"Бегущие огни" на ОСРВ

Статья рассматривает простой пример разработки программы на PIC-контроллере с использованием ОСРВ [1)]OSA и является пособием по применению ОСРВ в PIC-контроллерах для начинающих. В качестве аппаратной базы выбраны демо-платы из комплектов PicKit2 на базе контроллеров PIC16F886, PIC16F887 и PIC16F690:

- DV164120 - PICkit 2 Starter Kit
  Программатор PICkit2 + демонстрационная плата с PIC16F690
- DV164121 - PICkit™ 2 Debug Express
  Программатор PICkit2 + демонстрационная плата с контроллером PIC16F887

Введение

Чаще всего контроллеры, используемые в электронных устройствах, выполняют сразу несколько задач: интерфейс с пользователем, обмен данными с другими устройствами, слежение за внешними сигналами, управление какими-то электрическими цепями и т.д. Поэтому при написании программ программисту приходится задумываться не только о том, как выполнить какую-то конкретную задачу, но и о том, как распределить ресурсы контроллера между всеми возлагаемыми на него задачами. В первую очередь речь, конечно же, идет о временных ресурсах и о ресурсах памяти. Т.е. приходится думать о том, чтобы все задачи успевали все сделать вовремя, не мешая остальным задачам, и чтобы на все задачи хватило памяти (и программной и памяти данных). Кроме того, нужно будет продумывать способы синхронизации задач между собой (например, если мы делаем термометр, то температуру нельзя выводить на экран до того, как она будет измерена).

Получается, что большинство программ, написанных для микроконтроллеров, являются многозадачными приложениями. И программисту приходится каждый раз, начиная новую программу, прилагать немало усилий для продумывания механизмов обеспечения многозадачности, оптимальных для данной программы. Вполне резонно возникает вопрос: раз проектирование таких механизмов - это такая частая операция, то нет ли возможности его автоматизировать? Ответ: такая возможность есть - это использование многозадачной операционной системы, которая уже имеет механизмы распределения ресурсов контроллера между задачами, а также средства синхронизации задач между собой. И все, что программисту понадобится, - это изучить работу с операционной системой, а дальше работать с ней, избавив себя от лишней головной боли, связанной с организацией многозадачности и синхронизации задач, и позволив себе тем самым сконцентрироваться на решении конкретных задач.

Однако следует помнить, что операционная система (ОС), распределяя ресурсы контроллера между задачами, и сама тоже требует под свои нужды кое-какие ресурсы (и процессорное время и память). Поэтому, научившись использовать ОС для создания своих проектов, нужно помнить, что ее применение не всегда оправдано. Иногда это бывает просто неудобно, иногда малоресурсный контроллер не способен вместить в себя и все задачи и ОС, иногда время, отнимаемое операционной системой, не позволяет задачам работать с нужно скоростью. Со временем опыт позволит еще на этапе проектирования принять решение: использовать ОС или нет.

Более подробно об основах операционных систем реального времени можно прочитать здесь:

- Тим Уилмсхерст, «Разработка встроенных систем с помощью микроконтроллеров PIC» - в 18-ой главе неплохое описание основ ОСРВ. Книгу можно найти в сети.
- Описание ОСРВ Salvo [http://www.pumpkininc.com/content/doc/manual/SalvoUserManual-rus-v3.2.3.pdf] - рекомендую прочитать вторую главу;
- ОСРВ jacOS - стоит почитать описание от автора в папке «doc»;
- Программирование контроллеров с использованием ОСРВ OSA [http://wiki.pic24.ru/doku.php/osa/articles/rtos_usage]

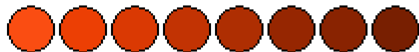
Здесь мы рассмотрим применение операционной системы реального времени OSA [http://www.pic24.ru/doku.php/osa/ref/intro] для разработки программы «Бегущие огни». В качестве аппаратной базы будем использовать демо-плату из комплекта PicKit2 с установленным на ней контроллером PIC16F886, PIC16F887 или PIC16F690.

Итак, в нашем распоряжении 4 светодиода (на демо-плате с контроллером 16F887 - 8 светодиодов), кнопка и резистор с переменным сопротивлением. Зададимся задачей сделать программу, управляющую светодиодами

так, чтобы:

1. каждый из них в один момент времени имел свою яркость;
2. яркость светодиодов циклически (по кругу) менялась, создавая эффект движения (вращения);
3. кнопкой менялось направление движения;
4. переменным сопротивлением регулировалась скорость движения.

Т.е. у нас получится некая «светодиодная картинка» примерно такого вида:



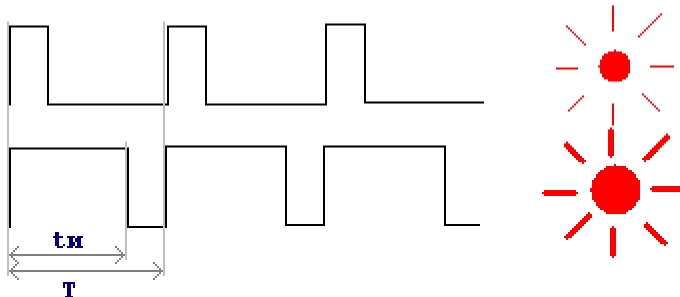
Как видно, яркость светодиодов постепенно убывает. Эту картинку мы и будем вращать так, чтобы самый яркий светодиод все время менял свое положение, а остальные следовали за ним шлейфом.

Как регулировать яркость светодиода

Два слова о том, как регулируется яркость светодиода. Есть два способа:

1. регулировкой силы тока, протекающей через светодиод;
2. регулировкой скважности импульсов постоянного тока.

Какой из этих способов использовать, разработчик решает, исходя из элементной базы и требований к устройству. В нашем случае будет удобнее использовать второй способ, т.к. у нас распоряжении только цифровое управление. В качестве источников постоянного тока будут выступать токоограничивающие резисторы в цепях светодиодов. Мы будем управлять яркостью светодиодов, подавая на них импульсы достаточно высокой частоты, чтобы глаз не замечал миганий. Обычно частоту в таких случаях выбирают в пределах 50-200 Гц. Т.е. каждый период длится от 5 до 20 мс. Сама яркость регулируется скважностью импульсов, т.е. отношением длительности периода ко времени импульса (активного состояния «1»). Очевидно, что чем скважность меньше (т.е. импульс длиннее), тем светодиод будет гореть ярче:



tи - время импульса; T - период.

Каково должно быть разрешение активного состояния, т.е. с каким минимальным шагом может изменяться длительность активного состояния? Понятно, что чем разрешение выше (шаг меньше), тем лучше. И опять же: требования к разрешающей способности управляющего светодиодом сигнала определяются назначением устройства. Если это LED-телевизор, то разрешение как минимум должно быть 8-бит. Если же это гирлянда, или какое-нибудь оформление вывески магазина или витрины (как раз то, для чего можно будет применить нашу программу), то тут хватит 4-5 бит.

Проектирование

В данном примере мы рассмотрим самый простой вариант применения ОСРВ: все задачи будут иметь одинаковый приоритет, обмен данными между задачами будет производиться через глобальные переменные. Т.е. сейчас будем использовать ОСРВ только для обеспечения параллельного выполнения нескольких подпрограмм (задач).

Задачи

Для начала определимся, какие задачи будет выполнять наш контроллер:

1. формирование 4-х (8-ми для 16F887) ШИМ[2)] каналов для управления каждым светодиодом; частотой в пределах 50-200 Гц и разрешением 5 бит;
2. «вращение» - подпрограмма, которая будет вращать «светодиодную картинку» с заданной скоростью;
3. опрос кнопки;
4. опрос позиции переменного резистора; другими словами: измерение напряжения на входе АЦП.

Учитывая то, что задача формирования ШИМ-сигналов требовательна к скорости, ее есть смысл поместить в обработчик прерывания. Остальные задачи будут в виде простых Си-функций с некоторыми особенностями, о которых поговорим позже.

Данные

Теперь подумаем о том, какими данными будет оперировать программа и какими данными будут обмениваться задачи.

1. очевидно, что нам нужны какие-то константы, определяющие длительности импульсов для разных яркостей (дело в том, что зависимость яркости светодиода от длительности импульсов нелинейная). Т.к. у нас 4 светодиода (или 8) то нам нужен массив из 4-х (8-ми) констант;
2. т.к. «светодиодная картинка» вращается, то нам нужна глобальная переменная, показывающая стадию вращения на данный момент;
3. учитывая, что вращение может происходить то в одну, то в другую сторону, нам нужна переменная, показывающая направление вращения;
4. наконец, нужна переменная, определяющая скорость вращения.

Итак, блок определения глобальных переменных (и констант) в нашей программе будет выглядеть так:

```
const char Brightness[] = {31, 11, 4, 0}; // Таблица яркостей
                                          // (длительностей импульсов)

char m_cPosition;       // Текущая фаза вращения (позиция яркости в
                        // таблице Brightness для первого светодиода)
char m_cDirection;      // Направление вращения (-1 или +1)
char m_cSpeed;          // Скорость вращения
```

Примечание:

- префикс «m_» в именах переменных говорит о том, что эти переменные глобальные;
- префикс «c» говорит о том, что переменные имеют тип **char**.

Реализация

Теперь, когда мы определились с количеством и назначением задач, приступим к их программной реализации. Для начала определимся с системными параметрами:

1. воспользуемся внутренним тактовым генератором контроллера (8 МГц);
2. все обрабатываемые светодиоды подключены к выводам одного порта последовательно. Эта особенность позволит работать с ними в цикле;
3. кнопка имеет активное состояние «0»;
4. напряжение на входе АЦП изменяется в пределах 0..VDD, т.е. для выбора скорости мы можем использовать весь диапазон значений 0..255.

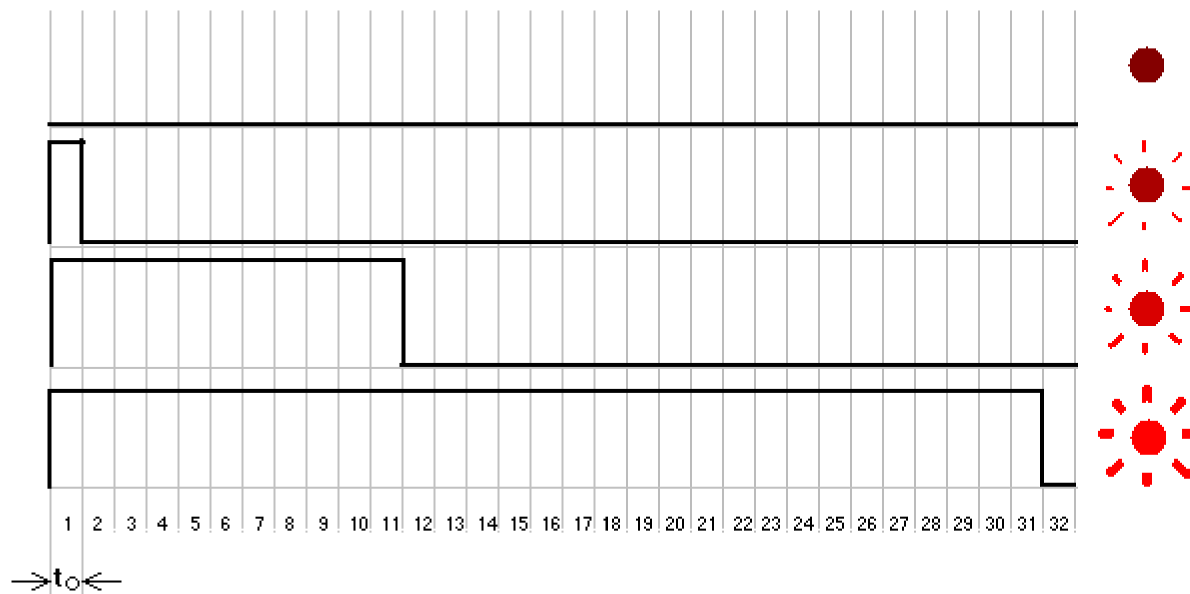
Примечания.

1. *Т.к. это наш первый проект с использованием ОСРВ, то мы постараемся обойтись минимальным набором системных сервисов;*
2. *Для лучшей читабельности и упрощения алгоритма работы программы некоторые фрагменты будут написаны несколько развернуто, в ущерб оптимальности.*

ШИМ

Итак, нам нужно сформировать 4 (или 8) ШИМ-каналов с частотой 50-200 Гц и разрешением 5 бит. Для этой цели воспользуемся прерыванием по TMR0.

Предлагаю это сделать так:



На рисунке видно, что один период ШИМ будет проходить за 32 периода TMR0 (на картинке обозначен как t0). Т.е. мы должны завести внутреннюю переменную-счетчик, которая будет увеличиваться при каждом такте ШИМ (каждом прерывании от TMR0) и сравниваться со значениями из таблицы **Brightness**, которую мы описали выше. Пока переменная меньше соответствующего каналу значения таблицы, светодиод горит.

Учитывая тактовую частоту, удобно выбирать значение одного шага ШИМ кратное одному периоду TMR0, т.е. 128 мкс. Но нужно помнить, что обработка 4-х (или 8-ми) каналов ШИМ требует времени, поэтому мы для начала возьмем период побольше, с запасом. Установим прескейлер для таймера 0 равным 4 и получим длительность шага ШИМ ~0.5мс (512 мкс). Т.е. период ШИМ будет равен 32*0.5мс = 16мс, т.е. частота = 62 Гц.

Не будем забывать также, что у нас есть переменные **m_cPosition**, показывающая текущую фазу вращения (позицию значения яркости из массива **Brightness** для первого светодиода), и m_cDirection, показывающя направление вращения. Их нужно будет учесть при написании обработчика.

```
void interrupt isr (void)
{
    // Нам понадобятся локальные переменные для обработки ШИМ

    static char cCounter;   // Счетчик шагов ШИМ.
           char cPosition;  // Переменная для обеспечения вращения
           char cLedsMask;  // Маска текущего светодиода
           char i;          // Переменная для организации цикла по
                            // всем ШИМ-каналам

    if (T0IF && T0IE)
    {
        T0IF = 0;
        cCounter++;

        cPosition = m_cPosition;    // Позиция яркости для первого
                                    // светодиода

        i = 4;                      // Цикл по всем светодиодам
        cLedsMask = 0x01;

        do
        {
            //-----------------------------------------------------
            // Проверяем, не пора ли гасить светодиод. Для этого
            // сравниваем текущий шаг ШИМ со значением яркости для
            // текущего светодиода
```

```
            //-----------------------------------------------------

            if (cCounter > Brightness[cPosition & 3])
                   PORTLEDS &= ~cLedsMask;
            else
                   PORTLEDS |=  cLedsMask;

            cLedsMask <<= 1;          // Берем следующий светодиод
            cPosition += m_cDirection;// В зависимости от направления
                                      // вращения берем позицию яркости
                                      // для следующего светодиода
        } while (--i);

        cCounter &= 0x1F;             // ШИМ 5-разрядный, старшие
                                      // разряды обнуляются
    }   // T0IF
}
```

Примечание: *учитывая специфику подпрограммы-прерывания, компилятор все переменные, объявленные в теле прерывания по умолчанию делает **static**. Тем не менее, сохраняя смысл эти переменных, мы нарочно поставили квалификатор **static** только перед **cCounter**, давая самим себе понять, что нам важно сохранение значения этой переменной после выхода из прерывания. Остальные переменные с точки зрения алгоритма - временные. Такая детализация может оказаться полезной, если код обработки ШИМ впоследствии будет вынесен из прерывания в обычную функцию.*

Чтение данных с АЦП: выбор скорости вращения

Эту задачу мы оформим в виде задачи операционной системы. Как уже говорилось выше, задачи в ОС - это обычные Си-функции с некоторыми особенностями, а именно:

1. тело функции должно содержать бесконечный цикл;
2. из этих функций нельзя выходить по **return**, можно только используя специальные сервисы ОС, переключающие контекст. Внутри каждой задачи обязательно должен быть вызов хотя бы одного такого сервиса;
3. функции не вызываются напрямую в теле программы, их вызывает планировщик ОС[3].

В нашей программе есть переменная **m_cSpeed**, которая определяет скорость вращения картинки. Эту переменную мы и будем формировать в данной задаче. Очевидно, что частота обновления этой переменной должна быть пропорциональна скорости изменения напряжения на входе АЦП. Медленнее делать нельзя, т.к. пользователю нужно сразу видеть, как меняется скорость при повороте потенциометра. Быстрее - нет смысла, т.к. на глаз разница не будет заметна, а система окажется перегружена слишком частыми запусками этой задачи. Поэтому мы выберем интервал обновления, равный 100 мс. Т.е. сделаем так, чтобы задача запускалась раз в 100 мс, измеряла напряжение на входе АЦП и формировала новое значение переменной **m_cSpeed**. Все остальное время задача будет находиться в ожидании и не будет мешать выполняться остальным задачам.

```
void Task_SetSpeed (void)
{
    for (;;)
    {
        CHS0 = 0;          // Выбираем нулевой канал АЦП. В принципе,
        CHS1 = 0;          // не очень нужный код, но если впоследствии
        CHS2 = 0;          // в программу будет добавлена обработка АЦП
        CHS3 = 0;          // по другим каналам, то этот код окажется
                           // очень полезным

        GODONE = 1;            // Запуск измерения

        OS_Cond_Wait(!GODONE); // Ожидание завершения измерения.
                               // Во время ожидания могут выполняться
                               // другие задачи

        m_cSpeed = ADRESH;     // Установка нового значения скорости

        OS_Delay(100 ms);      // Говорим ОС, что следующий раз эту
                               // задачунадо запустить через 100 мс.
    }
}
```

Примечание: *ms - макрос, определенный как «/ 1». Подробнее о нем - в параграфе «Таймер».*

Здесь мы использовали два системных сервиса: **OS_Cond_Wait [http://www.pic24.ru/doku.php/osa/ref/allservices/OS_Cond_Wait]** и **OS_Delay [http://www.pic24.ru/doku.php/osa/ref/allservices/OS_Delay]**. Остановимся на них и рассмотрим, как они работают.

Сервис **OS_Cond_Wait** - это сервис, который переводит задачу в **режим ожидания** до тех пор, пока не будет выполнено условие в скобках. Что такое режим ожидания? Как уже было сказано ОС имеет встроенный механизм обеспечения многозадачности, что позволяет всем задачам выполняться параллельно (т.е., конечно, они выполняются последовательно, но быстро сменяют друг друга, что создает эффект параллельности). Если какая-то задача ждет выполнения какого-то условия, то ее нет смысла запускать, пока условие не выполнено. Это позволит системе больше внимания уделять остальным задачам. Сервис **OS_Cond_Wait** сообщает системе, что эту задачу не нужно запускать, пока условие **!GODONE** не будет выполнено. Одновременно с этим этот сервис осуществляет выход из функции-задачи (с запоминанием точки выхода), после чего управление передается планировщику ОС. Когда планировщик обнаружит, что бит **GODONE** сброшен, он переводит задачу в **режим готовности** - задача становится в очередь выполняемых задач и получит управление, когда до нее дойдет очередь, причем она продолжит свое выполнение с того места, откуда был совершен выход, т.е. сразу же за сервисом **OS_Cond_Wait**.

Сервис **OS_Delay** - это сервис, который переводит задачу в режим ожидания на указанное время. Управление передается планировщику. В течение указанного времени задача не будет получать управление, позволяя системе больше времени уделять остальным задачам. Время задается в так называемых **системных тиках** - периодах вызова системного сервиса **OS_Timer**. Подробнее см. параграф «Таймер».

Вращение с заданной скоростью

Эта задача должна выдерживать паузу в соответствии с установленным значением переменной **m_cSpeed**, после чего увеличивает значение переменной **m_cPosition** (позиция элемента яркости в массиве Brightness для первого светодиода). Напрашивается красивое использование сервиса OS_Delay:

```
    OS_Delay(m_cSpeed);
```

Однако здесь есть подвох. Если на момент вызова сервиса переменная m_cSpeed имеет большое значение, то пока идет длинная задержка, программа не будет реагировать на изменение скорости. А это будет немного раздражать. Поэтому правильнее организовать задачу так:

```
void Task_Rolling (void)
{
    static char cDelay;

    for (;;)
    {
        cDelay = 0;
        while (cDelay++ < m_cSpeed)      // Ждем нужное время
            OS_Delay(1);

        m_cPosition ++;                  // Изменяем позицию яркости
    }
}
```

Такой подход обеспечит более быструю реакцию на изменение переменной **m_cSpeed**. Однако и он не лишен недостатков: при больших значениях **m_cSpeed** задача будет часто получать управление впустую (каждый системный тик). Но с точки зрения интерфейса пользователя такой подход более удачный.

Обратим внимание на объявление переменной **cDelay**. Эта переменная объявлена как **static**, т.к. нам важно сохранение ее значения после выхода из функции. Если мы не напишем **static**, то после выхода из функции эта переменная может затереться локальными переменными других функций.

Опрос кнопки: выбор направления вращения

Еще одна простая задача, функцией которой является изменение направления вращение при нажатии кнопки. Т.к. кнопка имеет активный уровень «0», то сперва мы ждем установки входа на ножке pin_BUTTON в «0». После того, как дождались нам нужно подавить дребезг, чтобы исключить ложные срабатывания. Для этого мы будем выжидать 40 мс и делать повторную проверку. Если состояние входа так и останется в «0», значит, кнопка нажата и можно продолжать работать. Если нет, то делаем повторное ожидание. После обработки кнопки мы таким же способом ждем отпускание кнопки.

```
void Task_Button (void)
{
    for (;;)
```

```
    {
        //------------------------------------------------------------
        //  Ожидаем нажатие кнопки
        //------------------------------------------------------------
        do
        {
            OS_Cond_Wait(!pin_BUTTON);
            OS_Delay(40 ms);      // Для подавления дребезга контактов
                                  // ждем 40 мс и делаем повторную
        } while (pin_BUTTON);     // проверку

        //------------------------------------------------------------
        //  Меняем направление вращения на противоположное
        //------------------------------------------------------------
        //
        m_cDirection = -m_cDirection;

        //------------------------------------------------------------
        //  Ожидаем отпускание кнопки
        //------------------------------------------------------------
        do
        {
            OS_Cond_Wait(!pin_BUTTON);
            OS_Delay(40 ms);      // Для подавления дребезга контактов
                                  // ждем 40 мс и делаем повторную
        } while (!pin_BUTTON);    // проверку

    }
}
```

Обратим внимание на особенность этой задачи. Если кнопка не будет нажата, то задача никогда не получит управление и не будет загружать процессор, в отличие от двух других задач, которые периодически управление будет получать.

Функция main()

Итак, мы написали все подпрограммы для работы нашего алгоритма. Пока что все они - обычные функции в стиле языка Си. Операционная система еще не знает, какие из этих функций следует рассматривать как задачи ОС, а какие - сами по себе. Для того чтобы она знала, какими функциями ей предстоит оперировать (т.е. какие функции должны выполняться параллельно), ей нужно сообщить их названия. Для этого есть сервис **OS_Task_Create [http://www.pic24.ru/doku.php/osa/ref/allservices/OS_Task_Create]**, которому в параметрах передается имя функции-задачи и ее приоритет. Т.к. изначально мы условились, что все задачи будут равноприоритетными, то всем задачам присвоим высший (нулевой) приоритет.

У нас 4 задачи, которые должны будут выполняться параллельно. Но учитывая, что код одной из задач расположен в прерывании, т.е. она и так уже сама по себе будет выполняться в фоновом режиме, то сервисом **OS_Task_Create** нужно создать только три задачи.

Теперь отсталость только добавить инициализацию контроллера и системы. И все: можно запускать планировщик в работу.

```
void main (void)
{
    char prs;

    Init();                 // Инициализация периферии

    OS_Init();              // Инициализация операционной системы

    //------------------------------------------------------------
    // Создаем функции-задачи, которые будут работать
    // параллельно. В нашем случае все имеют одинаковый
    // высший приоритет (0)
    //------------------------------------------------------------

    OS_Task_Create(0, Task_Rolling);
    OS_Task_Create(0, Task_SetSpeed);
    OS_Task_Create(0, Task_Button);

                            // Начальные значения:
    m_cPosition  = 0;       // для фазы вращения
    m_cDirection = 1;       // для направления вращения

    OS_EI();                // Разрешаем прерывания
```

```
    OS_Run();              // Запускаем планировщик в работу.

}
```

Помимо OS_Task_Create, мы тут встречаем еще три сервиса ОС: **OS_Init [http://www.pic24.ru/doku.php/osa/ref/allservices/OS_Init]**, **OS_EI [http://www.pic24.ru/doku.php/osa/ref/allservices/OS_EI]** и **OS_Run [http://www.pic24.ru/doku.php/osa/ref/allservices/OS_Run]**.

OS_Init - начальная инициализация операционной системы. Здесь обнуляются все системные переменные, подготавливаются дескрипторы задач. Этот сервис должен вызываться первым из всех системных сервисов.

OS_EI - разрешение прерываний. Можно было бы просто воспользоваться строчкой «GIE = 1;», но тогда, если мы когда-нибудь захотим перевести программу на PIC 18-ой серии, то нам придется вспомнить, что там два уровня прерываний и что, возможно, нужно добавить еще и разрешение GIEL. Этот же сервис все делает автоматически.

OS_Run - этот сервис должен вызываться в самом конце функции main(). Т.к. этот сервис является макросом, содержащем внутри себя бесконечный цикл, то все, что будет написано после него никогда не получит управление. Внутри этого сервиса происходит перебор все задач ОС, проверка их готовности или условий выхода из режима ожидания, сравнение приоритетов, выбор самой приоритетной из готовых задач и передача ей управления.

Функция Init()

Весь текст я здесь приводить не буду (его можно посмотреть в исходных текстах, прилагаемых к статье), т.к. из-за того, что эта функция предусматривает работу на 4-х разных контроллерах (16F886, 16F887, 16F690 и, «по совету друзей», 16F88), то код ее довольно громоздкий из-за наличия условных директив #ifdef…#endif.

Скажу только, что в этой функции производится инициализация:

- потов ввода/вывода;
- АЦП;
- таймеров;
- прерываний.

Таймер

Последнее, что нам осталось сделать, - это добавить обработку системного таймера. Т.к. у нас в программе вызываются системные сервисы, использующие системный таймер, то нам нужно в периодическое место в программе добавить вызов сервиса **OS_Timer [http://www.pic24.ru/doku.php/osa/ref/allservices/OS_Timer]**. Этот сервис будет следить за всеми задержками, выполняющимися через **OS_Delay**. Теперь нам нужно организовать периодическое место в программе, т.е. то место, куда программа будет попадать с заданным интервалом. У нас, в принципе, такое место уже есть - это прерывание по TMR0. Но мы для сохранения наглядности не будем его трогать, а организуем прерывание по таймеру TMR2. Кроме наглядности, здесь есть еще одно преимущество: мы можем отдельно изменять скорость работы ШИМ и интервал вызовов **OS_Timer**, если понадобится.

В подпрограмму обработки прерываний добавляем такой код:

```
    if (TMR2IF)
    {
        TMR2IF = 0;
        OS_Timer();
    }
```

Теперь все значения, передаваемые сервису **OS_Delay** в задачах, будут измеряться именно в интервалах вызова сервиса **OS_Timer** - системных тиках.

Я не случайно упомянул возможность появления необходимости изменения интервала вызова системного таймера. Причины могут быть разные, ну, например, мы используем контроллер, имеющий на борту всего один таймер, и нам, хочешь - не хочешь, придется использовать один таймер и для генерации ШИМ-сигналов и для системного таймера. А тут может понадобиться некоторая настройка периода ШИМ. Получается, что каждый раз, меняя период прерывания (и, следовательно, - период вызова **OS_Timer**), нужно будет пересчитывать все константы в параметрах вызовов сервиса **OS_Delay**? Вот тут нам на помощь придет константа **ms**, о которой речь шла выше. В нашем случае эта константа определена так:

```
#define ms    / 1
```

, т.к. период таймера 2 у нас равен 1 мс. Во всех вызовах сервиса **OS_Delay** мы пользуемся этой константой:

```
    OS_Delay(100 ms);   // Компилятор сделает подстановку "100 / 1"
    ...
    OS_Delay(20 ms);    // Компилятор сделает подстановку "20 / 1"
```

И если нам придется изменить период вызова **OS_Timer** (повторюсь: не важно, по какой причине), то нам не придется пересчитывать все константы в программе, а достаточно будет только изменить константу ms. Например, мы увеличили период втрое, тогда надо будет переопределить константу так:

```
#define ms    / 3
```

или мы уменьшили период вдвое:

```
#define ms    * 2
```

Конфигурация OSA

OSA - очень гибкая в настройке операционная система. Благодаря своей гибкости, она позволяет использовать ресурсы контроллера с максимальной эффективностью для конкретного проекта. Она позволяет программисту выбирать:

- типы используемых системных переменных (таймеров, семафоров, сообщений);
- банки RAM для хранения каждого типа системных переменных;
- какие сервисы и как будут использоваться системой;
- будут ли сервисы использованы в прерываниях;
- и т.д.

Все эти настройки программист указывает в файле OSAcfg.h, который должен быть расположен в папке проекта. Для каждого проекта - свой файл. Подробнее обо всех настройках можно почитать в описании OSA в параграфе Конфигурация OSAcfg.H [http://wiki.pic24.ru/doku.php/osa/ref/appendix/configuration]. Создавать и сопровождать этот файл вручную довольно сложно из-за большого количества различных определений, и это могут делать только имеющие опыт работы с OSA программисты. Удобнее для конфигурирования воспользоваться утилитой из комплекта поставки OSAcfg_Tool.

Запустив эту утилиту, мы увидим на экране все настройки, которые можно сделать при конфигурировании проекта. Итак, займемся созданием файла конфигурации для нашего проекта.

1. Выбираем папку, где располагается наш проект

Для этого в самом верху окна справа нажимаем кнопку **Browse**. Там выбираем путь к файлу OSAcfg.h - путь к нашему проекту («C:\TEST\PICKIT2\LIGHTS»). Нажимаем OK. Если файл еще не создан, то программа спросит у Вас, действительно ли Вы хотите создать этот файл. Смело отвечаем «Yes» и идем дальше. (Если файл уже существует, то он просто загрузится и установит в рабочем окне все галочки и параметры в соответствие со своей конфигурацией. Пока же мы считаем, что файл не создан.)

2. Выбираем имя проекта

В поле **Name** можно ввести имя проекта. Этот пункт необязателен, а имя вводится исключительно для наглядности, чтобы не путаться потом, какой файл от какого проекта. Мы введем в эту строку «Бегущие огни».

3. Выбираем платформу

Также необязательный пункт. Служит только для того, чтобы пользователь при конфигурировании файла в реальном времени наблюдал предполагаемый расход оперативной памяти операционной системой. Для успокоения выберем платформу: 14-бит (PIC12, PIC16)(ht-picc). Теперь при изменении настроек мы автоматически в рамке **RAM statistic** будем видеть, сколько байтов в каком банке памяти израсходовано.

4. Конфигурируем наш проект

Для этого есть 4 области в рабочем окне:

- **System** - системные настройки
- **Data services** - настройки сервисов обмена данными (счетные семафоры, сообщения, очереди сообщений)
- **Timers** - все настройки, относящиеся к таймерам
- **Binary semaphores** - настройка двоичных семафоров

Нам понадобятся только **System** и **Timers**, т.к. сервисы обмена данными (включая бинарные семафоры) мы не используем.

Учитывая, что мы решили не использовать приоритеты (т.е. все задачи сделать равноприоритетными), можно установить галочку напротив пункта **Disable priority**. Это сократит размер кода ядра операционной системы и ускорит работу планировщика.

Далее, нам обязательно нужно выбрать количество задач ОС, которые будут работать одновременно. В нашем случае - 3 (по количеству задач, создаваемых сервисом OS_Task_Create; как уже было сказано раньше, 4-я задача у нас не является задачей ОС и располагается в обработчике прерывания).

Последнее, что нам понадобится, - это включить таймер задач, т.е. поставить галочку напротив пункта **Task timers**.

5. Сохраняем и выходим

Жмем на кнопку **Save**, чтобы сохранить отредактированный файл конфигурации, и выходим из программы нажатием на кнопку **Exit**. Теперь, заглянув в созданный нами файл, мы увидим следующее:

```
//////////////////////////////////////////////////////////
//
// This file was generated by OSAcfg_Tool utility.
// Do not modify it to prevent data loss on next editing.
//
// PROJECT NAME: Бегущие огни
// PLATFORM: HT-PICC 14-bit
//
//////////////////////////////////////////////////////////


#ifndef _OSACFG_H
#define _OSACFG_H

#define OS_TASKS               3
#define OS_DISABLE_PRIORITY
#define OS_ENABLE_TTIMERS

#endif
```

Прошивка контроллера

Теперь, когда текст нашей программы готов, нам нужно выполнить компиляцию и прошить полученный код в контроллер.

Сборка проекта

Для работы с проектом нам нужно иметь установленную интегрированную среду MPLAB IDE [http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en019469&part=SW007002], установленный компилятор HI-TECH PICC STD [http://www.htsoft.com/microchip/products/compilers/picccompiler.php] (PRO-версия не подойдет).

Скачиваем, если еще не скачали, **файлы операционной системы OSA**, распаковываем этот архив на диск C: (должна получиться папка C:\OSA).

Распаковываем файл **lights.rar** в папку C:\TEST\PICKIT2. При этом внутри создастся папка LIGHTS. В MPLAB IDE открываем проект, в названии которого присутствует номер контроллера, который Вы собираетесь использовать: 886, 887, 690 или 88. Например, для демо-платы на базе 16F887 нам нужно открыть файл pk2_lights_887.mcp.

Примечание. *При распаковке в другую папку, отличную от C:\TEST\PICKIT2\LIGHTS, нужно будет через меню Project\Build options…\Project в закладке Directories в списке include-путей заменить путь к файлам проекта на тот, куда Вы распаковали файлы из архива.*

Выполняем сборку нажатием **Ctrl+F10**.

Прошивка

Здесь все просто:

1. подключаем программатор;
2. в меню «Programmer\Select» programmer выбираем PicKit2;
3. В настройках «Programmer→Settings» выбираем «3-State on «Release from Reset»
4. запускаем программирование «Programmer\Program»;
5. освобождаем вывод MCLR «Programmer\Release from reset».

Вот и все!

Заключение

Итак, в данной статье была приведена демонстрация применения операционной системы реального времени OSA для написания программы «Бегущие огни». Меньше чем за час мы спроектировали и написали программу, которая выполняет заданную задачу. При этом текст программы получился наглядным и легко читаемым.

Что хорошего мы извлекли из использования OSA:

- возможность организовать отдельные функциональные узлы программы в виде независимых и очень простых и наглядных задач;
- избавились от необходимости заводить несколько переменных для отсчета временных интервалов;
- мы не думали об обеспечении одновременного выполнения задач, т.к. все это на себя взяла ОС; кроме того, она это делает довольно эффективно, т.к. не запускает задачи, которые еще не готовы к выполнению.

Получившаяся программа может найти практическое применение, например, при оформлении вывески магазина. Для этого достаточно будет всего лишь применить кое-какие схемотехнические решения: в каждую цепь поставить не по одному светодиоду, а по несколько. Чередуя их четверками (или восьмерками, если программа для 16F887), т.е. сначала диод из первой цепи, затем из второй, потом из третей, дальше из четвертой, за ним - снова из первой, потом из второй и т.д., можно сделать длинную «гирлянду», которой оформить вывеску, витрину, объявление.

Введя небольшие изменения в программу, можно добавить свои эффекты над светодиодной картинкой, например, раскручивать ее с ускорением, или плавно гасить, а потом снова зажигать - в общем, как фантазия будет работать. Дерзайте!

Спасибо за внимание.

На сайте www.pickit2.ru [http://www.pickit2.ru] статья размещена с разрешения владельца сайта.

Виктор Тимофеев, март 2009
osa@pic24.ru [mailto:osa@pic24.ru]

---

[1] ОСРВ - операционная система реального времени

[2] ШИМ - Широтно-Импульсная Модуляция

[3] Планировщик - специальная подпрограмма ОСРВ, которая следит за готовностью задач к выполнению, выбирает наиболее приоритетную из них и передает ей управление