Продолжение. Начало № 3`2008

Игорь КРИВЧЕНКО, к. т. н.
ik@efo.ru
Елена ЛАМБЕРТ, к. т. н.
elena@efo.ru

Микроконтроллеры XMEGA — новые возможности проверенного решения. Часть 2

Конфигурируемые порты ввода/вывода

Порты ввода/вывода у всех микроконтроллеров AVR традиционно имеют три управляющих бита для каждого физического вывода: бит данных (PORTx), бит управления направлением передачи данных (DDRx) и бит для отображения логического уровня сигнала на физическом выводе микросхемы (PINx). Для конфигурации линии порта также используется бит PUD регистра SFIOR, с помощью которого вывод подтягивается к шине питания через внутренний резистор. Это позволяет обеспечить истинную функциональность вида «чтение — модификация — запись» и независимо выполнять битовые операции на любой линии порта. Порты ввода/вывода AVR, их конфигурирование, управляющие регистры и особенности работы подробно описаны в многочисленных изданиях (например, [1]), и здесь мы на этом останавливаться не будем.

Удачное построение портов ввода/вывода AVR было использовано и для XMEGA, причем разработчики кристалла сохранили весь базовый набор регистров. Но для микроконтроллеров нового поколения был сделан ряд важных усовершенствований:

- Изменились символические имена управляющих регистров, они стали более «естественными» для восприятия. Так, бит данных теперь обозначается OUTx, бит управления направлением передачи данных — DIRx, а бит для отображения логического уровня сигнала на физическом выводе микросхемы — INx.
- Расширились возможности работы портов ввода/вывода: каждая линия порта теперь имеет 6 конфигураций и 8 режимов работы. Гибкая конфигурация обеспечивается благодаря наличию нового специального регистра PINnCTRL.
- Для более эффективного битового управления линиями портов ввода/вывода для OUTx и DIRx добавлены функции «установить» — SET, «сбросить» — CLR и «переключить» — TGL. Соответственно, введены дополнительные наборы регистров для OUT (OUTSET, OUTCLR, OUTTGL) и DIR (DIRSET, DIRCLR, DIRTGL).
- С помощью виртуальных регистров реализована возможность отображения регистров порта в адресное пространство I/O памяти, доступное для манипуляции битами.
- Доступно синхронное или асинхронное обнаружение изменения входного сигнала на внешнем выводе с формированием события или запроса на прерывание, включая выведение микроконтроллера из режимов пониженного энергопотребления.
- Реализовано управление скоростью нарастания входного сигнала.
- Добавлено групповое управление конфигурацией сразу нескольких линий порта за одну операцию при помощи маски.
- Имеется возможность подключения к линии порта сигнала периферийной тактовой частоты и выхода одного из каналов системы событий.

Все описанные функции доступны для каждой линии портов ввода/вывода XMEGA. Отметим, что в процессе работы микроконтроллера операции с какой-то отдельной линией порта не оказывают влияния на функционирование других линий этого же порта и других портов ввода/вывода. Это относится к изменению направления передачи данных, к изменению логического уровня на выводе микросхемы при конфигурации на выход и к подключению (отключению) внутренних резисторов при конфигурации на вход.

Регистр PINnCTRL используется для реализации новых режимов конфигурации и работы линии порта ввода/вывода. Каждый вывод теперь может быть сконфигурирован как Push-Pull (двухтактный каскад с симметричной нагрузочной способностью), как монтажное «И» или как монтажное «ИЛИ». Можно разрешить инверсию сигнала на отдельной линии порта при вводе/выводе данных, а также разрешить ограничение скорости нарастания сигнала на ней при вводе данных. Управление скоростью нарастания входного сигнала снижает энергопотребление узла. В среднем после включения этого ограничения длительность фронта и среза входного сигнала увеличивается на 50–150% в зависимости от напряжения питания микроконтроллера, рабочей температуры и характера нагрузки.

Для режима Push-Pull есть 4 возможные конфигурации: собственно Push-Pull (или Totem-pole), подтяжка к «земле» (Pull-down), подтяжка к шине питания (Pull-up) и сохранение последнего активного состояния на шине (Bus-keeper). В конфигурации Bus-keeper порт может работать в двух направлениях — на прием и на передачу. Этот позволяет избежать колебаний сигнала на выводе при отключении нагрузки, а также когда мы запрещаем работу линии порта на выход. Слаботочные буферы Bus-keeper удерживают на выводе микросхемы логический уровень сигнала, который присутствовал последним: то есть реализуется подтяжка к «1», если последний уровень был «1», или подтяжка к «земле», если последний уровень был «0». Для конфигураций Pull-down и Pull-up подтяжка вывода к «земле» или к шине питания осуществляется через активный резистор только в том случае, если линия порта работает на вход. Такое решение позволяет снизить энергопотребление. Для режимов монтажное «И» и монтажное «ИЛИ» дополнительные резисторы подтяжки (pull-up и pull-down соответственно) могут быть подключены к выводу микросхемы как при работе на вход, так и при работе на выход.

Подключение и отключение внутренних резисторов подтяжки при помощи регистра конфигурации PINnCTRL является удачным решением разработчиков XMEGA. Благодаря этому на выводе микроконтроллера не наблюдается промежуточных состояний линии порта, когда мы переключаем направление работы порта или изменяем логическое значение на выводе.

Состав регистра PINnCTRL и назначение его битов подробно описаны в технической документации на микроконтроллер. На рис. 1 проиллюстрированы все возможные конфигурации линии порта ввода/вывода XMEGA и режимы работы.

Программист имеет возможность вывести сигнал периферийной тактовой частоты на любую линию порта ввода/вывода. К линии порта также можно программно подключить канал 7 системы событий (Event Channel 7). Если в этом канале генерируется событие, то соответствующий этому событию сигнал будет отображаться на выводе микроконтроллера в течение одного периода периферийного тактового сигнала.

Каждая линия порта ввода/вывода у XMEGA может быть запрограммирована на генерацию событий или запросов на прерывание по фронту, срезу, перепаду или низкому уровню входного сигнала на выводе микро-

контроллера. Если необходимо «почувствовать» высокий уровень, то следует использовать функцию инверсии путем установки бита INVEN в регистре PINnCTRL. Поддерживается синхронное и асинхронное обнаружение изменения входного сигнала. Синхронное детектирование требует наличия сигнала периферийной тактовой частоты, асинхронное — не требует.

Генерация событий возможна только при наличии сигнала периферийной тактовой частоты, поэтому асинхронная генерация событий при детектировании изменения состояния вывода микроконтроллера невозможна. Для генерации события по фронту, срезу или перепаду значение сигнала на выводе должно измениться в течение одного периода периферийной тактовой частоты. Если вывод запрограммирован для генерации события по наличию на нем сигнала низкого уровня, то данная линия порта непосредственно подключается к каналу системы событий. При этом Event System постоянно отслеживает текущее значение сигнала на выводе микроконтроллера: низкий уровень не генерирует событие, а при наличии высокого уровня события будут генерироваться постоянно. Это может вызвать «бесконечную» генерацию событий. Контроль такой ситуации и необходимые действия должны осуществляться программистом.

Немного сложнее обстоит дело с генерацией запросов на прерывание. Каждый порт ввода/вывода у микроконтроллеров XMEGA имеет два вектора прерывания. Программист может выбирать линии порта, которые будут использоваться для формирования запросов на прерывание по этим векторам. Тип изменения входного сигнала (фронт, срез, перепад или низкий уровень), которое вызывает генерацию, будет зависеть от вида детектирования — синхронное или асинхронное.

При синхронном детектировании запросы на прерывание генерируются для любого типа изменения входного сигнала. Как и в случае системы событий, для генерации запроса на прерывание по фронту, срезу или перепаду значение сигнала на выводе должно измениться в течение одного периода периферийной тактовой частоты.

Асинхронное детектирование интересно тем, что именно с его помощью можно вывести микроконтроллер из «спящих» режимов при изменении состояния сигнала на внешнем выводе. Отметим, что при асинхронном детектировании фронта, среза или перепада только линия 2 любого порта ввода/вывода XMEGA обладает возможностью действительно поддерживать полностью асинхронное детектирование. Это означает, что линия 2 будет определять и «ловить» любое изменение входного сигнала с последующей генерацией запроса на прерывание. Все другие линии порта имеют ограничение по выведению микроконтроллера из «спящего» режима. Значение входного сигнала, которое изменилось в виде любого перепада, должно удерживаться на выводе до тех пор, пока кристалл не «проснется» и не появится сигнал тактовой частоты. Если в течение этого времени сигнал вернется к первоначальному значению, то микроконтроллер все равно «проснется», но запрос на прерывание не будет сгенерирован.

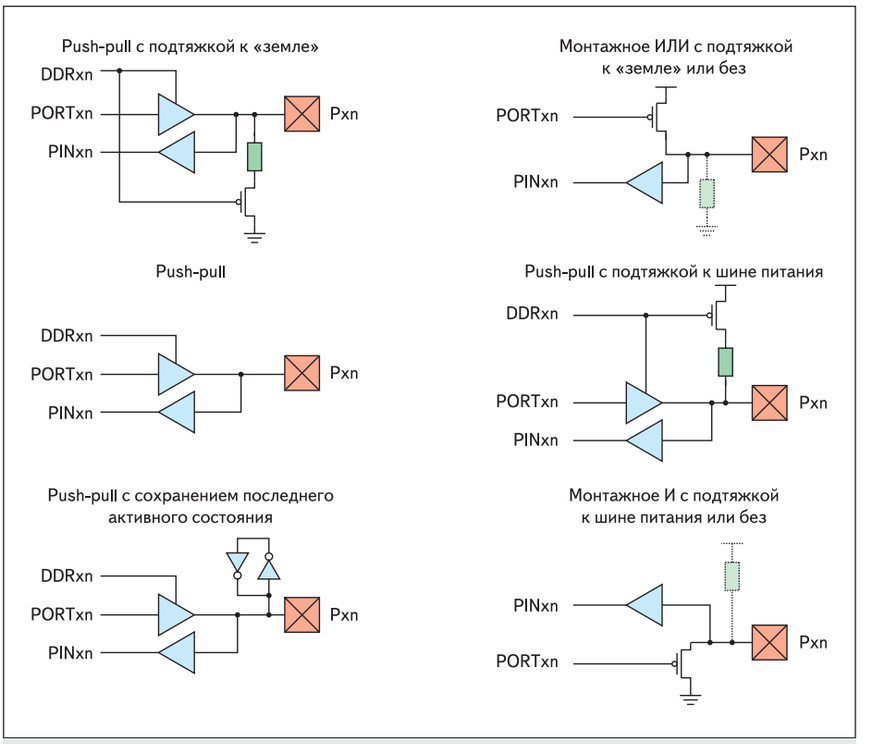

Рис. 1. Конфигурации порта ввода/вывода в микроконтроллерах XMEGA

Низкий уровень входного сигнала на выводе микроконтроллера может детектироваться всеми линиями порта ввода/вывода независимо от того, присутствует сигнал периферийной тактовой частоты или нет. Если линия порта сконфигурирована на генерацию прерывания по низкому уровню входного сигнала, запросы на прерывание будут возникать всегда, пока вывод удерживается в низком состоянии. Анализ состояния вывода в этом случае проводится поразному в зависимости от режима работы микроконтроллера. В активном режиме низкий уровень должен удерживаться, пока не завершится текущая инструкция CPU, и только потом будет сформирован запрос на прерывание. В режимах пониженного энергопотребления низкий уровень должен удерживаться на выводе в течение всего времени «просыпания» микроконтроллера, и только потом запрос на прерывание может быть сгенерирован. Если низкий уровень входного сигнала пропадет во время выхода из «спящего» режима, то микроконтроллер «проснется», но запроса на прерывание не будет.

В техническом описании на микроконтроллеры XMEGA приводятся все случаи, когда будет происходить генерация запросов на прерывание в зависимости от вида изменения входного сигнала на внешнем выводе микросхемы. Подробное описание последовательности действий для конфигурации линии порта ввода/вывода для генерации запроса на прерывание можно найти в Application Note AVR1313.

Рассмотрим еще два новшества у портов ввода/вывода XMEGA, которые позволяют снизить размер кода и увеличить скорость выполнения программы. Это групповая конфигурация нескольких выводов по маске и виртуальные регистры.

Идея добавления виртуальных регистров на кристалл XMEGA возникла из потребности обеспечить удобное и простое управление отдельными линиями каждого из многочисленных портов ввода/вывода новых микроконтроллеров.

Виртуальные регистры порта позволяют стандартным регистрам порта, расположенным в адресном пространстве расширенной области ввода/вывода (адреса старше 0x3F), быть отображенными на «базовое» для AVR адресное пространство области ввода/вывода (I/O Memory). Когда такое отображение осуществляется, запись в виртуальный регистр будет эквивалентна записи в реальный регистр порта. При этом для любого порта ввода/вывода XMEGA становится доступным весь ассортимент инструкций AVR для работы с регистрами, включая специальные

команды манипуляции битами. Всего у микроконтроллеров XMEGA есть 4 виртуальных порта, поэтому одновременно может быть отображено до 4 портов ввода/вывода. Отображаемые регистры — IN, OUT, DIR и INTFLAGS.

Почему так полезно использование виртуальных регистров? Потому что разница во времени исполнения и в размере генерируемого кода при работе с обычными и виртуальными портами может быть значительна. Некоторые инструкции из набора команд AVR могут работать только с регистрами, адреса которых в адресном пространстве AVR не старше 0x1F (0x3F). Использование именно этих команд вместо их эквивалентов (обращение к ячейкам памяти по этим же адресам) происходит быстрее и занимает меньше процессорного времени. А ведь все адреса регистров портов ввода/вывода XMEGA старше «заветной» границы пространства I/O Memory (0x3F). Вот для такой полезной «подмены» и используются виртуальные регистры. Поясним сказанное на простом примере. Установим в высокий уровень линию 0 порта C, не меняя при этом состояние других линий этого порта:

```
// Использование виртуального порта PVIRT0
sbi PVIRT0_OUT, 0;
// Код состоит из одного слова, время выполнения — 1 такт*)

// Прямая адресация порта C:
sbr r16, 0x01;
sts PORTC_base + PORT_OUTSET_offset, r16;
// Размер этого кода — 3 слова, время выполнения — 3 такта.
```

*) Отметим, что хотя CPU выполняет команду «sbi» за один такт, для появления результата на линии ввода/вывода требуется 2 такта

Этот пример показывает, что использование виртуального порта позволяет сократить требуемый объем памяти программ и сократить время исполнения для одинаковых операций. Это особенно важно, когда требуется быстрое выполнение программы и осуществляется постоянное обращение к регистрам порта.

Поговорим теперь о групповой конфигурации. Наличие нового регистра конфигурации PINnCTRL для каждой линии порта ввода/вывода, безусловно, удобно и полезно. Но при этом также возрастает количество операций, необходимых для последовательной конфигурации всего порта, разряд за разрядом. В результате размер генерируемого кода существенно увеличивается. В то же время очень часто встречаются ситуации, когда несколько линий одного порта должны быть сконфигурированы одинаковым образом. Для сокращения количества необходимых операций в таких случаях разработчики XMEGA добавили еще один специальный регистр — MPCMASK. Этот глобальный регистр является общим для всех портов ввода/вывода микроконтроллера и предназначен для одновременной конфигурации нескольких линий порта. Его использование позволяет создавать эффективный и компактный код.

Регистр MPCMASK загружается битовой маской, которая затем используется при работе с конфигурационными регистрами PINnCTRL. Делается это так. Сначала программистом определяется необходимая маска конфигурации (битовый шаблон) в регистре MPCMASK — то есть набор требуемых линий порта, которые нужно конфигурировать одинаковым образом. Запись «1» в бит «n» этого регистра означает, что линия порта «n» должна быть участником групповой конфигурации. Затем, когда любая линия этого порта конфигурируется путем записи информации в ее регистр PINnCTRL, во все остальные регистры PINnCTRL этого же порта, помеченные маской в регистре MPCMASK, записываются те же самые значения. Регистр MPCMASK очищается автоматически после окончания операции записи во все регистры PINnCTRL порта.

Отметим, что перед конфигурацией нескольких линий порта ввода/вывода с помощью маски рекомендуется глобально запретить все прерывания, а после завершения конфигурации — снова разрешить их. Это связано с аппаратными особенностями организации порта ввода/вывода XMEGA (подробнее см. в Application Note AVR1313).

В качестве иллюстрации рассмотрим практический пример работы с регистром MPCMASK. Для его реализации использовался стартовый набор разработчика STK600. Исходный текст написан на Си и компилирован в интегрированной среде IAR Systems EWAVR 4.30. Необходимые установки на плате STK600 (перемычки, соединительные кабели, переключатели и т. п.) в данной статье не описываются, мы рекомендуем использовать руководство пользователя для STK600 («User Manual»).

Пример 1. Одновременное конфигурирование нескольких выводов микроконтроллера

Задача заключается в том, чтобы считывать состояние кнопок SW на плате STK600 и копировать их состояние на светодиоды этого стартового набора. Нажатие кнопки SWn должно индицироваться включением светодиода LEDn. Первая функция примера показывает, как можно прочитать существующие значения из регистра OUT порта, очистить биты в соответствии с маской и затем вывести обратно новые значения. Такая конструкция очень часто используется при работе с заданными битовыми полями в регистрах, не «трогая» остальные биты.

Сначала рассмотрим вариант, который не использует возможности регистра MPCMASK:

```
void SetLEDPort( PORT_t volatile * ledPort, unsigned char value, unsigned char mask )
  {
      value &= mask;  // Очистка ненужных битов из value по маске
      ledPort->OUT = ledPort->OUT & ~mask | value;
  }

unsigned char GetSwitches( PORT_t volatile * switchPort, unsigned char mask )
  {
      // Считываем входное значение, удаляем ненужные биты
      // и возвращаем значение
      return (switchPort->IN & mask);
  }

void main( void )
{
  // Определяем указатели на порты ввода/вывода, которые мы
  // хотим использовать для работы с кнопками и светодиодами
  // на плате STK600:
  PORT_t volatile * ledPort = &PORTD;
  PORT_t volatile * switchPort = &PORTC;

  // Готовим битовую маску для тех выводов микроконтроллера,
  // которые будут использоваться.
  unsigned char const mask = 0x07;  // Работаем только с линиями
                                    // 0, 1, и 2.

  // Устанавливаем линии 0, 1, и 2 порта D: выход, монтажное «И»
  // с подтяжкой.
  ledPort->PIN0CTRL = ledPort->PIN0CTRL & ~PORT_OPC_gm |
                      PORT_OPC_WIREDANDPULL_gc;
  ledPort->PIN1CTRL = ledPort->PIN1CTRL & ~PORT_OPC_gm |
                      PORT_OPC_WIREDANDPULL_gc;
  ledPort->PIN2CTRL = ledPort->PIN2CTRL & ~PORT_OPC_gm |
                      PORT_OPC_WIREDANDPULL_gc;

  ledPort->DIR = mask;

  // Теперь копируем состояние кнопок на светодиоды...
  for (;;) {
      unsigned value = GetSwitches( switchPort, mask );
      SetLEDPort( ledPort, value, mask );
  }
}
```

После компиляции данного фрагмента кода и его запуска на STK600 нажатие одной из кнопок SW0…2 будет включать соответствующий светодиод LED0…2. При отпускании кнопки светодиод будет гаснуть. Программа очень простая, но требует много кода, так как три строчки конфигурации линий порта D однотипны. Попробуем сократить размер генерируемого кода — используем регистр MPCMASK. Теперь текст основной программы ***main(void)*** будет выглядеть так:

```
void main( void )
{
  PORT_t volatile * ledPort = &PORTD;
  PORT_t volatile * switchPort = &PORTC;

  unsigned char const mask = 0x07;

  PORTCFG.MPCMASK = mask;     // Включаем линии 0, 1 и 2
                              // в группу конфигурации
  // Теперь все равно, какой из конфигурирующих регистров
  // порта устанавливать. Пусть это будет линия 2. Регистры
  // PINnCTRL 0 и 1 будут сконфигурированы такими же
  // значениями автоматически!

  ledPort->PIN2CTRL = ledPort->PIN2CTRL & ~PORT_OPC_gm |
                      PORT_OPC_WIREDANDPULL_gc;

  ledPort->DIR = mask;

  for (;;) {
      unsigned value = GetSwitches( switchPort, mask );
      SetLEDPort( ledPort, value, mask );
  }
}
```

Программа выполняет те же самые действия, но размер конечного кода заметно сократился!

В заключение проиллюстрируем с помощью этой программы некоторые аппаратные особенности порта ввода/вывода XMEGA. Мы конфигурировали линии на выход в режим работы монтажное «И» с подтяжкой вывода к шине питания микроконтроллера. Это

позволяет непосредственно соединять линии порта. Воспользуемся 4-проводным шлейфом из комплекта поставки STK600. Соединим с его помощью линии 0, 1 и 2 порта D между собой и подключим получившуюся группу к любому из светодиодов (например, к LED1). Теперь после запуска программы нажатие любой кнопки SW0…2 (или нескольких одновременно) будет включать светодиод LED1. При отпускании всех кнопок светодиод будет гаснуть.

Дополнительную информацию об особенностях работы с портами ввода/вывода в микроконтроллерах XMEGA можно найти в документации Atmel — Application Note AVR1313.

Многоуровневый программируемый контроллер прерываний

Как мы уже говорили в первой части цикла, компания Atmel впервые реализовала у своих 8-разрядных микроконтроллеров AVR многоуровневый программируемый контроллер прерываний (PMIC), который управляет обслуживанием запросов на прерывание, включая уровень и приоритет.

Прерывание сигнализирует об изменении состояния периферийного модуля и может использоваться для изменения порядка выполнения основной программы. Периферийные блоки (источники) могут генерировать один или несколько типов запроса на прерывание, которые разрешаются установкой индивидуальных битов управления в соответствующих регистрах для каждого источника. У микроконтроллеров XMEGA каждый источник прерывания и сброс микроконтроллера имеют отдельный адрес (вектор прерывания) в памяти программ, в котором размещается команда перехода на подпрограмму обслуживания этого прерывания (обработчик). Самый младший адрес принадлежит вектору сброса.

Периферийным модулям в микроконтроллерах XMEGA может быть назначен один из четырех различных уровней приоритета прерываний: Off (нет прерываний), High (высокий), Medium (средний) и Low (низкий). Прерывания с уровнем Medium будут приостанавливать работу обработчиков прерываний с уровнем Low. Запросы на прерывание, которым присвоен уровень High, обслуживаются немедленно после поступления. Внутри каждого из трех уровней (Low, Medium, High) приоритет обслуживания прерывания определяется исходя из абсолютного адреса вектора прерывания — вектор с наименьшим адресом обслуживается в первую очередь. Для прерываний, имеющих одинаковый статус Low, дополнительно предусмотрена процедура диспетчеризации Round Robin, когда все процессы активизируются в фиксированном циклическом порядке. И несмотря на все нововведения прерывания в XMEGA по-прежнему должны быть предварительно глобально разрешены для того, чтобы контроллер PMIC давал право обработчику вмешиваться в ход выполнения основной программы. Это выполняется программно путем установки бита I в CPU Status Register, причем данный бит не сбрасывается в процессе обслуживания прерывания.

Когда в PMIC поступает запрос на прерывание, то сначала проводится анализ уровня поступившего запроса на прерывание и статуса текущих обслуживающихся прерываний. Затем поступивший запрос передается в обработку или помещается в очередь до момента, когда он получит право на обслуживание в соответствии со своим назначенным уровнем. После возвращения из обработчика прерывания основной ход программы продолжается из той же точки, где он был прерван. Важно отметить, что одна следующая инструкция основной программы CPU всегда выполняется перед тем, как передать обслуживание обработчикам отложенных прерываний в очереди. Корректный возврат из прерывания (по команде RETI) будет возвращать PMIC в состояние, которое он имел перед входом в прерывание.

Все прерывания в XMEGA имеют собственные флаги. Когда наступает условие для прерывания, соответствующий флаг устанавливается, даже если это прерывание не разрешено. Для большинства прерываний установленный флаг автоматически сбрасывается после обслуживания прерывания. Можно сбросить флаг программно, записав в него логическую «1». Некоторые флаги прерываний не сбрасываются после передачи управления обработчику, а некоторые сбрасываются автоматически, когда ассоциированный регистр изменяется по чтению или записи. Детали следует смотреть в техническом описании на периферийные модули XMEGA.

Флаги прерывания устанавливаются всегда. Если в текущий момент обслуживается прерывание более высокого уровня, флаг будет удерживаться до тех пор, пока не наступит очередь для обслуживания этого прерывания. Даже если соответствующее прерывание не разрешено, его флаг все равно устанавливается и запоминается до тех пор, пока прерывание не будет разрешено или пока флаг не будет сброшен программно. Если глобально запрещены все прерывания, то флаги все равно устанавливаются и запоминаются до тех пор, пока все прерывания будут разрешены. Все отложенные прерывания в очереди потом будут обслужены в строгом соответствии с их установленным приоритетом обслуживания.

В контроллере прерываний XMEGA реализована поддержка немаскируемых прерываний (NMI), которые также должны быть предварительно разрешены. Они жестко привязаны к аппаратным ресурсам микроконтроллера и к ряду системных функций процессора. Подробную информацию о составе и работе NMI следует смотреть в техническом описании на каждый микроконтроллер. Например, для семейства A1 XMEGA доступно всего одно немаскируемое прерывание — когда пропадает тактовая частота процессора.

Немаскируемые прерывания обслуживаются независимо от установки I-бита и никогда не изменяют этот бит. Они имеют самый высокий уровень приоритета — другие прерывания не могут повлиять на работу обработчика NMI. Если в одно и то же время обслуживания требуют несколько NMI, то порядок их обработки фиксирован — вектор с наименьшим адресом имеет высший приоритет.

Время отклика CPU на поступивший запрос на прерывание для всех разрешенных прерываний составляет как минимум 5 тактов системной частоты. В это время содержимое программного счетчика сохраняется в стеке. Затем выполняется безусловный переход на программу — обработчик прерывания, что занимает еще 3 такта. Если микроконтроллер находится в одном из энергосберегающих режимов, время реакции на исполнение увеличивается еще на 5 тактов плюс на время выхода микроконтроллера из текущего режима энергосбережения. Перед началом обслуживания любого запроса на прерывание процессорное ядро всегда завершает выполнение текущей инструкции.

Внутри каждого из уровней (Low, Medium, High) все прерывания имеют приоритет. Когда несколько запросов на прерывание находятся в очереди, порядок их обслуживания определяется и уровнем прерывания, и приоритетом. Обслуживание прерываний может быть организовано в статическом или динамическом (Round Robin) порядке. В микроконтроллерах XMEGA немаскируемые, High и Medium прерывания имеют только статический порядок обслуживания. Для прерываний с уровнем Low программисту доступна статическая и динамическая организация очередности обслуживания.

Вектора прерываний (IVEC) размещаются по фиксированным адресам в памяти программ микроконтроллера. Для статического порядка значение адреса вектора определяет очередность обслуживания внутри одного и того же уровня прерываний, где наименьший по абсолютному значению адрес вектора имеет наивысший приоритет. Таблицу векторов прерываний для каждого конкретного микроконтроллера XMEGA следует смотреть в технической документации.

Чтобы избежать возможных проблем с недоступностью обработчиков прерываний уровня Low в соответствии со статическим порядком обслуживания, PMIC предоставляет для прерываний этого уровня интересную возможность динамического перераспределения очередности — Round Robin или процедуру «карусельной диспетчеризации». Для этого в состав контроллера прерываний добавлен специальный регистр INTPRI. Когда процедура диспетчеризации Round Robin разрешена, в этом регистре запоминается вектор преры-

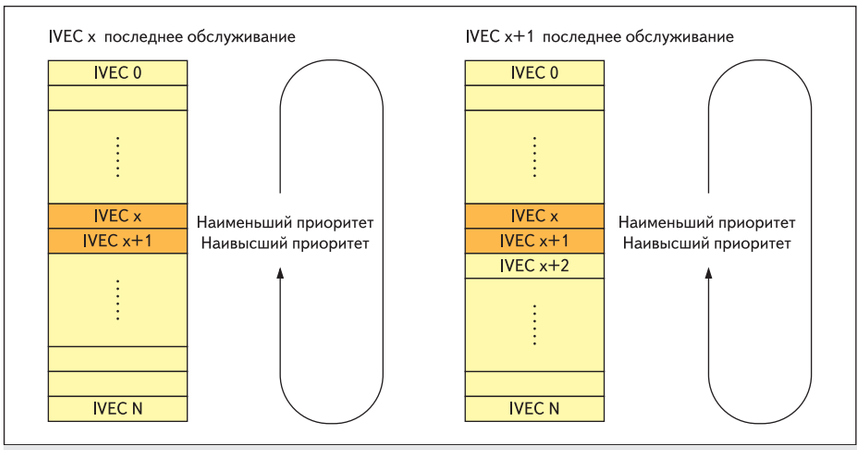

Рис. 2. Процедура «карусельной диспетчеризации» Round Robin

вания с уровнем Low, которое обслуживалось последним. Сохраненный вектор по отношению к его следующему вызову будет теперь иметь наименьший приоритет (рис. 2). Процедура Round Robin эффективно работает в тех случаях, когда в очереди находятся несколько разрешенных прерываний из уровня Low, которые будут обслуживаться в фиксированном циклическом порядке.

Отметим, что регистр PMIC.INTPRI доступен программисту для оперативного изменения порядка обслуживания и перестановок в очереди. Но он не сбрасывается к установкам по умолчанию, если процедуру «карусельной диспетчеризации» запретить в ходе выполнения программы. Поэтому для восстановления стандартного статического порядка (если необходимо) требуется записать в PMIC.INTPRI нулевое значение.

Практические примеры, демонстрирующие работу PMIC в микроконтроллерах XMEGA

По аналогии с разделом для портов ввода/вывода в качестве целевой платы использовался стартовый набор разработчика STK600 и интегрированная среда IAR Systems EWAVR 4.30. Для самостоятельной работы с примерами мы рекомендуем использовать руководство пользователя для STK600 («User Manual»).

Пример 1. Прерывание по переполнению таймера

Мигающие светодиоды на плате отладочного набора — это стандартная задача начального уровня. Реализуем ее с помощью управления прерываниями таймера/счетчика. Пример 1 показывает необходимые шаги для конфигурирования таймера (прерывание по переполнению). Используем таймер TCC0, порт D микроконтроллера для вывода и мигаем «младшим» светодиодом LED0:

```
#define LEDPORT PORTD
#define LEDMASK 0x01      // Используем LED0 для визуального
                          // отображения переключения

// Ассоциируем обработчик с вектором прерываний TCC0_OVF_vect
// (прерывание по переполнению):
#pragma vector = TCC0_OVF_vect
__interrupt void OverflowHandler( void )
  {
LEDPORT.OUTTGL = LEDMASK;  // Переключаем LED0 при
                           // переполнении таймера
                           // и вызове обработчика
                           // прерывания }


void main( void )
{
  // Конфигурируем порт вывода:
  LEDPORT.DIRSET = LEDMASK;
  LEDPORT.OUTSET = LEDMASK;

  // Конфигурируем таймер/счетчик 0:
  TCC0.PER = 10000; // Считаем до 10000 вместо 65536 для «ускорения».
  TCC0.CTRLA = TCC0.CTRLA & ~TC_CLKSEL_gm |
  TC_CLKSEL_DIV64_gc;  // Устанавливаем тактирование TCC0
                       // как CPUCLK/64.

  // Битовое поле Interrupt Level для периферийного узла XMEGA
  // должно быть установлено как Low, Medium или High.
  // Назначаем уровень прерывания для источника прерываний
  // (Low), но само прерывание пока не разрешаем:
  TCC0.INTCTRLA = TCC0.INTCTRLA & TC_OVFINTLVL_gm |
  TC_OVFINTLVL_LO_gc;  // Прерывание таймера/счетчика TCC0
                       // по переполнению, низкий уровень.

  // Разрешаем соответствующий уровень приоритета (Low)
  // для PMIC и разрешаем все прерывания
  PMIC.CTRL |= PMIC_LOLVLEN_bm;
  __enable_interrupt();

  // Запускаем бесконечный цикл…
  for (;;) {}
}
```

В целом, все просто и понятно, комментариев данный пример не требует.

Пример 2. Уровни прерываний

Если в программе обработки прерывания вдруг начинается бесконечный цикл, например ожидание нажатия клавиши пользователем, то это должно блокировать работу всего процессора, верно? Не совсем так для XMEGA! Пример 2 показывает, как взаимодействуют различные уровни прерываний. Здесь опять используются таймер TCC0 (в режиме сравнения) и порт D микроконтроллера для вывода. Но теперь переключаются три «младших» светодиода. Порт C микроконтроллера используем для ввода событий (нажатие клавиш на плате STK600), которые генерируют соответствующие запросы на прерывание. Тактирование TCC0 реализовано аналогично примеру 1.

```
#define LEDPORT PORTD
#define LEDMASKA 0x01 // переключение LED0 при прерывании
                      // от канала сравнения A.
#define LEDMASKB 0x02 // переключение LED1 при прерывании
                      // от канала сравнения B
#define LEDMASKC 0x04 // переключение LED2 при прерывании
                      // от канала сравнения C.

#define SWITCHPORT PORTC
#define SWITCHMASKA 0x01  // кнопка 0 для блокировки
                          // прерывания от канала сравнения A.
#define SWITCHMASKB 0x02  // кнопка 1 для блокировки
                          // прерывания от канала сравнения B.
#define SWITCHMASKC 0x04  // кнопка 2 для блокировки
                          // прерывания от канала сравнения C.

// Обработчик прерывания для TCC0, канал сравнения A.
#pragma vector = TCC0_CCA_vect
__interrupt void CompareMatchAHandler( void )
{
  // Переключаем соответствующий светодиод LED0
  LEDPORT.OUTTGL = LEDMASKA;
  // Бесконечный цикл, пока кнопка 0 остается нажатой.
  do {} while ((SWITCHPORT.IN & SWITCHMASKA) == 0x00);
}

// Обработчик прерывания для TCC0, канал сравнения B.
#pragma vector = TCC0_CCB_vect
__interrupt void CompareMatchBHandler( void )
{
  // Переключаем соответствующий светодиод LED1
  LEDPORT.OUTTGL = LEDMASKB;
  // Бесконечный цикл, пока кнопка 1 остается нажатой
  do {} while ((SWITCHPORT.IN & SWITCHMASKB) == 0x00);
}

// Обработчик прерывания для TCC0, канал сравнения C.
#pragma vector = TCC0_CCC_vect
__interrupt void CompareMatchCHandler( void )
{
  // Ппереключаем соответствующий светодиод LED2
  LEDPORT.OUTTGL = LEDMASKC;
  // Бесконечный цикл, пока кнопка 2 остается нажатой
  do {} while ((SWITCHPORT.IN & SWITCHMASKC) == 0x00);
}

void main( void )
{
  // Конфигурируем порты ввода/вывода:
  LEDPORT.DIRSET = LEDMASKA | LEDMASKB | LEDMASKC;
  LEDPORT.OUTSET = LEDMASKA | LEDMASKB | LEDMASKC;
  SWITCHPORT.DIRCLR = SWITCHMASKA | SWITCHMASKB |
                                          SWITCHMASKC;

  //Конфигурируем TCC0 :
  TCC0.PER = 10000;  // Значение «переполнения».
  TCC0.CCA = 5000;   // Устанавливаем одинаковые значения для
                     // всех каналов
  TCC0.CCB = 5000;   // сравнения, чтобы прерывания
                     // вызывались одновременно.
  TCC0.CCC = 5000;   // Это значение не должно быть больше
                     // содержимого регистра TCC0.PER.
  TCC0.CTRLB = TC_CCCEN_bm | TC_CCBEN_bm | TC_CCAEN_bm;

  // Назначаем различные уровни прерывания
  // (Low, Medium и High) для каналов сравнения:
  TCC0.INTCTRLB = (unsigned char) TC_CCCINTLVL_LO_gc |
          TC_CCBINTLVL_MED_gc | TC_CCAINTLVL_HI_gc;
  TCC0.CTRLA = TCC0.CTRLA & ~TC_CLKSEL_gm |
                                          TC_CLKSEL_DIV64_gc;

  // Разрешаем все уровни приоритета для PMIC и разрешаем
  // все прерывания.
  PMIC.CTRL |= PMIC_LOLVLEN_bm | PMIC_MEDLVLEN_bm |
                                          PMIC_HILVLEN_bm;
  __enable_interrupt();

  // Запускаем бесконечный цикл…
  for (;;) {}
}
```

После компиляции данного фрагмента кода и его запуска на STK600 можно наблюдать, как меняется поведение светодиодов. Если ни одна из кнопок SW0, SW1 и SW2 на плате STK600 не нажата, то все три светодиода «мигают» одновременно. При нажатии кнопки (или сразу нескольких кнопок) прерывание

«зацикливается» и состояние соответствующего светодиода фиксируется. Поведение остальных светодиодов будет определяться присвоенным статусом прерывания для соответствующих обработчиков. Так, например, при нажатии кнопки SW1 «зацикливается» обработчик прерывания с назначенным уровнем Medium. В результате состояние светодиодов LED1 (уровень Medium) и LED2 (уровень Low) «замораживается», а светодиод LED0 (уровень High) будет продолжать «мигать». Пример 2 показывает, как прерывания с более высоким уровнем приоритета могут приостанавливать работу обработчиков с более низким уровнем приоритета.

Пример 3. Round Robin или «карусельная диспетчеризация»

Когда несколько прерываний назначаются на один и тот же уровень, очередность их обслуживания определяет порядковый номер вектора прерываний. Чем меньше этот номер в абсолютном исчислении, тем раньше он обслуживается. Это — стандартный порядок. Но что произойдет, если какое-то прерывание требует очень много процессорного «внимания»? Тогда прерывания с более высоким порядковым номером никогда не будут обслуживаться. Это действительно будет так… до тех пор, пока не разрешена процедура Round Robin в PMIC. Теперь последний вектор прерываний, который обслуживался процессором, будет иметь самый низкий приоритет! Он перемещается в «хвост» кольцевой очереди. Пример 3 показывает, как использовать Round Robin для одновременной работы двух прерываний, даже если одно из них занимает почти 100% процессорного времени. Еще раз отметим, что процедура «карусельной диспетчеризации» доступна только для прерываний с установленным уровнем Low.

В примере 3 используются таймеры TCC0 и TCC1 (в режиме переполнения), порт D для вывода и порт C для ввода. Значения частоты тактирования таймеров здесь существенно различаются:

```
#define LEDPORT PORTD
#define LEDMASK0 0x01 // Светодиод LED0 ассоциирован с TCC0
#define LEDMASK1 0x02 // Светодиод LED1 ассоциирован с TCC1

#define SWITCHPORT PORTC
#define SWITCHMASK 0x04   // Кнопка SW3 используется для
                          // включения/выключения
                          // процедуры Round Robin.

// Ассоциируем обработчик с вектором прерываний TCC0_OVF_vect
// (прерывание по переполнению):
#pragma vector = TCC0_OVF_vect
__interrupt void Timer0Handler( void )
{
   // Так как этот обработчик вызывается очень часто, то мы
   // используем счетчик для понижения частоты «мигания»
   // светодиода до комфортного уровня восприятия.
   static long countdown = 0;
   if (countdown-- == 0) {
      countdown = 1000;
      LEDPORT.OUTTGL = LEDMASK0;
   }
}

// Ассоциируем обработчик с вектором прерываний TCC1_OVF_vect
// (прерывание по переполнению):
#pragma vector = TCC1_OVF_vect
__interrupt void Timer1Handler( void )
{
   LEDPORT.OUTTGL = LEDMASK1;
}

void main( void )
{
   // Конфигурируем порт вывода:
   LEDPORT.DIRSET = LEDMASK0 | LEDMASK1;

   // Устанавливаем TCC0 на очень частую генерацию запросов
   // на прерывание:
   TCC0.PER = 1;
   TCC0.CTRLA = TCC0.CTRLA & ~TC_CLKSEL_gm |
                                        TC_CLKSEL_DIV1_gc;
   TCC0.INTCTRLA = TCC0.INTCTRLA & ~TC_OVFINTLVL_gm |
                                        TC_OVFINTLVL_LO_gc;

   // Устанавливаем TCC1 на «умеренную» частоту генерации
   // запросов на прерывание:
   TCC1.PER = 10000;
   TCC1.CTRLA = TCC1.CTRLA & ~TC_CLKSEL_gm |
                                        TC_CLKSEL_DIV64_gc;
   TCC1.INTCTRLA = TCC1.INTCTRLA & ~TC_OVFINTLVL_gm |
                                        TC_OVFINTLVL_LO_gc;

   // Разрешаем уровень приоритета прерываний Low в PMIC
   // и разрешаем все прерывания:
   PMIC.CTRL |= PMIC_LOLVLEN_bm;
   __enable_interrupt();

   // Запускаем бесконечный цикл...
   for (;;) {
      // Считываем состояние кнопки и включаем/выключаем
      // процедуру Round Robin. Поскольку CPU будет выполнять
      // как минимум одну инструкцию между двумя вызовами
      // обработчиков, приведенный ниже код будет
      // выполняться, хотя и очень медленно.


      if ((SWITCHPORT.IN & SWITCHMASK) == 0x00) {
         PMIC.CTRL |= PMIC_RREN_bm;}
       else {
         PMIC.CTRL &= ~PMIC_RREN_bm;
         PMIC.INTPRI = 0;  // Сброс приоритета к значению
                           // по умолчанию. PMIC не сделает
                           // это за нас автоматически…}
      }
}
```

После компиляции фрагмента кода из примера 3 и его запуска на STK600 можно наблюдать, что светодиод LED0 постоянно переключается, а светодиод LED1 не меняет своего состояния. Если нажать и удерживать кнопку SW3, то светодиод LED1 начинает изредка «мигать». Это свидетельствует о том, что даже самое «забитое» по статусу прерывание иногда в порядке очереди получает право на обслуживание именно благодаря процедуре диспетчеризации Round Robin в микроконтроллерах XMEGA. При отпускании кнопки восстанавливается статический порядок обслуживания прерываний, и светодиод LED1 перестает «мигать» — процессор снова занят исключительно переключением LED0.

Дополнительную информацию об особенностях работы с программируемым многоуровневым контроллером прерываний PMIC в микроконтроллерах XMEGA можно найти в документации Atmel — Application Note AVR1305.

Система событий

Система обработки событий (Event System) у микроконтроллеров XMEGA разрабатывалась для того, чтобы максимально разгрузить центральный процессор. Она организована внутри кристалла как набор аппаратных средств, с помощью которых различные периферийные модули могут обмениваться между собой служебной или командной информацией, а также передавать и принимать данные. Система событий доступна в активном режиме работы микроконтроллера и в одном из энергосберегающих режимов — Idle. Индикация изменения состояния периферийного модуля у XMEGA носит название «событие». Система событий дает возможность по изменению состояния одного периферийного модуля автоматически вызывать определенные действия в других периферийных модулях. Причем характер принятия решения, а именно по какому из внешних возмущений может генерироваться определенное действие, программируется пользователем. При этом сохраняется исключительная гибкость в предоставлении прав каждому из периферийных узлов, которые подключены к системе событий.

В результате у разработчиков AVR XMEGA получилось предельно простое, но очень мощное аппаратное средство управления автономным функционированием периферии без участия CPU, DMA и контроллера преры-

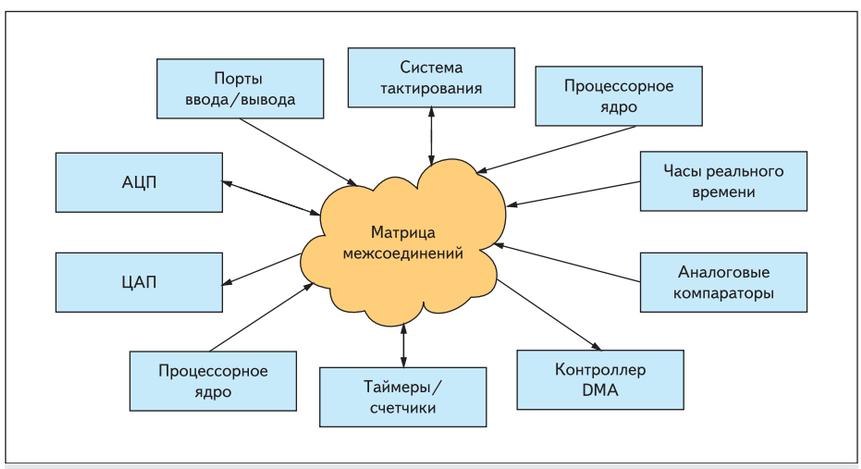

Рис. 3. Блок-схема Event System

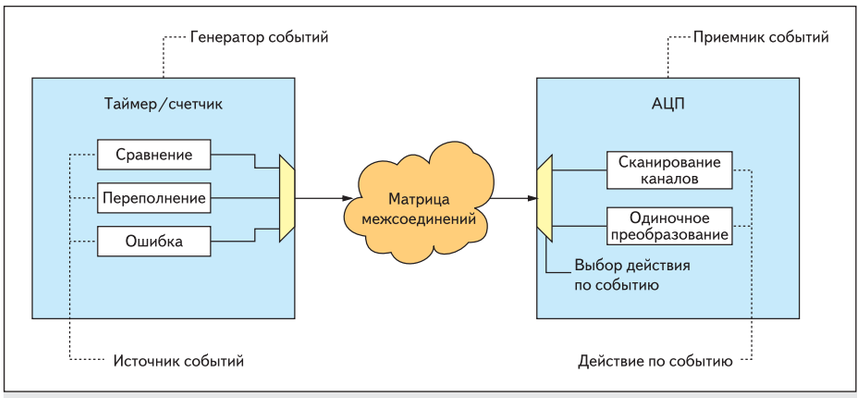

Рис. 4. Пример генератора, источника, приемника событий и действий по событию

ваний. С помощью Event System значительно экономится процессорное время и существенно разгружается система прерываний микроконтроллера, в основном за счет снижения количества запросов на прерывание. Блок-схема Event System показана на рис. 3.

Событие может генерироваться при изменении состояния сигнала на внешнем выводе микроконтроллера, при срабатывании блока захвата/сравнения таймера, при переключении аналогового компаратора и т. п. Поступающее на выбранный периферийный модуль событие также может инициировать различные действия — инкрементировать таймер, начать передачу данных через DMA, запретить выход сигнала ШИМ на внешний вывод микроконтроллера и т. п.

Аппаратный узел, от которого исходит событие, называется генератором события. Внутри каждого периферийного модуля может быть несколько источников событий. Например, у таймера/счетчика — это сравнение, переполнение или флаг ошибки. Аппаратный узел, на который поступает событие, называется приемником событий, а действие, которое он реализует, называется действием по событию. На рис. 4 проиллюстрированы все введенные понятия и их взаимодействие.

Существует два типа событий — Signaling и Data. События «Signaling» лишь показывают изменение состояния, а события «Data» содержат еще и дополнительную информацию. «Signaling» — это основной тип генерируемых событий. Подавляющее большинство периферийных модулей может генерировать и принимать лишь такие события. Поэтому, когда генерируется событие «Data» и оно приходит на периферийный модуль, который не может обрабатывать содержащуюся в событии информацию, оно автоматически интерпретируется как «Signaling». Приемники событий, которые могут обрабатывать оба типа событий, должны быть программным образом сконфигурированы на прием «Data» или «Signaling» перед началом работы. Подробную информацию следует смотреть в техническом описании на периферийные модули микроконтроллеров XMEGA.

В понятие «система событий» разработчики XMEGA включили генераторы событий, средства для передачи событий и приемники событий. События передаются между периферийными модулями при помощи специализированной матрицы соединений на кристалле XMEGA, которая носит название Event Routing Network. Она состоит из 8 мультиплексоров и собственно матрицы межсоединений, где реализовано до 8 параллельных каналов передачи сигналов (Event Channels) и все события «разводятся» на все мультиплексоры.

Центральный процессор (CPU) не является частью системы событий, но показан на блок-схеме Event System. Это означает, что у программиста имеется возможность управлять генерацией событий из программы микроконтроллера или в режиме debug, используя встроенные средства внутрисхемной отладки на кристалле XMEGA. Для этой цели в структуре Event System предусмотрены два специальных регистра — DATA и STROBE. Регистры доступны как в ходе выполнения программы, так и в режиме отладки, когда в AVR Studio можно изменять их содержимое в окне IO View и, соответственно, генерировать события. Каждый бит в регистрах соответствует определенному каналу Event System: бит 0 — каналу 0, бит 1 — каналу 1 и т. д. По умолчанию, в обоих регистрах записаны нулевые значения. Установка любого бита в любом из регистров инициирует определенное действие. Регистр DATA ответственен за тип генерируемого события и должен быть изменен первым, так как запись в регистр STROBE лишь генерирует событие. Подробнее соотношение и структура работы представлены в таблице 1.

Таблица 1. Соотношение и структура работы регистров DATA и STROBE

STROBE	DATA	События «Data»	События «Signaling»
0	0	Нет события	Нет события
0	1	Данные, тип 1	Нет события
1	0	Данные, тип 2	Генерация события
1	1	Данные, тип 3	Генерация события

Отметим, что все каналы в Event System аппаратно независимы. Поэтому у программиста имеется возможность генерировать до восьми синхронных событий, одновременно устанавливая биты желаемых каналов в регистрах DATA и STROBE.

Полная структурная схема системы событий микроконтроллеров XMEGA приведена на рис. 5. Здесь наглядно видно, как организована матрица межсоединений и каким образом мультиплексоры могут коммутировать события между генераторами и приемниками. Каждый мультиплексор имеет два регистра управления, доступные программисту. Регистр CHnMUX определяет, какое из входящих на мультиплексор событий коммутируется на соответствующий выходной канал событий Event Channel (n = 0 …7). Регистр CHnCTRL используется на выходе мультиплексора для организации дополнительных функций — фильтрации и квадратурного декодирования. Наличие восьми мультиплексоров означает, что у разработчика имеется возможность «развести» до восьми событий одновременно. Также можно «распараллелить» одно и то же событие на несколько мультиплексоров. Структура матрицы соединений в Event System одинакова для всех микроконтроллеров XMEGA. Конечно, представители разных подсемейств XMEGA имеют различные наборы периферии, но с точки зрения Event System это означает, что отсутствующий периферийный модуль не может генерировать и принимать события.

Как правило, генерация события занимает один период системной тактовой частоты. Отметим, что некоторые генераторы событий (например, изменившийся уровень на линии порта ввода/вывода) формально имеют возможность генерировать события постоянно. Забота об отслеживании таких ситуаций лежит на программисте. От момента генерации события до момента, когда оно «защелкивается» в приемнике, проходит два периода системной тактовой частоты. Первый — от наступления события до его регистрации матрицей межсоединений Event System по первому фронту тактового сигнала. Второй такт требуется на трансляцию события через Event Channel до приемника. Поэтому временны́е параметры работы системы событий практически точно предсказуемы. Это обстоятельство значительно повышает надежность принятия решений и обработки данных: чувствительные ко времени или к стабильности выполнения функции теперь становятся более прогнозируемыми, а значит, увеличивается надежность работы всего микроконтроллера в целом.

Рассмотрим подробнее дополнительные возможности Event System, которые «привязаны» к выходам мультиплексоров и соотносятся с каналами передачи событий Event Channels. Этими дополнительными возможностями управляет регистр CHnCTRL.

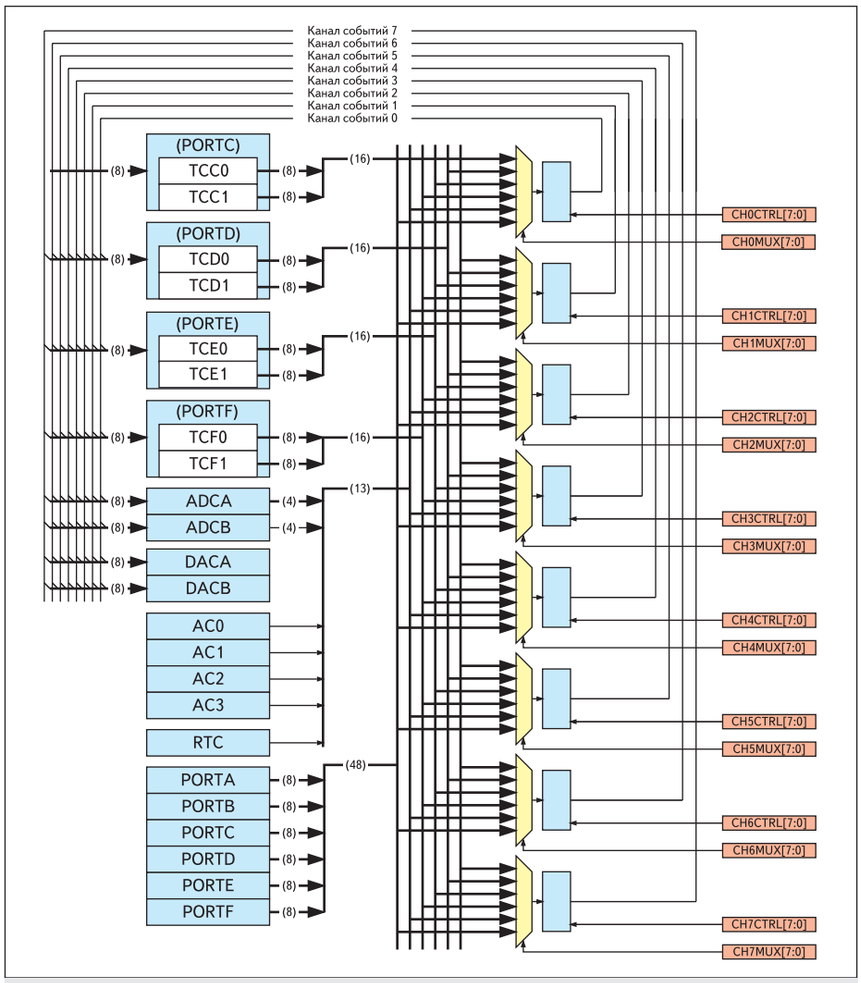

Рис. 5. Структурная схема системы событий в XMEGA

Первая дополнительная функция — цифровая фильтрация. Каждый канал системы событий снабжен программируемым цифровым фильтром. Поступающее событие будет опрашиваться определенное число раз перед тем, как оно будет передано дальше по направлению к приемнику события. Основное назначение такой фильтрации — опрос изменения состояния входных линий микроконтроллера. Другими словами, теперь можно аппаратно устранить дребезг контактов, которые непосредственно подключаются к выводам микроконтроллера. Такая функция избавит разработчика от необходимости писать соответствующую противодребезговую процедуру и снизит нагрузку на центральный процессор.

Работа цифрового фильтра разрешена всегда. Три младших бита регистра CHnCTRL определяют длительность временного промежутка, в течение которого источник события постоянно опрашивается. Событие будет отправлено в соответствующий канал только после того, как стабильность состояния источника события будет подтверждена в течение нескольких периодов периферийного тактового сигнала. Количество опросов (от одного до восьми) определяется тремя битами DIGFILT[2:0].

Вторая дополнительная функция — работа с квадратурно-кодированными сигналами. В состав системы событий для каналов с номерами 0, 2 и 4 включены три квадратурных декодера (QDEC). Это позволяет Event System декодировать входящие квадратурно-кодированные сигналы на линиях порта ввода/вывода и затем генерировать и посылать события «Data», которые могут интерпретироваться таймером/счетчиком и вызывать соответствующие действия по событию: считать «вверх», считать «вниз» или вырабатывать сигнал ошибки или сброса. Таблица 2 показывает, как события «Data» интерпретируются таймером/счетчиком при квадратурном декодировании.

Таблица 2. Интерпретация события «Data»

STROBE	DATA	События «Data»	События «Signaling»
0	0	Нет события	Нет события
0	1	Сигнал ошибки/сброс	Нет события
1	0	Считать «вниз»	Генерация события
1	1	Считать «вверх»	Генерация события

Напомним, что квадратурными называют сигналы, представляющие собой последовательности прямоугольных импульсов, которые смещены друг относительно друга на фазу 90°. Для создания таких сигналов применяют специальные изделия — энкодеры, которые используют кодирующие линейчатые маски и кодирующие диски. Например, двухканальные поворотные энкодеры широко используются для измерения углов поворота функциональных элементов в роботах и металлообрабатывающих станках. Выходные периодические импульсные последовательности, сдвинутые по фазе на 90°, позволяют с помощью внешних устройств обработки определять направление вращения вала и осуществлять двунаправленное позиционирование. Направление вращения определяется по фазовому соотношению сигналов в обоих каналах при анализе фронтов. Величина скорости определяется измерением интервала времени между импульсами или числом импульсов в пределах заданного временного интервала.

На рис. 6 показаны типичные квадратурные сигналы от поворотного энкодера

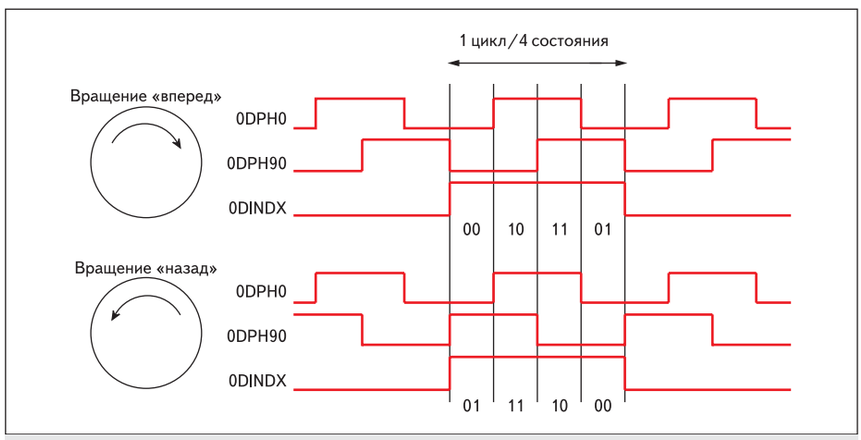

Рис. 6. Квадратурные сигналы от поворотного энкодера

и принцип распознавания направления вращения. В микроконтроллерах XMEGA им присвоены символические имена QDPH0 и QDPH90. В структуру Event System также включен еще один сигнал QDINDX, который генерируется один раз за оборот энкодера и может использоваться (в зависимости от требований конкретной задачи) для определения абсолютного положения, для генерации сигнала ошибки при декодировании, для индексации состояния счетчика (подсчет числа оборотов) или для его сброса.

Процедура организации квадратурного декодирования системой событий подробно описана в руководстве пользователя на микроконтроллеры XMEGA. Необходимо сконфигурировать как минимум две линии порта ввода/вывода микроконтроллера и таймер/счетчик на прием событий «Data» от Event System. В результате из регистра Count таймера/счетчика можно считывать угол поворота энкодера, а также принимать решения о необходимой реакции на изменение сигнала QINDX.

Практические примеры, демонстрирующие работу Event System в микроконтроллерах XMEGA

По аналогии с разделом для PMIC, в качестве целевой платы использовался стартовый набор разработчика STK600 и интегрированная среда IAR Systems EWAVR 4.30. Компания Atmel предлагает для знакомства с работой Event System ряд примеров, которые приведены в оригинальной документации Application Note AVR1001. Для самостоятельной работы с примерами мы также рекомендуем использовать руководство пользователя для STK600 («User Manual»).

Пример 1. Построение 32-разрядного таймера

Идея задачи очень проста — использовать событие, которое генерируется по переполнению одного таймера/счетчика, в качестве входного тактового сигнала для другого таймера/счетчика. Сигнал переполнения передается как событие «Signaling» по каналу Event System:

```
#define LEDPORT PORTD  // Используем порт D для вывода
                       // информации на светодиоды STK600

void main( void ){
   LEDPORT.DIR = 0xFF;   // Конфигурируем порт вывода
   LEDPORT.OUT = 0xFF;

// Конфигурируем приемник события (TCC1): какой канал будет
// источником тактирования периферийного узла и какое
// действие нужно осуществить при поступлении события:

   TCC1.CTRLA = TC_CLKSEL_EVCH0_gc; // Выбираем канал 0
// как источник тактирования TCC1

// Для системы событий необходимо явно указать, какое событие
// «разводится» на мультиплексор для каждого используемого
// канала событий. Выбираем переполнение таймера/счетчика
// TCC0 как входное событие для входа мультиплексора канала 0:

EVSYS.CH0MUX = EVSYS_CHMUX_TCC0_OVF_gc;
// Это весь код, который необходим для конфигурирования
// одного канала системы событий!

// Таймер/счетчик TCC0 используется в качестве 16 младших раз
// рядов нашего таймера. Для тактирования используем сигнал
// системной частоты, поделенной на 2:

   TCC0.PER = 0xFFFF;
   TCC0.CTRLA = TC_CLKSEL_DIV2_gc;

// Запускаем бесконечный цикл и «мигаем» светодиодами
 for(;;) {
    if (TCC0.CNT >= 0xFFF0) {
    __no_operation(); }

   LEDPORT.OUT = ~TCC1.CNTL;

   }

}
```

В целом все просто и понятно, комментариев данный пример не требует. Немного расширив код самостоятельно, можно организовать 48-разрядный или даже 64-разрядный счетчик.

Поскольку система событий не требует кода для пересылки событий, то момент переключения от TCC0 на TCC1 можно посмотреть в режиме debug, добавив в текст программы точку останова:

```
    if (TCC0.CNT >= 0xFFF0) {
    •__no_operation(); }
```

В процессе выполнения кода программа останавливается практически перед переполнением «младшего» таймера. Если сделать несколько последовательных шагов, таймер TCC0 переполнится и состояние светодиодов на плате STK600 изменится. Если раскрыть окно **IO view/TCC1** в AVR Studio, то можно наблюдать, как увеличивается CNT в момент переполнения TCC0.

Аналогично приведенному примеру с помощью системы событий можно реализовать 32-разрядный счетчик с функцией захвата. Программный код, реализующий эту задачу, приведен в Application Note AVR1001 (Example 3) и здесь он рассматриваться не будет.

Пример 2. Цифровая фильтрация

До сих пор мы использовали таймеры/счетчики для сравнения и генерации выходных цифровых последовательностей. В этом примере будет реализована функция захвата. Как известно, у таймеров/счетчиков имеются фиксированные входы, аппаратно соотнесенные с определенными выводами микроконтроллера. Это не всегда удобно при разводке целевой платы. Но можно использовать систему событий — подать на вход блока захвата таймера/счетчика входной сигнал от произвольной линии порта ввода/вывода, что значительно расширит диапазон доступных выводов микроконтроллера для реализации этой задачи.

Каждая линия порта ввода/вывода у XMEGA может быть запрограммирована на генерацию событий (или прерываний) по фронту, срезу, перепаду или низкому уровню сигнала. Генерируемое событие затем может быть направлено на таймер/счетчик, работающий в режиме захвата. Код, реализующий данную задачу, приведен в Application Note AVR1001 (Examples 1, 4) и копировать мы его здесь не будем. Кнопка на плате STK600, которая будет опрашиваться на нажатие, подключена к линии 0 порта D. Программируя цифровой фильтр на входе Event System на разные значения, можно наблюдать, сколько нажатий кнопки требуется для того, чтобы событие прошло далее на счетчик и инициировало захват. Удобнее всего это делать в режиме debug, раскрыв окно **IO view/Event System** в AVR Studio.

Пример 3. Программная генерация событий

События могут генерироваться «вручную» из программы или в режиме debug. Это осуществляется путем записи в регистры DATA и STROBE из программы или путем прямого доступа к содержимому регистров в окне отладчика AVR Studio (напомним, что регистр DATA должен записываться первым). Регистры DATA и STROBE содержат отдельные биты для каждого из каналов системы событий, то есть бит n соответствует каналу n. Поэтому существует возможность генерации событий на нескольких каналах одновременно, если установить в регистрах сразу несколько выделенных битов. Программная генерация может быть полезна для синхронизации событий и для отладки программы. Такие события завершаются в течение одного такта и будут превалировать над другими событиями в течение этого же такта:

```
#define LEDPORT PORTC  // Используем порт C для вывода
                       // информации на светодиоды STK600

volatile uint16_t capture_values[3];
volatile uint8_t adc_result;

// Простая функция, которая конфигурирует таймер в режим
// захвата. Канал событий — входной параметр:
void TC_init(volatile TC_t * tc, uint8_t EVSEL_CH){
    tc->CTRLD |= EVSEL_CH;
    tc->CTRLD |= TC_EVACT_CAPT_gc;
    tc->CTRLB = TC_CCAEN_bm;
    tc->CTRLA = TC_CLKSEL_DIV1_gc; }

void main( void ){

 uint16_t capture_values[3];
 uint8_t adc_result;

 // Конфигурируем порт вывода
LEDPORT.DIR = 0xFF;
LEDPORT.OUT = 0xFF;

// Теперь инициализацию таймера/счетчика (см. примеры 1 и 2)
// осуществляем с помощью функции:
TC_init(&TCC0, TC_EVSEL_CH0_gc);
TC_init(&TCC1, TC_EVSEL_CH1_gc);
TC_init(&TCD1, TC_EVSEL_CH2_gc);

/ * Эта функция конфигурирует ADCA на начало преобразования
  * в канале 0 по событию с канала 7 системы событий (подробно
  * инициализация АЦП будет рассмотрена в следующей части
  * статей) */
 ADC_init(&ADCA);

// Запускаем «медленный» таймер в качестве времязадающего
// узла для программной генерации событий
TCF0.PER = 0x3000;
TCF0.CTRLA = TC_CLKSEL_DIV256_gc;

do {} while ((TCF0.INTFLAGS & TC_OVFIF_bm) == 0);

for(;;) {

   do {} while ((TCF0.INTFLAGS & TC_OVFIF_bm) == 0);

    TCF0.INTFLAGS = TC_OVFIF_bm;
    LEDPORT.OUTTGL = 0xFF;
```

```
/* Код устанавливает биты в регистре STROBE для программной
  * генерации событий на каналах событий 0, 1, 2 и 7. Отметим,
  * что программно генерируемые события игнорируют установки
  * MUC канала событий. В действительности, мультиплексоры
  * вообще могут не конфигурироваться в случае использования
  * программно генерируемых событий */

EVSYS.STROBE = EVSYS_CHMUX0_bm | EVSYS_CHMUX1_bm |
                    EVSYS_CHMUX2_bm | EVSYS_CHMUX7_bm;

//… или проще, но не так наглядно то же самое можно написать
// как: EVSYS.STROBE =0x87;

   capture_values[0] = TCC0.CCA;
   capture_values[1] = TCC1.CCA;
   capture_values[2] = TCD1.CCA;
   adc_result = ADCA.CH0RESL;
   LEDPORT.OUT = ~adc_result;

 }

}
```

Выполнив компиляцию проекта, следует открыть файл ***.dbg*** (сгенерированный отладочный код в формате UBROF-8) в AVR Studio и установить точку останова, как показано ниже. Это необходимо для пошаговой отладки после генерации программно генерируемых событий:

```
    do {} while ((TCF0.INTFLAGS & TC_OVFIF_bm) == 0);
•   TCF0.INTFLAGS = TC_OVFIF_bm;
    LEDPORT.OUTTGL = 0xFF;
```

Теперь следует добавить переменные *adc_result* и *capture_values* в окно **Watch** для визуального наблюдения за изменениями результатов преобразования АЦП и захваченными значениями. После запуска кода на исполнение и остановки в назначенной точке следует открыть окно **IO view/Event System** и продолжать отладку по шагам. При этом будет видно, как регистр STROBE сначала записывается и затем сбрасывается на следующем такте. Если продолжать пошаговое выполнение программы, то можно наблюдать, как обновляются переменные *adc_result* и *capture_values* после поступления события на блок захвата таймера/счетчика и начала преобразования АЦП.

Содержимое регистра STROBE также может быть изменено непосредственно в ходе отладки программы, например, принудительной установкой битов этого регистра в окне **IO view/Event System**. Установленные биты будут автоматически сбрасываться на следующем такте. При таком способе «манипулирования» событиями следует иметь в виду, что захваченные значения таймера/счетчика не сохраняются, если буфер уже заполнен. В этом случае мы получим лишь сигнал ошибки.

Пример 4. Генерация синхронных событий

В предыдущем примере использовалась программная генерация событий для синхронизации событий, поступающих на различные каналы. Существует также возможность разрешать нескольким периферийным модулям использовать один и тот же канал системы событий в качестве источника событий. Тогда эти периферийные модули могут синхронизировать свои действия. Типичным примером может служить реализация функции захвата таймера/счетчика и одновременный старт преобразования АЦП, чтобы «проштемпелевать» факт начала преобразования меткой времени.

Приведенный ниже код во многом совпадает примером 3, но здесь конфигурируется мультиплексор седьмого канала системы событий на использование переполнения таймера/счетчика в качестве источника событий:

```
#define LEDPORT PORTC   // Используем порт C для вывода
                        // информации на светодиоды STK600

volatile uint16_t time_stamp;
volatile uint8_t adc_result;

// Конфигурируем таймер в режим захвата.
// Канал событий — входной параметр:
void TC_init(volatile TC_t * tc, uint8_t EVSEL_CH){
 tc->CTRLD |= EVSEL_CH;
 tc->CTRLD |= TC_EVACT_CAPT_gc;
 tc->CTRLB = TC_CCAEN_bm;
 tc->CTRLA = TC_CLKSEL_DIV8_gc; }

void main( void ){

// Переменные для значения захвата и результата
// преобразования АЦП
uint16_t capture_value;
uint8_t adc_result;

// Организуем небольшой кольцевой буфер для хранения результатов
uint8_t index = 0;
uint16_t result_buffer[2][32];


// Конфигурируем порт вывода
LEDPORT.DIR = 0xFF;
LEDPORT.OUT = 0xFF;

// Таймер/счетчик и АЦП должны использовать один и тот же
// канал системы событий по входу

// Инициализируем таймер/счетчик
TC_init(&TCC0, TC_EVSEL_CH7_gc);

/* Конфигурируем ADCA на начало преобразования в канале 0
  * по событию с канала 7 системы событий (подробно инициа-
  * лизация АЦП будет рассмотрена в следующей части статей) */
ADC_init(&ADCA);

/*****************************************************************
// Закомментированный в этом фрагменте код используется
// только во второй части примера
// Устанавливаем порт D на ввод информации, конфигурируем
// линию 0, назначаем событие для канала 7:
// PORTD.PIN0CTRL = PORT_ISC_FALLING_gc;
// EVSYS.CH7MUX = EVSYS_CHMUX_PORTD_PIN0_gc;
*****************************************************************/

/* Для системы событий необходимо явно указывать, какое
  * событие следует подавать на мультиплексор для каждого
  * используемого канала. Код ниже определяет это как перепол-
  * нение TCF0, что совпадает с установками АЦП и TCC0 */

EVSYS.CH7MUX = EVSYS_CHMUX_TCF0_OVF_gc;

/* Запускаем таймер, который будет использоваться в качестве
  * времязадающего узла для программно генерируемых событий */
TCF0.PER = 0x3000;
TCF0.CTRLA = TC_CLKSEL_DIV64_gc;

 for(;;) {

// Ждем, пока будет установлен флаг прерывания АЦП
// (т. е. преобразование было запущено и завершено)
   do{}
    while((ADCA.INTFLAGS & ADC_CH0IF_bm) != ADC_CH0IF_bm);

   ADCA.INTFLAGS |= ADC_CH0IF_bm;;
   LEDPORT.OUTTGL = 0xFF;

   capture_value = TCC0.CCA;      // считываем захваченную
                                  // метку времени
   adc_result = ADCA.CH0RESL;
   LEDPORT.OUT = ~adc_result; // Выводим результат
                              // преобразования
                              // на светодиоды STK600

   /* Сохраняем результат в кольцевом буфере — только для
     * того, чтобы иметь в окне Watch краткую «сводку» реальных
     * результатов преобразования АЦП и захваченных значений */

   result_buffer[0][index] = capture_value;
   result_buffer[1][index] = adc_result;

   index ++;
   index = index & 0x1F; // маскируем 3 старших бита для
                         // функционирования кольцевого буфера

 }

}
```

Выполним компиляцию проекта и загрузим файл ***.dbg*** в AVR Studio. Запустив программу на выполнение, можно наблюдать в окнах отладчика генерацию событий в ожидаемые моменты времени, начало преобразований аналого-цифрового преобразователя и захват в таймере/счетчике. Светодиоды на плате STK600 будут индицировать текущий результат преобразования АЦП.

Конечно, использование отдельного таймера для генерации событий с целью формирования метки времени с помощью другого таймера/счетчика, по меньшей мере, нелогично. Любой таймер/счетчик в микроконтроллерах XMEGA имеет свои собственные блоки захвата/сравнения, которые могут быть использованы для решения нашей задачи. Будем считать пройденное решение «академическим».

Модифицируем пример для того, чтобы старт преобразования АЦП и генерация временной метки инициировались событием от другого источника. В качестве генератора событий будет выступать кнопка на плате STK600, а источником события — факт нажатия этой кнопки. Для этого следует убрать комментарии со строк кода программы во фрагменте, отделенном символами «*», и закомментировать установки таймера TCF0 и предыдущие установки канала 7 системы событий. На плате STK600 необходимо также соединить гибким шлейфом контакт разъема PD0 с любой из кнопок SW. Тогда после компиляции текста, загрузки файла ***.dbg*** в AVR Studio и выполнения программы каждое нажатие выбранной нами кнопки запускает АЦП на преобразование и сохраняет метку времени об этом событии в буфере.

Дополнительную информацию об особенностях работы системы событий в микроконтроллерах XMEGA можно найти в документации Atmel — Application Note AVR1001.

Продолжение следует

Литература

1. Евстифеев А. В. Микроконтроллеры AVR семейств Tiny и Mega фирмы «Atmel». М.: Издательский дом «Додэка-XXI». 2004.
2. XMEGA Training Data // Atmel AVR Distributor Training. Atmel Norway, September 2007.
3. XMEGA Hands-On Session // Atmel AVR Distributor Training. Atmel Norway, September 2007.
4. www.atmel.com