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 (nonqueuqed). 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

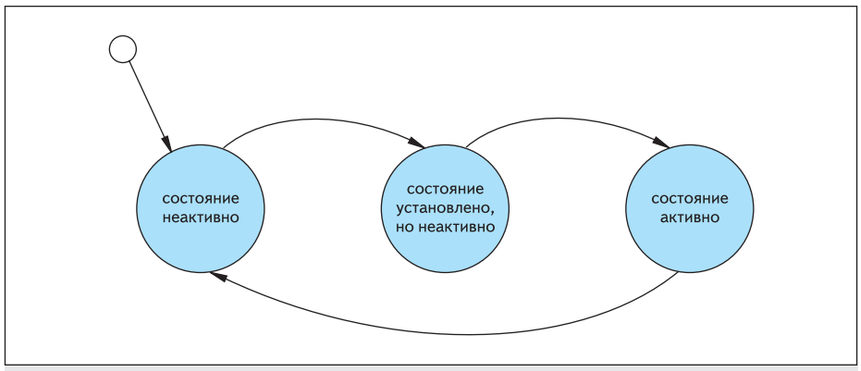

Рис. 3. Граф переходов сообщений

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; в двух томах. 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, vol. 13, №10, 2000.
4. Гуисов М. И., Кузнецов А. Б., Шалыто А. А. Интеграция механизма обмена сообщениями в Switch-технологию. 2003.