаем регистр IR1 адресом подпрограммы проверки терминального символа. Адрес этот предварительно переписали в ячейки **%a1** и **%a2**, и поэтому теперь загружаем их в регистры **#6** и **#7** с помощью команд **mtpr**. Сразу ответим на один каверзный вопрос, который, возможно, уже возник у читателя: «А нельзя ли было выполнить эту операцию напрямую, т.е **mptr #6,%d7**, **mtpr #7,%d7**?». Нет, нельзя! Причин для этого две: одна – общая для любого микроконтроллера, вторая – специфичная именно для КР1878ВЕ1. Общая причина заключается в том, что после первой же команды **mtpr** заменим младшую часть адреса строчки таблицы table, хранящуюся в регистре **#6**, младшей частью адреса подпрограммы. Такой «оригинальный» адрес будет указывать уже и не на таблицу, и еще не на подпрограмму – понятно, что поведение программы после этого станет абсолютно непредсказуемым. С тем же успехом можно, открыв один замок на двери своим ключом, пытаться открыть второй ключом от квартиры соседа. Специфичная же для микроконтроллера КР1878ВЕ1 причина заключается в том, что **в команде mtpr нельзя использовать** конструкцию **%d7**, если регистр IR1 указывает в это время на память программ (кстати говоря, очень жаль – иногда такая возможность была бы полезна, о чем автор в процессе написания программ неоднократно жалел).
7. И наконец – вызов подпрограммы проверки терминального символа **ijsr**. И вот теперь настало время поговорить поподробнее о том, что это такое – подпрограмма.

Понятие **подпрограммы** является одной из фундаментальных идей программирования. Представьте себе, что необходимо выполнить какую-то однотипную последовательность действий в нескольких местах программы (например, сыграть какую-нибудь мелодию). Первое решение заключается в том, чтобы во всех местах, где это необходимо, повторить набор команд, которым можно добиться требуемого эффекта (сделать это с помощью редактора не так сложно – надо просто скопировать фрагмент программы, играющий мелодию, во все нужные места программы). У этого очевидного решения есть только один недостаток – тем самым увеличивается (и очень значительно) объем программы, а память программ микроконтроллера – не резиновая! Выходом из положения явилось появление в программировании **механизма вызова подпрограмм**.

Подпрограмма – это как раз и есть тот кусок кода, который раньше надо было вставлять во все места программы, где он требовался. Теперь вставляем его куда-нибудь в «хвост» программы, где он не будет никому мешать, а в саму программу, во все места, где должен был бы быть этот кусок кода, вставляем специальные **команды вызова** этого куска кода – подпрограммы. Таким образом, экономится память программ – ведь теперь кусок кода у нас только один, а вызываем мы его из стольких мест, из скольких это необходимо. Когда процессор микроконтроллера встречает команду вызова подпрограммы (в микроконтроллере для этого существуют две команды: **jsr** и **ijsr**), он передает управление на указанную подпрограмму (в точности таким же способом, как это делает команда перехода **jmp**).

Затем подпрограмма выполняет все необходимые действия, а затем... А затем – что? Из подпрограммы надо возвратиться в то место, откуда она была вызвана, а мест, откуда ее могли вызвать, в программе могут быть десятки! Следовательно, вызывая подпрограмму, микроконтроллер должен каким-то образом запомнить место, откуда он ее вызвал! И именно для запоминания этого места и существует в микроконтроллере вторая сущность – стек адресов возврата. Это – тоже специальное ОЗУ, емкость которого – восемь слов (напоминаем, что адрес любого «места» в программе имеет размер два байта или одно слово). Когда процессор микроконтроллера встречает команды вызова подпрограмм, он в первую очередь «вталкивает» (наподобие команды **push**) в стек адресов возврата адрес команды, **следующей** за командой **jsr** или **ijsr**, и только потом передает управление подпрограмме. Когда же подпрограмма выполняет все необходимые действия, то для того, чтобы вернуться в то место, откуда ее вызвали, она должна выполнить специальную команду **rts** – возврат из подпрограммы. По сути своей, команда **rts** – это «хитрая» команда перехода, и передает управление она по тому адресу, который в данный момент находится в **вершине** стека адресов возврата, причем при выполнении этой команды адрес из вершины стека «выталкивается» (наподобие команды **pop**). А поскольку там всегда находится адрес команды, следующей за командой вызова подпрограммы (как раз тот адрес, куда необходимо вернуться из подпрограммы), команда и получила свое название – «возврат из подпрограммы».

Зачем в стеке адресов возврата целых восемь слов? Для того, чтобы изнутри подпрограмм тоже можно было вызывать другие подпрограммы – в подпрограмме ведь тоже могут оказаться повторяющиеся куски кода. По аналогии с вложенными циклами, для подпрограмм, вызванных из других подпрограмм, применяется термин **вложенные подпрограммы**. Число подпрограмм, вызвавших друг друга (**уровень вложенности подпрограмм**), для микроконтроллера КР1878ВЕ1 не должно превышать восьми (стек адресов, напомним, содержит только восемь слов).

Продолжим комментирование программы.

8. Давайте рассмотрим какую-нибудь подпрограмму, например `testDown`. Получив управление по команде **ijsr**, эта подпрограмма, в первую очередь, проверяет, нажата ли кнопка. Делает она это тем же способом, что и в `switch`-технологии, поэтому подробно разбирать этот фрагмент программы не будем. А вот следующий участок представляет интерес, потому что в нем вместо команды возврата из подпрограммы **rts** примена ее специальная разновидность **rtsc** – возврат с установкой флага `C` регистра состояния в определенное значение (логический 0 или логическая 1). Если посмотреть код аналогичной программы проверки `testUp`, то там тоже выполняется возврат таким же способом. Если читатель внимательно проанализирует обе подпрограммы, он заметит, что `C=1` устанавливается в них тогда, когда ожидаемый терминальный символ («Потянуть за веревочку» или «Отпустить веревочку» соответственно) обнаружен, а `C=0` – при его отсутствии. Таким образом, с помощью флага `C` регистра состояния вызванная подпрограмма **извещает** вызывающую программу о том, найден ли проверяемый ею терминальный символ. Такой способ извещения вызывающей программы в программистской литературе именуется **кодом возврата** или **кодом завершения**. Выполнив соответствующую подпрограмму, выполняется не только возврат обратно, но и получается информация о том, есть ли на входе автомата требуемый терминальный символ.

9. Сразу после возврата из подпрограммы выполняются команды **pop #7** и **pop #6**. В результате этого в регистре IR1 **восстановится** та информация, которая была в нем перед выполнением команд **push #6** и **push #7**. Обратите внимание – команды **pop** обязательно **должны выполняться в порядке, обратном тому**, в каком выполнялись в программе команды **push**! Это – общее правило работы со стеком, и исключения из него совершенно недопустимы!

10. Теперь настала пора **проанализировать код возврата** из подпрограммы проверки терминального символа: команда **jnc** передаст управление на метку $2, если флаг `C=0` – ожидаемый терминальный символ подпрограммой проверки не найден. К участку кода с меткой $2 применим тот же комментарий, что и к участку кода с меткой $1 – фиктивные сравнения предназначены исключительно для передвижения регистра IR1 на следующую строчку таблицы, но теперь передвигать его надо всего на три байта – на два байта его уже сдвинули ранее командами **mov %a1,%d7** и **mov %a2,%d7**.

11. Если же нужный терминальный символ подпрограммой проверки найден (флаг `C=1`), то команда условного перехода **jnc** процессором проигнорируется, и будет выполнена следующая команда **mov %a0,%d7**. Таким способом изменяется текущее состояние автомата на новое,

взятое из таблицы (регистр IR1 сдвинулся при этом еще на один байт, и он указывает теперь на адрес подпрограммы выхода).
12. Следующие две команды **mov**, две команды **mtpr** и команда **ijsr** вызывают на выполнение подпрограмму выхода, предусмотренную таблицей автомата. Этот код в точности повторяет аналогичный код для вызова подпрограммы проверки терминального символа, только теперь в нем нет команд **push** и **pop** – нужная строка в таблице автомата найдена, все необходимые действия сделаны, содержимое регистра IR1 нам больше не потребуется и сохранять его нет необходимости. Подпрограмм выхода в автомате три – одна включает лампочку, другая ее выключает, а третья – «ничего не делает» в самом прямом смысле этого слова. Такие «ничего не делающие» подпрограммы – один из **программистских трюков**: он позволяет выполнять в программе те или иные действия, не задумываясь (не анализируя с помощью дополнительных команд), надо ли вызывать в данном случае подпрограмму или нет.
13. После возврата из подпрограммы выхода происходит передача управление на метку again – начинается новый цикл работы интерпретатора автомата.
14. О назначении пяти команд фиктивного сравнения уже говорили. Команды же **subl** и **jnz** – это «хвост» цикла (хотя на самом деле это его заголовок). Цикл этот нужен, чтобы «не перестараться» и не обработать лишних строк – их в таблице всего четыре! Когда же цикл закончится, вступит в действие последняя команда **jmp again**, завершая таким образом бесконечный цикл работы автомата в целом.

Дочитав до этого места, читатель, думается, находится уже в крайней степени раздражения: вот так технология! Вот так программа! Выходит, для того, чтобы соорудить устройство, эквивалентное счетному триггеру (не правда ли?), надо, как предлагает автор, накатать опус чуть короче «Войны и мира»! Не торопитесь, уважаемый читатель, давайте все-таки откомпилируем и занесем в микроконтроллер любую из двух наших программ, и «подергаем за веревочку» у нашего «торшера». Не забудьте при загрузке программы в микроконтроллер указать строку конфигурации **/c:0x01C0**.

<В этом месте автор предполагает, что читатель внял просьбе автора и все-таки занес и подергал все, о чем его просили.>

Итак, посмотрим результаты. Мало того, что пришлось писать какую-то жуткую программу, так она еще и не работает – жмешь на кнопку, а лампочка не загорается. Или наоборот – не тухнет. В чем тут дело?

А дело очень и очень простое: мы забыли о самом элементарном – о «дребезге» контактов. Думается, нет нужды объяснять радиолюбителям, что это такое. Существует достаточно много способов бороться с дребезгом, и самый простой из них – дать кнопке после нажатия или отпускания некоторое время – успокоиться. Давайте нарисуем для нашего автомата новый граф переходов – уже с учетом дребезга (рис. 3). Для определенности будем предполагать, что 20 миллисекунд для «успокоения» кнопки будет достаточно.

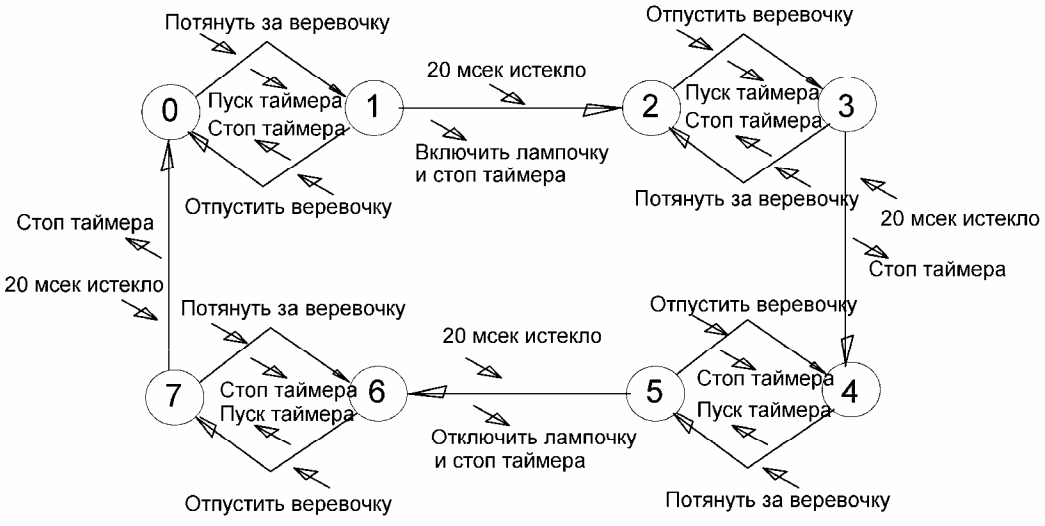

Рис. 3. Граф переходов автомата с учетом дребезга

4.Модификация автомата

Итак, новое описание автомата следующее:

1. Множество терминальных символов: {«Потянуть веревочку», «Отпустить веревочку», «20 мсек истекло»}.
2. Множество нетерминальных символов: {0,1,2,3,4,5,6,7}.
3. Множество выходных символов {«Включить лампочку и остановить таймер», «Отключить лампочку и остановить таймер», «Запустить таймер», «Остановить таймер» }.
4. Начальное состояние автомата – 0.
5. Множество конечных состояний автомата пусто.
6. Автоматная таблица (табл. 4) приведена ниже.

Таблица 4

Текущее состояние	Терминальный символ	Новое состояние	Программа выхода
0	«Потянуть веревочку»	1	«Запустить таймер»
1	«Отпустить веревочку»	0	«Остановить таймер»
1	«20 мсек истекло»	2	«Включить лампочку и остановить таймер»
2	«Отпустить веревочку»	3	«Запустить таймер»
3	«Потянуть веревочку»	2	«Остановить таймер»
3	«20 мсек истекло»	4	«Остановить таймер»
4	«Потянуть веревочку»	5	«Запустить таймер»
5	«Отпустить веревочку»	4	«Остановить таймер»
5	«20 мсек истекло»	6	«Отключить лампочку и остановить таймер»
6	«Отпустить веревочку»	7	«Запустить таймер»
7	«Потянуть веревочку»	6	«Остановить таймер»
7	«20 мсек истекло»	0	«Остановить таймер»

Итак, снова попробуем реализовать наш новый автомат теми же двумя способами, которыми мы реализовывали первый автомат. Начальный участок программы претерпевает некоторые изменения:

```
Start:  jmp     begin           ; Переход к началу программы
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
begin:  ldr     #D,18h          ; Сегмент D – управляющие регистры портов
        movl    %d1,00011011b   ; Доступ к подрегистру 3 в режиме автоинкремента
        movl    %d1,00000000b   ; Подрегистр 3: все линии - вводные
        movl    %d1,00000000b   ; Подрегистр 4: режим вывода безразличен
        movl    %d1,00000001b   ; Подрегистр 5: резистор A[0] подключен
        movl    %d1,00000000b   ; Подрегистр 6: прерывания нас не интересуют
        movl    %d1,00000000b   ; Подрегистр 7: прерывания нас не интересуют
        movl    %d2,00011011b   ; Доступ к подрегистру 3 в режиме автоинкремента
        movl    %d2,00000001b   ; Подрегистр 3: включить линию B[0] на вывод
        movl    %d2,00000001b   ; Подрегистр 4: режим вывода – «полноценный»
        movl    %d2,00000000b   ; Подрегистр 5: резисторы нагрузки нам не нужны
        movl    %d2,00000000b   ; Подрегистр 6: прерывания нас не интересуют
```

```
        movl    %d2,00000000b       ; Подрегистр 7: прерывания нас не интересуют
        ldr     #B,00               ; Сегмент B – регистры портов и таймера
        movl    %d4,00010000b       ; Подключить регистр конфигурации таймера
        movl    %d5,00000111b       ; 8-битный, с предделителем на 256
        movl    %d4,00000000b       ; Подключить регистр интервала
        movl    %d5,00000000b       ; Регистр интервала обнуляем
```

Изменения связаны в основном с тем, что теперь нам необходимо, помимо портов, еще и подготовить к работе таймер.

4.1. Switch-технология

Первый кусок программы реализации автомата по switch-технологии остается прежним, в нем только увеличивается список команд перехода:

```
labels: jmp     cond_0          ; Таблица перехода на участки, соответствующие
        jmp     cond_1          ; состояниям автомата
        jmp     cond_2
        jmp     cond_3
        jmp     cond_4
        jmp     cond_5
        jmp     cond_6
        jmp     cond_7
```

А вот как теперь выглядит «смысловая часть» автомата:

```
cond_0: bttl    %b1,00000001b ; Нажата ли кнопка «торшера» (лог.0)
        jnz     again         ; Пока нет – начать работу автомата сначала
        movl    %a0,1         ; Теперь новое состояние автомата = 1
        btsl    %b4,00001001b ; Запустить таймер на счет и подключить к
                              ; рабочему регистру счетный регистр
        jmp     again
cond_1: bttl    %b1,00000001b ; Отпущена ли кнопка «торшера» (логическая 1)
        jz      $1            ; Пока нет – проверить, закончен ли интервал
        movl    %a0,0         ; Теперь новое состояние автомата = 0
        btcl    %b4,00000001b ; Остановить таймер.
        jmp     again
$1:     cmpl    %b5,1         ; Истекло ли время?
        jnz     again         ; Пока нет...
        movl    %a0,2         ; Теперь новое состояние автомата = 2
        btsl    %b2,00000001b ; Включить лампочку
        btcl    %b4,00000001b ; Остановить таймер
        jmp     again
cond_2: bttl    %b1,00000001b ; Нажата ли кнопка «торшера» (логический 0)
        jz      again         ; Пока нет – начать работу автомата сначала
        movl    %a0,3         ; Теперь новое состояние автомата = 1
        btsl    %b4,00001001b ; Запустить таймер на счет и подключить к
                              ; рабочему регистру счетный регистр
        jmp     again
cond_2: bttl    %b1,00000001b ; Нажата ли кнопка «торшера» (логический 0)
        jnz     again         ; Пока нет – начать работу автомата сначала
        movl    %a0,3         ; Теперь новое состояние автомата = 3
        btcl    %b2,00000001b ; Отключить лампочку
        jmp     again
cond_3: bttl    %b1,00000001b ; Отпущена ли кнопка «торшера» (логическая 1)
        jz      again         ; Пока нет – начать работу автомата сначала
        movl    %a0,0         ; Теперь новое состояние автомата = 0
        jmp     again
        .end
```

Вот сколько изменений пришлось нам внести в программу, чтобы она могла адекватно реагировать на дребезг контактов! Можно только представить себе, во что это выльется для второго варианты реализации!

4.2. Интерпретатор

1. Начало программы придется изменить в любом случае – необходимо инициализировать таймер.
2. Автоматная таблица, разумеется, полностью изменилась, и выглядит теперь следующим образом:

```
table:  .byte   12              ; Число строчек в автоматной таблице
;       Первая строка таблицы:
        .byte   0               ; Текущее состояние = 0
        .word   testDown        ; Подпрограмма проверки факта нажатия кнопки
        .byte   1               ; Новое состояние = 1
        .word   startT          ; Подпрограмма запуска таймера
;       Вторая строка таблицы:
        .byte   1               ; Текущее состояние = 1
        .word   testUp          ; Подпрограмма проверки факта отпускания кнопки
        .byte   0               ; Новое состояние = 0
        .word   stopT           ; Подпрограмма останова таймера
;       Третья строка таблицы:
        .byte   1               ; Текущее состояние = 1
        .word   test20          ; Подпрограмма проверки временного интервала
        .byte   2               ; Новое состояние = 2
        .word   turnOn          ; Подпрограмма включения лампочки
;       Четвертая строка таблицы:
        .byte   2               ; Текущее состояние = 2
        .word   testUp          ; Подпрограмма проверки факта отпускания кнопки
        .byte   3               ; Новое состояние = 3
        .word   startT          ; Подпрограмма запуска таймера
;       Пятая строка таблицы:
        .byte   3               ; Текущее состояние = 3
        .word   testDown        ; Подпрограмма проверки факта нажатия кнопки
        .byte   2               ; Новое состояние = 2
        .word   stopT           ; Подпрограмма останова таймера
;       Шестая строка таблицы:
        .byte   3               ; Текущее состояние = 3
        .word   test20          ; Подпрограмма проверки временного интервала
        .byte   4               ; Новое состояние = 4
        .word   stopT           ; Подпрограмма останова таймера
;       Седьмая строка таблицы:
        .byte   4               ; Текущее состояние = 4
        .word   testDown        ; Подпрограмма проверки факта нажатия кнопки
        .byte   5               ; Новое состояние = 5
        .word   startT          ; Подпрограмма запуска таймера
;       Восьмая строка таблицы:
        .byte   5               ; Текущее состояние = 5
        .word   testUp          ; Подпрограмма проверки факта отпускания кнопки
        .byte   4               ; Новое состояние = 4
        .word   stopT           ; Подпрограмма останова таймера
;       Девятая строка таблицы:
        .byte   5               ; Текущее состояние = 5
        .word   test20          ; Подпрограмма проверки врекменного интервала
        .byte   6               ; Новое состояние = 6
        .word   turnOff         ; Подпрограмма выключения лампочки
;       Десятая строка таблицы:
        .byte   6               ; Текущее состояние = 6
        .word   testUp          ; Подпрограмма проверки факта отпускания кнопки
        .byte   7               ; Новое состояние = 7
```

```
        .word   startT          ; Подпрограмма запуска таймера
;       Одиннадцатая строка таблицы:
        .byte   7               ; Текущее состояние = 7
        .word   testDown        ; Подпрограмма проверки факта нажатия кнопки
        .byte   6               ; Новое состояние = 6
        .word   stopT           ; Подпрограмма останова таймера
;       Двенадцатая строка таблицы:
        .byte   7               ; Текущее состояние = 7
        .word   test20          ; Подпрограмма проверки временного интервала
        .byte   0               ; Новое состояние = 0
        .word   stopT           ; Подпрограмма останова таймера
```

3. В систему добавляются три подпрограммы:

```
Test20:         ; Проверка истечения 20 мсек.
        cmpl    %b5,2           ; 20 мсек истекло?
        jnz     $1              ; Пока нет...
        rtsc    1               ; Да!
$1:     rtsc    0               ; Нет!

startT: bisl    %b4,00001001b ; Запустить таймер на счет и подключить к рабочему
                                ; регистру счетный регистр
        rts

stopT:  bicl    %b4,00001001b ; Остановить таймер
        rts
```

4. Подпрограмма **nothing** из системы выбрасывается совсем.
5. Подпрограммы **turnOn** и **turnOff** исправляются следующим образом:

```
turnOn:         ; Подпрограмма включения лампочки
        btsl    %b2,00000001b ; Включить лампочку
        jsr     stopT           ; Вызвать подпрограмму отключения таймера
        rts


turnOff:        ; Подпрограмма отключения лампочки
        btcl    %b2,00000001b ; Отключить лампочку
        jsr     stopT           ; Вызвать подпрограмму отключения таймера
        rts
```

И ВСЕ!!!

Да-да! Это действительно **ВСЕ**! Результат кажется потрясающим. По большому счету во втором случае **программу** практически и пальцем не тронули – просто **заменили старую таблицу автомата новой**, да дописали пару-тройку малюсеньких подпрограмм. Вот где сказывается преимущество автоматной технологии – не в том, **как быстро мы напишем первый вариант** автомата, а в том, с **какой легкостью мы можем изменить уже написанную программу**.

Все вышесказанное и вышенаписанное кажется каким-то ловким фокусом, который автор проделал над читателем, но нет – никакого фокуса нет! В самом деле, посмотрите, как работает интерпретатор автомата – если текущее состояние автомата равно *какому-то-там* состоянию и *какая-то-там* подпрограмма проверки *чего-то-там* сказала, что *чего-то-там* имееет место, то состояние автомата нужно изменить на *какое-то-там* новое, а также выполнить *какую-то-там* подпрограмму, которая *что-то-там-такое* сделает! Что в этой работе специфически «торшерного», например?! Такому интерпретатору вообще нет никакой разницы, пирожки ли печь, или управлять прокатным станом. Кстати, теперь можно дать и пояснения относительно одного комментария автора – когда он отмечал, что вместо команды **mov %a3,%d7**, следующей за командами **ldal** и **ldah**, можно использовать и команду **movl %a3,4**, но лучше этого не делать. Если бы вставить в это место команду **movl**, то при любом изменении таблицы автомата эту команду тоже пришлось бы довольно часто изменять (если бы изменилось число строк автоматной таблицы). С командой **mov** даже этого делать не надо, интерпретатор с ее помощью сразу же сам определит, сколько реально строк содержит

автоматная таблица, и вносить в **интерпретатор** какие-либо изменения не потребуется больше **никогда**.

ВЫВОДЫ...

Автоматное программирование – одна из наиболее эффективных техник программирования микроконтроллеров, пригодная в первую очередь для реализации задач управления (а микроконтроллеры в большинстве практических случаев используются именно для управления, а не для обработки сигналов. Одно из его достоинств – возможность **табличного** программирования микроконтроллера, когда по разработанному графу переходов составляется автоматная таблица, а код ее интерпретатора переносится без изменений из системы в систему. Еще одно достоинство автоматного программирования – простота. Например, по автоматной таблице легко восстановить логику работы устройства и/или внести в нее необходимые изменения.

Завершая эту статью, небольшое «домашнее задание»: запрограммировать на основе программы-интерпретатора «умный» торшер, который ведет себя следующим образом:

1. Если при выключенной лампочке дернуть за веревочку, торшер включается на одну минуту, после чего пять раз мигает (мигание – по секунде на вспышку), и затем автоматически выключается (режим временного включения).
2. Если в то время, когда торшер горит, дернуть за веревочку, торшер выключается досрочно.
3. Если в то время, когда торшер начал мигать, дернуть за веревочку, торшер перестает мигать и горит уже постоянно, до тех пор, пока его не выключат, дернув за веревочку.
4. Если при включении торшера дернуть за веревочку дважды (наподобие двойного щелчка мыши), торшер включается и горит постоянно, пока его не выключат, дернув за веревочку. Для того, чтобы считать дергание веревочки двойным, будем считать, что второе дерганье должно произойти не позднее чем через 0.5 с после первого.
5. Если торшер горит постоянно, дважды дернув за веревочку, его переводят в режим автоматического погасания через минуту.

Обязательное требование к программе – **не изменять** интерпретатор автомата. При этом все возможности «умного» торшера должны быть реализованы через автоматную таблицу и соответствующие маленькие подпрограммы! Не смущайтесь размеров автоматной таблицы – устройства с такой сложной логикой простыми не бывают.

И, напоследок, «крамольный» вопрос – для чего нужна switch-технология, если интерпретатор дает такие великолепные результаты? `Switch`-технология тоже необходима и часто применяется, потому что у нее есть один неоспоримый плюс – скорость работы автомата, запрограммированного по switch-технологии, обычно **значительно выше**, чем при использовани00и интерпретатора автоматной таблицы (как всегда, за удовольствие приходится платить, и за применение интерпретатора расплачиваются скоростью работы программы, за ее компактность, переносимость и простоту модификации). Конечно, в радиолюбительской практике сложно представить себе задачи, в которых требуется экономить каждую микросекунду, но если такая необходимость возникнет, то switch-технология – реальный способ ее решения. Можно даже порекомендовать такой способ – вначале реализовывать автомат в виде интерпретатора, а затем, после отладки программы, переписать его на основе `switch`-технологии[1].

Источники

1. *Шалыто А.А.* Switch-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука. 1998. – 628 с.

[1] Если программа генерируется по графу переходов автоматически, то указанный недостаток `switch`-технологии исчезает, а достоинство остается [2]. В инструментальном средстве *UniMod* [3, 4] на языке *Java* реализованы оба подхода – интерпретационный и компилятивный. Во втором случае при генерации кода используется оператор `switch` (прим. ред. А. Шалыто)

2. *Гуров В.С., Мазин М.А., Нарвский А.С., Шалыто А.А.* UML. SWITCH-технология. Eclipse // Информационно-управляющие системы, 2004, № 6, с. 12-17.

3. http://unimod.sourceforge.net

4. http://is.ifmo.ru