**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.

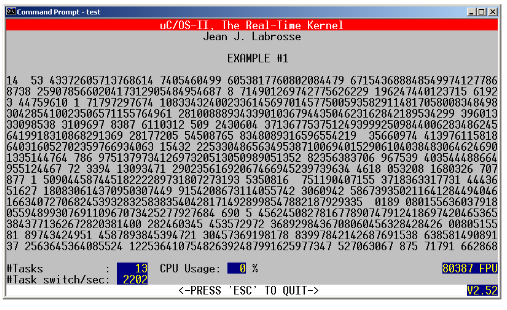
Код для примера # 1 находится в каталоге\SOFTWARE\uCOS-II\EX1\_x86L\BC45 установочного диска (по умолчанию - C:). Можно открыть окно DOS (называемое командной строкой в Windows 2000) и ввести:

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)

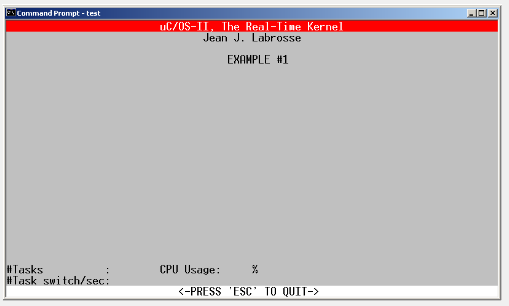
    }

}

(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

**Пример 2: Проверка стека**

Пример #2 демонстрирует функцию проверки стека, выполняемую в отношении uС/ОС-II. Объем пространства стека, используемого каждой задачей, отображается вместе с объемом свободного пространства стека. Кроме того, в примере 2 показано время выполнения функции проверки стека OSTaskStkChk(), поскольку оно зависит от размера каждого стека. Получается, что сильно используемый стек требует меньшего времени обработки.

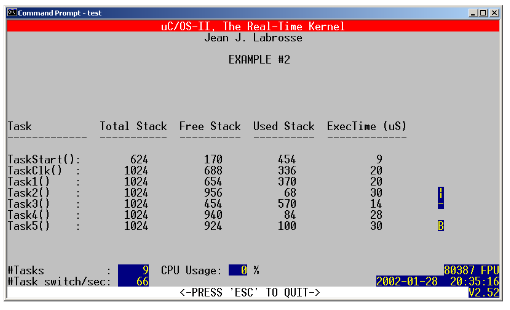
Код примера # 2 находится в каталоге \SOFTWARE\uCOS-II\EX2\_x86L\BC45. Можно открыть окно DOS и ввести:

CD \SOFTWARE\uCOS-II\Ex2\_x86L\BC45\TEST

Для выполнения примера # 2 просто введите TEST в командной строке. В окне DOS будет запущена программа TEST.EXE.

Примерно через одну секунду появится экран, показанный на рисунке 1.3.

**Рис. 1.3. Пример 2, выполняемый в окне DOS**



Пример 2 состоит из 9 задач, показанных в левой нижней части рисунка 1.3. Из этих 9 задач, в uC/OS-II создаются две внутренние задачи: задача простоя и задача, определяющая загрузку ЦП. В примере 2 создаются другие 7 задач.

В примере 2 показано, как можно отобразить статистику задач, выходящую за рамки количества созданных задач, количества контекстных переключателей и загрузки ЦП. В частности, в примере № 2 показано, как можно узнать, сколько места в стеке используется каждой задачей на самом деле и сколько времени требуется для выполнения, чтобы определить размер каждого стека задач.

В примере # 2 используется расширенная функция создания задачи (OSTaskCreateExt()) и функция проверки стека (OSTaskStkChk()). Проверка стека полезна, когда вы не знаете заранее, сколько места стека нужно выделить для каждой задачи. В этом случае вы бы выделили гораздо больше места в стеке, чем вам кажется нужным, и позволили бы uC/OS-II точно указать, сколько места в стеке используется на самом деле. Очевидно, вам нужно запустить приложение достаточно долго и в худшем случае, чтобы получить действительные номера. Окончательный размер стека должен соответствовать расширению системы, поэтому убедитесь, что вы выделяете от 10 до 25 процентов больше. Однако в критически важных для безопасности приложениях, возможно, вы даже захотите рассмотреть на 100% больше! То, что вы получаете от проверки стека, это фигура стадиодна; Вы не ищете точного использования стека.

При создании задачи функция проверки стека ((stack-checking) («Проверка стека») uC/OS-II заполняет стек задачи нулями. Вы делаете это, сообщая OSTaskCreateExt() , что вы хотите очистить стек при создании задачи и, что вы хотите проверить стек (то есть, установив OS\_TASK\_OPT\_STK\_CLR и OS\_TASK\_OPT\_STK\_CHK для аргумента функции opt ). Если предполагается создание и удаление задач, необходимо настроить эти параметры таким образом, чтобы каждый раз при создании задачи очищался новый стек. Следует отметить, что очистка стека OSTaskCreateExt() увеличивает затраты на выполнение, которые, очевидно, зависят от размера стека.

uC/OS-II сканирует стек, начиная с нижней части, пока не обнаружит ненулевую запись. При сканировании стека, uC/OS-II увеличивает счетчик, который показывает, сколько записей свободны.

Исходный код для примера# 2, находится в TEST.C в каталоге SOURCE . Туда можно попасть из каталога TEST , набрав:

CD ..\SOURCE

Части TEST.Cприведены в списке 1.6. Фактический код можно проверить с помощью избранного редактора кода.

**Листинг 1.6 Пример № 2, TEST.C**

#include "includes.h"                                                (1)

#define          TASK\_STK\_SIZE     512                               (2)

#define          TASK\_START\_ID       0                               (3)

#define          TASK\_CLK\_ID         1

#define          TASK\_1\_ID           2

#define          TASK\_2\_ID           3

#define          TASK\_3\_ID           4

#define          TASK\_4\_ID           5

#define          TASK\_5\_ID           6

#define          TASK\_START\_PRIO    10                               (4)

#define          TASK\_CLK\_PRIO      11

#define          TASK\_1\_PRIO        12

#define          TASK\_2\_PRIO        13

#define          TASK\_3\_PRIO        14

#define          TASK\_4\_PRIO        15

#define          TASK\_5\_PRIO        16

OS\_STK        TaskStartStk[TASK\_STK\_SIZE];                           (5)

OS\_STK        TaskClkStk[TASK\_STK\_SIZE];

OS\_STK        Task1Stk[TASK\_STK\_SIZE];

OS\_STK        Task2Stk[TASK\_STK\_SIZE];

OS\_STK        Task3Stk[TASK\_STK\_SIZE];

OS\_STK        Task4Stk[TASK\_STK\_SIZE];

OS\_STK        Task5Stk[TASK\_STK\_SIZE];

OS\_EVENT     \*AckMbox;                                               (6)

OS\_EVENT     \*TxMbox;

Исходя из того, что вы узнали в примере № 1, вы должны распознать:

(1) INCLUDES.H в качестве основного файла включения.

(2) Размер каждого стека задач (TASK\_STK\_SIZE). Опять-таки, для простоты я сделал все размеры стека одинаковыми, но при работе с uC/OS-II размер стека для каждой задачи может быть разным.

(5) Хранилище для стеков задач.

main() Для примера # 2, показан в листинге 1.7 и очень похож на main() примера # 1. Я только опишу различия.

**Листинг 1.7 Пример № 2, TEST.C, main()**

**void** main (**void**)

{

    OS\_STK \*ptos;

    OS\_STK \*pbos;

    INT32U  size;

    PC\_DispClrScr(DISP\_FGND\_WHITE);

    OSInit();

    PC\_DOSSaveReturn();

    PC\_VectSet(uCOS, OSCtxSw);

    PC\_ElapsedInit();                                              (1)

    ptos        = &TaskStartStk[TASK\_STK\_SIZE - 1];                (2)

    pbos        = &TaskStartStk[0];

    size        = TASK\_STK\_SIZE;

    OSTaskStkInit\_FPE\_x86(&ptos, &pbos, &size);                    (3)

    OSTaskCreateExt(TaskStart,                                     (4)

                   (**void** \*)0,

                   ptos,                                            (5)

                   TASK\_START\_PRIO,                                 (6)

                   TASK\_START\_ID,                                   (7)

                   pbos,                                            (8)

                   size,                                            (9)

                   (**void** \*)0,                                      (10)

                   OS\_TASK\_OPT\_STK\_CHK | OS\_TASK\_OPT\_STK\_CLR);     (11)

    OSStart();

}

(1) main() вызывает  PC\_ElapsedInit()для инициализации функции измерения прошедшего времени, которая будет использоваться для измерения времени выполнения OSTaskStkChk(). Эта функция в основном измеряет время выполнения (т.е. накладные расходы) двух функций: PC\_ElapsedStart() и PC\_ElapsedStop(). Измеряя это время, мы можем точно определить, сколько времени занимает выполнение кода между этими двумя вызовами.

(2) & (3) TaskStart()в примере № 2 будет вызывать библиотеку эмуляции с плавающей запятой вместо использования блока с плавающей запятой (FPU), который присутствует на 80486 и более поздних компьютерах. Компилятор Borland по умолчанию использует свою библиотеку эмуляции, если FPU не обнаружен. Другими словами, если бы вы запускали TEST.EXE на машине на базе DOS, оснащенной 80386EX Intel (без сопроцессора 80387), то блок с плавающей запятой был бы эмулирован. Библиотека эмуляции, к сожалению, не повторяется, и мы должны «обмануть» ее, чтобы позволить нескольким задачам выполнять математику с плавающей запятой. Это достигается вызовом OSTaskStkInit\_FPE\_x86() (см. главу 14, порт 80x86). На рисунке 1.3 показано, что размер стека для TaskStart()равен 624 вместо 1024. Это потому, что OSTaskStkInit\_FPE\_x86() резервирует разницу для библиотеки эмуляции с плавающей запятой.

(4) Вместо вызова OSTaskCreate() для создания TaskStart() мы должны вызвать OSTaskCreateExt() (расширенная версия OSTaskCreate()), потому что мы изменили стек, а также потому, что мы хотим проверить размер стека во время выполнения (описано ниже).

(5) OSTaskStkInit\_FPE\_x86() изменяет указатель сверху стека, поэтому мы должны передать новый указатель  OSTaskCreateExt().

(6) Вместо передачи хардкодного приоритета (как я это сделал в примере № 1), я создал символ #define (см. L1.6 (4)).

(7) OSTaskCreateExt()требует, чтобы вы передали идентификатор задачи (ID). Фактическое значение может быть любым, так как в настоящее время это поле фактически не используется uС/ОС-II.

(8) OSTaskStkInit\_FPE\_x86() изменяет указатель снизу стека, поэтому мы должны передать новый указатель OSTaskCreateExt().

(9) OSTaskStkInit\_FPE\_x86() также изменяет размер стека, поэтому мы должны передать новый размер OSTaskCreateExt ().

(10) Одним из аргументов OSTaskCreateExt() является указатель расширения блока управления задачами (TCB). Этот аргумент не используется в примере # 2, поэтому мы просто передаем указатель NULL.

(11) Наконец, последним аргументом для OSTaskCreateExt() является набор опций (т.е. битов), которые сообщают OSTaskCreateExt() , что мы будем выполнять проверку размера стека и что мы хотим очистить стек, когда задача будет создана.

TaskStart() аналогичен описанному в примере № 1 и показан в списке 1.8. Опять же, я только опишу различия.

**Листинг 1.8 Пример 2, TEST.C, TaskStart()**

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

{

#if OS\_CRITICAL\_METHOD == 3

    OS\_CPU\_SR  cpu\_sr;

#endif

    INT16S     key;

    pdata = pdata;

    TaskStartDispInit();                                          (1)

    OS\_ENTER\_CRITICAL();

    PC\_VectSet(0x08, OSTickISR);

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

    OS\_EXIT\_CRITICAL();

    OSStatInit();

    AckMbox = OSMboxCreate((**void** \*)0);                            (2)

    TxMbox  = OSMboxCreate((**void** \*)0);

    TaskStartCreateTasks();                                       (3)

**for** (;;) {

        TaskStartDisp();

**if** (PC\_GetKey(&key)) {

**if** (key == 0x1B) {

                PC\_DOSReturn();

            }

        }

        OSCtxSwCtr = 0;

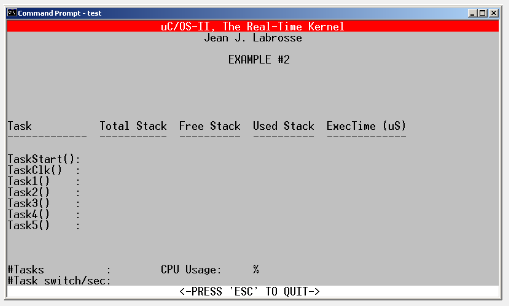
        OSTimeDly(OS\_TICKS\_PER\_SEC);                              (4)

    }

}

(1) Хотя вызов функции идентичен TaskStartDispInit() инициализирует дисплей, как показано на рис. 1.4.

**Рисунок 1.4 Инициализация дисплея TaskStartDispInit()**



(2) uC/OS-II позволяет выполнять задачи или отправлять сообщения ISR другим задачам. В примере No 2 задача No 4 отправит сообщение задаче No 5, и задача No 5 ответит на задачу No 4 сообщением подтверждения (описано ниже). Для этого необходимо создать два объекта ядра, которые называются почтовыми ящиками. Почтовый ящик позволяет задаче или ISR отправлять «указатель» на другую задачу. В почтовом ящике есть место только для одного указателя. Указатель указывает на специфичность приложения и, конечно, отправитель и получатель должны согласовать содержание сообщения.

(3) TaskStartCreateTasks() создает шесть задач с помощью OSTaskCreateExt(). Эти задачи не будут выполнять операции с плавающей запятой, поэтому нет необходимости вызывать OSTaskStkInit\_FPE\_x86() для изменения стеков. Тем не менее, я буду делать стековую проверку этих задач, поэтому я вызываю OSTaskCreateExt()  с соответствующим набором параметров.

(4) В примере № 1 я вызвал  OSTimeDlyHMSM() , чтобы задержать TaskStart() на одну секунду. Я решил использовать  OSTimeDly(OS\_TICKS\_PER\_SEC) , чтобы показать вам, что вы можете использовать любой метод. Однако OSTimeDly() немного быстрее, чем  OSTimeDlyHMSM() .

Код Task1() приведен в листинге 1.9. Task1() проверяет размер стека для каждой из семи задач приложения (шесть задач, созданных TaskStart() в том числе и для TaskStart() )

**Листинг 1.9 Пример № 2, TEST.C, Task1()**

**void**  Task1 (**void** \*pdata)

{

    INT8U       err;

    OS\_STK\_DATA data;

    INT16U      **time**;

    INT8U       i;

**char**        s[80];

    pdata = pdata;

**for** (;;) {

**for** (i = 0; i < 7; i++) {

            PC\_ElapsedStart();                                        (1)

            err  = OSTaskStkChk(TASK\_START\_PRIO + i, &data);          (2)

**time** = PC\_ElapsedStop();                                  (3)

**if** (err == OS\_NO\_ERR) {

**sprintf**(s, "%4ld        %4ld        %4ld        %6d", (4)

                        data.OSFree + data.OSUsed,

                        data.OSFree,

                        data.OSUsed,

**time**);

                PC\_DispStr(19, 12 + i, s, DISP\_FGND\_YELLOW);          (5)

            }

        }

        OSTimeDlyHMSM(0, 0, 0, 100);                                  (6)

    }

}

(1) & (3) Время выполнения OSTaskStkChk()измеряется путем обтекания OSTaskStkChk()вызовами PC\_ElapsedStart()и PC\_ElapsedStop().PC\_ElapsedStop() возвращает разность времени в микросекундах.

(2) OSTaskStkChk()- это сервис, предоставляемый uС/OS-II, позволяющая вашему коду определить фактическое использование стека задачи. Вы вызываете OSTaskStkChk() , передавая ему приоритет задачи, которую хотите проверить. Вторым аргументом функции является указатель на структуру данных, которая будет содержать информацию о стеке задачи. В частности, OS\_STK\_DATA содержит количество используемых байтов и количество свободных байтов. OSTaskStkChk() возвращает код ошибки, указывающий на успешность вызова. Это не было бы успешным, если бы я сдал номер приоритета задачи, которой не было.

(4) & (5) Информация, полученная OSTaskStkChk() , форматируется в строку и отображается.

(6) Я решил выполнять эту задачу 10 раз в секунду, но в реальном продукте или приложении вы, скорее всего, запускаете проверку стека каждые несколько секунд или около того. Другими словами, может оказаться нецелесообразным использовать ценное время обработки ЦП для определения наихудшего роста стека.

Код  Task2()и  Task3()показан в листинге 1.10. Обе эти задачи отображают вращающееся колесо. Эти две задачи практически идентичны. Task3() выделяет и инициализирует фиктивный массив размером 500 байт. Я хотел «использовать» пространство стека, чтобы показать вам, что OSTaskStkChk() сообщит, что Task3()имеет 502 байт меньше, чем Task2() в своем стеке (500 байт для массива и 2 байта для 16-битного целого числа). Колесо Task2() вращается по часовой стрелке при 5 вращениях в секунду, а колесо Task3() вращается против часовой стрелки при 2,5 вращениях в секунду.

**Примечание**.При запуске примера # 2 в окне Windows 95, 98, Me, NT, 2000 или XP вращение может появиться не так быстро. Просто нажмите и удерживайте клавишу «Alt», а затем нажмите клавишу «Enter» на клавиатуре, чтобы окно DOS использовало весь экран. Можно вернуться в режим окна, повторив операцию.

**Листинг 1.10 Пример № 2, TEST.C, Task2 () и Task3 ()**

**void**  Task2 (**void** \*data)

{

    data = data;

**for** (;;) {

        PC\_DispChar(70, 15, '|',  DISP\_FGND\_WHITE + DISP\_BGND\_RED);

        OSTimeDly(10);

        PC\_DispChar(70, 15, '/',  DISP\_FGND\_WHITE + DISP\_BGND\_RED);

        OSTimeDly(10);

        PC\_DispChar(70, 15, '-',  DISP\_FGND\_WHITE + DISP\_BGND\_RED);

        OSTimeDly(10);

        PC\_DispChar(70, 15, '\\', DISP\_FGND\_WHITE + DISP\_BGND\_RED);

        OSTimeDly(10);

    }

}

**void**  Task3 (**void** \*data)

{

**char**    dummy[500];

    INT16U  i;

    data = data;

**for** (i = 0; i < 499; i++) {

        dummy[i] = '?';

    }

**for** (;;) {

        PC\_DispChar(70, 16, '|',  DISP\_FGND\_WHITE + DISP\_BGND\_BLUE);

        OSTimeDly(20);

        PC\_DispChar(70, 16, '\\', DISP\_FGND\_WHITE + DISP\_BGND\_BLUE);

        OSTimeDly(20);

        PC\_DispChar(70, 16, '-',  DISP\_FGND\_WHITE + DISP\_BGND\_BLUE);

        OSTimeDly(20);

        PC\_DispChar(70, 16, '/',  DISP\_FGND\_WHITE + DISP\_BGND\_BLUE);

        OSTimeDly(20);

    }

}

Task4() и Task5()приведены в Листинге 1.11.

**Листинг 1.11 Пример № 2, TEST.C, Task4 () и Task5 ()**

**void**  Task4 (**void** \*data)

{

**char**   txmsg;

    INT8U  err;

    data  = data;

    txmsg = 'A';

**for** (;;) {

        OSMboxPost(TxMbox, (**void** \*)&txmsg);                         (1)

        OSMboxPend(AckMbox, 0, &err);                               (2)

        txmsg++;                                                    (3)

**if** (txmsg == 'Z') {

            txmsg = 'A';

        }

    }

}

**void**  Task5 (**void** \*data)

{

**char**  \*rxmsg;

    INT8U  err;

    data = data;

**for** (;;) {

        rxmsg = (**char** \*)OSMboxPend(TxMbox, 0, &err);                  (4)

        PC\_DispChar(70, 18, \*rxmsg, DISP\_FGND\_YELLOW + DISP\_BGND\_RED);(5)

        OSTimeDlyHMSM(0, 0, 1, 0);                                    (6)

        OSMboxPost(AckMbox, (**void** \*)1);                               (7)

    }

}

(1) Task4() отправляет сообщение (ASCII-символ) в Task5()путем отправки сообщения в TxMbox.

(2) Task4()затем ожидает подтверждения от Task5(), ожидая AckMbox. Второй аргумент вызова OSMboxPend() указывает тайм-аут, и я указал ожидание вечно, так как я передал значение 0. Указав ненулевое значение, Task4() прекратил бы ожидание после указанного тайм-аута. Тайм-аут задается как целое число тактов.

(3) Сообщение изменяется, когда Task5()подтверждает предыдущее сообщение.

(4) Когда Task5()начинает выполнение, он немедленно ожидает (навсегда) поступления сообщения через почтовый ящик TxMbox.

(5) При поступлении сообщения Task5() отображает его на экране.

(6) и (7) Task5() затем ожидает в течение одной секунды, прежде чем подтвердить Task4() . Я решил подождать одну секунду, чтобы вы видели, как она меняется на экране. На самом деле, должна быть задержка в Task5() или одна в Task5() , иначе все задачи с более низким приоритетом не будут запускаться!

Наконец, код TaskClk() отображен в листинге 1.12. Эта задача выполняется каждую секунду и просто получает текущую дату и время от услуги ПК под названием PC\_GetDateTime() (см. главу 18,  PC Services) и отображает их на экране.

**Листинг 1.12 Пример № 2, TEST.C, TaskClk ()**

**void**  TaskClk (**void** \*data)

{

**char** s[40];

    data = data;

**for** (;;) {

        PC\_GetDateTime(s);

        PC\_DispStr(60, 23, s, DISP\_FGND\_BLUE + DISP\_BGND\_CYAN);

        OSTimeDly(OS\_TICKS\_PER\_SEC);

    }

}

Если компилятор Borland C/C + + V4.5x 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 изменения в TEST.MAK, INCLUDES.H и TEST.LNK соответственно.

Каталог SOURCE содержит четыре файла:  INCLUDES.H, OS\_CFG.H, TEST.C и TEST.LNK. OS\_CFG.H , используется которые используются для определения опций конфигурации (параметров конфигурации, параметров конфигурации, параметров). TEST.LNK- командный файл компоновщика для компоновщика Borland, TLINK.

**Пример 3: Расширение функциональных возможностей с помощью TCB**

В примере # 3 показано, как можно расширить функциональные возможности uC/OS-II. В частности, в примере #3 используется функция расширения TCB (блок управления задачами(task control block)) OSTaskCreateExt(), пользовательская сцепка переключения контекста задач (context switch hook) [OSTaskSwHook()], пользовательская сцепка задачи сбора статистики(statistic task hook) [OSTaskStatHook() ] и очереди сообщений. В этом примере показано, как легко определить, сколько раз выполняется задача и сколько времени занимает выполнение задачи. Время выполнения может использоваться для определения загрузки ЦП задачи относительно других задач.

Код примера # 3 находится в каталоге\SOFTWARE\uCOS-II\EX3\_x86L\BC45. Можно открыть окно DOS и ввести:

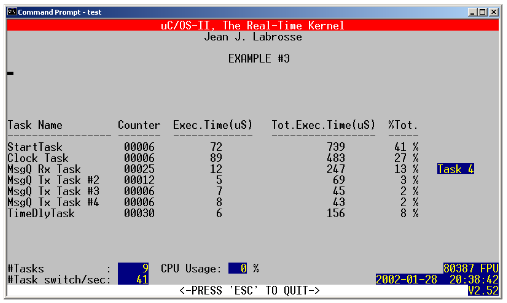
CD \SOFTWARE\uCOS-II\Ex3\_x86L\BC45\TEST

Как обычно, для выполнения примера # 3 просто введите TEST в командной строке. В окне DOS будет запущена программа TEST.EXE.

Примерно через одну секунду появится экран, показанный на рисунке 1.5. Я позволил  TEST.EXE работать в течение пары секунд, прежде чем я захватил снимок экрана. Семь задач показаны вместе с тем, сколько раз они выполнялись (столбец Counter ), время выполнения каждой задачи в микросекундах (столбец Exec.Time(uS)), общее время выполнения с момента запуска (столбец Tot.Exec.Time(uS)) и, наконец, процент времени выполнения каждой задачи относительно других задач (%Tot. столбец).

Пример 3 состоит из 9 задач, показанных в левой нижней части рисунка 1.5. Из этих 9 задач, в uC/OS-II создаются две внутренние задачи: задача простоя и задача, определяющая загрузку ЦП. В примере № 3 создаются другие 7 задач.

**Рис. 1.5. Пример 3, выполняемый в окне DOS**



Части TEST.C приведены в листинге 1.13. Фактический код можно проверить с помощью избранного редактора кода.

**Листинг 1.13 Пример № 3, TEST.C**

#include  "includes.h"

#define          TASK\_STK\_SIZE     512

#define          TASK\_START\_ID       0

#define          TASK\_CLK\_ID         1

#define          TASK\_1\_ID           2

#define          TASK\_2\_ID           3

#define          TASK\_3\_ID           4

#define          TASK\_4\_ID           5

#define          TASK\_5\_ID           6

#define          TASK\_START\_PRIO    10

#define          TASK\_CLK\_PRIO      11

#define          TASK\_1\_PRIO        12

#define          TASK\_2\_PRIO        13

#define          TASK\_3\_PRIO        14

#define          TASK\_4\_PRIO        15

#define          TASK\_5\_PRIO        16

#define          MSG\_QUEUE\_SIZE     20

**typedef** **struct** {                                                    (1)

**char**    TaskName[30];

    INT16U  TaskCtr;

    INT16U  TaskExecTime;

    INT32U  TaskTotExecTime;

} TASK\_USER\_DATA;

OS\_STK          TaskStartStk[TASK\_STK\_SIZE];

OS\_STK          TaskClkStk[TASK\_STK\_SIZE];

OS\_STK          Task1Stk[TASK\_STK\_SIZE];

OS\_STK          Task2Stk[TASK\_STK\_SIZE];

OS\_STK          Task3Stk[TASK\_STK\_SIZE];

OS\_STK          Task4Stk[TASK\_STK\_SIZE];

OS\_STK          Task5Stk[TASK\_STK\_SIZE];

TASK\_USER\_DATA  TaskUserData[7];                                    (2)

OS\_EVENT     \*MsgQueue;                                             (3)

**void**         \*MsgQueueTbl[20];

(1) Создается структура данных для хранения дополнительной информации о задаче. В частности, структура данных позволяет добавить имя к задаче (в uC/OS-II эта функция не предоставляется напрямую), отслеживать, сколько раз задача выполнялась, сколько времени занимает выполнение задачи и, наконец, общее время выполнения задачи.

(2) Массив TASK\_USER\_DATA выделяется для хранения информации о каждой созданной задаче (за исключением задачи ожидания и статистической задачи).

(3) uC/OS-II обеспечивает другой механизм передачи сообщений, называемый очередью сообщений. Очередь сообщений подобна почтовому ящику, за исключением того, что вместо возможности отправки одного указателя очередь может содержать более одного сообщения (т.е. указателей). Таким образом, очередь сообщений позволяет вашим задачам или прерываниям(ISRs ) отправлять сообщения другим задачам. Каждый из указателей указывает на специфичность приложения и, конечно, как отправитель, так и получатель должны согласовать содержание сообщений. Для создания очереди сообщений необходимы два элемента: структура OS\_EVENT  и массив указателей. Глубина очереди определяется количеством указателей, выделенных в массиве указателей. В этом случае очередь сообщений содержит 20 записей.

main()показан в Листинге 1.14. Далее будут описаны только новые функции.

**Листинг 1.14 Пример № 3, TEST.C, main ()**

**void**  main (**void**)

{

    PC\_DispClrScr(DISP\_BGND\_BLACK);

    OSInit();

    PC\_DOSSaveReturn();

    PC\_VectSet(uCOS, OSCtxSw);

    PC\_ElapsedInit();

**strcpy**(TaskUserData[TASK\_START\_ID].TaskName, "StartTask");      (1)

    OSTaskCreateExt(TaskStart,

                    (**void** \*)0,

                    &TaskStartStk[TASK\_STK\_SIZE - 1],

                    TASK\_START\_PRIO,

                    TASK\_START\_ID,

                    &TaskStartStk[0],

                    TASK\_STK\_SIZE,

                    &TaskUserData[TASK\_START\_ID],                   (2)

                    0);

    OSStart();

}

(1) Перед созданием задачи назначим ей имя с помощью функции библиотеки ANSI C strcpy() . Имя сохраняется в структуре данных (см. L1.13 (1)), присвоенной задаче.

(2) TaskStart() создается с помощью OSTaskCreateExt()и передает указатель на структуру пользовательских данных. Блок управления задачами (TCB) каждой задачи в uC/OS-II может сохранять указатель на пользовательскую структуру данных (для получения более подробной информации см. главу 3, Структура ядра). Это позволяет расширить функциональные возможности uC/OS-II, как вы увидите в ближайшее время.

Код TaskStart()показан в листинге 1.15.

**Листинг 1.15 Пример 3, TEST.C, TaskStart ()**

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

{

#if OS\_CRITICAL\_METHOD == 3

    OS\_CPU\_SR  cpu\_sr;

#endif

    INT16S     key;

    pdata = pdata;

    TaskStartDispInit();

    OS\_ENTER\_CRITICAL();

    PC\_VectSet(0x08, OSTickISR);

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

    OS\_EXIT\_CRITICAL();

    OSStatInit();

    MsgQueue = OSQCreate(&MsgQueueTbl[0], MSG\_QUEUE\_SIZE);         (1)

    TaskStartCreateTasks();                                        (2)

**for** (;;) {

        TaskStartDisp();

**if** (PC\_GetKey(&key)) {

**if** (key == 0x1B) {

                PC\_DOSReturn();

            }

        }

        OSCtxSwCtr = 0;

        OSTimeDly(OS\_TICKS\_PER\_SEC);

    }

}

(1) Добавлено не так много, кроме создания очереди сообщений, которая будет использоваться Task1() , Task2() , Task3() и Task4() .

(2) Как и в примере № 2, TaskStartCreateTasks() создает 6 задач. Отличие состоит в том, что каждой задаче назначается запись в массиве TaskUserData []. При создании каждой задачи ей присваивается имя, как и мне при создании TaskStart () (см. L1.14 (1)).

Как только приложение TaskStart() вызовет OSTimeDly (OS\_TICKS\_PER\_SEC), µC/OS-II вызовет следующую задачу с наивысшим приоритетом, которая готова к запуску. Это будет Task1() . Список 1.16 показывает код для Task1() , Task2() , Task3() и Task4() , потому что я буду обсуждать их далее.

**Листинг 1.16 Пример № 3, TEST.C, Task1 () - Task4 ()**

**void**  Task1 (**void** \*pdata)

{

**char**  \*msg;

    INT8U  err;

    pdata = pdata;

**for** (;;) {

        msg = (**char** \*)OSQPend(MsgQueue, 0, &err);                   (1)

        PC\_DispStr(70, 13, msg, DISP\_FGND\_YELLOW + DISP\_BGND\_BLUE); (2)

        OSTimeDlyHMSM(0, 0, 0, 100);                                (3)

    }

}

**void**  Task2 (**void** \*pdata)

{

**char**  msg[20];

    pdata = pdata;

**strcpy**(&msg[0], "Task 2");

**for** (;;) {

        OSQPost(MsgQueue, (**void** \*)&msg[0]);                          (4)

        OSTimeDlyHMSM(0, 0, 0, 500);                                 (5)

    }

}

**void**  Task3 (**void** \*pdata)

{

**char**  msg[20];

    pdata = pdata;

**strcpy**(&msg[0], "Task 3");

**for** (;;) {

        OSQPost(MsgQueue, (**void** \*)&msg[0]);                           (6)

        OSTimeDlyHMSM(0, 0, 0, 500);

    }

}

**void**  Task4 (**void** \*pdata)

{

**char**  msg[20];

    pdata = pdata;

**strcpy**(&msg[0], "Task 4");

**for** (;;) {

        OSQPost(MsgQueue, (**void** \*)&msg[0]);                           (7)

        OSTimeDlyHMSM(0, 0, 0, 500);

    }

}

(1) Task1() ожидает получения сообщения через очередь сообщений.

(2) При поступлении сообщения оно отображается на экране.

(3) Задача задерживается на 100 мс, чтобы вы могли увидеть полученное сообщение.

(4) Task2() передает сообщение «Задача 2» в Task1() через очередь сообщений.

(5) Task1() ожидает полсекунды, прежде чем отправить другое сообщение.

(6) & (7) Task3() и Task4() посылает свое сообщение и также ожидает полсекунды между сообщениями.

Другая задача, Task5() (не показана), не делает ничего полезного, кроме задержки на 1/10 секунды. Обратите внимание на то, что все задачи, выполняемые с использованием системы uС/OS-II, должны вызывать службу, предоставляемую системой uС/OS-II, для ожидания истечения времени или возникновения события. Если этого не сделать, задача предотвращает выполнение всех задач с более низким приоритетом.

Наконец, TaskClk() (также не показан) отображает текущую дату и время один раз в секунду.

События происходят за кулисами, которые не являются очевидными, просто глядя на задачи в TEST.C. На V2.52 существует 9 функций захвата(hooks), и прототип этих функций показан в листинге 1.17.

**Листинг 1,17 функции зацепки(hooks) uC/OS-II**

**void**  OSInitHookBegin(**void**);

**void**  OSInitHookEnd(**void**);

**void**  OSTaskCreateHook(OS\_TCB \*ptcb);

**void**  OSTaskDelHook(OS\_TCB \*ptcb);

**void**  OSTaskIdleHook(**void**);

**void**  OSTaskStatHook(**void**);

**void**  OSTaskSwHook(**void**);

**void**  OSTCBInitHook(OS\_TCB \*ptcb);

**void**  OSTimeTickHook(**void**);

Функции захвата(hook) обычно находятся в файле под названием OS\_CPU\_C.C и, как правило, записываются пользователем, который использует порт для процессора, который вы намереваетесь использовать. Однако если для константы конфигурации с именем OS\_CPU\_HOOKS\_EN задано значение 0, можно объявить функции захвата в другом файле. OS\_CPU\_HOOKS\_EN является одной из многих констант конфигурации, найденных в файле заголовка  OS\_CFG.H. Для каждого проекта, использующего uC/OS-II, требуется своя версия  OS\_CFG.H, так как для каждого проекта может потребоваться по-разному настроить uC/OS-II. Каждый пример, приведенный в этой книге, содержит собственные  OS\_CFG.H в каталоге SOURCE .

В примере #3 я установил значение OS\_CPU\_HOOKS\_EN , равное 0, и переопределил функциональность функций захвата в TEST.C. Как показано в листинге 1.18, семь из девяти функций захвата фактически ничего не делают и, таким образом, не содержат никакого кода.

**Листинг 1.18 Пример 3, TEST.C, пустые функции захвата(hook)**

**void**  OSInitHookBegin (**void**)

{

}

**void**  OSInitHookEnd (**void**)

{

}

**void**  OSTaskCreateHook (OS\_TCB \*ptcb)

{

    ptcb = ptcb;

}

**void**  OSTaskDelHook (OS\_TCB \*ptcb)

{

    ptcb = ptcb;

}

**void**  OSTaskIdleHook (**void**)

{

}

**void**  OSTCBInitHook (OS\_TCB \*ptcb)

{

    ptcb = ptcb;

}

**void**  OSTimeTickHook (**void**)

{

}

Код для OSTaskSwHook() показан в листинге 1.19 и позволяет нам измерить время выполнения каждой задачи, отслеживать, как часто каждая задача выполняется, и накапливать общее время выполнения каждой задачи. Вызов OSTaskSwHook() происходит при переходе с низкоприоритетной задачи на высокоприоритетную.

**Листинг 1.19 задача захвата переключения контекста, OSTaskSwHook ()**

**void**  OSTaskSwHook (**void**)

{

    INT16U           **time**;

    TASK\_USER\_DATA  \*puser;

**time**  = PC\_ElapsedStop();                                    (1)

    PC\_ElapsedStart();                                           (2)

    puser = OSTCBCur->OSTCBExtPtr;                               (3)

**if** (puser != (TASK\_USER\_DATA \*)0) {                          (4)

        puser->TaskCtr++;                                        (5)

        puser->TaskExecTime     = **time**;                          (6)

        puser->TaskTotExecTime += **time**;                          (7)

    }

}

(1) Таймер на ПК получает время выполнения задачи, которая «выключается» через PC\_ElapsedStop().

(2) Предполагается, что таймер был запущен путем вызова PC\_ElapsedStart(), когда задача была «включена». Первый переключатель контекста, вероятно, прочитает неверное значение, но это не очень важно.

(3) При вызове OSTaskSwHook() глобальный указатель OSTCBCur указывает на TCB текущей задачи, в то время как OSTCBHighRdy указывает на TCB новой задачи. В этом случае, однако, мы не используем OSTCBHighRdy. OSTaskSwHook() извлекает указатель на расширение TCB , переданное в OSTaskCreateExt() .

(4) Затем мы проверяем, чтобы убедиться, что мы не передаем нулевой указатель. Фактически, некоторые задачи в этом примере не содержат указателя расширения TCB: задача ожидания(idle ) и задача статистики(statistic ).

(5) Мы увеличиваем счетчик, который показывает, сколько раз задача выполнялась. Этот счетчик полезен для определения того, выполняется ли конкретная задача.

(6) Измеренное время выполнения (в микросекундах) сохраняется в TCB дополнении.

(7) Общее время выполнения (в микросекундах) задачи также сохраняется в дополнении TCB . Это позволяет определить процент времени, которое каждая задача занимает по отношению к другим задачам в приложении (будет обсуждаться в ближайшее время).

Если функция включена (см. OS\_TASK\_STAT\_EN в OS\_CFG.H), статистическая задача, OSTaskStat() вызывает определяемую пользователем функцию OSTaskStatHook() , которая показана в листинге 1.20. OSTaskStatHook()вызывается каждую секунду.

**Листинг 1.20 захват задачи статистики, OSTaskStatHook ()**

**void**  OSTaskStatHook (**void**)

{

**char**    s[80];

    INT8U   i;

    INT32U  total;

    INT8U   pct;

    total = 0L;

**for** (i = 0; i < 7; i++) {

        total += TaskUserData[i].TaskTotExecTime;                    (1)

        DispTaskStat(i);                                             (2)

    }

**if** (total > 0) {

**for** (i = 0; i < 7; i++) {

            pct = 100 \* TaskUserData[i].TaskTotExecTime / total;     (3)

**sprintf**(s, "%3d %%", pct);

            PC\_DispStr(62,                                           (4)

                       i + 11,

                       s,

                       DISP\_FGND\_BLACK + DISP\_BGND\_LIGHT\_GRAY);

        }

    }

**if** (total > 1000000000L) {

**for** (i = 0; i < 7; i++) {

            TaskUserData[i].TaskTotExecTime = 0L;

        }

    }

}

(1) Вычисляется общее время выполнения всех задач (за исключением статистической задачи).

(2) Отдельная статистика отображается в нужном месте на экране командой  DispTaskStat() , которая выполняет преобразование значений в ASCII. Кроме того,  DispTaskStat() также отображает имя каждой задачи.

(3) & (4) Процент времени выполнения вычисляется для каждой задачи и отображается.

Если компилятор Borland C/C + + V4.5x 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 построит новый TEST.EXE. Если у вас нет V4.5x Borland C/C + + или вы установили его в другой каталог, вы можете внести изменения в TEST.MAK, INCLUDES.H и TEST.LNK соответственно.

Каталог SOURCE содержит четыре файла: INCLUDES.H, OS\_CFG.H, TEST.C and TEST.LNK. OS\_CFG.H используется для определения опций конфигурации (параметров конфигурации, параметров конфигурации, параметров). TEST.LNK- командный файл компоновщика для компоновщика Borland, TLINK

**Пример 4: Порты**

uC/OS-II написан полностью на языке C и требует определенного кода процессора для его адаптации к различным процессорам. Этот код процессора называется портом. Эта книга поставляется с двумя портами для семейства процессоров Intel 80x86: Ix86L (см. главу 14) и Ix86L-FP (см. главу 15). Ix86L используется с процессорами 80x86, которым не повезло иметь FPU (блок с плавающей запятой) и Ix86L использовался во всех примерах до сих пор. Следует отметить, что Ix86L по-прежнему работает на процессорах 80x86 с FPU. Ix86L-FP позволяет приложениям использовать аппаратные возможности с плавающей запятой в процессорах, совместимых с более высокими процессорами 80x86. В примере 4 используется Ix86L-FP.

В этом примере я создал 10 одинаковых задач, каждая из которых выполняется 200 раз в секунду. Каждая задача вычисляет синус и косинус угла (в градусах). Угол, вычисляемый каждой задачей, смещен друг от друга на 36 градусов (360 градусов, разделенных на 10 задач). Каждый раз при выполнении задачи угол для вычисления увеличивается на 0,01 градуса.

Код примера # 4 находится в каталоге \SOFTWARE\uCOS-II\EX4\_x86L\BC45. Можно открыть окно DOS и ввести:

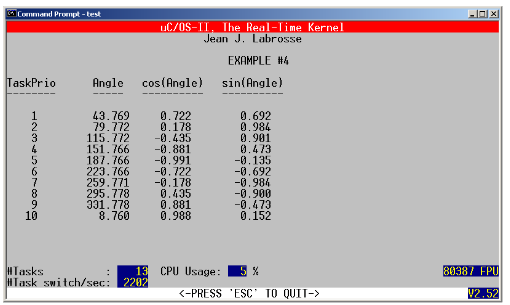
CD \SOFTWARE\uCOS-II\Ex4\_x86L\BC45\TEST

Как обычно, для выполнения примера # 4 просто введите TEST в командной строке. В окне DOS будет запущена программа TEST.EXE.

Примерно через две секунды появится экран, показанный на рис. 1.6. Я позволил TEST.EXE работать в течение нескольких секунд, прежде чем я захватил снимок экрана.

Пример #4 состоит из 13 задач, как показано в левой нижней части рис. 1.6. Из этих 13 задач, в uC/OS-II создаются две внутренние задачи: задача простоя и задача, определяющая загрузку ЦП. В примере #4 создаются другие 11 задач.

**Рисунок -1.6 Пример # 4, выполняемый в окне DOS**



К настоящему времени вы должны найти способ обойти TEST.C. Пример # 4 не вводит слишком много новых понятий. Тем не менее, есть несколько тонкостей, сделанных за сценой, которые я опишу после обсуждения нескольких пунктов в TEST.C. Листинг 1.21 показывает код для создания 10 идентичных задач приложений.

**Листинг 1.21 Пример № 4, TEST.C, TaskStartCreateTasks ()**

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

{

    INT8U  i;

    INT8U  prio;

**for** (i = 0; i < N\_TASKS; i++) {

        prio        = i + 1;                                        (1)

        TaskData[i] = prio;                                         (2)

        OSTaskCreateExt(Task,

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

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

                        prio,

                        0,

                        &TaskStk[i][0],

                        TASK\_STK\_SIZE,

                        (**void** \*)0,

                        OS\_TASK\_OPT\_SAVE\_FP);                       (4)

    }

}

(1) Из-за того, что в uC/OS-II не допускается выполнение нескольких задач с одинаковым приоритетом, приоритет идентичных задач смещается на 1, поскольку приоритет задачи # 0 назначен TaskStart() .

(2) Приоритет каждой задачи помещается в массив.

(3) uC/OS-II позволяет передавать аргумент задаче при первом запуске задачи. Этот аргумент является указателем, и я обычно называю его pdata  (указатель на данные). Сохраненный в массиве приоритет задачи фактически передается в качестве аргумента задачи pdata .

(4) Каждая из задач будет выполнять вычисления с плавающей запятой, и мы хотим сообщить порту (см. главу 15) сохранить регистры с плавающей запятой во время переключения контекста.

В Листинге 1.22 указан фактический код задачи.

**Листинг 1.22 Пример № 4, TEST.C, Task()**

**void**  Task (**void** \*pdata)

{

    FP32   x;

    FP32   y;

    FP32   angle;

    FP32   radians;

**char**   s[81];

    INT8U  ypos;

    ypos  = \*(INT8U \*)pdata + 7;

    angle = (FP32)(\*(INT8U \*)pdata) \* (FP32)36.0;                           (1)

**for** (;;) {

        radians = (FP32)2.0 \* (FP32)3.141592 \* angle / (FP32)360.0;         (2)

        x       = **cos**(radians);

        y       = **sin**(radians);

**sprintf**(s, "   %2d       %8.3f  %8.3f     %8.3f",

                \*(INT8U \*)pdata, angle, x, y);

        PC\_DispStr(0, ypos, s, DISP\_FGND\_BLACK + DISP\_BGND\_LIGHT\_GRAY);

**if** (angle >= (FP32)360.0) {

            angle  =   (FP32)0.0;

        } **else** {

            angle +=   (FP32)0.01;

        }

        OSTimeDly(1);                                                       (3)

    }

}

(1) Аргумент pdata указывает на 8-битовое целое число, содержащее приоритет задачи. Чтобы каждая задача рассчитывала разные углы (не то, чтобы это действительно имело значение), я решил компенсировать каждую задачу на 36 градусов.

(2) sin() и cos() предполагают радианы вместо градусов и, таким образом, преобразование.

(3) Каждая задача задерживается на 1 такт (т.е. 50 миллисекунд) и, таким образом, каждая задача будет выполняться 200 раз в секунду.

Кроме указания OS\_TASK\_OPT\_SAVE\_FP в TaskStartCreateTasks() , вы не можете сказать, что из TEST.C мы используем порт, отличный от других примеров. На самом деле, может быть хорошей идеей всегда указывать опцию OS\_TASK\_OPT\_SAVE\_FP при создании задачи (с помощью OSTaskCreateExt() ), и, если порт поддерживает аппаратное обеспечение с плавающей запятой, для того, чтобы сохранить и извлечь регистры с плавающей запятой во время переключения контекста, будет предприняты необходимые шаги. На самом деле, это одна из красот uC/OS-II: портируемость ваших приложений на разных процессорах.

Чтобы использовать другой порт (по крайней мере, для 80x86), необходимо изменить только следующие файлы:

INCLUDES.H (в каталоге SOURCE ):

Вместо включения:

\software\ucos-ii\**ix86l**\bc45\os\_cpu.h

просто необходимо «указать» на другой каталог:

\software\ucos-ii\**ix86l-fp**\bc45\os\_cpu.h

TEST.LNK (в каталоге SOURCE ):

Файл линковщика включает библиотеку эмуляции с плавающей запятой в версии без плавающей запятой:

C:\BC45\LIB\EMU.LIB

и на аппаратную библиотеку с плавающей запятой необходимо ссылаться для кода, который использует FPU:

C:\BC45\LIB\FP87.LIB

TEST.MAK (в каталоге TEST ):

Каталог порта изменяется с:

PORT=\SOFTWARE\uCOS-II\Ix86L\BC45

на:

PORT=\SOFTWARE\uCOS-II\Ix86L-FP\BC45

Флаги компилятора в макросе C\_FLAGS включают -f287 для версии кода с плавающей запятой и пропускают его в версии без плавающей запятой.