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.