**Начало работы с uC/OS-II**

В этой главе приведены четыре примера использования uC/OS-II. Я решил включить эту главу в начале книги, чтобы вы могли как можно скорее начать использовать uC/OS-II. На самом деле, я предполагаю, что вы мало знаете о uС/ОС-II и многозадачности; концепции вводятся по мере необходимости.

Образец кода был скомпилирован с помощью компилятора Borland C/C + + V4.51 и были выбраны варианты генерации кода для процессора Intel/AMD 80186 (модель большой памяти). Код был фактически запущен и протестирован на 300MHz ПК Intel Pentium II, работающем в окне DOS под управлением Windows 2000. Для всех целей, Pentium можно рассматривать как сверхбыстрый процессор 80186. Borland C/C + + V4.52 (называемый Borland Turbo C++ 4.5) доступен от JK Microsystems (www.jkmicro.com и конкретно: http://stores.jkmicro.com/borland-c-development-kit/.

Компьютер был выбран в качестве целевой системы по ряду причин. Прежде всего, когда я писал первоначальную версию книги, тестировать код на ПК было намного проще, чем на любой другой встраиваемой среде (то есть evaluation board, эмуляторе и т. д.): не было EPROM для записи и не было загрузок на эмуляторы EPROM, эмуляторы CPU и т. д. Вы просто компилируете, связываете и запускаете. Во-вторых, объектный код 80186 (реальный режим, большая модель), созданный с помощью компилятора Borland C/C + +, совместим со всеми производными процессорами 80x86 от Intel, AMD и др. Сегодня вы можете получить оценочную плату за просто так, и 80x86 может не быть лучшей встроенной платформой, но, тем не менее, будет освещена в этом документе.

**Установка uC/OS-II**

Вы можете загрузить пример кода с веб-сайта Micrium, и вы должны обратиться к Приложению F за инструкциями о том, как установить на Ваш компьютер исходный код, а также исполняемые файлы примеров. Предполагается, что программное обеспечение будет установлено на компьютере под управлением Windows 95, 98, Me, NT, 2000 или XP.

**Пример 1: Базовая многозадачность**

Пример #1 демонстрирует основные возможности многозадачности uC/OS-II. Десять задач отображают число от 0 до 9 в случайных местах на экране. Каждая задача отображает только один номер. Другими словами, одна задача отображает 0 в случайных местах, другая задача отображает 1 и так далее.

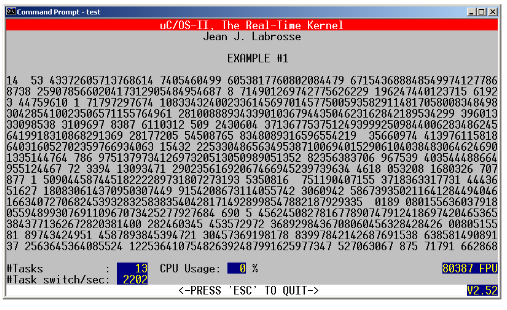
Код для примера # 1 находится в каталоге\SOFTWARE\uCOS-II\EX1\_x86L\BC45 установочного диска (по умолчанию - C:). Можно открыть окно DOS (называемое командной строкой в Windows 2000) и ввести:

CD \SOFTWARE\uCOS-II\Ex1\_x86L\BC45\TEST

Команда CD позволяет изменить каталог и в этом случае перейти к каталогу TEST примера № 1. Каталог TEST содержит четыре файла: MAKETEST.BAT, TEST.EXE, TEST.LNK и TEST.MAK. Чтобы выполнить пример # 1, просто введите TEST в командной строке. В окне DOS будет запущена программа TEST.EXE.

Примерно через одну секунду окно DOS будет заполнено цифрами от 0 до 9, как показано на рис. 1.1.

**Рисунок - Рисунок 1.1 Пример # 1, выполняемый в окне DOS**



Пример # 1 в основном состоит из 13 задач, как показано в левой нижней части рис. 1.1. uC/OS-II создает две внутренние задачи: задачу простоя и задачу, определяющую загрузку ЦП. Код в примере № 1 создает остальные 11 задач.

Исходный код, например # 1, находится в TEST.C в каталоге SOURCE. Туда можно попасть из каталога TEST, набрав:

CD ..\SOURCE

Части TEST.C приведены в Листинге 1.1. Фактический код можно проверить с помощью избранного редактора кода.

**Примечание**: Чтобы описать объявления и цифры, я помещу ссылку на поле. Ссылка соответствует элементу списка или фигуры, на которую я хочу обратить ваше внимание. Например, L1.1 (1) означает: «Пожалуйста, обратитесь к листингу 1.1 и найдите товар (1)». Это обозначение также относится к цифрам и, таким образом, F3.1 (2) будет означать: «Пожалуйста, посмотрите на рис. 3.1 и изучите пункт (2)».

**Листинг - Листинг 1.1 Пример № 1, TEST.C**

#include  "includes.h"                                           (1)

#define       TASK\_STK\_SIZE            512                       (2)

#define       N\_TASKS                   10

OS\_STK        TaskStk[N\_TASKS][TASK\_STK\_SIZE];                   (3)

OS\_STK        TaskStartStk[TASK\_STK\_SIZE];                       (4)

**char**          TaskData[N\_TASKS];                                 (5)

OS\_EVENT     \*RandomSem;                                         (6)

(1) Во-первых, вы заметите, что существует только одна инструкция #include. Это потому, что я люблю помещать все мои заголовочные файлы в «главный» заголовочный файл под названием INCLUDES.H. Каждый исходный файл всегда ссылается на этот единственный include-файл и, таким образом, мне никогда не нужно беспокоиться о том, какие заголовки мне нужны; все они включаются через INCLUDES.H. Вы можете использовать редактор кода для просмотра содержимого INCLUDES.H, которое также находится в каталоге SOURCE.

Я вернусь к листингу 1.1 позже, когда это потребуется. Как и большинству программ C, нам нужен main(), как показано в листинге 1.2.

**Листинг - листинг 1.2 Пример № 1, TEST.C, main ()**

**void**  main (**void**)

{

    PC\_DispClrScr(DISP\_FGND\_WHITE + DISP\_BGND\_BLACK);                        (1)

    OSInit();                                                                (2)

    PC\_DOSSaveReturn();                                                      (3)

    PC\_VectSet(uCOS, OSCtxSw);                                               (4)

    RandomSem   = OSSemCreate(1);                                            (5)

    OSTaskCreate(TaskStart, (**void** \*)0, &TaskStartStk[TASK\_STK\_SIZE - 1], 0); (6)

    OSStart();                                                               (7)

}

(1) µC/OS-II - многозадачное ядро и позволяет Вам иметь до 255 прикладных задач. µC/OS-II решает, когда переключиться от одной задачи к другой на основе информации, которую вы обеспечили для µC/OS-II. Один из пунктов, которые Вы должны сказать µC/OS-II - является приоритет Ваших задач. Изменение между задачами называют переключением контекста.

main()начинается, очищая экран, чтобы гарантировать, что никакие знаки не перенесены от предыдущей сессии DOS. Функция PC\_DispClrScr()найходится в файле под названием  PC.C (см. Главу 18, PC Services для деталей).  PC.C содержит функции, которые предоставляют услуги, если Вы запустили программу в окружающей среде DOS (или окно под Windows 95, 98, Me, NT, 2000 или операционной системы XP). Префикс **PC\_** позволяет Вам легко определять название файла, куда ведет функция; в этом случае, PC.C. Вы должны отметить, что я определил белые письма о черном фоне. Так как экран будет очищен, я просто, возможно, определил черный фон и не определил передний план. Если бы я сделал это, и Вы решили возвратиться к командной строке DOS, то Вы ничего не видели бы на экране! Поэтому всегда лучше определить видимый передний план.

(2) Требование µC/OS-II - то, что Вы вызовите OSInit() прежде чем Вы вызовите любую из ее других сервисов. OSInit()создает две задачи: задача простоя(idle), которая выполняется, когда никакая другая задача не готова к запуску, и статистическая задача, которая вычисляет использование центрального процессора.

(3) Текущая DOS 'окружающая среда' сохраняется, вызовом PC\_DOSSaveReturn() .Это позволяет Вам вернуться к DOS, как будто Вы никогда не запускали µC/OS-II. Вы можете обратиться к Главе 18, PC Services для описания того, что делает PC\_DOSSaveReturn().

(4)  main()вызывает PC\_VectSet() (см. Главу 18, PC Services), чтобы установить обработчик переключения контекста в µC/OS-II. Переключение контекста задачи будет сделано µC/OS-II, путем выполнения 80x86 инструкции прерывания INT в векторе прерываний. Я решил использовать вектор 0x80 (т.е. 128), потому что он не используется ни DOS ни BIOS.

(5) Двойной семафор создан, чтобы охранять доступ к функции генератора случайных чисел, обеспеченной библиотекой Borland C/C ++. Семафор - объект, обеспеченный ядром, чтобы препятствовать нескольким задачам получить доступ к тому же ресурсу (в этом случае функция) одновременно. Я решил использовать семафор, потому что я не знал, будет ли случайная функция генератора повтоно использована; я предположил, что нет. Инициализируя семафор единицией, я сообщаю µC/OS-II что она должна позволять одной задаче получить доступ к случайной функции генератора в любой момент времени. Семафор должен быть 'создан', прежде чем он сможет использоваться. Это сделано вызовом OSSemCreate() и определением его начального значения. OSSemCreate() возвращает обработчки (см. Листинг 1.1 (6)) семафора, который должен использоваться, чтобы сослаться на этот конкретный семафор.

(6) Перед стартом многозадачности Вы должны создать по крайней мере одну задачу. Для этого примера я назвал эту задачу TaskStart(). Вы 'создаете' задачу, потому что Вы хотите сказать µC/OS-II управлять задачей.  OSTaskCreate() функция получает четыре аргумента. Первый аргумент - указатель на адрес задачи, в этом случае TaskStart(). Второй аргумент - указатель на данные, которые Вы хотите передать к задаче, когда она впервые начинается. В этом случае нет ничего, чтобы передать и таким образом, я передал NULL указатель, но это, могло бы быть что угодно другое. Я буду обсуждать использование этого аргумента в Примере № 4. Третий аргумент - вершина стека (TOS) задачи. У µC/OS-II, как у большинства приоритетных ядер, каждая задача требует своего собственного пространства стека. У каждой задачи в µC/OS-II может быть различный размер, но для простоты я сделал их всех одинаковыми. На 80x86 центральном процессоре, стек растет вниз и таким образом, мы должны передать самый верхний 'достоверный' адрес TOS в OSTaskCreate() . В этом случае стек называют TaskStartStk[] и выделяют во время компиляции. Стек должен быть объявлен, имея тип OS\_STK  (см. Листинг 1.1 (4)). Размер стека объявлен в Листинге 1.1 (2). Для 80x86, OS\_STK  - 16 битовых значений и таким образом, размер стека составляет 1 024 байта. Наконец, мы должны определить приоритет создаваемой задачи. Чем ниже приоритетное число, тем выше приоритет (т.е. его важность).

Как ранее упомянуто, µC/OS-II позволяет Вам создавать до 255 задач. Однако у каждой задачи должно быть уникальное приоритетное число между 0 и 254. Вы - тот, который на самом деле решает какой приоритет дать Вашей задаче на основе Ваших основных эксплуатационных характеристик. Приоритетный уровень 0 - самый высокий приоритет.

(7) Затем вызывается OSStart() , чтобы начать многозадачность и дать управление uC/OS-II. Перед вызовом функции OSStart() очень важно создать хотя бы одну задачу. Если этого не сделать, приложение, безусловно, завершится сбоем. На самом деле, если вы планируете использовать задачу статистики использования ЦП, то всегда может потребоваться создать только одну задачу.

Задание OSStart() - определяет, какая из всех созданных задач является наиболее важной (наивысший приоритет), и начинает выполнение этой задачи. В нашем случае, uС/ОС-II создал две низкоприоритетные задачи: задачу простоя(Idle) и задачу статистики.  main() создал TaskStart() с приоритетом 0. Как я уже упоминал, приоритет 0 является наивысшим приоритетом, и, таким образом, OSStart() начнет выполнение TaskStart().

Следует отметить, что OSStart() не возвращается обратно к  main(). Однако при вызове PC\_DOSReturn() многозадачность будет остановлена, а приложение вернется в DOS (но не в  main()). Во встроенной системе нет необходимости в эквивалентной функции для PC\_DOSReturn(), поскольку вы, скорее всего, не вернетесь ни к чему!

Как я уже упоминал в предыдущем разделе, OSStart() выберет TaskStart() в качестве наиболее важной задачи для запуска. TaskStart() отображается в листинге 1.3.

**Листинг 1.3 Пример № 1, TEST.C, TaskStart ()**

**void**  TaskStart (**void** \*pdata)

{

#if OS\_CRITICAL\_METHOD == 3

    OS\_CPU\_SR  cpu\_sr;

#endif

**char**       s[100];

    INT16S     key;

    pdata = pdata;                                               (1)

    TaskStartDispInit();                                         (2)

    OS\_ENTER\_CRITICAL();                                         (3)

    PC\_VectSet(0x08, OSTickISR);                                 (4)

    PC\_SetTickRate(OS\_TICKS\_PER\_SEC);                            (5)

    OS\_EXIT\_CRITICAL();                                          (6)

    OSStatInit();                                                (7)

    TaskStartCreateTasks();                                      (8)

**for** (;;) {                                                   (9)

        TaskStartDisp();                                        (10)

**if** (PC\_GetKey(&key) == TRUE) {                          (11)

**if** (key == 0x1B) {                                  (12)

                PC\_DOSReturn();                                 (13)

            }

        }

        OSCtxSwCtr = 0;                                         (14)

        OSTimeDlyHMSM(0, 0, 1, 0);                              (15)

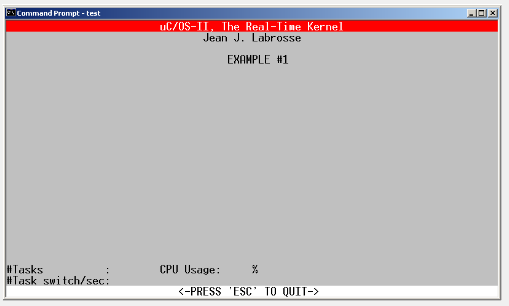
    }

}

(1) TaskStart()начинается с установки pdata указывающим на себя. Я делаю это, потому что некоторые компиляторы жалуются (ошибка или предупреждение), если pdata не ссылается никуда. Другими словами, я «подделываю» использование pdata ! pdata - это указатель, передаваемый на задачу при ее создании. Второй аргумент, переданный в OSTaskCreate (), не является другим аргументом pdata задачи (см. L1.2 (6)). Поскольку я передал указатель NULL (снова см. L1.2 (6)), я ничего не передаю TaskStart().

(2) Затем TaskStart() вызывает TaskStartDispInit()для инициализации дисплея, как показано на рис. 1.2. TaskStartDispInit()выполняет 25 последовательных вызовов PC\_DispStr() (см. Главу 18, Службы ПК) для заполнения 25 строк текста типичного окна DOS.

**Рисунок - Рисунок 1.2 Инициализация дисплея TaskStartDispInit ()**



(3) TaskStart()вызывает макрос OS\_ENTER\_CRITICAL().OS\_ENTER\_CRITICAL()является в основном определенным для процессора макросом и используется им, чтобы отключить преывания (см. Главу 13. Porting µC/OS-II).

(4) uC/OS-II, как и все ядра, требует источника времени для отслеживания задержек и тайм-аутов. В реальном режиме ПК предлагает такой источник времени, который встречается каждые 54,925 мс (18,20648 Гц) и называется tick. PC\_VectSet()позволяет заменить адрес, по которому ПК отправляется к обработчику DOS tick , на адрес, который будет использоваться uС/ОС-II. Тем не менее, uC/OS-II будет по-прежнему вызывать обработчик tick-ов DOS каждые 54.925 мс. Этот метод называется chaining  и устанавливается PC\_DOSSaveReturn() (см. главу 18, Услуги ПК).

(5) Затем мы изменяем частоту деления с 18,2 Гц до 200 Гц. Я выбрал 200 Гц, потому что это почти точно кратно 18,2 Гц (то есть в 11 раз быстрее). Я так и не понял, почему IBM выбрала 18,2 Гц вместо 20 Гц в качестве частоты tick  на оригинальном ПК. Вместо того, чтобы устанавливать таймер 82C54 для деления входной частоты таймера на 59 659, чтобы получить хорошую частоту 20 Гц, кажется, что они оставили 16-битное переполнение таймера каждые 65 536 импульсов! Изменение скорости tick-ов обрабатывается другим сервисом ПК под названием  PC\_SetTickRate() и передается требуемая скорость tick-ов (OS\_TICKS\_PER\_SEC  устанавливается равной 200 в OS\_CPU.H).

(6) Затем вызывается макрос OS\_EXIT\_CRITICAL(). OS\_EXIT\_CRITICAL() в основном является специфичным для процессора макросом и используется для повторного включения прерываний (см. главу 13, Porting uC/OS-II). OS\_ENTER\_CRITICAL()и OS\_EXIT\_CRITICAL() должны использоваться попарно.

(7) OSStatInit()вызывается для определения скорости процессора (см. Главу 3, Структура ядра). Это позволяет uC/OS-II знать, какой процент ЦП фактически используется всеми задачами.

(8) Затем TaskStart()вызывает TaskStartCreateTasks(), чтобы позволить uC/OS-II управлять большим количеством задач. В частности, мы добавим колчиество N\_TASKS  идентичных задач (см. Листинг 1.1 (2)). TaskStartCreateTasks()отображается в Листинге 1.4.

**Список 1.4 Пример № 1, TEST.C, TaskStartCreateTasks ()**

**static**  **void**  TaskStartCreateTasks (**void**)

{

    INT8U  i;

**for** (i = 0; i < N\_TASKS; i++) {

        TaskData[i] = '0' + i;                                   (1)

        OSTaskCreate(Task,                                       (2)

                    (**void** \*)&TaskData[i],                        (3)

                    &TaskStk[i][TASK\_STK\_SIZE - 1],              (4)

                    i + 1);                                      (5)

    }

}

(1) Массив инициализируется и содержит символы ASCII от «0» до «9» (см. также Список 1.1 (5)).

(2) Цикл инициализирует идентичные задачи N\_TASKS  называемые Task(). Task() будет отвечать за размещение символа ASCII в случайном месте на экране. Фактически, каждый экземпляр Task() помещает свой символ.

(3) Каждая из этих задач получит указатель на массив символов ASCII. Каждая задача фактически получит указатель на другой символ.

(4) Для каждой задачи требуется собственное пространство стека (см. Листинг 1.1 (3)).

(5) При работе с uС/ОС-II каждая задача должна иметь уникальный приоритет. Поскольку номер приоритета 0 уже используется TaskStart(), я решил создать задачи с приоритетами от 1 до 10.

По мере создания каждой задачи ее значение зависит от того, является ли созданная задача более важной, чем создатель. Если создаваемая задача имеет более высокий приоритет, то uС/ОС-II будет немедленно ее выполнять. Однако, поскольку  TaskStart() имеет наивысший приоритет (приоритет 0), ни одна из созданных задач еще не будет выполнена.

Теперь мы можем возобновить обсуждение листинга 1.3.

(9) При работе с uС/ОС-II каждая задача должна представлять собой бесконечный цикл.

(10) Для отображения информации в нижней части окна DOS вызывается TaskStartDisp() (см. рис. 1.1). В частности, TaskStartDisp() будет печатать количество созданных задач, текущее использование ЦП в процентах, количество контекстных переключателей, версию, и, наконец имеет ли процессор блок с плавающей запятой (FPU) или нет.

(11) Затем TaskStart() проверяет, нажата ли клавиша путем вызова PC\_GetKey()

(12) & (13) TaskStart() определяет, нажата ли клавиша «ESC» на клавиатуре, и, если да, вызывает PC\_DOSReturn() для выхода из этого примера и возврата к запросу DOS. Чтобы узнать, как это делается, обратитесь к главе 18 PC Services.

(14) Если вы не нажали клавишу «Esc», глобальная переменная OSCtxSwCtr  (счетчик переключения контекста) очищается, чтобы мы могли отобразить количество переключений контекста за одну секунду.

(15) Наконец, TaskStart() приостанавливается (не будет выполняться) на одну полную секунду путем вызова OSTimeDlyHMSM(). HMSM обозначает часы, минуты, секунды и миллисекунды и соответствует аргументам, переданным OSTimeDlyHMSM() Так как  TaskStart() приостанавливается на одну секунду, в качестве приоритетной задачи () будет выполнена следующая наиболее важная задача, в данном случае  Task() c приоритетом 1. Следует отметить, что без OSTimeDlyHMSM() (или других подобных функций) TaskStart() будет истинным бесконечным циклом, а другие задачи никогда не получат шанса на запуск.

Код Task() показан в листинге 1.5.

**Листинг 1.5 Пример № 1, TEST.C, Задача ()**

**void**  Task (**void** \*pdata)

{

    INT8U  x;

    INT8U  y;

    INT8U  err;

**for** (;;) {                                                       (1)

        OSSemPend(RandomSem, 0, &err);                               (2)

        x = random(80);                                              (3)

        y = random(16);                                              (4)

        OSSemPost(RandomSem);                                        (5)

        PC\_DispChar(x, y + 5, \*(**char** \*)pdata, DISP\_FGND\_LIGHT\_GRAY); (6)

        OSTimeDly(1);                                                (7)

    }

}

(1) Как я уже упоминал ранее, задача uС/ОС-II обычно представляет собой бесконечный цикл.

(2) Задача начинается с получения семафора, который охраняет доступ к функции генератора случайных чисел компилятора Борланд. Это делается путем вызова OSSemPend() и передачи ему «дескриптора» (см. L1.1 (6)) семафора, который был создан для защиты доступа к функции генератора случайных чисел. Второй аргумент OSSemPend() используется для указания тайм-аута. Значение 0 означает, что эта задача будет ждать семафора вечно. Поскольку семафор был инициализирован с числом, равным одному, и никакая другая задача не запросила семафор, Task() может продолжать выполнение. Если бы семафор «принадлежал» другой задаче, uC/OS-II приостановил бы эту задачу и выполнил бы следующую по значимости задачу.

(3) Вызывается функция генератора случайных чисел, и возвращается значение от 0 до 79 (включительно). Это координата «X», где мы хотим отобразить символ «0» (для этой задачи) на экране.

(4) Снова вызывается генератор случайных чисел и возвращает число между 0 и 15 (включительно). Это значение будет использоваться для определения позиции «Y» отображаемого символа.

(5) Семафор освобождается путем вызова OSSemPost(). Здесь нужно просто указать обработчик семафора.

(6) Теперь мы можем отобразить символ, который был передан  Task() при создании  Task(). Для первого экземпляра  Task() символ будет равен «0», а для последнего экземпляра - «9». Я добавил смещение в 5 строк от верхней части, чтобы не перезаписывать заголовок в верхней части дисплея (см. рисунок 1.1).

(7) Наконец,  Task() вызывает OSTimeDly() ,чтобы сообщить uC/OS-II, что она выполнена, и дать другим задачам шанс на выполнение. Значение 1 означает, что я хочу, чтобы эта задача задержалась для одного такта или, 5 мс, так как частота тиков составляет 200 Гц. При вызове функции OSTimeDly(), uC/OS-II приостанавливает выполнение вызывающей функции и выполняет следующую наиболее важную задачу. В этом случае это будет другой экземпляр Task(), который будет отображать «1». Этот процесс продолжается для всех экземпляров Task() и, таким образом, поэтому рисунок 1.1 отображает как это происходит.

Если компилятор Borland C/C + + V4.5 установлен в каталог C:\BC45, вы можете экспериментировать с TEST.C. После изменения TEST.C, вы можете ввести MAKETEST  в командной строке каталога TEST , который строит новый TEST.EXE. Если у вас нет V4.5x Borland C/C + + или вы установили его в другой каталог, вы можете внести соответствующие изменения в  TEST.MAK, INCLUDES.H и  TEST.LNK.

Каталог SOURCE  содержит четыре файла: INCLUDES.H,  OS\_CFG.H,  TEST.C и TEST.LNK. OS\_CFG.H  используемых для определения опций конфигурации  µC/OS-II. TEST.LNK - командный файл компоновщика для компоновщика Borland, TLINK