Применение SWITCH-технологии при разработке прикладного программного обеспечения для микроконтроллеров. Часть 1

В предыдущей своей статье [1] автор предпринял попытку рассмотрения ряда проблем, возникающих при разработке прикладного программного обеспечения (ПО) встроенных систем (в дальнейшем будем называть его «программное обеспечение»). Многие положения статьи могли показаться спорными, но, без сомнения, затронутые в ней вопросы являются актуальными, что демонстрирует ряд читательских откликов. Между тем, существуют технологии программирования, существенно облегчающие разработку встроенного ПО.

Владимир ТАТАРЧЕВСКИЙ
arktur04@mail.ru

Одной из технологий, облегчающих разработку встроенного ПО, является SWITCH-технология, упоминавшаяся в работе [1]. Сегодня мы рассмотрим один из вариантов SWITCH-технологии, применяемый при разработке ПО микроконтроллеров. Данный вариант реализации SWITCH-технологии был разработан автором для создания собственных проектов, и, конечно же, не является единственно возможным. Автор намерен и в дальнейшем развивать предлагаемую концепцию программирования, делая ее более гибкой и приспособленной к весьма широкому кругу задач. Автор также надеется, что данная публикация послужит своего рода примером для других программистов, и побудит их к публикации своих разработок.

В последнее время, по мере роста мощности микроконтроллеров, все большую популярность приобретают операционные системы реального времени (ОСРВ), такие как uC/OS II, Embedded Linux и т. п. Однако их применение не решает всех проблем, возникающих при разработке ПО. Во-первых, применение ОСРВ ограничено сравнительно мощными микроконтроллерами и практически исключено для наиболее массового сегмента микроконтроллеров — 8-разрядных устройств. Во-вторых, ОСРВ применяют там, где необходимо организовать выполнение и взаимодействие нескольких программных потоков, при этом многие проблемы разработки не только снимаются, но и могут усугубиться. Разработка приложений, состоящих из множества асинхронных потоков, выполняющихся зачастую с разными приоритетами, требует от программиста применения сложных средств организации взаимодействия потоков, что приобретает особую сложность при организации доступа различных потоков у аппаратным ресурсам микроконтроллера. Впрочем, достоинствам и недостаткам ОСРВ можно посвятить отдельную статью, а здесь достаточно сказать, что применение SWITCH-технологии в ряде случаев снимает необходимость использования ОСРВ за счет того, что, во-первых, взаимодействие между автоматами осуществляется более простым образом, чем между потоками ОСРВ, во-вторых, применение автоматов делает поведение программы абсолютно детерминированным, а взаимодействие потоков в ОСРВ требует дополнительных (и немалых) усилий для устранения коллизий. Однако SWITCH-технология вовсе не исключает применение операционных систем (в частности, она может использоваться при разработке ПО для MS Windows). Например, возможен вариант реализации программ по SWITCH-технологии, в котором различные автоматы выполняются в различных потоках, обмениваясь между собой сообщениями. SWITCH-технология может применяться для разработки систем со сложным поведением в различных предметных областях, она подходит и для создания систем жесткого реального времени. Технология может также использоваться для построения сложных обработчиков прерываний, что будет рассмотрено в последующих статьях.

В чем же заключаются преимущества SWITCH-технологии вообще и предлагаемой реализации в частности?

Преимущество первое (и главное): программы, построенные по предлагаемой технологии, легко документировать. В рамках этого стиля программирования [4] программа представляет собой совокупность конечных автоматов, взаимодействующих друг с другом и с «внешним миром». При этом в наглядной графической форме могут быть выражены как связи между автоматами, так и их внутренняя структура.

Преимущество второе: возможность повторного использования кода. Задумывались ли вы о том, почему столь высокую популярность завоевали средства быстрой разработки (Rapid Application Development — RAD) — такие, как Borland Delphi? Ответ прост: потому что программа в них состоит из компонентов, являющихся в высокой степени автономными «сущностями». Компонент име-

Как уже упоминалось в работе [1], SWITCH-технология, которая также называется «автоматное программирование», является отечественной технологией программирования, созданной и разрабатываемой А. А. Шалыто и его соавторами. Отличным введением в SWITCH-технологию может служить книга [2]. Кроме того, Анатолий Абрамович Шалыто является основателем «Движения за открытую проектную документацию» [3]. С материалами по этой технологии и «Движению» можно ознакомиться на сайте **http://is.ifmo.ru/**, а также в Википедии (**http://ru.wikipedia.org/**, статья «Switch-технология»).

ет ограниченное количество связей с остальной программой, его можно разрабатывать и тестировать отдельно, а применять многократно, и именно это свойство подобных систем делает разработку быстрой и удобной.

Автомат в предлагаемом стиле программирования также является своего рода «кирпичиком», автономной единицей программы. Его связи с остальными автоматами сведены к минимуму и унифицированы. Его, как и компонент RAD-системы, можно разработать отдельно, а затем применять в различных программных проектах.

Из сказанного вытекает и третье преимущество: программы, построенные по SWITCH-технологии, легко поддаются модификации. Так как количество связей между автоматами минимально, изменения в одном из них чаще всего не влекут за собой необходимость коррекции кода в других автоматах.

В данной части настоящей статьи рассматривается общая структура программы, построенной на основе SWITCH-технологии, и ее базовая конструкция — автоматы.

В дальнейшем будут рассмотрены механизм обработки сообщений и механизм таймеров, необходимый для придания программе временно́го детерминизма. После рассмотрения базовых понятий будет приведен пример проектирования реальной программы.

Все исходные тексты ПО в статье приведены на языке Си. Это не означает, что рассматриваемый стиль программирования может быть реализован только на языке Си — его можно переложить и на любой язык программирования, включая язык ассемблера и C++, однако именно язык Си наиболее подходит для демонстрации возможностей данной технологии благодаря своей широкой распространенности среди специалистов по микроконтроллерам.

Все исходные тексты программ взяты из реального проекта, написанного для микроконтроллера AT91SAM7S256 (на ядре ARM7). При разработке использовался компилятор IAR C/C++ Compiler for ARM 4.40A.

Для того чтобы не затруднять понимание исходных текстов особенностями архитектуры данного процессора, из текстов по возможности исключен аппаратно-зависимый код, который заменен в соответствующих местах комментариями. Это сделано еще и для того, чтобы подчеркнуть независимость технологии от архитектуры конкретного процессора. Читатель при желании сможет восполнить эти «пробелы», написав соответствующий код для своего «любимого» микроконтроллера.

Общая структура программы

Опишем излагаемый стиль программирования. Его суть можно сформулировать следующим образом: программа представляет собой совокупность конечных автоматов, выполняющихся параллельно и обменивающихся между собой сообщениями. Другим ключевым свойством предлагаемой концепции является широкое использование таймеров, которые предназначены для привязки работы программы к реальному времени.

Итак, автоматы должны выполняться параллельно. Как достичь данного эффекта без использования многозадачной ОС? На самом деле, ничего сложного здесь нет. Все дело в особой структуре автоматной программы. В работе [5] приведены слова известного разработчика ядра ОС Linux Алана Кокса: «Потоковое программирование нужно тем, кто не умеет использовать конечные автоматы».

Рассмотрим структуру программ рассматриваемого класса более подробно. Будем для простоты считать, что каждый конечный автомат (КА) описан в отдельном модуле программы и имеет, как минимум, две внешние функции:

```
void InitFSM(void);
void ProcessFSM(void);
```

Вместо букв FSM в объявлении функций подставим имя автомата. При этом, например, функции:

```
void InitPasswordEditor(void);
void ProcessPasswordEditor(void);
```

будут принадлежать автомату PasswordEditor, описанному в модуле password_editor.c.

При этом функция InitFSM, как следует из названия, инициализирует автомат (это что-то вроде конструктора в объектно-ориентированном программировании), а функция ProcessFSM отвечает за работу автомата. Главная особенность последней функции: она не должна выполнять продолжительных во времени действий, связанных с ожиданием какого-либо флага или с истечением временного интервала — то есть в ней не должно быть конструкций типа:

```
while(flag == 0);
или
for(i = 0; i < delay; i++);
```

Как будет показано далее, в таких конструкциях просто нет необходимости.

Задачей главного цикла программы является поочередный вызов функций ProcessFSM всех автоматов, составляющих программу:

```
//main.c
#include «messages.h»      //модуль обработки сообщений
#include «timers.h»        //модуль таймеров
#include «fsm1.h»          //модуль автомата fsm1
#include «fsm2.h»          //модуль автомата fsm2
#include «fsm3.h»          //модуль автомата fsm3

void main()
{
 InitTimers();             //инициализация таймеров
 InitMessages();           //инициализация механизма
                           //обработки сообщений
 InitFSM1();               //инициализация автомата FSM1
 InitFSM2();               //инициализация автомата FSM2
 InitFSM3();               //инициализация автомата FSM3

 SendMessage(MSG_FSM1_ACTIVATE); //активируем автомат FSM1

 // главный цикл программы
 while(1)
 {
   ProcessFSM1();          //итерация автомата FSM1
   ProcessFSM2();          //итерация автомата FSM2
   ProcessFSM3();          //итерация автомата FSM3
   ProcessMessages();      //обработка сообщений
 };
}
```

Теперь становится понятно, каким образом обеспечивается «многопоточность» системы: в каждой итерации главного цикла поочередно вызываются Process-функции каждого автомата — каждому автомату выделяется время для выполнения какого-либо элементарного действия (или, возможно, просто для передачи управления далее по списку). Такой порядок работы напоминает кооперативную многозадачность в ранних версиях Windows: программа выполняет какое-либо действие и передает управление дальше, но если она «зависает», то «зависает» вся система. Именно для предотвращения подобной ситуации вводится явный запрет на действия в автоматах, которые занимают продолжительное или неопределенное время.

Рассмотрим базовую структуру, используемую в приведенном выше коде — автоматы.

Автоматы

Вне сомнений, большинству читателей «КиТ» теория КА знакома не понаслышке. Однако в различных литературных источниках для терминов теории КА приводятся несколько различные определения. Поэтому, не приводя здесь строгих формальных определений, уточним некоторые базовые понятия теории КА. Итак, автомат Мура — это КА, у которого выход является функцией состояния — выходные воздействия определены

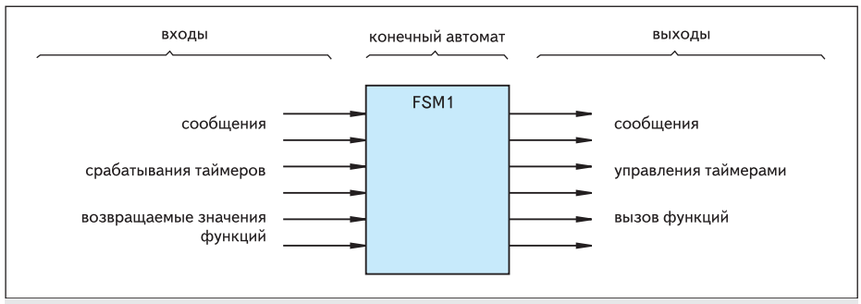

Рис. 1. Конечный автомат с входами и выходами

Ключевым понятием в рассматриваемой технологии программирования является конечный автомат (КА). Теория КА интенсивно развивалась в 50-е — 60-е годы прошлого века и нашла широкое применение во многих областях техники. Особенно плодотворной она оказалась при разработке трансляторов.

При этом КА рассматривается как своеобразное средство для «перевода» цепочек символов языка А в цепочки символов языка В, а правила перевода задаются структурой КА. Такое применение КА привело к тому, что до настоящего времени наиболее полное изложение теории КА можно найти в учебниках по компиляторам и математической лингвистике.

Достаточно широкое применение КА нашли также в задачах синтеза цифровых устройств, в задачах описания поведения сложных систем (в отрасли связи даже был создан специальный язык описания телекоммуникационных систем, базирующийся на КА).

Однако в программировании (в том числе встроенных систем) КА не нашли широкого применения. По сути, единственным инструментальным средством программирования микроконтроллеров, основанным на КА и доступным на сегодняшний день, является IAR Visual State.

Несколько иначе обстоят дела с программируемыми логическими контроллерами (ПЛК). Для них создан стандарт IEC 61131-3, реализованный во множестве систем программирования, из которых, пожалуй, самыми известными являются системы CoDeSys (фирма Smart Software Solutions) и ISaGRAF (фирма ICS Triplex ISaGRAF).

Стандарт IEC 61131-3 включает в себя несколько языков, в том числе и язык SFC (Sequential Function Chart). В рамках этого языка программа представлена как множество шагов (steps), между которыми осуществляются условные переходы (transitions и jumps). Программа может совершать действия (actions) как в состояниях (шагах), так и на переходах. Язык SFC поддерживает также параллельное выполнение участков программы.

Следует, однако, отметить, что языки стандарта IEC 61131-3 не являются идеальным средством программирования, им присущи и некоторые недостатки. Часть из них устраняется за счет введения расширений стандарта. Одно из таких расширений, призванное придать языкам стандарта объектно-ориентированные свойства, описано в работе [6].

В рамках данной статьи не будут рассматриваться достоинства и недостатки IEC 61131-3. Отметим лишь тот факт, что данный стандарт, хорошо зарекомендовавший себя при разработке ПО для ПЛК, не очень хорошо подходит для программирования микроконтроллеров, и дело тут не только в отсутствии соответствующих программных средств.

Это связано с тем, что микроконтроллерные системы зачастую более сложны и предполагают более гибкое использование аппаратных ресурсов, чем позволяет стандарт IEC 61131-3. Именно поэтому базовыми средствами программирования микроконтроллеров до сих пор являются языки ассемблера и Си.

SWITCH-технология делает программирование независимым как от применяемых аппаратных средств, так и языка программирования [2].

в состояниях. Автомат Мили — это автомат, у которого выходные воздействия определены для переходов между состояниями. И, наконец, смешанный автомат — это автомат, у которого выходные воздействия могут быть определены как в состояниях, так и на переходах. Также напомним читателям, что в английском языке термину конечный автомат соответствует термин Finite State Machine, или, сокращенно FSM.

Под словом «автомат» здесь и далее будем понимать исключительно КА. Прежде чем перейти к рассмотрению программных реализаций автоматов, укажем, что автомат имеет входы, выходы и переменную состояния (рис. 1).

Под входом понимается сообщение, срабатывание таймера, результат выражения, которое имеет логическое значение (в том числе возвращаемое функцией логическое значение). Доступ к аппаратным ресурсам микроконтроллера (регистрам периферийных устройств, портам ввода-вывода и т. п.) выполняется также путем вызова функций.

Под выходом автомата понимается отправка сообщения; запуск, останов или сброс таймера; вызов функции.

При этом выделяется особая переменная состояния — переменная, которая определяет текущее состояние автомата. Эта переменная должна быть доступна только своему автомату, ее изменение из других автоматов недопустимо.

Автомат может осуществлять действия (action) и деятельности (activity) автомата. И деятельность, и действие автомата относятся к его выходной активности.

При этом действием называется выходное воздействие, выполняемое однократно при входе в состояние или на переходе, а деятельностью — выходное воздействие, выполняющееся непрерывно в течение всего времени нахождения автомата в определенном состоянии. Также возможна реализация действий, выполняющихся на выходе из состояния.

Автомат Мура [2] и автомат смешанного типа могут выполнять как действия, так и деятельности, а автомат Мили может выполнять только действия.

В минимальном виде объявление автомата выглядит следующим образом:

```
//файл fsm1.h
#ifndef FSM1_h
#define FSM1_h

void InitFSM1(void);
void ProcessFSM1(void);
#endif
```

Перейдем к описанию автомата:

```
//файл fsm1.c
#include «fsm1.h»
#include «timers.h»
#include «messages.h»

char fsm1_state; //переменная состояния

void InitFSM1(void)
{
   fsm1_state = 0;
      //здесь можно выполнить инициализацию других
      //переменных автомата при их наличии
}

void ProcessFSM1(void)
{
   switch(fsm1_state)
   {
      case 0: //неактивное состояние
         if(GetMessage(MSG_FSM1_ACTIVATE))
         {
            fsm1_state = 1;
               //здесь выполняются действия, связанные
               // с активизацией компонента
            …
         };
         break;
   case 1: //активное состояние
      if(GetMessage(MSG_FSM1_DEACTIVATE))
      {
         fsm1_state = 0;
            //здесь выполняются действия, связанные с
            //деактивизацией компонента
         …
      };
      break;
   }
}
```

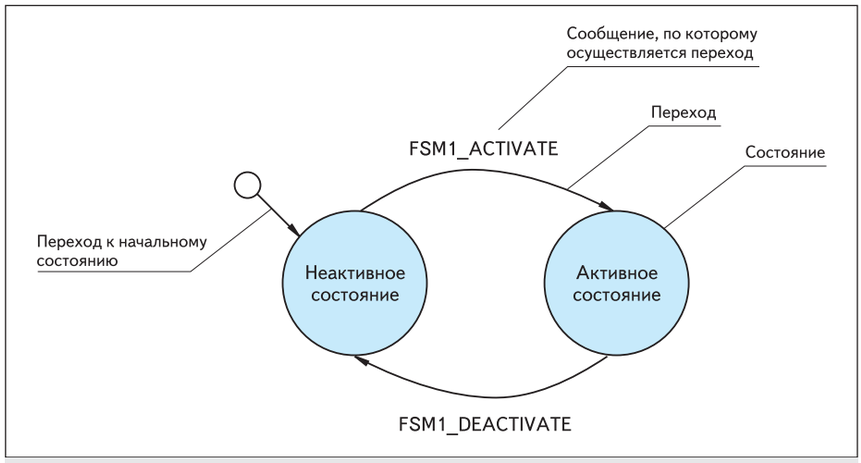

Рис. 2. Граф автомата

Приведенный исходный текст соответствует автомату Мили, который имеет два состояния: активное и неактивное, а также два входа: сообщения MSG_FSM1_ACTIVATE и MSG_FSM1_DEACTIVATE, которые переводят автомат в активное и неактивное состояние соответственно. Граф этого автомата изображен на рис. 2.

Этот автомат выполняет действия на переходах из одного состояния в другое (именно поэтому это автомат Мили), однако его достаточно легко преобразовать в автомат Мура или в смешанный автомат. Пока этот автомат не делает ничего полезного, это просто шаблон, на основе которого можно строить автоматы с более сложным поведением, что и будет сделано в дальнейшем.

В следующей части статьи будет продолжено рассмотрение реализации автоматов, а также механизмов таймеров и сообщений.

Автор выражает глубокую благодарность Анатолию Абрамовичу Шалыто за редактирование статьи и ценные замечания.

Литература

1. Татарчевский В. Некоторые мысли по поводу программирования встроенных систем //Компоненты и технологии. 2006. № 8.
2. Шалыто А. А. Switch-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука. 1998. **http://is.ifmo.ru/books/switch/1**
3. Шалыто А. А. Новая инициатива в программировании. Движение за открытую проектную документацию // Мир ПК. 2003. № 9. **http://is.ifmo.ru/works/open_doc/**
4. Непейвода Н.Н. Стили и методы программирования. М.: Интернет-Университет Информационных технологий. 2005. **http://www.intuit.ru/department/se/progstyles/9/1.html**
5. Зубинский А. FSM. **http://itc.ua/article.phtml?ID=19921&IDw=19**
6. Хесс Д. Объектно-ориентированные расширения МЭК 61131-3 // Современные технологии автоматизации. 2006. № 2.

Применение SWITCH-технологии при разработке прикладного программного обеспечения для микроконтроллеров. Часть 2

Владимир ТАТАРЧЕВСКИЙ
arktur04@mail.ru

В предыдущей статье цикла мы начали рассматривать применение SWITCH-технологии для программирования микроконтроллерных устройств. В данной статье мы продолжим рассмотрение реализации различных конструкций, лежащих в основе предлагаемой концепции программирования.

Итак, мы привели код простейшего конечного автомата, имеющего два состояния и представляющего собой автомат Мили. Автомат переходит из одного состояния в другое под воздействием двух сообщений: MSG_FSM1_ACTIVATE и MSG_FSM1_DEACTIVATE. У читателя может возникнуть вопрос: откуда берутся и как обрабатываются сообщения, если мы не используем операционную систему? Ответ на этот вопрос очень важен, поэтому разберем его подробно.

Обработка сообщений

Широкое применение сообщений является ключевой особенностью всех современных операционных систем — от Windows до «маленьких» ОСРВ, предназначенных для встраиваемых приложений. Механизм сообщений настолько удобен для программиста, что было бы странно отказаться от его применения в автоматной технологии. Перед тем как перейти к реализации собственного механизма обработки сообщений, рассмотрим, как устроены соответствующие механизмы в ОС Windows. В Windows сообщение может быть отправлено с помощью одной из двух функций: SendMessage и PostMessage:

```
LRESULT SendMessage(
      HWND hWnd,          // идентификатор (handle) окна,
                          // которому адресовано сообщение
      UINT Msg,           // идентификатор сообщения
      WPARAM wParam,      // первый параметр
      LPARAM lParam       // второй параметр
  );

BOOL PostMessage(
      HWND hWnd,          // идентификатор (handle) окна,
                          // которому адресовано сообщение
      UINT Msg,           // идентификатор сообщения
      WPARAM wParam,      // первый параметр
      LPARAM lParam       // второй параметр
  );
```

Эти функции имеют одинаковые списки параметров. Параметр hWnd определяет окно (а точнее, процедуру обработки сообщений окна, window procedure), которому адресовано сообщение, параметр Msg является собственно идентификатором сообщения, wParam и lParam определяют параметры сообщения. Параметры могут передаваться непосредственно в виде числовых значений в полях wParam и/или lParam, могут содержаться в отдельной структуре, при этом wParam и/или lParam содержат указатель на эту структуру, и, наконец, могут вовсе отсутствовать. Конкретный смысл полей wParam и lParam определяется сообщением и приведен в документации Windows для каждого сообщения. Также мы можем заметить, что функции SendMessage и PostMessage имеют несколько различные возвращаемые значения, но для наших дальнейших рассуждений это не столь важно. Главное различие между обеими функциями состоит в том, что они отправляют сообщение по-разному. Функция SendMessage отправляет сообщение непосредственно адресату (то есть при ее вызове, по сути, напрямую вызывается соответствующая оконная процедура), а функция PostMessage размещает сообщение в очереди сообщений. Соответственно, в зависимости от способа отправки сообщения делятся на синхронные (queued) и асинхронные (nonqueued). Очередь сообщений представляет собой FIFO-буфер. В ОС Windows существует системная очередь сообщений и отдельные очереди сообщений для каждого потока (с целью экономии системных ресурсов поток после инициализации не имеет очереди сообщений, она создается по мере надобности). Таким образом, передача сообщений в Windows осуществляется посредством взаимодействия множества очередей сообщений: системной очереди и очередей приложений. Передача сообщений может осуществляться от ОС к приложению, от приложения к ОС, от приложения к другим приложениям и внутри одного приложения. Более подробно с механизмом обмена сообщениями Windows можно ознакомиться в [1].

Очередь сообщений

В различных вариантах многопоточных систем программирования для микроконтроллеров можно встретить различные реализации механизма обмена сообщениями. Например, весьма популярная ОСРВ uC/OS II поддерживает два механизма обмена сообщениями: Message Mailbox (почтовый ящик сообщений) и Message Queue (очередь сообщений). Почтовый ящик фактически представляет собой переменную-указатель, содержащую ссылку на собственно сообщение. Очередь сообщений представляет собой буфер фиксированного размера, также содержащий указатели. Потоки и обработчики прерываний могут создавать и удалять очереди и почтовые ящики, размещать в них сообщения и получать их. В uC/OS II существует и специальный механизм, освобождающий очереди и ящики от сообщений, не принятых в течение определенного времени. Система предоставляет программисту полную свободу действий в создании ящиков и очередей сообщений: любой поток может создавать любое количество этих объектов, размер очередей и время «жизни» сообщений определяется программистом [2].

Другой подход к организации обмена сообщениями описан в статье «Get by without an RTOS» Майкла Мелкониана [3]. В рамках концепции, предлагаемой Мелконианом,

каждая задача имеет собственную очередь сообщений, представляющую собой простой FIFO-буфер. При этом само сообщение является переменной структурного типа, определяемой программистом для каждого потока отдельно. Поток может извлекать сообщения только из «своей» очереди и размещать сообщения в очередях любых других потоков. Очевидно, что такой подход имеет целый ряд недостатков. Мы не можем вызовом одной функции послать сообщения всем потокам сразу, нам придется посылать сообщения каждому потоку в отдельности. Если поток не «принял» сообщение, то есть не извлек его из очереди, оно останется там неограниченно долгое время, что может вызвать серьезные сбои в работе программы.

Возможен и другой способ обработки сообщений в системе. По аналогии с ОС Windows мы можем все посланные потоками сообщения накапливать в едином системном буфере, а затем передавать их (с помощью специального системного механизма — менеджера сообщений) очередям сообщений потоков. Эту задачу можно существенно облегчить (в плане затрат ресурсов микроконтроллера), если каждое сообщение будет иметь параметр — идентификатор потока, которому оно адресовано. Если же мы хотим передать сообщение всем потокам сразу, то можем выделить особый идентификатор, указывающий менеджеру сообщений, что данное сообщение следует скопировать во все локальные очереди потоков.

А сейчас зададим провокационный вопрос: а нужны ли вообще очереди сообщений? Не является ли их применение в ПО встроенных систем скорее результатом устоявшегося стереотипа, чем жизненной необходимостью? И в каких случаях их применение оправдано?

Рассмотрим этот вопрос на примере нашей концепции программирования. Все потоки в программе представлены в виде автоматов, которые «физически» реализованы в виде функций типа ProcessFSM (будем называть эти функции функциями автоматов). На каждой итерации главного цикла программы вызываются все функции автоматов. Таким образом, сообщение, отправленное любым автоматом, может быть принято любым другим автоматом за время, не превышающее полное время исполнения главного цикла программы. Соответственно, мы можем сделать вывод: во всяком случае, в рамках описываемой нами концепции очередь сообщений не нужна. Здесь может возникнуть два вопроса. Первый: что делать с сообщениями, не принятыми в течение цикла? Второй: если в течение цикла будет передано несколько сообщений с различными параметрами, то какое из них будет активно при приеме? Ответы на данные вопросы станут ясны из последующего обсуждения.

Простейшие сообщения: без очереди, без параметров

Во многих случаях передача сообщений с параметрами просто не нужна. При этом каждое сообщение представляет собой, по сути, флаг. Автомат, передающий сообщение, устанавливает соответствующий флаг, а автомат, принимающий сообщение, проверяет состояние этого флага. Очередь сообщений тоже не нужна, так как несколько флагов (то есть несколько различных сообщений) могут быть установлены одновременно без всякой очереди, а отсутствие у сообщений параметров приводит к тому, что держать в очереди два одинаковых сообщения бессмысленно. Разумеется, автомат, принимающий сообщение, должен его удалить, иначе оно останется активным навсегда. Теперь вспомним из предыдущей статьи, как реализован главный цикл программы:

```
// главный цикл программы
while(1)
{
    ProcessFSM1();          //итерация автомата FSM1
    ProcessFSM2();          //итерация автомата FSM2
    ProcessFSM3();          //итерация автомата FSM3
    ProcessMessages();      //обработка сообщений
};
```

Пусть автомат FSM1 передает некоторое сообщение автомату FSM3, а автомат FSM3 его принимает, то есть переходит под его действием в новое состояние. В этом случае все работает правильно, то есть флаг, соответствующий данному сообщению, устанавливается и сбрасывается в пределах одного цикла программы.

```
//»messages.h»
#ifndef MESSAGES_h
#define MESSAGES_h

#define MAX_MESSAGES 64

#define MSG_SOME_MESSAGE 0

void InitMessages(void);
void SendMessage(char Msg);
char GetMessage(char Msg);
#endif

//»messages.c»
char Messages[MAX_MESSAGES];

void InitMessages(void)
{
  char i;
  for(i = 0; i < MAX_MESSAGES; i++)
    Messages[i] = 0;
}

void SendMessage(char Msg)
{
  Messages[Msg] = 1;
}

char GetMessage(char Msg)
{
  if(Messages[Msg] == 1)
  {
    Messages[Msg] = 0;
    return 1;
  };
  return 0;
}
//автомат FSM1
void ProcessFSM1(void)
{
…
SendMessage(MSG_SOME_MESSAGE);
…
};

//автомат FSM3
void ProcessFSM3(void)
{
  switch(fsm3_state)
  {
    …
    case OLD_STATE:
    if(GetMessage(MSG_SOME_MESSAGE))
      {
        DoSomething();
        Fsm3_state = NEW_STATE;
      };
    break;
    case NEW_STATE:
    …
    break;
  };
}
```

В том случае если автомат FSM3 передает сообщение автомату FSM1, все тоже сработает как надо. Однако на практике такая реализация все же непригодна. И вот почему. Представим себе ситуацию, в которой автомат FSM1 передал сообщение, но оно не принято ни одним автоматом. Такая ситуация может произойти, например, в том случае, если пользователь нажал какую-либо кнопку микроконтроллерного устройства, автомат, контролирующий клавиатуру, послал сообщение MSG_KEY_PRESSED, но устройство в этот момент находится в состоянии, не предусматривающем какое-либо вмешательство пользователя, и, соответственно, ни один автомат не «ждет» сообщение MSG_KEY_PRESSED. В результате флаг, соответствующий MSG_KEY_PRESSED, останется установленным в единичное состояние до тех пор, пока по логике работы программы какой-либо автомат не попадет в состояние, условием выхода из которого является именно это сообщение, и этот переход сработает. Причем случиться это может когда угодно: через миллисекунду или через неделю. В первом случае пользователь, может быть, ничего и не заметит, зато во втором он будет немало удивлен. И в любом случае нормальная логика работы программы будет нарушена. Вот почему все необработанные сообщения-флаги должны сбрасываться в конце каждого цикла. Для этого и служит функция ProcessMessages(), вызываемая в главном цикле после вызовов функций ProcessFSM() всех автоматов. Но если мы будем просто удалять все сообщения, не принятые в течение рабочего цикла программы, например, вот так:

```
void ProcessMessages(void)
{
  char i;
  for(i = 0; i < MAX_MESSAGES; i++)
    Messages[i] = 0;
}
```

то нас ждет еще один сюрприз, а именно сообщения в этом случае смогут передаваться только сверху вниз по списку вызовов автоматов, но не в обратном направлении. Для того чтобы избежать такого неприятного

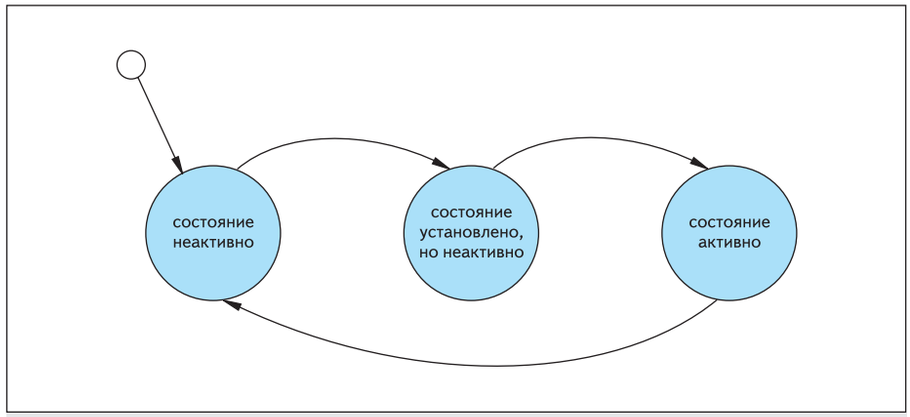

Рис. 3. Граф переходов сообщений

эффекта, нужно несколько усложнить механизм передачи сообщений — сделать его двухступенчатым. Пусть сообщение-флаг имеет не два состояния: «неактивно» и «активно», а три: «неактивно», «установлено, но неактивно» и «активно». Тогда сообщение, изначально находящееся в неактивном состоянии, при вызове функции SendMessage перейдет в состояние, «установлено, но неактивно», и автоматы, находящиеся ниже по списку, на него реагировать не будут. При достижении программой конца цикла сообщение переходит в состояние «активно» и находится в этом состоянии в следующем цикле программы до его сброса принявшим его автоматом либо до конца цикла (рис. 3). Такой подход совпадает с принятым в [4]: «все сообщения должны быть обработаны на следующем шаге после генерации».

Приведем и программную реализацию такого механизма:

```
// «messages.c»

char Messages[MAX_MESSAGES];

void InitMessages(void)
{
   char i;
   for(i = 0; i < MAX_MESSAGES; i++)
        Messages[i] = 0;
}

void SendMessage(char Msg)
{
   Messages[Msg] = 1;
}

void ProcessMessages(void)
{
   char i;
   for(i = 0; i < MAX_MESSAGES; i++)
   {
        if(Messages[i] == 2) Messages[i] = 0;
        if(Messages[i] == 1) Messages[i] = 2;
   }
}

char GetMessage(char Msg)
{
   if(Messages[Msg] == 2)
   {
        Messages[Msg] = 0;
        return 1;
   };
   return 0;
}
```

У такой реализации, тем не менее, есть одно свойство: принять сообщение может только один автомат, причем находящийся выше по списку вызовов функций ProcessFSM(). Однако это не является большой проблемой. В большинстве случаев в каждый конкретный момент времени каждое конкретное сообщение и должен принимать какой-либо конкретный автомат. К примеру, если программа реализует пользовательский интерфейс устройства, то сообщения, передаваемые автоматом-менеджером клавиатуры, должен принимать только автомат, отвечающий за элемент интерфейса, находящийся в фокусе ввода. Если же необходима передача сообщения сразу всем автоматам программы (назовем такие сообщения широковещательными, или broadcast messages), то и такой вариант вполне реализуем:

```
// «messages.c»

char Messages[MAX_MESSAGES];
char BroadcastMessages[MAX_BROADCAST_MESSAGES];

void InitMessages(void)
{
   char i;
   for(i = 0; i < MAX_MESSAGES; i++)
        Messages[i] = 0;
   for(i = 0; i < MAX_BROADCAST_MESSAGES; i++)
        BroadcastMessages[i] = 0;
}

void SendMessage(char Msg)
{
   Messages[Msg] = 1;
}

void SendBroadcastMessage(char Msg)
{
   BroadcastMessages[Msg] = 1;
}

void ProcessMessages(void)
{
   char i;
   for(i = 0; i < MAX_MESSAGES; i++)
   {
        if(Messages[i] == 2) Messages[i] = 0;
        if(Messages[i] == 1) Messages[i] = 2;
   }
   for(i = 0; i < MAX_BROADCAST_MESSAGES; i++)
   {
        if(BroadcastMessages[i] == 2) BroadcastMessages[i] = 0;
        if(BroadcastMessages[i] == 1) BroadcastMessages[i] = 2;
   }
}

char GetMessage(char Msg)
{
   if(Messages[Msg] == 2)
   {
        Messages[Msg] = 0;
        return 1;
   };

   return 0;
}

char GetBroadcastMessage(char Msg)
{
   if(Messages[Msg] == 2)
        return 1;
   return 0;
}
```

Передача параметров: два варианта

Однако в ряде случаев без параметров не обойтись. Простейшим примером является та же обработка сообщений клавиатуры: нам мало знать, что пользователь нажал на клавишу, нам нужно знать, на какую именно клавишу он нажал. Конечно, можно присвоить отдельное сообщение каждой клавише, например, так:

```
//»messages.h»
#ifndef MESSAGES_h
#define MESSAGES_h

#define MAX_MESSAGES 64

#define MSG_KEY_0          0
#define MSG_KEY_1          1
…
#define MSG_KEY_9          9
#define MSG_KEY_LEFT       10
#define MSG_KEY_RIGHT      11
#define MSG_KEY_UP         12
#define MSG_KEY_DOWN       13
…
```

Но тогда можно представить себе, во что выльется условие типа «пользователь нажал любую клавишу»:

```
if(GetMessage(MSG_KEY_0) || GetMessage(MSG_KEY_1) || … ||
GetMessage(MSG_KEY_DOWN))…
```

Как-то не очень красиво, правда? Гораздо лучше иметь только одно сообщение MSG_KEY_PRESSED, а код конкретной клавиши получать с помощью специальной функции:

```
if(GetMessage(MSG_KEY_PRESSED))
   switch (GetKeyCode())
   {
        case KEY_LEFT:
             //обработка нажатия LEFT
             break;
        case KEY_RIGHT:
             //обработка нажатия RIGHT
             break;
        case KEY_UP:
             //обработка нажатия UP
             break;
        case KEY_DOWN:
             //обработка нажатия DOWN
             break;
   };
```

В данном случае передача параметра (кода клавиши) отделена от механизма трансляции сообщений. Код клавиши хранится в отдельной переменной в модуле сканирования клавиатуры, заполняется автоматом модуля сканирования непосредственно перед передачей сообщения MSG_KEY_PRESSED и доступен для получателя по вызову функции GetKeyCode. Такой способ передачи параметров прост, но можно и передавать параметры непосредственно вместе с сообщениями. Все, что для этого нужно: заменить в вышеприведенном модуле messages.c строку

```
char Messages[MAX_MESSAGES];
```

на:

```
MSG_DATA Messages[MAX_MESSAGES];
```

где MSG_DATA может быть определено, например, как:

```
typedef struct
{
   char Msg;
   void *ParamPtr;
}
```

и, соответственно, скорректировать функции InitMessages, SendMessage, ProcessMessages, GetMessage.

Теперь мы можем ответить на второй из вопросов, заданных выше: что происходит с параметрами в том случае, если в одном цикле работы программы передано два сообщения с различными параметрами? Активным станет, разумеется, второе сообщение: новые значения параметров просто затрут установленные ранее. Однако, как правило, такая ситуация совершенно приемлема.

У внимательного читателя может возникнуть следующий вопрос: как сообщения передаются в главный цикл программы из обработчиков прерываний? Не могут ли при этом возникать какие-либо коллизии? С обсуждения этого вопроса мы начнем следующую статью.

Литература

1. Программирование для Windows 95; 2 т. СПб.: BHV — Санкт-Петербург, 1997.
2. MicroC/OS II: The Real Time Kernel. Jean J. Labrosse, CMP Books, 2002.
3. Get by Without an RTOS. Michael Melkonian.// Embedded System Programming, 2000, vol. 13, N 10.
4. Гуисов М. И., Кузнецов А. Б., Шалыто А. А. Интеграция механизма обмена сообщениями в Switch-технологию. СПб.. 2003.

Применение SWITCH-технологии при разработке прикладного программного обеспечения для микроконтроллеров. Часть 3

В предыдущей статье цикла был рассмотрен ряд вопросов, связанных с передачей сообщений между автоматами. Мы выяснили, что в ряде случаев (а на самом деле в большинстве) можно обойтись без очередей сообщений — если использовать гораздо более простой механизм их обработки. Однако за рамками материала остался ряд нюансов, в частности, вопрос организации обмена сообщениями между главным циклом программы и обработчиками прерываний. Данная статья посвящена проблемам, возникающим при обмене сообщениями, и другому важному аспекту предлагаемой технологии программирования. а именно таймерам.

Владимир ТАТАРЧЕВСКИЙ
arktur04@mail.ru

Обмен сообщениями между автоматами: заключение

Перед тем как приступить к обсуждению вопроса, вынесенного в подзаголовок, подведем краткие итоги материала, изложенного в предыдущей статье цикла.

Итак, мы условно делим сообщения на два типа: «обычные» (далее — просто сообщения) и широковещательные. Широковещательное сообщение может быть принято неограниченным количеством автоматов в цикле, в то время как обычное сообщение — только одним автоматом, после чего оно удаляется (сбрасывается в неактивное состояние). Реализация широковещательных сообщений довольно проста, но обычным сообщениям мы уделим еще немного нашего внимания, отметив ряд нюансов.

Нюанс первый: на программисте лежит обязанность организовать программу таким образом, чтобы в каждый конкретный момент времени каждое сообщение могло приниматься только одним автоматом в цикле (или не приниматься ни одним). Несоблюдение этого правила ведет к следующему. Допустим, некоторое сообщение MSG_SOME_MESSAGE принимается двумя автоматами FSM1 и FSM3 в одном цикле программы. При этом первый автомат, приняв сообщение, сбросит его, а второй автомат, соответственно, его не примет. Очевидно, что в данном случае поведение программы будет определяться последовательностью вызовов автоматов в главном цикле, если автоматы будут вызываться в последовательности FSM3, … FSM1, поведение программы станет другим. Между тем одним из основных свойств предлагаемой концепции является именно независимость поведения программы от порядка вызовов функций автоматов (потоков).

Итак, каждое сообщение представлено конечным автоматом (КА), имеющим три состояния:

- Состояние 0: состояние неактивно.
- Состояние 1: состояние установлено, но неактивно.
- Состояние 2: состояние активно.

Автомат, посылающий сообщение, фактически переводит соответствующий КА из состояния 0 в состояние 1, а в конце цикла программы функция ProcessMessages переводит все сообщения, находящиеся в состоянии 1, в состояние 2. Таким образом, сообщение является активным с начала цикла, следующего после его генерации, до момента его приема либо до конца цикла, если оно не было принято. Также функция ProcessMessages сбрасывает все не принятые сообщения в конце цикла (переход 2–0).

Благодаря такому разделению «жизненного цикла» сообщения на фазы состояния, отладка программы становится простым и удобным процессом, так как мы можем постоянно отслеживать в отладчике, какой автомат и в каком состоянии сгенерировал сообщение, а какой его принял. Процесс отладки можно сделать еще более удобным, если организовать передачу отладочной информации из функций GetMessage и SendMessage через, например, порт RS-232, тогда мы сможем получить протокол работы программы на компьютере.

Существуют еще некоторые нюансы, связанные с обработкой сообщений. Допустим, мы записали условия переходов из некоторого состояния следующим образом:

```
…
case OLD_STATE:
    if(GetMessage(MSG_SOME_MESSAGE) && GetInput(IN0))
    {
      DoSomething();
      Fsm_state = NEW_STATE;
    };
    if(GetMessage(MSG_SOME_MESSAGE) && GetInput(IN1))
    {
      DoSomethingElse();
      Fsm_state = ANOTHER_STATE;
    };
    break;
```

Здесь должны выполняться следующие условия переходов: если принято сообщение MSG_SOME_MESSAGE и выполнено некоторое дополнительное условие GetInput(IN0), то автомат выполняет действие DoSomething() и переходит в состояние NEW_STATE, а при выполнении дополнительного условия GetInput(IN1) выполняется действие DoSomethingElse() и осуществляется переход в состояние ANOTHER_STATE. Однако ясно, что второе условие никогда не сработает, так как сообщение MSG_SOME_MESSAGE будет сброшено первой проверкой перехода

при вызове GetMessage, вне зависимости от выполнения дополнительного условия GetInput(IN0). Поэтому так писать нельзя. Правильной реализацией переходов с такими условиями будет следующий код:

```
…
case OLD_STATE:
      if(GetMessage(MSG_SOME_MESSAGE))
      {
         if(GetInput(IN0))
         {
            DoSomething();
            Fsm_state = NEW_STATE;
         };
         if(GetInput(IN1))
         {
            DoSomethingElse();
            Fsm_state = ANOTHER_STATE;
         };
      };
      break;
```

И, разумеется, нельзя использовать условия переходов типа следующего:

```
…
case OLD_STATE:
      if(GetMessage(MSG_SOME_MESSAGE) &&
GetMessage(MSG_ANOTHER_MESSAGE))
      {
         DoSomething();
         Fsm_state = NEW_STATE;
      };
      break;
```

Другими словами, никогда нельзя исходить из предположения, что два разных сообщения поступят одновременно, в одном цикле.

Особенностью предлагаемой модели взаимодействия конечных автоматов является то, что они в определенном смысле синхронны, и если автомат генерирует сообщение, мы можем с уверенностью утверждать, что его сможет принять любой автомат в следующем цикле работы программы. Но в том случае, если мы передаем сообщение из главного цикла программы в обработчик прерывания, мы не можем быть уверены, что прерывание наступит в течение следующей итерации главного цикла, или в течение определенного времени, или вообще когда-либо. Поэтому такие передачи сообщений нужно применять с крайней осторожностью. Автору представляется, что в случае, если необходимо управление процессами в обработчике прерывания из главного цикла, лучше пожертвовать чистотой концепции и обойтись обычным набором флагов. Передача сообщений в обратном направлении, из обработчика прерывания в главный цикл, допускает применение сообщений (а вот применение простых флагов здесь крайне нежелательно в силу того, что из значения могут изменяться обработчиком прерываний асинхронно, в самые непредсказуемые моменты). Однако в силу асинхронности процессов в такой передаче сообщений скрываются некоторые «подводные камни». Представим себе такую ситуацию: обработчик прерывания послал сообщение MSG_SOME_MESSAGE автомату FSM3 в главном цикле программы. При этом КА, соответствующий данному сообщению, перешел в состояние 1. В конце цикла функция ProcessMessages перевела его в состояние 2 (активное состояние), в котором он должен находиться до окончания следующего цикла либо до приема сообщения автоматом FSM3. Теперь представим себе, что обработчик прерывания снова послал это же сообщение до его приема автоматом. В результате оно снова переводится в неактивное состояние и не может быть принято автоматом. Если обработчик прерывания будет повторять отправку сообщения с тем же периодом, с которым выполняется главный цикл программы, или чаще, сообщение никогда не перейдет в активное состояние и, соответственно, никогда не будет принято. Выход из данного положения очень прост. Немного изменим функцию SendMessage, чтобы не допустить повторной активизации сообщения:

```
void SendMessage(char Msg)
{
  if(Messages[Msg] == 0)
     Messages[Msg] = 1;
}
```

На этом тему манипуляций с сообщениями пока оставим и перейдем к рассмотрению другого важного раздела — программных («виртуальных») таймеров.

Таймеры

Механизм таймеров может быть эффективно использован в SWITCH-программировании для эффективного описания условий переходов между состояниями автомата в том случае, если переход должен произойти по истечении определенного промежутка времени.

Под таймером мы будем подразумевать не аппаратное устройство микроконтроллера, а «виртуальный» объект, имеющий уникальный идентификатор и представляющий, по сути, переменную, увеличивающую свое значение на единицу через определенные интервалы времени. Этот интервал определяется настройками аппаратного таймера. Все «виртуальные» таймеры программы обслуживаются одним прерыванием аппаратного таймера.

```
#define MAX_TIMERS 16                    //количество таймеров
unsigned int timers[MAX_TIMERS];         //переменные
                                         //«виртуальных»
                                         //таймеров

void ProcessTimers(void)                 //обработчик прерывания
                                         //таймера/счетчика
{
 char i;
 unsigned int dummy;                     //вспомогательная
                                         //переменная
 dummy = AT91C_BASE_TC0->TC_SR; //читаем флаг статуса
                                         //прерывания
 //-------------------------------------------------
 //увеличиваем значение всех переменных-таймеров на 1
 //-------------------------------------------------
 for(i = 0; i < MAX_TIMERS; i++)
 timers[i]++;
}
```

Полную процедуру инициализации аппаратного таймера мы здесь приводить не будем, так как она зависит от типа применяемого микроконтроллера и реализуется для каждого типа микроконтроллеров по-разному (вы можете обратиться к фирменной документации, в которой вопросы подобного рода, как правило, рассматриваются очень подробно).

Инициализация виртуальных таймеров состоит в их обнулении:

```
void InitTimers(void)
{
   char i;
   for(i = 0; i < MAX_TIMERS; i++)
   timers[i] = 0;
   //инициализация аппаратного таймера
   timer_init();
}

   Введем еще две функции — GetTimer и ResetTimer:
   unsigned int GetTimer(char Timer)
{
   return Timers[Timer];
}

void ResetTimer(char Timer)
{
   Timers[Timer] = 0;
}
```

Функция GetTimer позволяет получить текущее значение таймерной переменной, а ResetTimer сбрасывает таймер в начальное (нулевое) состояние. Заголовочный файл модуля таймеров может выглядеть, например, следующим образом:

```
#ifndef TIMERS_h
#define TIMERS_h
//------------------------------
//определения единиц времени
#define sec 100 //период таймера 10 мс, т.е. 1 с соответствует 100
периодам
#define min 60 * sec
#define hour 60 * min
#define day 24 * hour

#define MAX_TIMERS 16 //максимальное количество таймеров
//в этом разделе объявляются константы, служащие идентификаторами таймеров.
#define KEYB_TIMER      0
#define CURSOR_TIMER    1
#define FLASH_TIMER     2
#define MENU_TIMER      3
#define UPDATE_TIMER    4

//функции работы с таймерами
void InitTimers(void);

unsigned int GetTimer(char Timer);

void ResetTimer(char Timer);

#endif
```

В следующей статье мы продолжим рассмотрение механизма действия виртуальных таймеров согласно SWITCH-технологии и приведем конкретные примеры их применения.

Автор благодарит Анатолия Абрамовича Шалыто за поддержку при написании данного цикла статей.

Применение SWITCH-технологии при разработке прикладного программного обеспечения для микроконтроллеров. Часть 4

Владимир ТАТАРЧЕВСКИЙ
arktur04@mail.ru

В предыдущих статьях цикла [1–3] мы подробно обсудили организацию обмена сообщениями в программном обеспечении, построенном на основе SWITCH-технологии, и начали рассмотрение механизма таймеров. В этом материале мы продолжим обсуждение таймеров, а также приведем пример реализации SWITCH-программы на конкретном примере.

Как уже упоминалось в предыдущей статье [3], под таймером мы будем понимать не аппаратный узел микроконтроллера, а программную конструкцию (виртуальный таймер), позволяющую ввести в программу, основанную на SWITCH-технологии, условия переходов, зависящие от времени. При этом мы будем подразделять таймеры на локальные и глобальные. Локальным таймером будем считать такой таймер, действие которого ограничено одним состоянием конечного автомата, и мы можем использовать его значение для описания условий переходов только данного состояния. Область действия глобального таймера распространяется на несколько состояний автомата. Программа может содержать множество локальных и глобальных таймеров, но все они управляются прерыванием единственного аппаратного таймера микроконтроллера.

Рассмотрим вопросы, связанные с реализацией и применением таймеров.

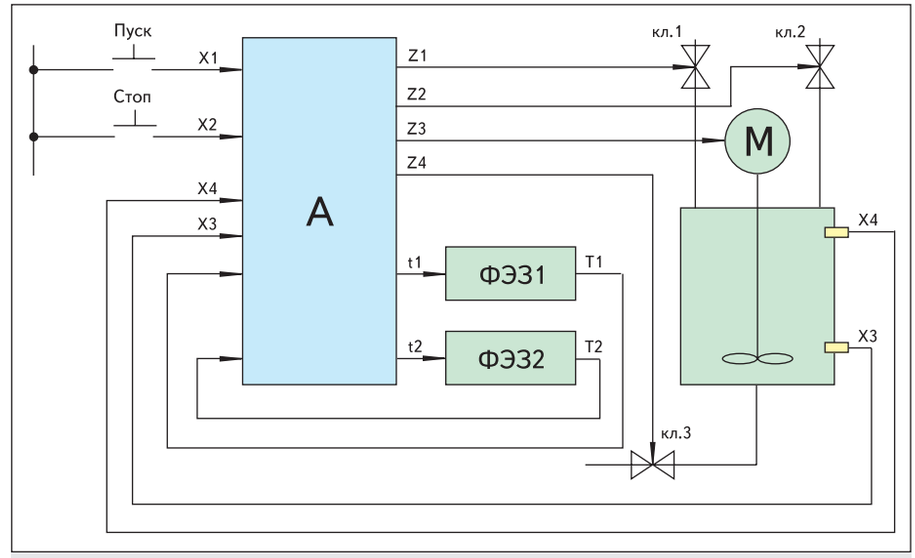

Рис. 1. Функциональная схема технологической установки: А — управляющий автомат; ФЭЗ1, ФЭЗ2 — функциональные элементы задержки; М — мотор; х3, х4 — датчики уровня жидкости; кл1, кл2, кл3 — клапаны

Локальные таймеры

Локальным таймером в предлагаемой модели является выделенная переменная, увеличивающее свое значение через фиксированные интервалы времени (например, каждую секунду). При входе в состояние значение этой переменной обнуляется (таймер инициализируется). В результате появляется возможность указывать в качестве условия выхода из состояния определенное значение данной переменной, соответствующее необходимому интервалу времени. Это позволяет упростить граф переходов конечного автомата. Наряду с локальными таймерами возможно также применение и глобальных таймеров, которые инициализируются в одном, а являются условиями выхода — в другом состоянии (состояниях).

Рассмотрим автомат, использующий таймеры. В качестве примера возьмем технологическую установку, описанную в работе [4]. Ее функциональная схема приведена на рис. 1.

Приведем словесное описание работы системы:

1. В исходном состоянии все клапаны закрыты, мотор (двигатель) «смешивателя» выключен.
2. Этап 1. При нажатии кнопки «Пуск» осуществляется налив жидкости путем открытия клапана 1 до срабатывания датчика х3.
3. Этап 2. Осуществляется налив жидкости путем открывания клапана 2 до срабатывания датчика х4.
4. Этап 3. Включается двигатель «мешалки» на время t1.
5. Этап 4. Производится слив смеси путем открывания клапана 3 на время t2.
6. После выполнения пункта 5 система возвращается в исходное состояние.
7. При нажатии кнопки «Стоп» во время выполнения пунктов 2–4 система переходит к сливу смеси (пункт 5).

Построим по словесному описанию автомат Мура, использующий локальные таймеры (рис. 2).

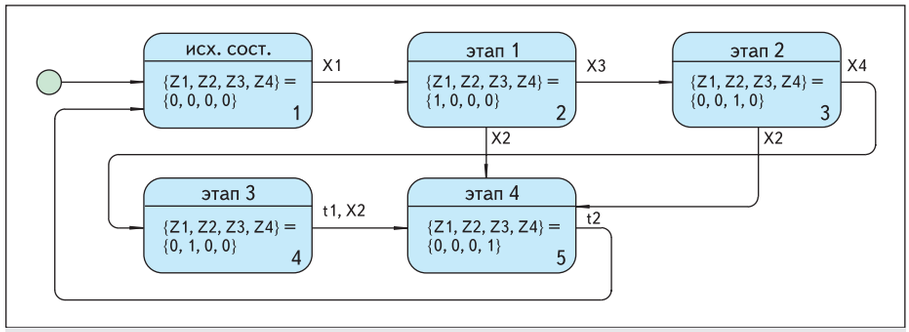

Рис. 2. Граф конечного автомата системы управления

Заметим, что граф переходов на рис. 2 состоит всего из пяти состояний. При этом за счет введения локального таймера (всего одной переменной на весь автомат) мы избавились от «искусственных» входов T1, T2, выходов t1, t2 и двух элементов задержки — ФЭЗ1 и ФЭЗ2. Можно заметить, что у данной схемы есть один недостаток — из всех «рабочих» состояний, кроме последнего, (из состояний 2–4) в состояние 5 ведут переходы с условием x2, соответствующим нажатию кнопки «Стоп». С учетом того, что в сложных технологических установках число состояний автомата в рабочем цикле может достигать нескольких десятков (и даже сотен), такие переходы могут сильно загромождать схему, затрудняя ее читаемость. Также подобный «прямой» подход может привести к ошибкам при разработке программы и особенно при ее модификации (например, из какого-либо состояния забыли провести переход по кнопке «Стоп»). Теперь представим себе, что в процессе разработки было решено использовать в качестве кнопки «Стоп» не нормально-разомкнутую, а нормально-замкнутую систему (как это и имеет место в стандартных двухкнопочных постах «пуск-стоп»). Для этого необходимо произвести в соответствующих переходах из состояний 2–4 замену x2 на ~x2 (x2 инверсное). При этом небольшое, казалось бы, изменение является «ошибкоопасным», так как замену придется производить во всех рабочих состояниях. Поэтому целесообразно преобразовать граф переходов на рис. 2 в схему с параллельными потоками, изображенную на рис. 3.

Несмотря на то, что количество состояний в новой схеме увеличилось до шести, число переходов в ней уменьшилось, а ее наглядность возросла. И, что самое главное, возросла модифицируемость схемы. Теперь можно ввести любое количество состояний рабочего цикла вместо состояний 2–4, не заботясь о корректности срабатывания кнопки «Стоп», а замена условия останова системы (например, замена x2 на ~x2) влечет за собой изменение только в одном месте программы.

Рассмотрим реализацию данного автомата (рис. 3) на языке C:

```
while(1) {
//Здесь должен производиться опрос входов системы управления

//основной поток программы (состояния 1–5 на рис. 3)
 switch(state){
  case 1: //состояние 1
       //устанавливаем выходы автомата
       z1 = z2 = z3 = z4 = 0;
       //условие перехода
       if(x1) state = 2;
       break;
  case 2: //состояние 2
       //устанавливаем выходы автомата
       z1 = 1;
       z2 = z3 = z4 = 0;
       //условие перехода
       if(x3) state = 3;
       break;
  case 3: //состояние 3
       //устанавливаем выходы автомата
       z1 = z2 = z4 = 0;
       z3 = 1;
       //условие перехода
       if(x4) state = 4;
       break;
  case 4: //состояние 4
       //устанавливаем выходы автомата
       z1 = z3 = z4 = 0;
       z2 = 1;
       //запускаем локальный таймер
       starttimer();
       //условие перехода
       if(GetTimer(TimerID) >= t_1) state = 5;
       break;
  case 5: //состояние 5
       //устанавливаем выходы автомата
       z1 = z2 = z3 = 0;
       z4 = 1;
       //запускаем локальный таймер
       starttimer();
       //условие перехода
       if(GetTimer(TimerID) >= t_2) state = 1;
       break;
 };
//контроль кнопки «стоп»
/*этот поток состоит из одного состояния и выполняется параллельно основному потоку программы. При активации входа x2 (кнопка «стоп»), этот поток принудительно переводит основной КА в состояние 5 (слив смеси и завершение работы).*/
 if((state > 1)&&(state < 5) && x2)
   state = 5;
//Здесь должна производиться установка выходов системы управления

};
```

Следует отметить, что в данном исходном тексте приведена скорее «условная» реализация, чем реальный код, в нем опущены многие подробности реализации, такие как объявления переменных и функций, опрос входов автомата и выдача воздействий на выходы.

Однако на примере данного исходного текста мы еще раз можем убедиться в преимуществе SWITCH-технологии перед традиционным подходом. В самом деле, SWITCH-программа всегда выполняется циклически, и на каждом цикле мы считываем состояние входных линий контроллера и устанавливаем выходные сигналы контроллера. Программа нигде не имеет «петель», вложенных циклов ожидания того или иного события, которые помешали бы нам обеспечить корректную работу параллельных потоков и функций обслуживания ввода/вывода.

Вернемся к таймерным условиям переходов. На рис. 3 из состояний 4 и 5 ведут переходы с пометками t1 и t2 соответственно. Это означает, что переход происходит по истечении времени t после *входа* автомата в состояние. Программно такое поведение реализуется достаточно просто:

```
case N;; //N — номер текущего состояния
   //---------------
   //здесь выполняем действие (деятельность) состояния
   //---------------
   starttimer(); //обнуляем таймер при входе в состояние
   if(GetTimer(TimerID) >= t) state = N_1; //если время t истекло, совершаем переход в состояние N_1
   break;
```

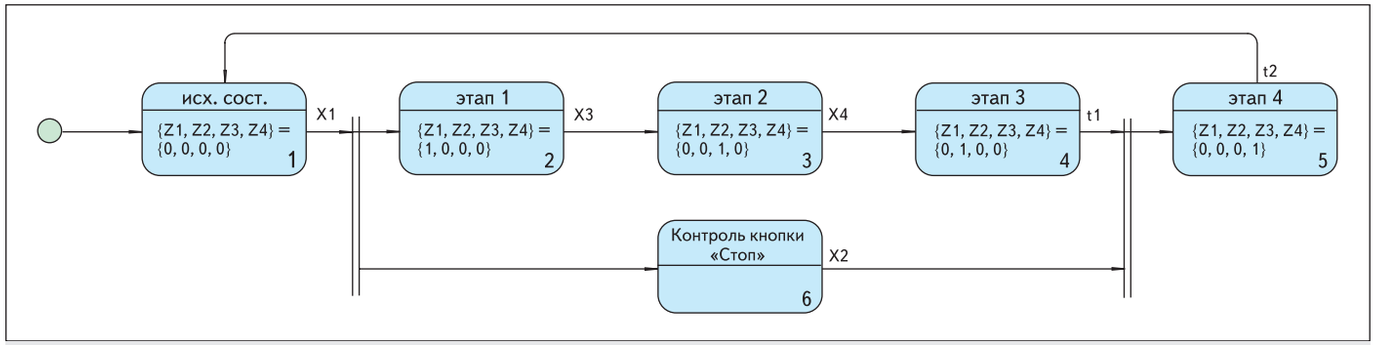

Рис. 3. Граф конечного автомата системы управления с параллельными потоками

Время t можно указывать как непосредственно в периодах аппаратного таймера (t >= 1234), так и в «естественных» единицах, объявив их в тексте программы:

```
#define sec 100 // период аппаратного таймера 10 мс
#define min 60 * sec
#define hour 60 * min
#define day 24 * hour
```

Тогда условие перехода может выглядеть, например, так: (GetTimer(TimerID) >= 5*min + 30*sec), что гораздо нагляднее. Разумеется, мы можем описывать и сложные условия, например:

```
If((GetTimer(TimerID) >= t) && GetInput(1)) state = N_1; //если время t истекло и вход 1 активен, совершаем переход в состояние N_1
```

Функция starttimer(), обнуляющая таймерную переменную, должна выполняться только при первом входе в состояние, что решается, например, следующим образом:

```
int state, _state; //state — переменная состояния автомата
//_state — вспомогательная переменная, предназначенная для определения факта входа автомата в состояние
void starttimer(char TimerID)
{
  if(state <> _state)
      ResetTimer(TimerID); // обнуляем таймерную переменную при смене состояния автомата
  _state = state;
}
```

Разумеется, мы должны инкрементировать переменную таймера через равные промежутки времени в обработчике прерывания *аппаратного* таймера микроконтроллера [3].

И наконец, следует заметить, что при использовании в программе нескольких параллельных автоматных потоков мы должны ввести отдельную таймерную переменную для каждого потока.

У приведенной программы есть один недостаток. В состояниях 2, 3 (рис. 3) программа ожидает срабатывания датчиков x3 и x4 соответственно. Если какой-либо из этих датчиков не сработает (например, по причине отсутствия жидкости в трубопроводе), программа будет находиться в соответствующем состоянии неограниченное время. Приведенную ранее программу можно несколько улучшить, если ввести переходы по времени из состояний 2, 3 (рис. 3) в некое состояние, которое можно назвать «аварийным». Вообще, локальные таймеры очень удобны для описания реакции системы на различные нештатные ситуации, для контроля исправности системы по времени выполнения каких-либо операций и т. п. И в данном случае SWITCH-технология демонстрирует явное преимущество перед «традиционным» подходом. Приведем цитату из [5]: «…алгоритм должен быть дополнен «нехорошими» условиями и, соответственно, действиями для выхода из «нехороших» ситуаций. Это отразить на схеме алгоритма, сильно не переделывая его, не просто… Такая схема алгоритма обычно очень громоздка, и поэтому она на практике почти никогда не используется: схему алгоритма один раз еще можно построить, но учитывать в ней *все* особенности реального алгоритма, а тем более вносить в нее изменения на всех этапах жизненного цикла программы не будет практически *никто*».

Изложенный подход к применению таймеров в SWITCH-программах имеет ряд очевидных преимуществ перед «прямым» использованием аппаратного таймера микроконтроллера. Во-первых, появляется возможность организации неограниченного количества «виртуальных» таймеров, управляемых одним аппаратным таймером (экономия ресурсов контроллера). Во-вторых, не требуется перепрограммировать аппаратный таймер для того, чтобы производить отсчет различных интервалов времени. В-третьих, в программе может производиться одновременный отсчет нескольких временных интервалов в разных потоках.

Как уже упоминалось ранее, локальные таймеры имеют ограниченную функциональность, с их помощью мы можем всего лишь описывать условия переходов по времени, прошедшему с момента входа автомата в состояние. Гораздо большей гибкостью обладают глобальные таймеры, которые будут рассмотрены в следующей статье цикла.

Автор выражает глубокую благодарность Анатолию Абрамовичу Шалыто за ценные замечания и редактирование статьи.

Литература

1. Татарчевский В. А. Применение SWITCH-технологии при разработке прикладного программного обеспечения для микроконтроллеров. Часть 1 // Компоненты и технологии. 2006. № 11.
2. Татарчевский В. А. Применение SWITCH-технологии при разработке прикладного программного обеспечения для микроконтроллеров. Часть 2 // Компоненты и технологии. 2006. № 12.
3. Татарчевский В. А. Применение SWITCH-технологии при разработке прикладного программного обеспечения для микроконтроллеров. Часть 3 // Компоненты и технологии. 2007. № 1.
4. Шалыто А. А. SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука. 1998.
5. Вавилов К. В., Шалыто А. А. Что плохого в неавтоматном подходе к программированию контроллеров? //Промышленные АСУ и контроллеры. 2007. № 1.

Применение SWITCH-технологии при разработке прикладного программного обеспечения для микроконтроллеров. Часть 5

Владимир ТАТАРЧЕВСКИЙ
arktur04@mail.ru

В предыдущих статьях цикла [1, 2] рассматривался механизм реализации таймеров в SWITCH-программах. В данной статье будет продолжено обсуждение реализации таймеров.

В статье [1] при рассмотрении механизма работы таймеров виртуальные таймеры в SWITCH-программе были условно разделены на два типа: локальные и глобальные. При этом локальным таймером был назван таймер, отсчитывающий время с момента входа в состояние[1] конечного автомата, который используется для определения условий выхода из данного состояния по истечении определенного интервала времени. Также было отмечено, что глобальный таймер, область действия которого охватывает несколько состояний, обладает гораздо большей гибкостью (рис. 1). Итак, рассмотрим пример применения глобального таймера и его программную реализацию.

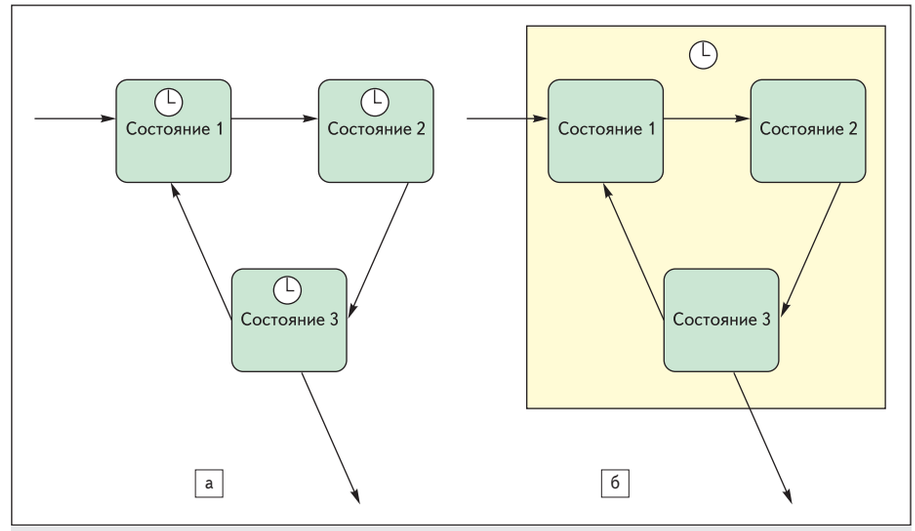

Рис. 1. Локальные и глобальные таймеры:
а) локальный таймер запускается в каждом состоянии при входе в него конечного автомата;
б) глобальный таймер является общим для группы состояний автомата

Глобальные таймеры

Глобальным будем называть виртуальный таймер, значение которого инициализируется в одном состоянии автомата, а применяется в качестве условия перехода — в другом. С его помощью можно придать временной детерминизм процессу, который управляется группой состояний автомата.

Для того чтобы понять, зачем нужны глобальные таймеры, рассмотрим в качестве примера систему управления установкой подачи воды в трубопровод. Функциональная схема установки подачи воды в трубопровод приведена на рис. 2. Вода поступает в бак через клапан налива, а затем поступает из бака в технологический трубопровод через клапан подачи.

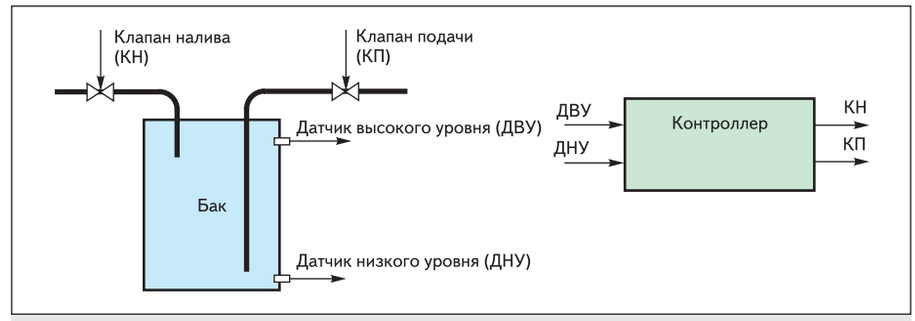

Рис. 2. Функциональная схема установки подачи воды в трубопровод

Система должна подавать в трубопровод воду в количестве (предположим) десяти объемов бака за один рабочий цикл. С целью экономии оборудования обойдемся без расходомера, а будем вычислять объем воды исходя из времени заполнения бака. При этом расход воды в единицу времени заранее неизвестен, но предполагается, что он существенно

[1] В каждое состояние автомата, разумеется (*прим. автора*).

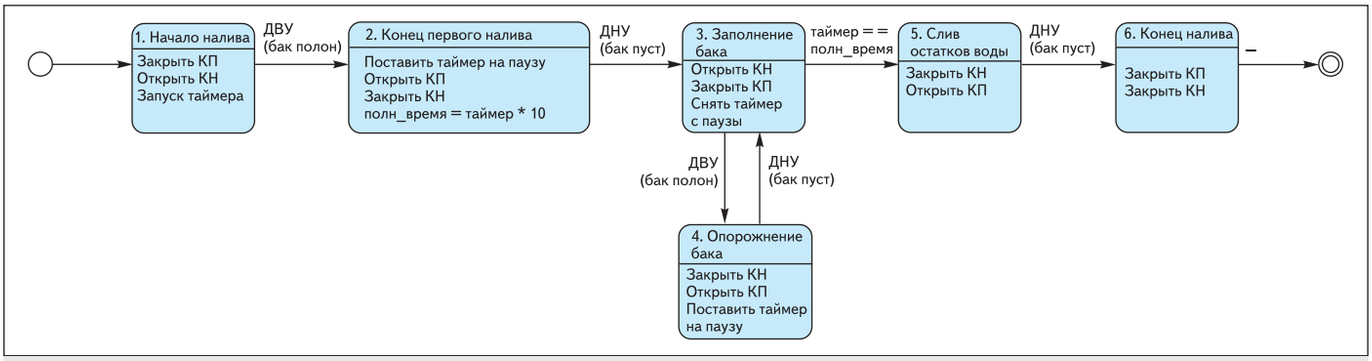

Рис. 3. Конечный автомат системы управления установкой подачи воды

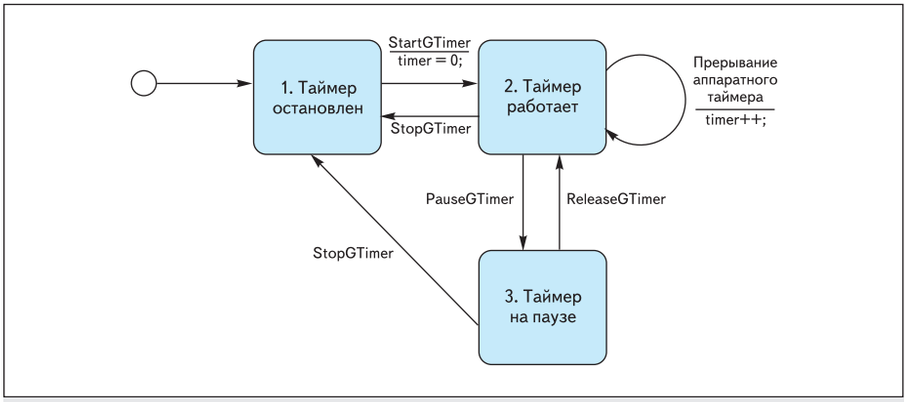

Рис. 4. Конечный автомат, описывающий глобальный таймер

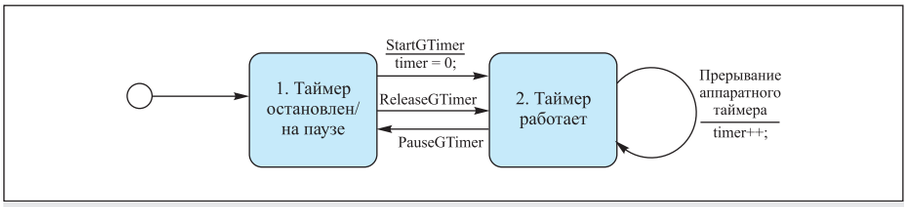

Рис. 5. Упрощенный конечный автомат глобального таймера

не изменяется за время одного рабочего цикла. Скорость заполнения бака легко измерить по времени его заполнения (от срабатывания датчика нижнего уровня до срабатывания датчика верхнего уровня) при отсутствии отбора воды из бака (клапан подачи закрыт). Пусть при этом скорость набора воды в бак выше скорости отбора воды из бака. Поэтому нельзя просто открыть клапан налива и отсчитать определенное время, так как бак переполнится. Следовательно, необходимо применить более гибкий подход.

Изобразим алгоритм работы установки в виде конечного автомата с глобальным таймером (рис. 3).

Цикл системы начинается с открывания клапана налива. При этом клапан подачи закрыт. Одновременно с началом рабочего цикла начинается отсчет времени таймером (состояние 1 «Начало налива»). Поступление воды продолжается до достижения верхнего уровня бака (состояние 2 «Конец первого налива»). После этого закрывается клапан налива, измеряется время заполнения бака, вычисляется полное время налива требуемого количества воды, открывается клапан подачи в трубопровод. После осушения бака автомат переходит в состояние «Продолжение налива». По мере заполнения бака и его опустошения автомат переключается между состояниями 3 «Заполнение бака» и 4 «Опорожнение бака». При этом отсчет времени происходит только при открытом клапане налива. По прохождении полного времени налива автомат переходит в состояние 5 «Слив остатка воды», а затем завершает работу (состояние 6 «Конец налива»).

На приведенном примере хорошо видны отличия глобального таймера от локального. Первое отличие, уже обсуждавшееся выше, заключается в том, что запуск таймера происходит в одном состоянии автомата, а его значение служит условием перехода в другое состояние (или несколькj состояниq). Второе отличие состоит в том, что отсчет времени глобальным таймером можно приостановить («поставить на паузу»)[2], а потом снова запустить. Благодаря этому мы можем контролировать время протекания не только непрерывных процессов, но и таких процессов, выполнение которых периодически приостанавливается (как набор воды в приведенном примере). Третья особенность глобального таймера состоит в том, что мы можем в любой момент времени прочитать его текущее значение и использовать в вычислениях (для локального таймера мы тоже можем прочитать его текущее значение, но так как он «существует» только в пределах одного состояния, это не так интересно с практической точки зрения).

Итак, поняв, что такое глобальный таймер и для чего он нужен, опишем его функционирование более формально. Для обеспечения работы с глобальными таймерами используем следующие функции:

```
void StartGTimer(unsigned int GTimerID); //Запуск таймера
void StopGTimer(unsigned int GTimerID); //Останов таймера
void PauseGTimer(unsigned int GTimerID); //Приостановка работы таймера
void ReleaseGTimer(unsigned int GTimerID); //Продолжение работы таймера
unsigned int GetGTimer(unsigned int GTimerID); //Получение текущего значения таймера
```

Теперь мы легко сможем описать поведение глобального таймера в виде конечного автомата (рис. 4).

Отметим, что на практике конечный автомат, приведенный на рис. 3, можно упростить, объединив состояния 1 и 3 и избавившись от функции StopGTimer (так как она становится эквивалентной функции PauseGTimer). Упрощенный таким образом конечный автомат изображен на рис. 5.

Перейдем к программной реализации механизма глобальных таймеров.

[2] Может быть, выражение «поставить таймер на паузу» несколько громоздко и не слишком удачно, но позволяет более четко разграничить состояние полного останова таймера, в котором он сброшен «в ноль», и состояние временной приостановки его работы, в котором он сохраняет текущее значение счетчика времени. Здесь можно провести аналогию с магнитофонной кассетой, которую можно перемотать к началу (перевести таймер в состояние останова) или нажать кнопку «пауза», а затем, через какое-то время, продолжить воспроизведение (*прим. автора*).

Программная реализация механизма глобальных таймеров

```
//------------------
// Модуль timers.h
//------------------
#ifndef TIMERS_h
#define TIMERS_h

.
.
.

#define MAX_GTIMERS 16 //Максимальное к-во глобальных таймеров в системе

//Идентификаторы глобальных таймеров
#define GTIMER1              0
#define GTIMER2              1
.
.
.
#define GTIMER15             11

//Функции работы с глобальными таймерами
void InitGTimers(void); //Инициализация глобальных таймеров
void StartGTimer(unsigned int GTimerID); //Запуск таймера
void StopGTimer(unsigned int GTimerID); //Останов таймера
void PauseGTimer(unsigned int GTimerID); //Приостановка работы таймера
void ReleaseGTimer(unsigned int GTimerID); //Продолжение работы таймера
unsigned int GetGTimer(unsigned int GTimerID); //Получение текущего значения таймера
.
.
.
#endif

//------------------
// Модуль timers.c
//------------------
#include «timers.h»

unsigned int GTimers[MAX_GTIMERS]; //В массиве хранятся текущие значения глобальных таймеров

//Состояния таймера
#define TIMER_STOPPED 0 //Таймер остановлен
#define TIMER_RUNNING 1 //Таймер работает
#define TIMER_PAUSED  2 //Таймер на паузе

char GTStates[MAX_GTIMERS]; //В массиве хранятся текущие состояния глобальных таймеров

void InitGTimers(void) //Инициализация глобальных таймеров
{
   char i;
   for(i = 0; i < MAX_GTIMERS; i++)
       GTStates[i] = TIMER_STOPPED;
}
void StartGTimer(unsigned int GTimerID) //Запуск таймера
{
   if(GTStates[GTimerID] == TIMER_STOPPED)
   {
       GTimers[GTimerID] = 0;
       GTStates[GTimerID] = TIMER_RUNNING;
   }
}
void StopGTimer(unsigned int GTimerID) //Останов таймера
{
   GTStates[GTimerID] = TIMER_STOPPED;
}
void PauseGTimer(unsigned int GTimerID) //Приостановка работы таймера
{
   if(GTStates[GTimerID] == TIMER_RUNNING)
       GTStates[GTimerID] = TIMER_PAUSED;
}
void ReleaseGTimer(unsigned int GTimerID) //Продолжение работы таймера
{
   if(GTStates[GTimerID] == TIMER_PAUSED)
       GTStates[GTimerID] = TIMER_RUNNING;
}
unsigned int GetGTimer(unsigned int GTimerID) //Получение текущего значения таймера
{
   return GTimers[GTimerID];
}

void ProcessTimers(void) //обработчик прерывания таймера/счетчика
{
   char i;
…
   for(i = 0; i < MAX_GTIMERS; i++)
       if(GTStates[i] == TIMER_RUNNING)
           GTimers[i]++;
…
}
```

Небольшое отступление. Как видно из приведенного выше листинга, программа, которая реализует управление глобальными таймерами, построена по графу конечного автомата, показанного на рис. 4, предназначена для использования в SWITCH-программах, однако сама по себе построена не по SWITCH-технологии. Это вызвано тем, что переходы между состояниями таймера происходят под воздействием вызовов функций и наиболее удобной в данном случае является реализация этих переходов непосредственно в функциях управления таймером, а не в едином операторе switch. Таким образом, мы несколько пожертвовали «чистотой концепции» ради простоты реализации.

Сейчас мы легко сможем построить программу, реализующую конечный автомат, изображенный на рис. 3.

```
char state; //переменная состояния автомата
char _state; //предыдущее состояние автомата
char entry; //флаг = 1 при входе автомата в новое состояние

unsigned int full_time;//полное время налива
void InitFSM(void)
{
   state = _state = 0;
   entry = 0;
}
void ProcessFSM(void)
{
   if(state != _state) entry = 1; else entry = 0;
   switch(state)
   {
      case 0: //Неактивное состояние
         if(GetMessage(MSG_ACTIVARE)) state = 1;
         break;
      case 1: //Начало налива
         if(entry) //при входе в состояние
            {
               StartGTimer(TIMER1);//Запускаем таймер
               SetOut(KP, off);//Закрываем клапан подачи
               SetOut(KN, on);//Открываем клапан налива
            };
         if(GetInput(DVU))//Если бак заполнен
            state = 2;//переходим к состоянию 2
         break;
      case 2: //Конец первого налива
         if(entry) //при входе в состояние
            {
               PauseGTimer(TIMER1); //Ставим таймер на паузу
               SetOut(KP, on);// Открываем клапан подачи
               SetOut(KN, off);// Закрываем клапан налива
               full_time = GetGTimer(TIMER1) * 10; //Вычисляем полное время налива
            };
         if(GetInput(DNU))//Если бак пуст
            state = 3;//переходим к состоянию 3
         break;
      case 3: //Заполнение бака
         if(entry) //при входе в состояние
            {
               ReleaseGTimer(TIMER1); //Снимаем таймер с паузы
               SetOut(KP, off);//Закрываем клапан подачи
               SetOut(KN, on);//Открываем клапан налива
            };
         if(GetInput(DVU))//Если бак заполнен
            state = 4;//переходим к состоянию 4
         if(GetGTimer(TIMER1) >= full_time)//Если время налива истекло
            state = 5;//переходим к состоянию 5
         break;
      case 4: //Опорожнение бака
         if(entry) //при входе в состояние
            {
               PauseGTimer(TIMER1); //Ставим таймер на паузу
               SetOut(KP, on);// Открываем клапан подачи
               SetOut(KN, off);// Закрываем клапан налива
            };
         if(GetInput(DNU))//Если бак пуст
            state = 3;//переходим к состоянию 3
         break;
      case 5: //Слив остатка воды
         if(entry) //при входе в состояние
            {
               SetOut(KP, on);// Открываем клапан подачи
               SetOut(KN, off);// Закрываем клапан налива
            };
         if(GetInput(DNU))Если бак пуст
            state = 6; переходим к состоянию 6
         break;
      case 6: //Конец налива и завершение работы
         if(entry) //при входе в состояние
            {
               SetOut(KP, off);//Закрываем клапан подачи
               SetOut(KN, off/ Закрываем клапан налива
            };
         state = 0;//Завершаем работу и переходим в неактивное состояние
         break;
   };
   state = state;
}
```

Сделаем некоторые пояснения к коду. В программе присутствует флаг entry, принимающий значение 1 на следующей итерации цикла после изменения автоматом состояния. Он нужен для того, чтобы действия в состояниях выполнялись однократно после входа автомата в состояние. В самом деле, если таймер будет сбрасываться на каждой итерации цикла, то он просто не будет работать! Для отслеживания изменения состояний служит вспомогательная переменная _state. После каждой итерации переменная _state приравнивается к переменной состояния автомата state.

О некоторых аспектах применения таймеров в автоматном программировании вы можете прочитать в [3–7].

На этом мы закончим обсуждение таймеров и перейдем к описанию некоторых практических примеров применения SWITCH-технологии.

Автор выражает глубокую благодарность Анатолию Абрамовичу Шалыто за ценные замечания и редактирование статьи.

Литература

1. Татарчевский В. А. Применение SWITCH-технологии при разработке прикладного программного обеспечения для микроконтроллеров. Часть 3 // Компоненты и технологии. 2007. № 1.
2. Татарчевский В. А. Применение SWITCH-технологии при разработке прикладного программного обеспечения для микроконтроллеров. Часть 4 // Компоненты и технологии. 2007. № 2.
3. Шахов В. Моделирование программно-аппаратных «реактивных» систем раскрашенными сетями Петри // RSDN Magazine. 2006. № 3.
4. Петриковский А. Субъектное программирование // Компьютера. 2006. № 13.
5. Козаченко В. Ф. Эффективный метод программной реализации дискретных управляющих автоматов во встроенных системах управления // **w ww.motorcontrol.r u**
6. Шалыто А. А., Туккель Н. И. SWITCH-технология — автоматный подход к созданию программного обеспечения «реактивных» систем // Программирование. 2001. № 5.
7. Альтерман И. З., Шалыто А. А. Формальные методы программирования логических контроллеров // Промышленные АСУ и контроллеры. 2005. № 10.

Применение SWITCH-технологии при разработке прикладного программного обеспечения для микроконтроллеров.

Часть 6. Реализация протокола Modbus

В предыдущих статьях цикла мы подробно рассмотрели ряд аспектов, связанных с применением SWITCH-технологии в микроконтроллерных устройствах. Были рассмотрены такие важные программные структуры, как механизм передачи сообщений, локальные и глобальные таймеры. Сейчас мы можем перейти к рассмотрению ряда распространенных алгоритмов на базе SWITCH-технологии, находящих применение в современных микроконтроллерных устройствах. В качестве первого примера рассмотрим реализацию широко распространенного в системах промышленной автоматики протокола Modbus.

Владимир ТАТАРЧЕВСКИЙ
arktur04@mail.ru

Краткое введение в протокол Modbus

Протокол Modbus получил широкое распространение в современных системах автоматики. Спецификация протокола не определяет тип физического уровня сети передачи данных, оставляя выбор за разработчиком. На практике широкое распространение получила связка, состоящая из сети RS-485 в качестве физической среды передачи данных и протокола Modbus в качестве логического уровня сети. Такое решение позволяет использовать Modbus в самом широком спектре микроконтроллерных устройств, начиная с самых простых, таких как различные интеллектуальные датчики, и заканчивая сложными распределенными системами на базе программируемых логических контроллеров.

Протокол построен по схеме «ведущий–ведомый» (master–slave). В системе выделяется одно ведущее устройство (мастер), которое инициализирует любую транзакцию в сети. Все остальные устройства являются ведомыми и выполняют команды мастера или передают информацию в ответ на запрос мастера. Протокол предусматривает два типа адресации: индивидуальную, когда сообщение адресуется одному ведомому устройству и широковещательную (broadcast messages), при которой сообщение адресуется всем устройствам сети. При индивидуальной адресации ведомое устройство возвращает мастеру ответное сообщение, при широковещательной адресации ответные сообщения не посылаются.

Сообщения, посылаемые мастером, имеют следующую структуру: адрес ведомого устройства (или код широковещательного сообщения), код, определяющий действия ведомого устройства, данные и контрольная сумма. Ответное сообщение состоит из поля, подтверждающего выполнение действия, данных и контрольной суммы. Если при приеме сообщения от мастера произошла ошибка, либо ведомое устройство по каким-либо причинам не может выполнить запрашиваемое действие, ведомое устройство посылает мастеру сообщение об ошибке.

Существует также расширенная версия протокола, называемая Modbus Plus. Протокол Modbus Plus формирует одноранговую сеть, в которой любое устройство может инициализировать транзакцию и, таким образом, каждое устройство может выступать в роли как ведущего, так и ведомого в разных транзакциях. Однако мы не будем рассматривать Modbus Plus в рамках данной статьи.

Сеть Modbus может работать в одном из двух режимов: RTU и ASCII. В режиме RTU информация передается «как есть», в двоичном коде. В режиме ASCII информация передается в текстовом виде как последовательность символов '0'–'9', 'A'–'F' в ASCII-кодировке.

Выбор режима RTU или ASCII, а также настройки сети, такие как скорость передачи, бит четности и т. п., выбираются пользователем при конфигурировании контроллеров и должны быть одинаковы для всех устройств в сети.

Рассмотрим оба режима протокола Modbus.

Режим ASCII

Главным преимуществом данного режима является то, что символы могут передаваться с интервалом вплоть до одной секунды без возникновения ошибки передачи данных. Недостатком данного режима является его сниженная (более чем в два раза) информационная пропускная способность по сравнению с режимом RTU при равной скорости физической линии. Передача символа осуществляется в следующем формате: 1 стартовый бит, 7 бит данных (младший разряд передается первым), 1 бит контроля четности (или нечетности; при отсутствии контро-

Таблица 1. Передача символа ASCII-кода

№ бита	0	1–7	8	9
С контролем четности	Стартовый бит	7 бит данных	Бит контроля четности	Стоповый бит
Без контроля четности	Стартовый бит	7 бит данных	Стоповый бит	Стоповый бит

Таблица 2. Структура сообщения в формате ASCII

Начало	Адрес	Код функции	Данные	Контрольная сумма	Конец
':' 3Ah	2 символа	2 символа	n символов	2 символа	CRLF 0D0Ah

Таблица 3. Передача символа ASCII-кода

№ бита	0	1–8	9	10
С контролем четности	Стартовый бит	8 бит данных	Бит контроля четности	Стоповый бит
Без контроля четности	Стартовый бит	8 бит данных	Стоповый бит	Стоповый бит

ля четности данный бит отсутствует), 1 стоповый бит, если контроль четности присутствует, 2 стоповых бита при отсутствии контроля четности (табл. 1).

Сообщение в данном режиме начинается с символа двоеточия (код 3Ah), заканчивается последовательностью символов «возврат каретки», «перевод строки» (CRLF, код 0D0Ah). Между передачей символов возможны интервалы времени до 1 секунды. При превышении тайм-аута принимающее устройство фиксирует ошибку передачи. Структура сообщения приведена в таблице 2.

В некоторых ранних реализациях протокола сообщение заканчивается контрольной суммой, без последовательности CRLF. Таким образом, принимающее устройство должно выждать как минимум 1 секунду после приема контрольной суммы, и если последовательность CRLF не получена, сообщение считается успешно принятым.

Режим RTU

Передача символа в режиме RTU состоит из следующих этапов: 1 стартовый бит, 8 бит данных (младший разряд передается первым), 1 бит контроля четности (нечетности), при отсутствии контроля четности данный бит отсутствует, 1 стоповый бит, если контроль четности присутствует, 2 стоповых бита при отсутствии контроля четности (табл. 3).

Сообщение представляет собой последовательность символов, передаваемых непрерывно, без пауз. При возникновении паузы длительностью более 1,5 t, где t — время передачи одного символа при заданной скорости передачи, принимающее устройство фиксирует ошибку в приеме сообщения и начинает прием нового сообщения после паузы. Структура сообщения приведена в таблице 4.

Таблица 4. Структура сообщения в формате RTU

Адрес	Код функции	Данные	Контрольная сумма
1 символ	1 символ	n символов	2 символа

Сообщение предваряется и заканчивается паузой не менее 3,5t при типичном значении 4t.

Адресация в протоколе Modbus

Значение адреса ведомого устройства находится в диапазоне 1–247. Широковещательные сообщения передаются с адресом 0.

Код команды Modbus

Код команды может принимать значение в диапазоне 1–255. Когда ведомое устройство отвечает на запрос мастера, оно использует код команды для индикации правильности выполнения команды. Если операция выполнена успешно, ведомое устройство просто повторяет код команды. Если во время выполнения операции по какой-либо причине произошла ошибка или операцию выполнить невозможно, ведомое устройство устанавливает старший бит команды в 1.

Поле данных в протоколе Modbus

Содержание и структура поля данных зависит от конкретной команды, и для ряда команд может вообще отсутствовать. Структуры данных для каждой конкретной команды описаны в спецификации протокола [1] и в данной статье не рассматриваются.

Контрольная сумма сообщения в протоколе Modbus

Контрольная сумма для режимов ASCII и RTU рассчитывается по разным алгоритмам. Алгоритм подсчета контрольной суммы для режима ASCII носит название LRC (Longitudinal Redundancy Check), результатом его работы являются 2 ASCII-символа, а для режима RTU контрольная сумма вычисляется по алгоритму CRC (Cyclical Redundancy Check), результатом работы которого является 16-битное число. Оба алгоритма подробно описываются в руководстве [1] и здесь не приводятся.

Реализация протокола Modbus

Рассмотрим реализацию протокола Modbus на основе SWITCH-технологии. Следует отметить, что преимуществом реализации алгоритма обмена данными в виде конечного автомата с применением SWITCH-технологии является возможность работы прикладной программы одновременно с приемом

Таблица 5. Структура буфера данных для передачи

№ поля	Наименование	Назначение поля	Размер поля, байт
1	Addr	Адрес ведомого устройства	1
2	Command	Код команды	1
3	Num_bytes	Количество байт данных	1
4	Data[i]	Данные	Определяется полем 3

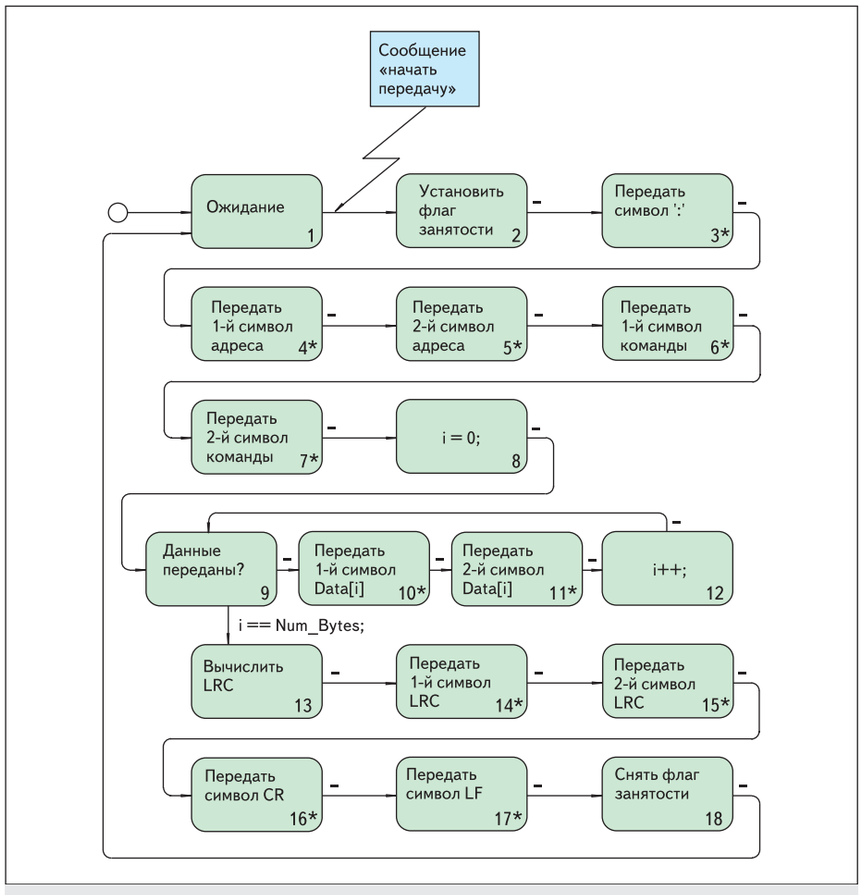

Рис. 1. Передача сообщения в режиме ASCII

и передачей сообщений, что сводит к сокращению до минимума задержки в работе прикладной программы на время приема или передачи сообщения, в особенности в режиме ASCII. Минимизация задержек при обмене данными имеет особенно важное значение при построении систем реального времени, таких как системы управления технологическими процессами.

Реализация передачи сообщен ия мастером в режиме ASCII

При передаче сообщения мастеров в режиме ASCII программа контроллера подготавливает данные, содержащие адрес устройства, код команды, данные команды и размер поля данных в байтах, и записывает их в выделенный буфер, имеющий структуру, которая представлена в таблице 5.

Указатель на буфер передается автомату Modbus в качестве параметра сообщения передачи. Во время передачи автомат Modbus устанавливает специальный флаг занятости, который должна проверять прикладная программа перед передачей. В случае, если флаг установлен, прикладная программа должна ждать его освобождения. Для преодоления данного ограничения возможно введение автомата, реализующего очередь сообщений и служащего промежуточным слоем между прикладной программой и автоматом Modbus. Граф автомата передачи сообщения в режиме ASCII изображен на рис. 1. Из рисунка видно, что символы передаются по одному, в различных состояниях, при этом прикладная программа (также написанная по SWITCH-технологии) может выполняться одновременно с передачей сообщения. Состояния, номера которых отмечены звездочкой, являются сложными состояниями (суперсостояниями), и имеют для данной граф-схемы одну структуру, показанную на рис. 2. Смысл введения суперсостояний в данном случае заключается в том, что UART может не успеть передать предыдущий символ, поэтому перед передачей очередного символа осуществляется проверка занятости UART. В силу свойств SWITCH-технологии остановки прикладной программы при этом не происходит. Единственным ограничением на работу цикла прикладной программы является то, что длительность одной итерации данного цикла не может превышать 1 секунды в соответствии со спецификацией протокола Modbus.

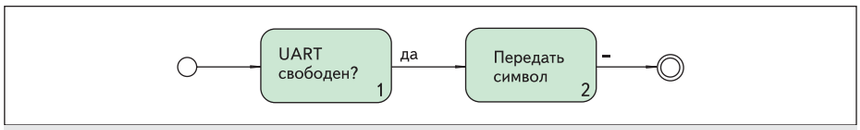

Рис. 2. Передача символа

В следующей статье мы продолжим рассмотрение реализации протокола Modbus на основе SWITCH-технологии.

Автор выражает глубокую благодарность Анатолию Абрамовичу Шалыто за ценные замечания и редактирование статьи.

Литература

1. Modicon Modbus Protocol Reference Guide. MODICON, Inc., Industrial Automation Systems, 1996.

Применение SWITCH-технологии при разработке прикладного программного обеспечения для микроконтроллеров. Часть 7

В предыдущей статье цикла [1] было начато рассмотрение реализации протокола Modbus на основе SWITCH-технологии. Мы кратко рассмотрели два режима работы протокола Modbus: ASCII и RTU, а также описали алгоритм передачи сообщения устройством-мастером в режиме ASCII. В этой статье мы продолжим описание реализации протокола Modbus на основе SWITCH-технологии.

Владимир ТАТАРЧЕВСКИЙ
arktur04@mail.ru

Обобщенный алгоритм обмена данными по протоколу Modbus

Обобщенный алгоритм обмена данными между ведущим и ведомым устройством можно представить в виде, изображенном на рис. 1. В предыдущей статье было приведено краткое описание работы протокола Modbus. Сформулируем порядок работы ведущего (master) и ведомого (slave) устройств в краткой форме.

Итак, ведущему устройству в процессе выполнения целевой программы (состояние 1.1 на рис. 1) потребовалось запросить данные удаленного модуля либо установить выходы удаленного модуля в определенное состояние. Ведущее устройство (далее *ведущий* или *мастер*) формирует сообщение Modbus, которое в общем случае содержит: адрес ведомого, код команды, дополнительные данные и контрольную сумму. Сформированное сообщение передается через UART микроконтроллера и физическую линию связи ведомому устройству (состояние 1.2). После передачи сообщения ведущий переходит в состояние ожидания ответа ведомого (состояние 1.3). Ведомое устройство, находящееся в состоянии ожидания команды ведущего (состояние 2.1) при поступлении данных в буфер UART переходит в состояние приема (состояние 2.2) и в случае успешного завершения приема переходит к декодированию и выполнению команды (состояние 2.3). После выполнения команды ведомое устройство формирует ответное сообщение, которое в общем случае содержит адрес ведомого устройства (то есть собственный адрес), код команды, дополнительные данные и контрольную сумму. Сформированное сообщение передается ведущему устройству (состояние 2.4), и после успешного приема ответа ведомого (состояние 1.4) ведущее устройство продолжает работу по основной программе, а ведомое переходит к ожиданию следующей команды. На этом транзакция Modbus завершается.

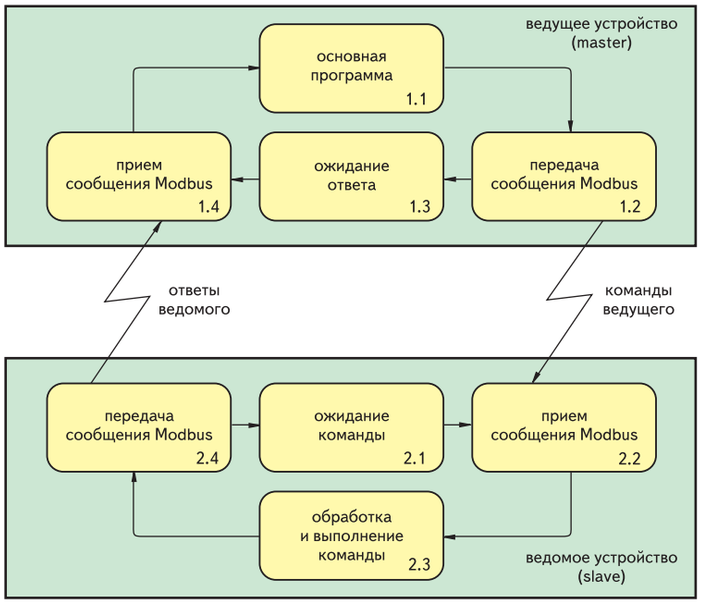

Рис. 1. Обобщенный алгоритм обмена данными между ведущим и ведомым устройствами

В предыдущей статье были упомянуты два режима работы протокола Modbus: ASCII и RTU. Напомним читателю, что в режиме ASCII сообщения передаются в виде текста, состоящего из чисел в шестнадцатеричной форме (символы '0'–'9', 'A'–'F'), сообщение начинается со «стартового» символа — двоеточия ':' и завершается «стоповой» последовательностью CR LF (возврат строки, перевод каретки, соответствующий код 0A 0D). При передаче допускаются паузы между символами продолжительностью до 1 с, в режиме RTU информация передается «как есть» в виде байтовых значений; специальные стартовые и стоповые символы отсутствуют; су-

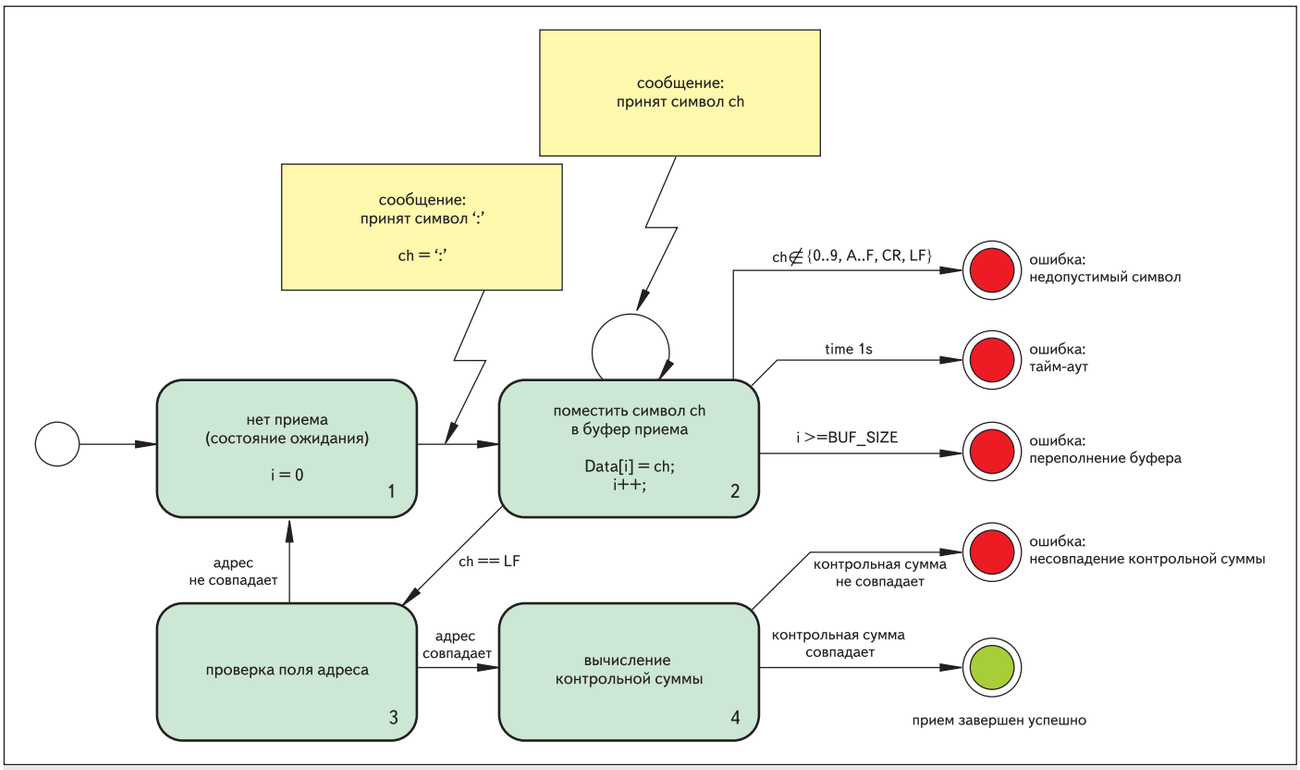

Рис. 2. Конечный автомат приема сообщения Modbus в режиме ASCII

щественные паузы между отдельными байтами сообщения недопустимы, сами же сообщения разделяются паузами длительностью не менее 3,5t, где t — время передачи одного байта при данной скорости UART [2].

Эти различия между режимами ASCII и RTU имеют важные следствия:

- **Следствие 1.** Так как для передачи одного байта информации в режиме ASCII требуется передать по линии связи два символа (то есть два байта), режим ASCII примерно в два раза медленнее, чем режим RTU (при одинаковой скорости UART).
- **Следствие 2.** В силу того, что в режиме ASCII существуют особые «стартовые» и «стоповые» символы, программа может фиксировать начало и конец передачи с гораздо более высокой степенью достоверности, чем в режиме RTU, где единственным признаком конца передачи служит небольшая пауза.
- **Следствие 3.** Так как в режиме RTU паузы между передачей отдельных байтов в сообщении недопустимы, это может вызвать некоторые трудности в программной реализации данного режима. Например, если передача сообщения прервана каким-либо обработчиком прерывания, устройство — приемник сообщения — сочтет эту паузу концом передачи со всеми вытекающими отсюда последствиями в виде несовпадения контрольной суммы, тогда всю транзакцию придется начинать заново. В результате в программу приходится вводить дополнительные механизмы, призванные обеспечить непрерывность передачи сообщения. В ряде случаев достаточно заблокировать прерывания на время передачи сообщения, если этого сделать нельзя, возможно отложить передачу на временной промежуток, в котором гарантированно не может возникнуть прерывание с длительной обработкой, способной нарушить процесс передачи сообщения.

Мы видим, что оба режима имеют свои преимущества и недостатки, и невозможно однозначно рекомендовать применение одного из них «на все случаи жизни». Наилучшим решением будет поддержка в разрабатываемом приборе обоих режимов протокола Modbus.

Вернемся к анализу рис. 1. Из него следует, что для полноценной поддержки Modbus в ведомом и ведущем устройстве нам необходимо реализовать два базовых алгоритма: передачу сообщения (состояния 1.2 и 2.4) и прием сообщения (состояния 1.4 и 2.2), причем каждый из них должен быть реализован в двух вариантах: для режима ASCII и для режима RTU. Алгоритм передачи сообщения ведущим устройством в режиме ASCII уже рассматривался в предыдущей статье, передача ответного сообщения ведомым устройством полностью аналогична, поэтому сейчас можно перейти к описанию приема сообщения в режиме ASCII (данный алгоритм также одинаков как для ведущего, так и для ведомого устройства). Конечный автомат, выполняющий прием сообщения Modbus в режиме ASCII, представлен на рис. 2. Его работа проста. В состоянии 1 обнуляется счетчик буфера I, и автомат ждет приема символа ':' (символ начала сообщения). В состоянии 2 каждый принятый символ записывается в буфер Data до получения символа LF (код xx), служащего признаком конца сообщения, после чего автомат переходит в состояние 3, в котором происходит проверка поля адреса сообщения. Для ведомого устройства это поле должно совпадать с адресом устройства. Если же прием осуществляется ведущим устройством, то есть сообщение является ответом ведомого на ранее посланную команду, в поле адреса должен содержаться адрес ответившего ведомого. Если это не так, то сообщение адресовано другому устройству, и автомат снова переходит в состояние 1 и ожидает начало следующего сообщения. Если же адрес совпадает, автомат переходит в состояние 4 «вычисление контрольной суммы». Если контрольная сумма совпадает с переданной в сообщении, прием сообщения считается успешно завершенным и автомат прекращает работу. Автомат также осуществляет обработку ряда нештатных ситуаций, которые могут возникнуть при приеме сообщения. К таким ситуациям относятся следующие:

- принят недопустимый символ (принятый символ не принадлежит множеству {'0'–'9', 'A'–'F', ':', CR, LF});
- после приема символа, не являющегося завершающим (LF), прошло более 1 с;

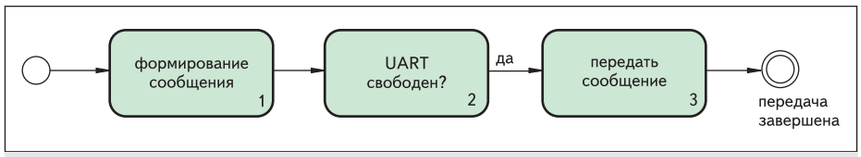

Рис. 3. Конечный автомат передачи сообщения Modbus в режиме RTU

- размер буфера превысил максимально допустимый (константа BUF_SIZE на рис. 2);
- не совпадает контрольная сумма.

Возникновение ошибок такого рода при работе устройства свидетельствует о плохом качестве связи (сильные электромагнитные помехи, плохое экранирование, слишком большая длина линии связи, несоблюдение правил монтажа и т. п.). Появление ошибок может также свидетельствовать о некорректной работе передающего устройства. Обработка ошибок будет рассмотрена чуть позже, а сейчас перейдем к описанию алгоритмов передачи и приема сообщений в режиме RTU.

Передача и прием сообщений в режиме RTU

Передача сообщения в режиме RTU осуществляется простейшим образом (рис. 3): передающее устройство формирует сообщение (состояние 1), дожидается, в случае необходимости, освобождения буфера передатчика UART (состояние 2) и отправляет сообщение (состояние 3). Каких-либо нештатных ситуаций и ошибок здесь ожидать не приходится.

Прием сообщений в режиме RTU (рис. 4) в целом аналогичен приему сообщения в режиме ASCII. Разница между ними состоит в следующем:

- Переход из состояния 2 (помещение сообщения в буфер) в состояние 3 (проверка поля адреса) осуществляется не по приему завершающего символа, а по тайм-ауту, величина которого должна быть равна 3,5 длительности передачи одиночного байта.
- Отсутствует проверка на допустимость символа, так как в режиме RTU все возможные коды в диапазоне 0…255 являются допустимыми.
- Отсутствует ошибка тайм-аута, так как пауза между символами интерпретируется в данном режиме как признак завершения передачи сообщения.
- Прием сообщения начинается при получении любого символа (переход из состояния 1 в состояние 2), так как специальный «стартовый» символ в режиме RTU отсутствует.

В завершение обсуждения реализации протокола Modbus приведем обобщенные алгоритмы поддержки протокола для ведущего и ведомого устройства, детализирующие изображенный на рис. 1 порядок обмена данными.

Обобщенный алгоритм работы ведущего устройства Modbus

Обобщенный алгоритм работы ведущего устройства Modbus изображен на рис. 5. Перед тем как перейти к подробному обсуждению его работы, напомним читателю, что мы рассматриваем алгоритмы в рамках SWITCH-технологии как конечные автоматы, выполняющиеся поочередно в главном цикле программы, аналогично тому, как выполняются задачи (потоки) в операционной системе с кооперативной многозадачностью. Поэтому автомат можно считать аналогом потока и представить себе, что все такие автоматы-потоки выполняются параллельно. На рис. 5 изображен именно такой автомат-поток, который выполняется параллельно с основной программой контроллера и обслуживает ввод/вывод сообщений по протоколу Modbus. Работа автомата управляется двумя сообщениями «извне»: «начать транзакцию» — сообщение от основной программы (предполагается, что посылка Modbus

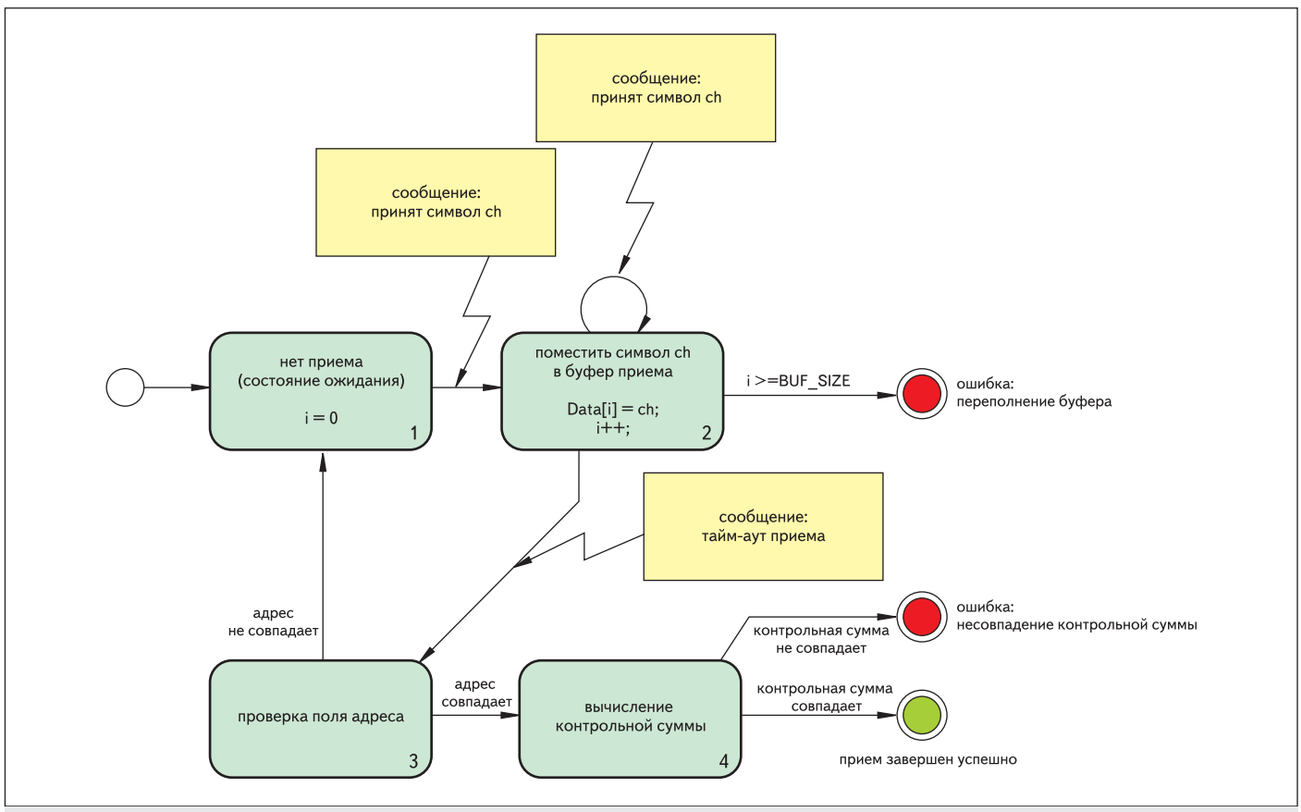

Рис. 4. Конечный автомат приема сообщения Modbus в режиме RTU

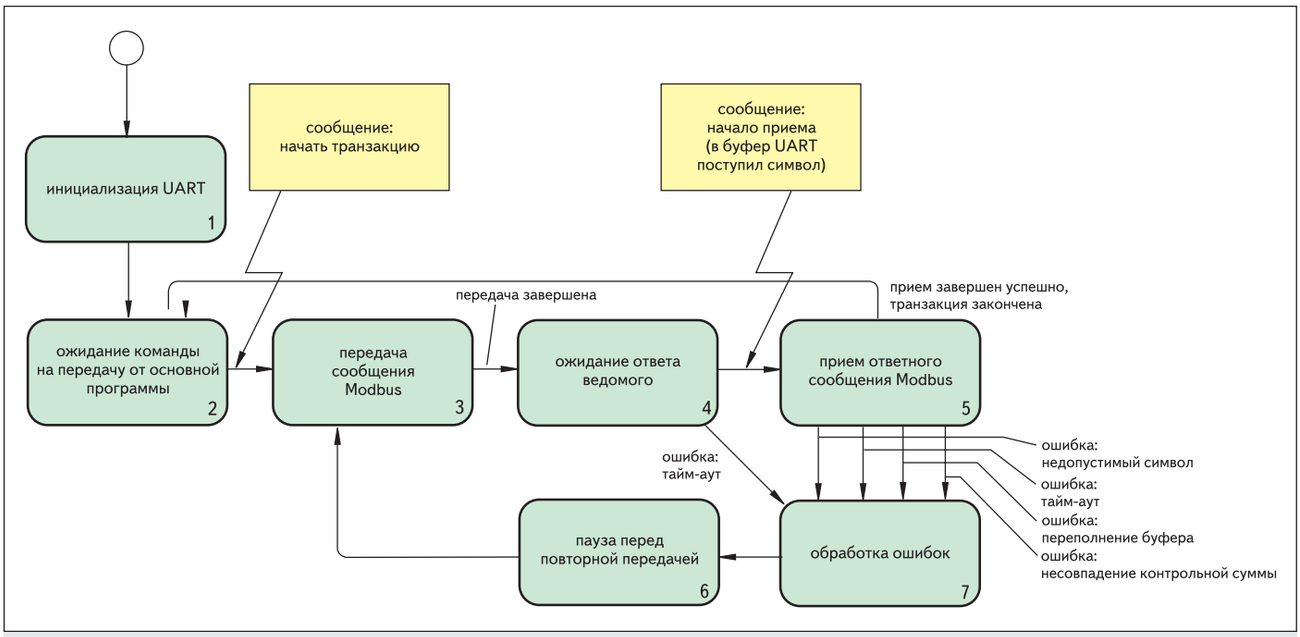

Рис. 5. Обобщенный алгоритм работы ведущего устройства Modbus

формируется и размещается в буфере передачи основной программой) и «начало приема» — сообщение, которое сигнализирует о поступлении символа в приемный буфер UART и формируется соответствующим обработчиком прерывания UART. Итак, поток после инициализации UART (состояние 1) переходит в состояние ожидания 2. Как только основная программа отдает команду на начало транзакции, автомат переходит в состояние 3, в котором происходит передача сообщения (в этом состоянии «заключен» один из рассмотренных выше алгоритмов передачи сообщения в режиме ASCII либо RTU). После передачи сообщения контроллер ожидает ответа ведомого устройства (состояние 4). Ожидание ответа ограничено по времени и обычно не превышает нескольких секунд. При поступлении первого символа ответного сообщения в буфер UART автомат переходит в состояние 5 (в этом состоянии размещен один из рассмотренных выше алгоритмов приема сообщения). При успешном приеме сообщения транзакция считается завершенной, и автомат переходит в состояние 2, ожидая следующей команды основной программы. При возникновении ошибки при приеме автомат выдерживает паузу (состояние 6) и повторяет передачу сообщения (состояние 3). Можно обратить внимание на то, что между состояниями 5 и 6 на рис. 5 изображено состояние 7 «обработка ошибок». В принципе, протокол Modbus не предусматривает какой-либо специфической обработки ошибок в принятых сообщениях (в случае возникновения ошибки при приеме ведущее устройство должно просто повторить передачу команды). Однако мы можем ввести в программу устройства, например,

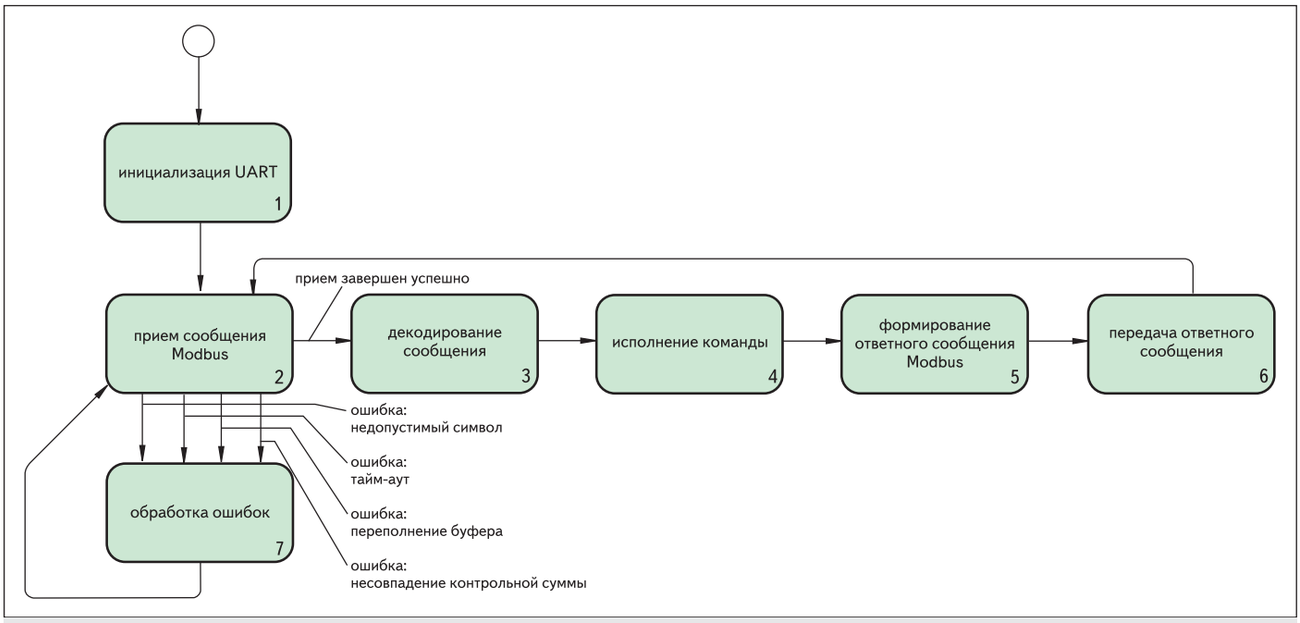

Рис. 6. Обобщенный алгоритм работы ведомого устройства Modbus

журнал ошибок, причем с подразделением причин возникновения той или иной ошибки (несовпадение контрольной суммы и т. п.). Наличие такой информации может быть очень полезно при отладке системы. Например, наблюдая за интенсивностью возникновения ошибок в канале связи, мы можем подобрать оптимальную скорость передачи данных в сети Modbus.

Обобщенный алгоритм работы ведомого устройства Modbus

Обобщенный алгоритм работы ведомого устройства Modbus изображен на рис. 6. Он похож на соответствующий алгоритм ведущего устройства, с той лишь разницей, что операции происходят в обратном порядке: сначала принимается сообщение (состояние 2), затем, после декодирования команды, ее исполнения и формирования ответного сообщения (состояния 3–5) происходит передача ответа ведущему устройству. Отличие заключается и в том, что при возникновении ошибки приема ведомое устройство не обязано посылать ответное сообщение, оно просто должно дождаться, когда мастер передаст команду повторно. Состояние 7, как и в предыдущем случае, не является обязательным, но весьма полезно при отладке.

На базе SWITCH-технологии могут быть реализованы и другие протоколы передачи данных [3, 4].

В следующей статье будет рассмотрена реализация на основе SWITCH-технологии ряда алгоритмов, полезных при разработке программного обеспечения для микроконтроллеров.

Автор выражает глубокую благодарность Анатолию Абрамовичу Шалыто за замечания и научное редактирование статьи.

Литература

1. Татарчевский В. А. Применение SWITCH-технологии при разработке прикладного программного обеспечения для микроконтроллеров. Часть 6. Реализация протокола Modbus // Компоненты и технологии. 2006. № 4.
2. Modicon Modbus Protocol Reference Guide. MODICON, Inc., Industrial Automation Systems, 1996.
3. Жданов А. Д., Коломейцева Т. М., Шалыто А. А. Реализация надежного протокола передачи данных. **ht tp://is.ifmo. ru**
4. Агафонов К. А., Порох Д. С., Шалыто А. А. Реализация протокола SMTP на основе SWITCH-технологии. **ht tp://is.ifmo. ru**

Применение SWITCH-технологии при разработке прикладного программного обеспечения для микроконтроллеров. Часть 8

В предыдущих статьях цикла был рассмотрен ряд аспектов использования SWITCH-технологии при построении программного обеспечения встраиваемых систем на базе микроконтроллеров. В этой статье представлены алгоритмы, часто применяемые в программном обеспечении микроконтроллерных систем, и их реализация на основе SWITCH-технологии.

Владимир ТАТАРЧЕВСКИЙ
arktur04@mail.ru

В подавляющем большинстве устройств на базе микроконтроллеров применяется клавиатура, имеющая, как правило, матричную организацию. Следовательно, в программе такого устройства должен содержаться алгоритм, производящий опрос клавиатурной матрицы и осуществляющий подавление дребезга контактов. Во многих случаях этот алгоритм также должен корректно обрабатывать различные сочетания нажатых клавиш. Алгоритмы такого рода не очень сложны и могут быть реализованы на основе циклов и задержек. Примеры программ подобного рода можно найти в [1, стр. 156]. Однако такая «очевидная» реализация имеет недостаток: будучи встроена в основную программу, эта подпрограмма блокирует исполнение основной программы на время своего цикла. Этого недостатка лишена реализация на основе SWITCH-технологии.

В качестве примера рассмотрим программу обслуживания ввода для клавиатуры, подключенной к микроконтроллеру по схеме, приведенной на рис. 1. Программа должна производить опрос клавиатурной матрицы, иметь защиту от дребезга контактов клавиатуры, корректно обрабатывать сочетания двух клавиш (например, сочетание клавиш Shift+6 должно определяться как нажатие «виртуальной» клавиши «стрелка влево»). Также программа должна иметь функцию автоповтора, подобно тому, как работает ввод с клавиатуры компьютера.

На рис. 2 приведен граф автомата, реализующего перечисленные функции.

Из рис. 2 видно, что устройство представляет собой автомат Мили (все действия производятся на переходах). Автомат использует один таймер (KEYB_TIMER). Величины задержек определяются следующими константами (они описаны в программе):

```
//период таймера 10 мс
#define debouce 10          //задержка для подавления дребезга
                            //(10 * 10 = 100 мс)
#define first_delay 50      //задержка первого повтора кода клавиши
                            //при ее удержании (50 * 10 = 500 мс)
#define auto_repeat 10      //задержка автоповтора кода клавиши при
                            //ее удержании (10 * 10 = 100 мс)
```

На рис. 3 показана временная диаграмма работы автомата при подаче на вход сигнала от кнопки с дребезгом.

На рис. 3 числа обозначают состояния автомата в соответствии с рис. 2. Временные задержки обозначены следующим образом: td (debounce) — задержка подавления дребезга, tfr (first repeat) — задержка перед первым повтором нажатия клавиши, tar (auto repeat) — период автоповтора нажатий клавиши.

При анализе рис. 2, 3 может сложиться впечатление, что такая программа предназначена для обработки нажатий только одной клавиши, тогда как необходимо обрабатывать нажатие всех клавиш клавиатуры. Кроме того, на рис. 2 не показан сам цикл сканирования клавиатурной матрицы. Однако никакой ошибки здесь нет. Цикл сканирования (он не приводится здесь, так как очень прост) активируется перед вызовом функции ProcessKeyFSM, реализующей автомат, изображенный на рис. 2. В нем формируется скан-код — число, соответствующее текущему («мгновенному») состоянию клавиатурной матрицы. Такой скан-код может представлять собой, например, 16-битовое беззнаковое целое, 12 младших разрядов которого соответствуют 12 кнопкам клавиа-

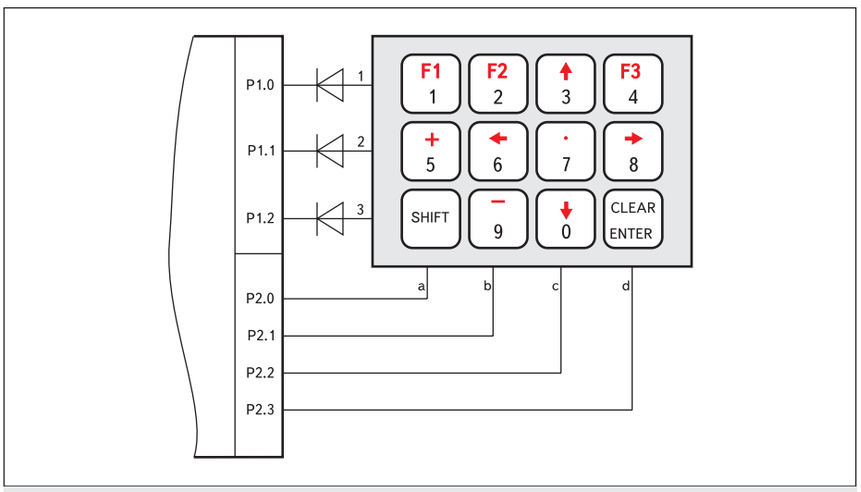

Рис. 1. Пример подключения матричной клавиатуры к микроконтроллеру

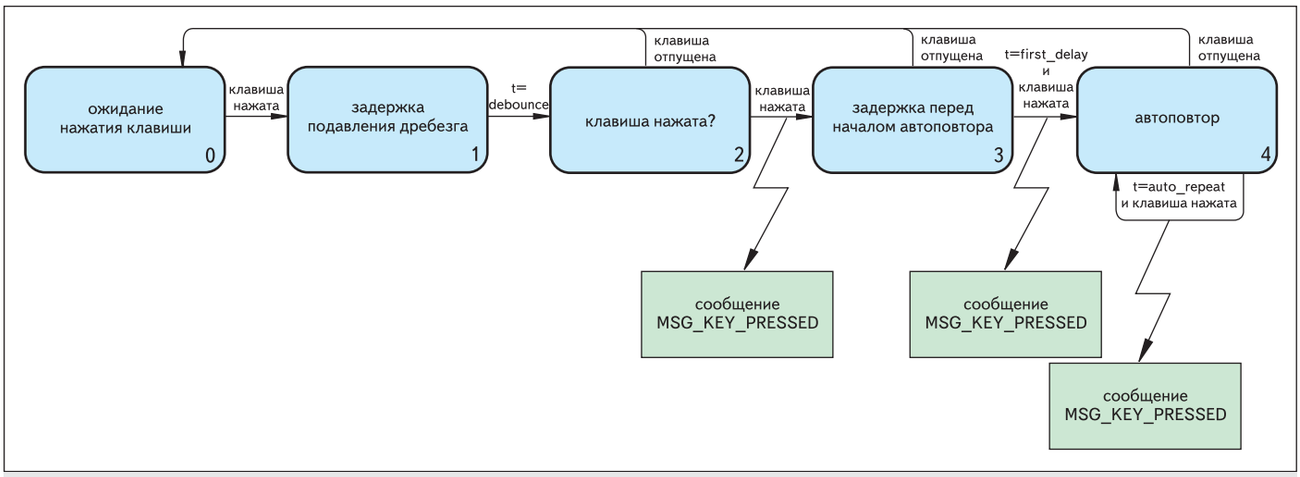

Рис. 2. Автомат драйвера клавиатуры

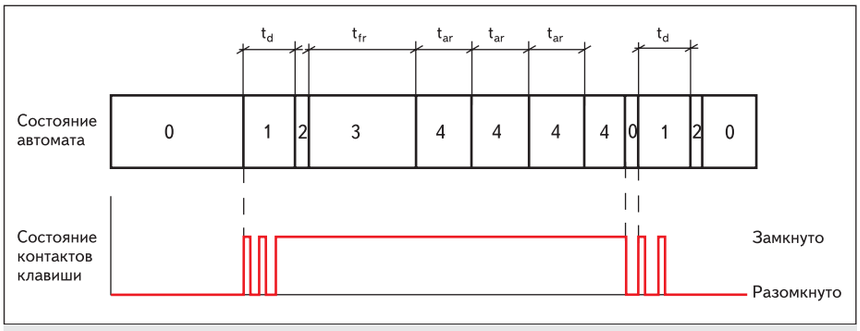

Рис. 3. Временная диаграмма работы автомата

туры (рис. 1), при этом нулю в разряде соответствует разомкнутая кнопка, а единице — замкнутая. Таким образом, каждой клавише и каждому сочетанию нажатых клавиш соответствует уникальный скан-код (если ни одна клавиша не нажата, скан-код равен нулю). В листинге программы (он приведен далее) скан-код клавиатуры содержится в переменной key_code. Также имеется переменная _key_code, в которой содержится состояние клавиатуры в предыдущем цикле. Она нужна для того, чтобы автомат мог корректно обработать ситуацию, при которой пользователь нажимает клавишу, удерживая при этом другую (или отпускает одну клавишу, удерживая нажатой другую). Для корректной обработки подобных ситуаций достаточно полагать, что изменение скан-кода клавиатуры равнозначно отпусканию клавиши. Тогда при изменении скан-кода автомат перейдет в состояние 0, и следующая комбинация клавиш будет обработана вновь. Таким образом, только переход 0-1 («клавиша нажата») реализуется буквально:

```
if (key_code > 0) {...};
```

Переходы 2-0, 3-0, 4-0 («клавиша отпущена») и 2-3, 3-4, 4-4, включающие в себя условие «клавиша нажата», реализуются как проверка совпадения текущего скан-кода и предыдущего его значения:

```
if (key_code == _key_code){
//клавиша нажата
}
else{
//клавиша отпущена
};
```

Теперь можно записать реализацию автомата на языке Си.

```
char key_state;//переменная состояния автомата

void ProcessKeyFSM(void){
switch (key_state){
/*состояние 0. В этом состоянии автомат ожидает нажатия на клавишу*/
case 0: //клавиша не нажата
if (key_code > 0){
_key_code = key_code;
ResetTimer(KEYB_TIMER);
key_state = 1;
};
break;
/*состояние 1. В этом состоянии вводится задержка на время debounce (ожидание завершения переходного процесса при замыкании контактов клавиши)*/
case 1: //задержка подавления дребезга
if (GetTimer(KEYB_TIMER) > debounce)
key_state = 2;
break;
/*состояние 2. В этом состоянии фиксируется факт нажатия клавиши и формируется сообщение MSG_KEY_PRESSED. Если клавиша не нажата, возврат в состояние 0*/
case 2: //если клавиша нажата, посылаем сообщение
if (key_code == _key_code){
ResetTimer(KEYB_TIMER);
SendMessage(MSG_KEY_PRESSED);
key_state = 3;
}
else
key_state = 0;
break;
/*состояние 3. В этом состоянии автомат формирует сообщение MSG_KEY_PRESSED, если пользователь удерживает клавишу в течение времени first_delay. Если клавиша отпущена, возврат в состояние 0*/
case 3:
if (key_code == _key_code){
if (GetTimer(KEYB_TIMER) >= first_delay){
ResetTimer(KEYB_TIMER);
SendMessage(MSG_KEY_PRESSED);
key_state = 4;
};
}
else
key_state = 0;
break;
/*состояние 4. В этом состоянии автомат формирует последовательность сообщений MSG_KEY_PRESSED с периодом auto_repeat, если пользователь продолжает удерживать клавишу. Если клавиша отпущена, возврат в состояние 0*/
case 4:
if (key_code == _key_code){
if (GetTimer(KEYB_TIMER) >= auto_repeat){
ResetTimer(KEYB_TIMER);
SendMessage(MSG_KEY_PRESSED);
};
}
else
key_state = 0;
break;
};
}
```

Следует отметить еще один важный момент. Представленный автомат посылает сообщения основной программе при нажатии клавиш, но не сообщает, какая именно клавиша (или сочетание клавиш) нажата. Если в программе реализован какой-либо механизм сообщений с параметрами, то код нажатой клавиши можно передавать в качестве параметра сообщения. При использовании в программе простейшего механизма передачи сообщений без параметров, получение кода нажатой клавиши основной программой может быть реализовано следующим образом.

В программе перечисляются все используемые клавиши и их сочетания (сочетание кла-

виш, имеющее самостоятельный смысл, будем называть «виртуальной клавишей»):

```
#define KEY_0 1
#define KEY_1 2
...
#define KEY_9 10
#define KEY_ENTER 11
#define KEY_F1 12
#define KEY_F2 13
#define KEY_F3 14
//коды виртуальных клавиш (сочетания клавиша + «Shift»)
#define KEY_UP 15
#define KEY_DOWN 16
#define KEY_LEFT 17
#define KEY_RIGHT 18
#define KEY_MINUS 19
#define KEY_DOT 20
#define KEY_CLEAR 21
```

Также в программу вводится функция, преобразующая скан-код в код клавиши (ее реализация здесь не приводится):

```
char GetKeyCode(void);
```

Теперь основная программа, получив сообщение MSG_KEY_PRESSED, может получить код нажатой клавиши и предпринять какие-либо действия:

```
if(GetMessage(MSG_KEY_PRESSED)){
switch(GetKeyCode()){
case KEY_ENTER: //ENTER — завершение работы
...
break;
case KEY_CLEAR: //CANCEL — отмена
...
break;
```

Приведенный здесь модуль работы с клавиатурой играет в SWITCH-программе роль своеобразного драйвера клавиатуры. По сравнению с «традиционной» реализацией такой подход имеет ряд преимуществ. Одно из них уже было приведено: при работе драйвера клавиатуры основная программа не испытывает существенных задержек, связанных с работой антидребезгового алгоритма. Другие преимущества заключаются в следующем. Драйвер имеет высокую степень автономности: его работа не зависит от функционирования основной программы. Основная программа также не обязана «знать» о внутреннем алгоритме работы драйвера, она только получает от него сообщения о нажатии клавиш, и все, что нужно для обеспечения корректного взаимодействия основной программы и драйвера — описать несколько сообщений и коды виртуальных клавиш. Отсюда вытекает еще одно важное преимущество использования автоматов-драйверов: очень высокая степень модифицируемости программы. При внесении изменений в драйвер не требуется никаких изменений в основной программе и наоборот. При изменении аппаратной части устройства изменения вносятся только в драйвер. Даже при переносе программы на микроконтроллер другого типа, с другой организацией портов ввода/вывода потребуется только корректно переписать драйверы с учетом изменившейся архитектуры системы; те части программы, которые отвечают непосредственно за логику функционирования программы, остаются без изменений. Еще одним большим преимуществом автоматного подхода является высокая степень повторного использования ранее написанного кода. Написав драйвер один раз, его возможно использовать в последующих разработках многократно, уже не вникая в алгоритм его работы. Автономность драйвера, ограниченность его связей с основной программой также позволяют достичь эффективного разделения труда между программистами-разработчиками драйверов и программистами-разработчиками «основной» программы, то есть высокой степени параллельности их работы.

Автор выражает благодарность Анатолию Абрамовичу Шалыто за ценные замечания и научное редактирование статьи.

Литература

1. Сташин В. В., Урусов А. В., Мологонцева О. В. Проектирование цифровых устройств на однокристальных микроконтроллерах. М.: Энергоатомиздат. 1990.