Введение..2

Общие принципы построения ОСРВ...5

Процессы, потоки, задачи µC/OS-II...5

Состояние задачи..7

Блок контроля задач (OS_TCBs)...9

Лист готовности (Ready List)..12

Планировщик задач...13

Создание задач..15

Стеки задач...17

Портирование µC/OS-II...17

OS_CPU.H, зависимые от компилятора типы данных..................................18

OS_ENTER_CRITICAL() и OS_EXIT_CRITICAL()...19

OS_STK_GROWTH..20

OS_CPU_A.ASM...21

OS_CPU_C.C...21

OSTaskStkInit()..21

OSTaskCreateHook()..22

OSTaskDelHook()..23

OSTaskSwHook()..23

OSTaskStatHook()..23

OSTimeTickHook()...24

Введение

uC/OS-II - это операционная система(ядро) реального времени, специально разработанная для встроенных (embedded) систем.

Основные характеристики:

- по специальной лицензии распространяется с исходными кодами;
- портирована на множество систем (процессоров, микроконтроллеров и т.д.);
- масштабируемость и расширяемость - можно использовать только те функции ядра, которые необходимы путем объявления соответствующих #define. Это позволяет сократить объем используемой памяти данных и памяти программы. Например, есть пример расчета ресурсов (Excel) для 80x86, Real Mode, Large Model. Уменьшая, вплоть до нуля, количество ресурсов и "ненужных" функций можно существенно сократить требования к железу;
- fully-preemptive real-time kernel;
- детерминированное время запуска функций ядра. За исключением одной функции, время запуска всех сервисов ядра можно просчитать и оно не зависит от количества задач;
- отдельный стек для каждой задачи;
- множество функций: работа с очередями, семафорами, временем, выделением памяти и так далее;
- управление прерыванием: по результатам работы прерывания возможно приостановка выполнения текущей задачи и запуск более приоритетной.

Недостатки (относительные, без этого можно обойтись):

отсутствие так называемого, time slicing или round-robin планировщика (когда две задачи имеют одинаковый приоритет и процессорное время распределяется между ними определёнными квантами времени).

В Таблице 1 представлены сравнительные характеристики uC/OS-II и других ОСРВ.

uC/OS-II поддерживает значительно большее число процессоров по сравнению с другими, причем как с RISC архитектурой (MIPS, ARM), так и с CISC архитектурой (Motorola, Zilog).

Процессор поддерживает μC/OS-II, если он удовлетворяет следующим основным требованиям:

1. Должен быть компилятор языка С для данного процессора, и этот компилятор должен понимать реентерабельный код

2. Возможно отменять и разрешать прерывания на С

3. Процессор должен поддерживать прерывания и предусматривать (обрабатывать) прерывания, происходящие через регулярные промежутки времени (обычно между 10 и 100 Гц).

4. Процессор должен поддерживать аппаратный стек, и иметь возможность сохранять значительное количество данных в нем (до нескольких Кб)

5. Процессор должен иметь команды для сохранения и загрузки указателя стека или другого регистра ЦП в стек или в память

На Рис.1 показана архитектура μC/OS-II и ее взаимодействие с аппаратурой.

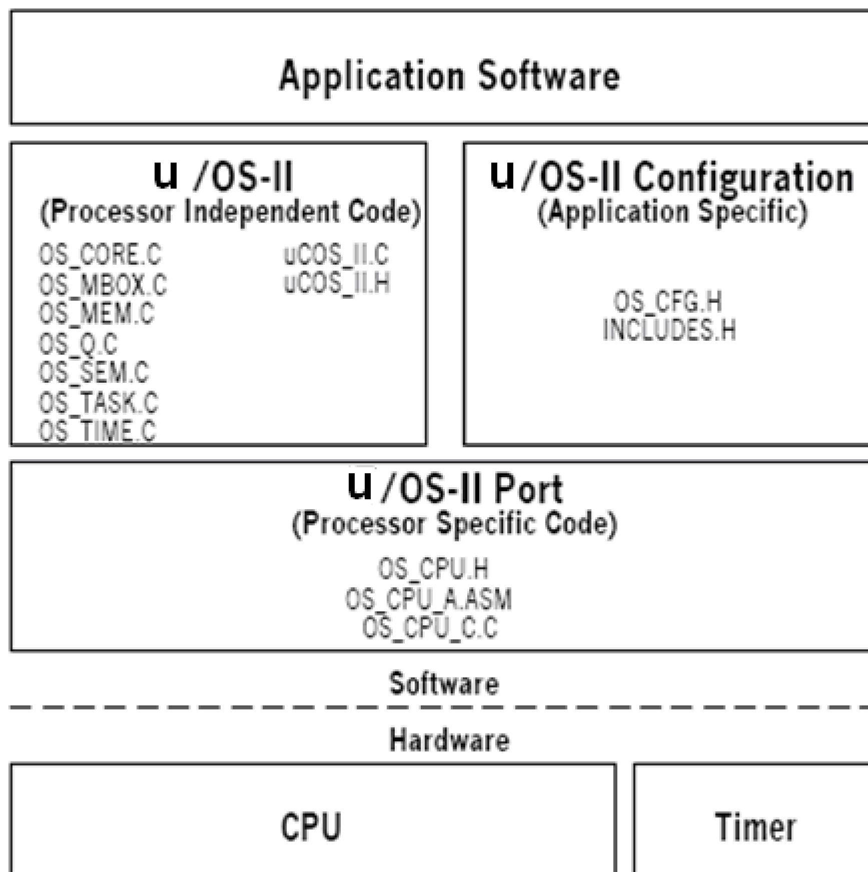

Рис.1 Архитектура μC/OS-II

Название ОСРВ	***Номер версии***	***Поддерживаемые процессоры***	***Поддерживаемые базовые ОС***	***Алгоритм планировщика***	***Число уровней приоритета***	***Минимальный объем ОЗУ***	***Минимальный объем ПЗУ***
Nucleus	1.12	386, 486, 80186, ARC, ARM, Atmel, AVR, Fujitsu SPARClite, Hitachi H8S, Intel i960, MIPS R3000, MIPS R4000, MIPS R5000, Mitsibishi M16C, Motorola 68000, Motorola 68HC11, Motorola ColdFire, PowerPC, RISCore3000, Siemens166 и др.	Win 95/98/NT	Fixed, round, time	256	2K	14K
OSE RealTime Operating Sysytm	4.11	8051, ARM, MIPS (R3000, R4000), Motorola 68000, Motorola ColdFire, Motorola M-CORE, PowerPC	Win 95/98/NT, Solaris, Unix	Fixed, round, time	32	20K Min, 200K Max	60K
Jbed	1.2x	ARM, Motorola68000, Motorola ColdFire, PowerPC	Win 95/98/NT	Fixed, round, time	10	10K	10K
OS-9	2.2	MIPS, Motorola, PowerPC, STMicroelectronix и др.	Win 95/98/NT	Fixed, round, time	65.536	128K	32K
QNX Neutrino Realtime OS	2.0	386, 486, MIPS (R4000, R5000, R7000), PowerPC	Win 95/98/NT, Linux, Solaris, Unix	round	63	Типично 128K, зависит от платформы	Типично 256K, зависит от платформы

Название ОСРВ	***Номер версии***	***Поддерживаемые процессоры***	***Поддерживаемые базовые ОС***	***Алгоритм планировщика***	***Число уровней приоритета***	***Минимальный объем ОЗУ (byte)***	***Минимальный объем ПЗУ (byte)***
MicroC/ OS-II	2,03	386, 486, 80186, ARC, ARM, Atmel, AVR, Fujitsu SPARClite, Hitachi H8S, Intel i960, MIPS R3000, MIPS R4000, MIPS	Win 95/98/NT	fixed	64	1000	3000

Общие принципы построения ОСРВ

Операционные системы реального времени (ОСРВ) предназначены для обеспечения интерфейса к ресурсам критических по времени систем реального времени. Основной задачей в таких системах является своевременность (timeliness) выполнения обработки данных.

В качестве основного требования к ОСРВ выдвигается требование обеспечения предсказуемости или детерминированности поведения системы в наихудших внешних условиях, что резко отличается от требований к производительности и быстродействию универсальных ОС. Хорошая ОСРВ имеет предсказуемое поведение при всех сценариях системной загрузки (одновременные прерывания и выполнение потоков).

Существует некое различие между системами реального времени и встроенными системами. От встроенной системы не всегда требуется, чтобы она имела предсказуемое поведение, и в таком случае она не является системой реального времени. Однако даже беглый взгляд на возможные встроенные системы позволяет утверждать, что большинство встроенных систем нуждается в предсказуемом поведении, по крайней мере, для некоторой функциональности, и таким образом, эти системы можно отнести к системам реального времени.

Как правило, большинство современных ОСРВ построено на основе микроядра (kernel или nucleus), которое обеспечивает планирование и диспетчеризацию задач, а также осуществляет их взаимодействие. Несмотря на сведение к минимуму в ядре абстракций ОС, микроядро все же должно иметь представление об абстракции процесса. Все остальные концептуальные абстракции операционных систем вынесены за пределы ядра, вызываются по запросу и выполняются как приложения.

Процессы, потоки, задачи µC/OS-II

Концепция многозадачности (псевдопараллелизм) является существенной для системы реального времени с одним процессором, приложения которой должны быть способны обрабатывать многочисленные внешние события, происходящие практически одновременно. Потоки существуют в одном контексте процесса, поэтому переключение между потоками происходит очень быстро, а вопросы безопасности не принимаются во внимание. Потоки являются легковесными, потому что их регистровый контекст меньше, т.е. их управляющие блоки намного компактнее.

Уменьшаются накладные расходы, вызванные сохранением и восстановлением управляющих блоков прерываемых потоков. Объем управляющих блоков зависит от конфигурации памяти. Если потоки выполняются в разных адресных пространствах, система должна поддерживать отображение памяти для каждого набора потоков.

Итак, в системах реального времени процесс распадается на задачи или потоки. В любом случае каждый процесс рассматривается как приложение. Между этими приложениями не должно быть слишком много взаимодействий, и в большинстве случаев они имеют различную природу – жесткого реального времени, мягкого реального времени, не реального времени.

Под задачами обычно понимается бесконечный цикл (как показано на листинге 1). Задача представляет собой функцию на языке С, имеющую аргументы, но не возвращающую значения. Возвращаемый тип обычно объявляется как void.

Листинг 1

```
void YourTask (void *pdata)                    (1)
{
   for (;;) {                                   (2)
      /* USER CODE */
      Call one of uC/OS-II's services:
      OSMboxPend();
      OSQPend();
      OSSemPend();
      OSTaskDel(OS PRIO SELF);
      OSTaskSuspend(OS PRIO SELF);
      OSTimeDly();
      OSTimeDlyHMSM();
      /* USER CODE */
   }
}
```

С другой стороны, задача может быть удалена по завершении (листинг 2). Отметим, что код задачи на самом деле не удаляется, µC/OS-II просто не знает о других задачах и, следовательно, о том, что код не будет выполняться. Также если задача вызывает **OSTaskDel(),** процедура ничего не возвращает.

Листинг 2

```
void YourTask (void *pdata)
{
   /* USER CODE */
   OSTaskDel(OS PRIO SELF);
}
```

Надо отметить, что аргументом является указатель типа void. Это позволяет приложению принимать любой тип данных: адрес переменной, структуру или даже адрес функции, если необходимо.

Возможно создание большого числа одинаковых задач, использующих одинаковые функции (тело задачи). Например, имеется 4 последовательных порта, каждый из которых управляется собственной задачей, причем коды этих задач одинаковы. Вместо копирования четырех раз кода, возможно создание задачи, которая будет получать в качестве аргумента указатель на структуру данных, которая определят параметры последовательного порта (частоту передачи, адрес порта ввода/вывода, номер векторов прерываний и т.д.).

µC/OS-II может управлять 64 задачами, однако для системных целей используется только 2 задачи.

Также для дальнейшего использования зарезервированы приоритеты *0, 1, 2, 3,* **OS_LOWEST_PRIO-3***,* **OS_LOWEST_PRIO-2***,* **OS_LOWEST_PRIO-1** и **OS_LOWEST_PRIO***.* **OS_LOWEST_PRIO** объявлен с помощью *#define* константой, которая определена в файле **OS_CFG.H**. Таким образом, существует 56 задач приложения. Каждой задаче должен быть присвоен приоритет от 0 до **OS_LOWEST_PRIO – 2***.* µC/OS-II всегда выполняет задачу с наивысшим приоритетом, готовую к исполнению.

В µC/OS-II задача создаются передачей ее адреса в качестве аргумента в одну из двух функций: **OSTaskCreate()** или **OSTaskCreateExt(). OSTaskCreateExt()** является расширенной версией **OSTaskCreate(***)*, но имеет некоторые особенности.

Состояние задачи

На Рис.2 показана диаграмма перемещения задачи для µC/OS-II. В определенные моменты времени задача может иметь одно из пяти состояний.

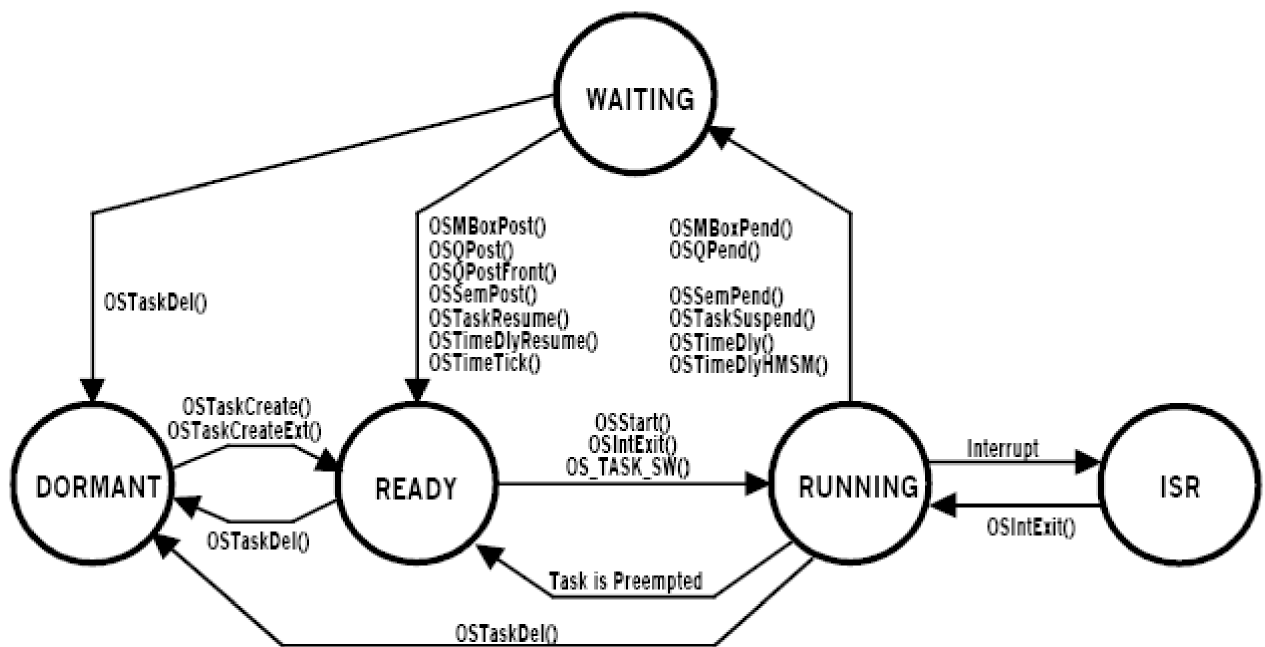

Рис. 2 Состояние задачи

DORMANT *(ожидающий)* состояние соответствует задачи, которая располагается в пространстве программы (ОЗУ или ПЗУ), но не доступная для µC/OS-II. Задача делается доступной для µC/OS-II после вызова функции **OSTaskCreate()** или **OSTaskCreateExt()**. Когда задача создана, она делается доступной для выполнения. Задача может быть создана перед стартом многозадачности или динамически при запуске задачи. Если после создания задача имеет приоритет выше, чем ее создатель, созданная задача немедленно передает управление ЦП. Задача может возвращать себя или другую задачу, с ожиданием вызова **OSTaskDel().**

Многозадачность начинается после вызова **OSStart(). OSStart()** начинает выполнять задачу с наивысшим приоритетом, которая имеет статус готовности **READY**. Эта задача перемещается в состояние выполнения **RUNNING**. Только одна задача может выполняться в это время. Готовая задача не будет выполняться, до тех пор, пока все задачи с высшими приоритетами не будут перемещены либо в состояние ожидания или не будут удалены.

Выполняемая задача может задерживаться не некоторое время при вызове процедур **OSTimeDly()** или **OSTimeDlyHMSM().** Эта задача находится в состоянии ожидания **WAITING** некоторое время и следующие высокоприоритетные задачи, готовые к выполнению, немедленно передают контроль ЦП. Задержанная задача переходит в

состояние готовности при вызове **OSTimeTick(),** когда необходимое время заканчивается.

Выполнение задачи может также требовать ожидание до тех пор, пока происходит событие. Это происходит при вызове функций **OSSemPend()**, **OSMboxPend()** или **OSQPend().** Задача при этом находится в состоянии ожидания **WAITING** при наступлении события. Когда задача ожидает выполнения события, задача со следущим высшим приоритетом немедленно передает управление ЦП. Задача переходит в состояние готовности когда событие происходит.

Выполнение задачи может всегда быть прервано, пока задача или µC/OS-II не отменит прерывания. Задача находится в состоянии ISR. Когда происходит прерывание, выполнение задачи временно останавливается, и ISR передает управление ЦП. ISR может делать одну или несколько задач готовыми к выполнению предупреждая одно или несколько событий. В этом случае, перед возвращением из ISR, µC/OS-II определяет если прерванная задача все еще с высшим приоритетом и готова к выполнению. Если задача с высшим приоритетом делается готовой к выполнению ISR, тогда новая задача с высоким приоритетом продолжается. С другой стороны, прерванная задача продолжается также. Когда все задачи либо ожидают события или времени окончания, µC/OS-II выполняет пустую задачу **OSTaskIdle().**

Блок контроля задач (OS_TCBs)

Когда задача создана, она присваивается Блоку контроля задач **OS_TCB** (Листинг 3). Блок задач – специальная структура данных, которая используется µC/OS-II для сохранения состояния задачи, когда она прервана. Когда задача восстанавливает управление, блок контроля задач позволяет возобновить выполнение задачи точно в том месте, где оно было прервано. Весь **OS_TCB** находится в ОЗУ, и инициализируется тогда, когда задача создана.

Листинг 3

```
typedef struct os_tcb {
    OS_STK        *OSTCBStkPtr;

#if OS_TASK_CREATE_EXT_EN
    void          *OSTCBExtPtr;
    OS_STK        *OSTCBStkBottom;
    INT32U         OSTCBStkSize;
    INT16U         OSTCBOpt;
    INT16U         OSTCBId;
#endif

    struct os_tcb *OSTCBNext;
    struct os_tcb *OSTCBPrev;

#if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN || OS_SEM_EN
    OS_EVENT      *OSTCBEventPtr;
#endif

#if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN
    void          *OSTCBMsg;
#endif

    INT16U         OSTCBDly;
    INT8U          OSTCBStat;
    INT8U          OSTCBPrio;

    INT8U          OSTCBX;
    INT8U          OSTCBY;
    INT8U          OSTCBBitX;
    INT8U          OSTCBBitY;

#if OS_TASK_DEL_EN
    BOOLEAN        OSTCBDelReq;
#endif
} OS_TCB;
```

OSTCBStkPtr содержит указатель на текущую вершину стека или задачи. μC/OS-II позволяет каждой задаче иметь свой собственный стек, причем каждая задача может иметь стек любого размера.

OSTCBStkPtr – это только область структуры данных **OS_TCB***,* доступная из языка ассемблера (из контекстно переключаемого кода). Перемещая **OSTCBStkPtr** в первое поле структуры, возможно обращаться к этому полю с помощью языка Ассемблера.

OSTCBStkSize – это переменная, которая содержит размер стека, причем не в байтах, а в словах. Это значит, что если стек содержит 1000 элементов, и каждый является 32-разрядным, то реальный размер стека будет 4000 байт.

OSTCBId – это переменная, содержащая идентификатор задачи. Это поле не используется и вводится для дальнейшего расширения.

OSTCBStat содержит состояние задачи. Когда **OSTCBStat** равен 0, задача готова к выполнению Другие значения могут быть назначены из значений, описанных в **uCOS_II.H**.

OSTCBPrio содержит приоритет задачи. Высший приоритет имеет значение **OSTCBPrio.**

OSTCBX, **OSTCBY**, **OSTCBBitX** и**OSTCBBitY** используются для ускорения процесса создания выполняемой задачи или задачи, ожидающей события. Значения этих полей вычисляются, когда задача создана или когда изменяется ее приоритет.

Листинг 4

```
OSTCBY      = priority >> 3;
OSTCBBitY   = OSMapTbl[priority >> 3];
OSTCBX      = priority & 0x07;
OSTCBBitX   = OSMapTbl[priority & 0x07];
```

OSTCBDelReq – это логическая переменная, используемая для индикации, удален или нет запрос задачи или сама задача.

OSTCBNext и **OSTCBPrev** используется для двойного связывания **OS_TCB.** Эта цепочка из **OS_TCB** используется **OSTimeTick()** для обновления полей **OSTCBDly** для каждой задачи. **OS_TCB** для каждой задачи связывается когда задача создана и удаляется когда удаляется задача. Двусвязный список используется для вставки или удаления элемента из цепочки.

Максимальное число задач (**OS_MAX_TASKS**), которое может иметь приложение, определено в **OS_CFG.H** и определяет число **OS_TCB**, размещаемое µC/OS-II для данного приложения. Все **OS_TCB** помещаются в массив **OSTCBTbl[ ]**.

При инициализации µC/OS-II все **OS_TCB** в этой таблице поступают в односвязный список свободных **OS_TCB** (как показано на Рис.3). Когда задача создана, **OS_TCB** указывает на **OSTCBFreeList**. Когда задача удалена, **OS_TCB** возвращает список свободных **OS_TCB.**

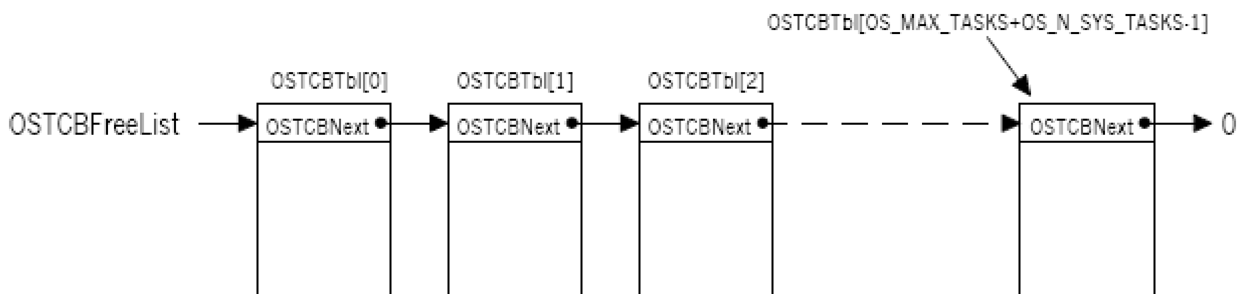

*Рис.3 Список свободных **OS_TCB***

Лист готовности (Ready List)

Каждой задаче ставится в соответствие уникальный уровень приоритета между 0 и **OS_LOWEST_PRIO** (см. **OS_CFG.H**). Задача с приоритетом **OS_LOWEST_PRIO** всегда определяется как невостребованная задача при инициализации ОС. Следует отметить, что **OS_MAX_TASKS** и **OS_LOWEST_PRIO** не связаны.

Каждая задача, готовая к выполнению, помещается в список готовности, состоящий из 2-х переменных: **OSRdyGrp** и **OSRdyTbl[].** Приоритеты задач группируются (по 8 задач в группу) в **OSRdyGrp.** Каждый бит в **OSRdyGrp** используется для обозначения того, что какая-либо из задач в группе готова к выполнению. Когда задача готова к выполнению, устанавливается соответствующий бит в таблице готовности **OSRdyTbl[]**. Размер **OSRdyTbl[]** зависит от **OS_LOWEST_PRIO** (см. **uCOS_II.H**). Это позволяет уменьшить количество ОЗУ, необходимого µC/OS-II в том случае, когда приложению требуется несколько уровней задач.

Для определения того, какой приоритет (или какая задача) будет выполняться следующей, планировщик определяет самый низкий номер приоритета, который имеет установленный бит в **OSRdyTbl[]**. Взаимосвязь между **OSRdyGrp** и **OSRdyTbl[]** показано на Рис.4 и следует следующим правилам:

Бит 0 in **OSRdyGrp** равен 1 когда какой-либо бит в **OSRdyTbl[0]** равен 1.

Бит 1 in **OSRdyGrp** равен 1 когда какой-либо бит в **OSRdyTbl[1]** равен 1.

Бит 2 in **OSRdyGrp** равен 1 когда какой-либо бит в **OSRdyTbl[2]** равен 1.

Бит 3 in **OSRdyGrp** равен 1 когда какой-либо бит в **OSRdyTbl[3]** равен 1.

Бит 4 in **OSRdyGrp** равен 1 когда какой-либо бит в **OSRdyTbl[4]** равен 1.

Бит 5 in **OSRdyGrp** равен 1 когда какой-либо бит в **OSRdyTbl[5]** равен 1.

Бит 6 in **OSRdyGrp** равен 1 когда какой-либо бит в **OSRdyTbl[6]** равен 1.

Бит 7 in **OSRdyGrp** равен 1 когда какой-либо бит в **OSRdyTbl[7]** равен 1.

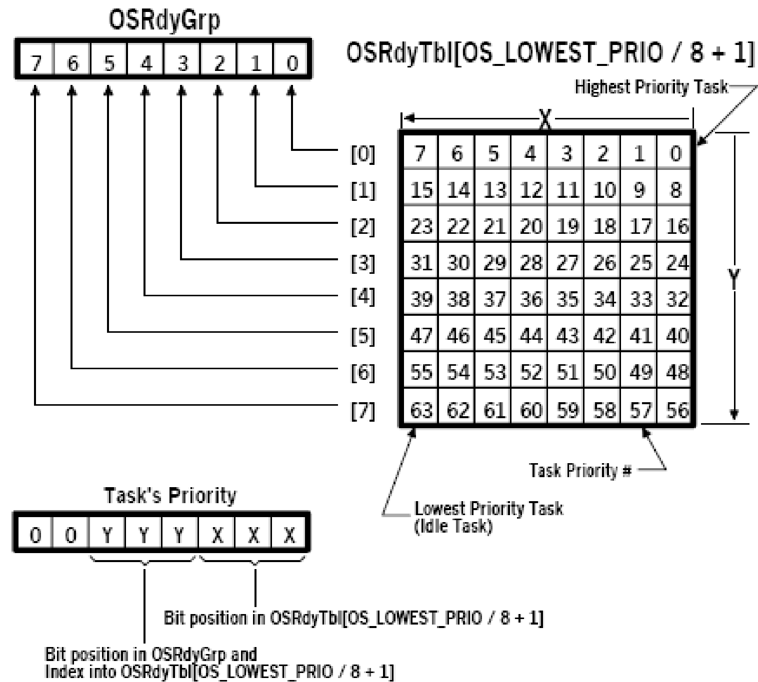

*Рис.4 Лист готовности **µC/OS-II***

Приведенный ниже код показывает, как поместить задачу в лист готовности.

Листинг 5

```
    OSRdyGrp            |= OSMapTbl[prio >> 3];
    OSRdyTbl[prio >> 3] |= OSMapTbl[prio & 0x07];
```

Удаление задачи из списка показано в листинге 6:

Листинг 6

```
if ((OSRdyTbl[prio >> 3] &= ~OSMapTbl[prio & 0x07]) == 0)
    OSRdyGrp &= ~OSMapTbl[prio >> 3];
```

Планировщик задач

µC/OS-II всегда выполняет задачу с наивысшим приоритетом, готовую к выполнению. Определение того, какая из задач имеет наивысший приоритет, и, таким образом, будет исполняться следующей, определяется планировщиком. Планирование уровня задач выполняется **OSSched().** Стандартная программа обслуживания прерываний (ISR) вызывается другой функцией **OSIntExit().** Код **OSSched()** показан на листинге 7.

Листинг 7

```
void OSSched (void)
{
    INT8U y;


    OS ENTER CRITICAL();
    if ((OSLockNesting | OSIntNesting) == 0) {                       (1)
        y             = OSUnMapTbl[OSRdyGrp];                        (2)
        OSPrioHighRdy = (INT8U)((y << 3) + OSUnMapTbl[OSRdyTbl[y]]); (2)
        if (OSPrioHighRdy != OSPrioCur) {                            (3)
            OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];              (4)
            OSCtxSwCtr++;                                            (5)
            OS TASK SW();                                            (6)
        }
    }
    OS EXIT CRITICAL();
}
```

Время планирования задач в µC/OS-II не зависит от числа задач в приложении. **OSSched()** заканчивает работу если вызвано из ISR ранее (другими словами, **OSLockNesting > 0**). Если **OSSched()** не вызвана из ISR и планировщик выполняется, то **OSSched()** определяет наивысший уровень приоритета задачи, готовой к выполнению. Задача, готовая к выполнению, имеет соответствующий бит, установленный в **OSRdyTbl[]**. Если задача с наивысшим приоритетом не найдена, **OSSched()** проверяет, что задача с наивысшим приоритетом не является текущей задачей.

Для выполнения контекстного переключателя, необходимо определить поле **OSTCBHighRdy** в **OS_TCB** задачи с наивысшим приоритетом, что осуществляется индексированием **OSTCBPrioTbl[]** используя **OSPrioHighRdy. OS_TASK_SW()** осуществляет реальное контекстное переключение.

Контекстное переключение состоит из сохранения регистров процессора в стеке (когда задача приостановлена) и восстановления регистров задачи с наивысшим приоритетом из стека. В µC/OS-II стековый фрейм для выполняемой задачи аналогичен прерыванию, если регистры процессора были сохранены. Другими словами, что требуется µC/OS-II для выполнения задачи, это восстановить регистры из стека задачи и выполнить возврат из прерывания. Для переключения контекста необходимо вызвать **OS_TASK_SW(),** при этом происходит эмуляция прерывания. Большинство процессоров обеспечивают либо программное, либо системное прерывание для выполнения этого. Программа обслуживания прерывания (ISR) или

обработчик системных прерываний должны иметь свой вектор в ассемблерной функции **OSCtxSw()**.

Следует отметить, что код **OSSched()** должен быть написан на языке Ассемблера для уменьшения времени планирования, однако он был написан на языке С для читабельности, портируемости, а также для минимизации ассемблерного кода.

Создание задач

Задача представляет собой бесконечный цикл или функцию, которая удаляется при завершении работы (на самом деле удаления не происходит, ОС просто ничего не знает о задаче и она не выполняется). Задача представляет собой функцию на языке С. Задача должна представлять функцию, представленную на листинге 8.

Листинг 8

```
void YourTask (void *pdata)
{
   for (;;) {
      /* USER CODE */
      Call one of uC/OS-II's services:
      OSMboxPend();
      OSQPend();
      OSSemPend();
      OSTaskDel(OS_PRIO_SELF);
      OSTaskSuspend(OS_PRIO_SELF);
      OSTimeDly();
      OSTimeDlyHMSM();
      /* USER CODE */
   }
}
```

Задача создается при передаче ее адреса вместе с другими аргументами в одну из 2-х функций: **OSTaskCreate()** или **OSTaskCreateExt()** (**OSTaskCreateExt()** - расширенная версия **OSTaskCreate()**). Задача может быть создана при старте многозадачности или другой задачей. Необходимо создать по крайней мере одну задачу перед стартом многозадачности. ISR не может сама создать задачу.

Код **OSTaskCreate()** показан на листинге 9. Эта функция требует 4 аргумента:

task – это указатель на код задачи;

pdata – это аргумент, передаваемый в задачу при начале выполнения;

ptos – это указатель на вершину стека, соотнесенного с задачей;

prio – желательный приоритет задачи.

Листинг 9

```
INT8U OSTaskCreate (void (*task)(void *pd), void *pdata, OS_STK *ptos, INT8U prio)
{
    void   *psp;
    INT8U   err;


    if (prio > OS_LOWEST_PRIO) {                                         (1)
        return (OS_PRIO_INVALID);
    }
    OS_ENTER_CRITICAL();
    if (OSTCBPrioTbl[prio] == (OS_TCB *)0) {                             (2)
        OSTCBPrioTbl[prio] = (OS_TCB *)1;                                (3)
        OS_EXIT_CRITICAL();                                              (4)
        psp = (void *)OSTaskStkInit(task, pdata, ptos, 0);               (5)
        err = OSTCBInit(prio, psp, (void *)0, 0, 0, (void *)0, 0);       (6)
        if (err == OS_NO_ERR) {                                          (7)
            OS_ENTER_CRITICAL();
            OSTaskCtr++;                                                 (8)
            OSTaskCreateHook(OSTCBPrioTbl[prio]);                        (9)
            OS_EXIT_CRITICAL();
            if (OSRunning) {                                            (10)
                OSSched();                                              (11)
            }
        } else {
            OSTCBPrioTbl[prio] = (OS_TCB *)0;                           (12)
        }
        return (err);
    } else {
        OS_EXIT_CRITICAL();
        return (OS_PRIO_EXIST);
    }
}
```

OSTaskCreate() начинает выполняться с проверки того, какой приоритет задачи является верным. Приоритет задачи может быть между 0 и LOWEST_PRIO. Далее **OSTaskCreate()** проверяет, что задача уже создана с нужным приоритетом.

Далее **OSTaskCreate()** вызывает **OSTaskStkInit()**, отвечающую за инициализацию стека задач. Эта функция зависит от процессора и поэтому располагается в файле **OS_CPU_C.C.**

µC/OS-II поддерживает процессоры, имеющие стек, который растет от от младших адресов к старшим (или наоборот). При вызове OSTaskCreate() необходимо точно

знать, как растет стек (см. в **OS_CPU.H** переменную **OS_STACK_GROWTH**) в используемом процессоре.

Стеки задач

Каждая задача должна иметь свой собственный стек. Стек должен иметь тип **OS_STK** и располагаться в соседних участках памяти. Размер стека можно определять статически (во время компиляции) или динамически. Статическое размещение стека показано на листинге 10.

Листинг 10

```
static OS_STK  MyTaskStack[stack_size];
```

Динамически стек размещается при помощи функции языка C malloc() (листинг 11). Однако, надо следить за фрагментацией.

Листинг 11

```
OS_STK  *pstk;


pstk = (OS_STK *)malloc(stack_size);
if (pstk != (OS_STK *)0) {            /* Make sure malloc() had enough space */
    Create the task;
}
```

Портирование µC/OS-II

Портирование µC/OS-II на различные процессоры требует детальной разборки архитектуры процессора и используемого C компилятора. Портирование µC/OS-II состоит в:

- установке значения одной константы (**OS_CPU.H**);
- объявление 10 типов данных (**OS_CPU.H**);
- объявление 3 **#define** макросов (**OS_CPU.H**);
- написание 6 простых функций на C (**OS_CPU_C.C**);
- написание 4 функций на языке ассемблера (**OS_CPU_A.ASM**).

В зависимости от процессора, код портирования содержит от 50 до 300 строк кода! Портирование µC/OS-II может занять от нескольких часов до недели.

OS_CPU.H, зависимые от компилятора типы данных

Поскольку различные микропроцессоры имеют различную длину слова, портация μC/OS-II включает ряд определений типов для гарантии переносимости. В особенности, коды μC/OS-II никогда не используют типы данных С **short**, **int** и **long**, потому что они не переносимые. Вместо этого, определены целочисленные типы данных, которые и переносимы, и интуитивно понятны (см. листинг 12). Также, для удобства, включены типы данных с плавающей точкой (см. листинг 12), даже при том, что μC/OS-II не использует такие числа.

Тип данных **INT16U**, к примеру, всегда представляет 16-битное целое число без знака. μC/OS-II и приложение может теперь предполагать, что диапазон значений для переменных этого типа — от 0 до 65535. В μC/OS-II, портированной на 32-битный процессор, надо полагать, что **INT16U** на самом деле объявлен как **unsigned short** вместо **unsigned int**. Там, где в этом заинтересована μC/OS-II, однако, всё равно будет работать с **INT16U**.

Необходимо сообщить μC/OS-II тип данных стека задач. Это делается определением соответствующего типа данных С для **OS_STK**. Если элементы стека на вашем процессоре 32-битные и в документации вашего компилятора указано, что **int** имеет 32-бита, вы можете определить **OS_STK** как тип **unsigned int**. Все стеки задач должны быть объявлены с использованием **OS_STK** как типом данных.

Листинг 12

```
#ifdef  OS CPU GLOBALS
#define OS CPU EXT
#else
#define OS CPU EXT  extern
#endif

/*
*********************************************************************************************************
*                                              DATA TYPES
*                                         (Compiler Specific)
*********************************************************************************************************
*/

typedef unsigned char  BOOLEAN;
typedef unsigned char  INT8U;                    /* Unsigned  8 bit quantity                           */  (1)
typedef signed   char  INT8S;                    /* Signed    8 bit quantity                           */
typedef unsigned int   INT16U;                   /* Unsigned 16 bit quantity                           */
typedef signed   int   INT16S;                   /* Signed   16 bit quantity                           */
typedef unsigned long  INT32U;                   /* Unsigned 32 bit quantity                           */
typedef signed   long  INT32S;                   /* Signed   32 bit quantity                           */
typedef float          FP32;                     /* Single precision floating point                    */  (2)
typedef double         FP64;                     /* Double precision floating point                    */

typedef unsigned int   OS STK;                   /* Each stack entry is 16-bit wide                    */

/*
*********************************************************************************************************
*                                          Processor Specifics
*********************************************************************************************************
*/

#define  OS ENTER CRITICAL()  ???                /* Disable interrupts                                 */  (3)
#define  OS EXIT CRITICAL()   ???                /* Enable  interrupts                                 */

#define  OS STK GROWTH        1                  /* Define stack growth: 1 = Down, 0 = Up              */  (4)

#define  OS TASK SW()         ???                                                                          (5)
```

Всё, что необходимо сделать, это почитать руководство к компилятору и найти стандартные типы данных, соответствующие типам, ожидаемым µC/OS-II.

OS_ENTER_CRITICAL() и OS_EXIT_CRITICAL()

µC/OS-II, подобно всем ядрам реального времени, должна отключать прерывания, когда обращается к критическим участкам кода, и затем включать прерывания по завершении. Это позволяет µC/OS-II защищать критический код от одновременного входа в него из нескольких задач или программ обработки прерываний. Время, на которое запрещены прерывания, является одной из наиболее важных характеристик, которые должен предоставлять продавец ядра, так как это время оказывает непосредственное влияние на скорость реакции ядра на события реального времени.
µC/OS-II пытается удержать длительность отключения прерываний по минимуму, но в µC/OS-II длительность отключения прерываний в большей степени определяется архитектурой процессора и качеством генерации кода компилятором. Каждый процессор имеет инструкции для разрешения/запрета прерываний и C компилятор должен иметь механизм проведения этих операций непосредственно из C. Некоторые компиляторы позволяют вставить ассемблерные команды в исходник на C. Это значительно облегчит вставку команды разрешения/запрета прерываний. Другие компиляторы будут, фактически, содержать языковые расширения для

разрешения/запрета прерываний непосредственно из C. Для скрытия метода выполнения этих операций, выбранного производителем компилятора, в µC/OS-II определены два макроса для запрета и разрешения прерываний: **OS_ENTER_CRITICAL()** и **OS_EXIT_CRITICAL()**, см. листинг 13.

Критические секции µC/OS-II заключены в **OS_ENTER_CRITICAL()** и **OS_EXIT_CRITICAL()**, как показано ниже:

Листинг 13

```
µC/OS-II Service Function
{
    OS ENTER CRITICAL();
    /* µC/OS-II critical code section */
    OS EXIT CRITICAL();
}
```

Первый и самый простой способ осуществлять эти две макрокоманды состоит в том, чтобы вызвать команду процессора для отключения прерываний по **OS_ENTER_CRITICAL()** и команду разрешения прерываний по **OS_EXIT_CRITICAL()**. Есть, однако, небольшая проблема с этим сценарием. Если вызвана µC/OS-II функцию с отключенными прерываниями, тогда, после возврата из µC/OS-II, прерывания бы разрешились!

Второй способ осуществить **OS_ENTER_CRITICAL()** состоит в том, чтобы сохранить состояние отключенного прерывания в стек, и затем отключить прерывания. **OS_EXIT_CRITICAL()** было бы просто реализовать восстановлением состояния прерывания из стека. При использовании этой схемы, если вызвана сервисная функция µC/OS-II с разрешенными или запрещёнными прерываниями, тогда это состояние было бы защищено на время вызова. Если вызывается µC/OS-II сервис с заблокированными прерываниями, потенциально продлевается латентность прерываний приложения. Приложение может использовать **OS_ENTER_CRITICAL()** и **OS_EXIT_CRITICAL()** чтобы также защитить критические участки кода. Следует отметить, что приложение зависнет, если прерывания перед вызовом сервиса типа **OSTimeDly()** запрещены. Это произойдёт, потому что задача будет приостановлена на некоторый интервал времени, но так как прерывания запрещены, таймер никогда не сможет отсчитать нужное.

OS_STK_GROWTH

Стек большинства микропроцессоров и микроконтроллеров растёт в памяти с верху вниз. Есть, однако, некоторые процессоры, работающие по-другому. µC/OS-II была

разработана для любых разновидностей процессоров. Это обеспечивается тем, что необходимо указать µC/OS-II, в каком направлении растёт стек, изменяя значение конфигурационной константы **OS_STK_GROWTH** (листинг 14):
Необходимо установить **OS_STK_GROWTH** в 0 если стек растёт снизу вверх, или в 1, если сверху вниз.

OS_CPU_A.ASM

Для портации µC/OS-II требуется написать четыре довольно простые функции на языке ассемблера:

OSStartHighRdy()
OSCtxSw()
OSIntCtxSw()
OSTickISR()

Если компилятор поддерживает встраивание кода ассемблера, фактически, можно расположить все процессорно-зависимые коды в **OS_CPU_C.C** вместо отдельного ассемблерного файла.

OS_CPU_C.C

Портация µC/OS-II требует, чтобы были написаны шесть довольно простых функций на C:

OSTaskStkInit()
OSTaskCreateHook()
OSTaskDelHook()
OSTaskSwHook()
OSTaskStatHook()
OSTimeTickHook()

Единственная функция, которая действительно необходима — **OSTaskStkInit()**. Остальные пять функций ДОЛЖНЫ быть объявлены, но не обязаны содержать код.

OSTaskStkInit()
Эта функция вызывается из **OSTaskCreate()** и **OSTaskCreateExt()** для инициализации фрейма стека задачи так, чтобы стек выглядел как будто прерывание только что произошло и все регистры процессора были помещены в стек. Рис. 5 показывает, что **OSTaskStkInit()** помещает в стек создаваемой задачи. Отметьте, что на Рис.5 принят

стек растущий вниз. Дальнейшее обсуждение также относится и к стеку, растущему вверх.

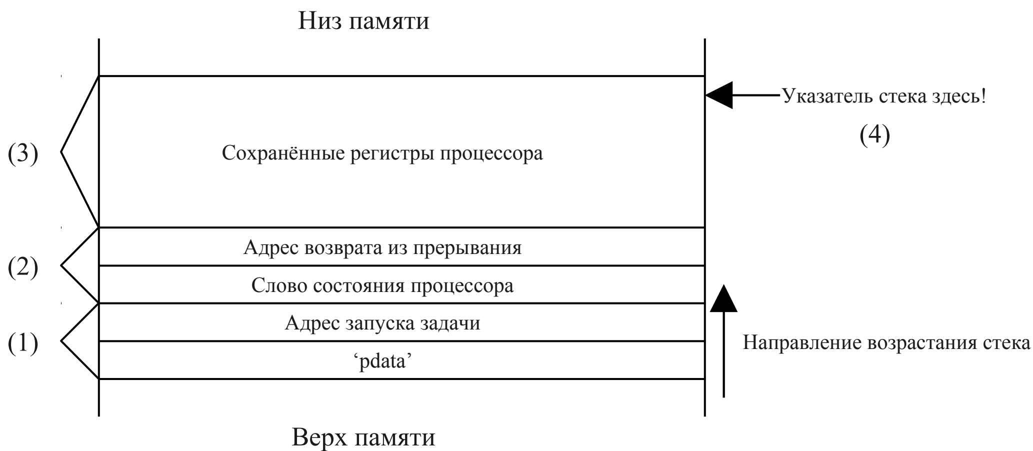

Рис.5 Инициализация фрейма стека параметром 'pdata', передаваемым в стек.

При создании задачи, в **OSTaskCreate()** или **OSTaskCreateExt()** определяется начальный адрес задачи, передается указатель **pdata**, верхняя граница стека задачи и приоритет задачи. **OSTaskCreate()** требует дополнительных аргументов. Для правильной инициализации фрейма стека, **OSTaskStkInit()** требует только три первых аргумента.

OSTaskCreateHook()

OSTaskCreateHook() вызывается всякий раз, когда создаётся задача через **OSTaskCreate()** или **OSTaskCreateExt()**. Это позволяет расширять функциональность µC/OS-II. **OSTaskCreateHook()** вызывается когда µC/OS-II завершает настройку внутренней структуры, но до вызова планировщика. Прерывания блокируются на время работы этой функции, поэтому надо иметь как можно меньше кода в этой функции, так как это непосредственно влияет на латентность прерываний.

При вызове, **OSTaskCreateHook()** принимает указатель на **OS_TCB** созданной задачи и т.о. может обращаться ко всем элементам структуры. **OSTaskCreateHook()** имеет ограниченные возможности, когда задача создаётся через **OSTaskCreate()**. Однако, с **OSTaskCreateExt()** можно получить доступ к указателю на расширение TCB (**OSTCBExtPtr**) в структуре **OS_TCB**, который может быть использован для доступа к дополнительной информации о задаче, такой как содержимое регистров с плавающей

точкой, MMU (МУП — модуль управления памятью) регистрам, счётчику задач, отладочной информации, и т.п.

OSTaskDelHook()

OSTaskDelHook() вызывается при каждом удалении задачи. **OSTaskDelHook()** вызывается перед отделением задачи от внутреннего списка активных задачи µC/OS-II. При вызове **OSTaskDelHook()** получает указатель на **OS_TCB** подлежащей удалению задачи и, таким образом, можно получить доступ ко всем элементам структуры. **OSTaskDelHook()** может проверить, был ли создан расширенный TCB (по ненулевому указателю). Таким образом **OSTaskDelHook()** отвечает за операцию очистки. От функции **OSTaskDelHook()** не ожидается возврат какого-либо результата.
Код **OSTaskDelHook()** генерируется, только если **OS_CPU_HOOKS_EN** установлено в 1 в файле **OS_CFG.H**.

OSTaskSwHook()

OSTaskSwHook() вызывается при каждом переключении задач. Это происходит, если переключение задач вызвано функциями **OSCtxSw()** или **OSIntCtxSw()**. **OSTaskSwHook()** имеет прямой доступ к **OSTCBCur** и **OSTCBHighRdy**, потому что это глобальные переменные. Конечно, **OSTCBCur** указывает на **OS_TCB** задачи, из которой система переключается, а **OSTCBHighRdy** указывает на **OS_TCB** задачи, в которую переключается. Следует отметить, что прерывания всегда запрещаются при вызове **OSTaskSwHook()** и, таким образом, необходимо иметь как можно меньше дополнительного кода, так как от этого зависит латентность прерываний. **OSTaskSwHook()** не имеет аргументов и ничего не возвращает.

Код для **OSTaskSwHook()** генерируется, только если **OS_CPU_HOOKS_EN** установлено в 1 в файле **OS_CFG.H**.

OSTaskStatHook()

OSTaskStatHook() вызывется раз в секунду из **OSTaskStat()**. С помощью **OSTaskStatHook()** можно расширить возможности статистики. Например, можно контролировать и отображать время выполнения каждой задачи, процент загрузки