# *Портирование микроСи/ОС-II*

Эта глава описывает основные понятия, которые нужно знать для адаптации микроСи/ОС-II для различных процессоров. Адаптация ядра реального времени на микропроцессор или микроконтроллер называется портированием. Большая часть микроСи/ОС-II написана на С для совместимости, однако всё же необходимо писать некоторые процессорно-зависимые коды на С или языке ассемблера. Портирование микроСи/ОС-II на различные процессоры относительно легко, так как микроСи/ОС-II была разработана легко переносимой. Если вы уже имеете портированную микроСи/ОС-II на процессор, который вы хотите использовать, вам не нужно читать эту главу, если конечно, вы не хотите знать, как работают процессорно-зависимые участки микроСи/ОС-II.

На процессоре можно запустить микроСи/ОС-II, если он удовлетворяет следующим основным требованиям:

1. Вы должны иметь компилятор С для выбранного процессора и компилятор должен понимать реентерабельный код.
2. Вы должны иметь возможность включать и выключать прерывания из С.
3. Процессор должен поддерживать прерывания и вам нужно обеспечить регулярно возникающее прерывание (с частотой примерно 10…100 Гц).
4. Процессор должен аппаратно поддерживать стек, и процессор должен иметь возможность сохранять в стеке большие объёмы данных (возможно несколько КБ).
5. Процессор должен иметь команды загрузки и сохранения указателя стека и других регистров в стеке или в памяти.

Процессоры, подобные серии Motorola 6805, не удовлетворяют требованиям #4 и #5, так что микроСи/ОС-II не может быть запущена на таких процессорах.

(Какой-то рисунок или схема)

В зависимости от процессора, Портирование может состоять из написания или изменения от 50 до 300 строк кода! Портирование микроСи/ОС-II может занять вас где-то этак на несколько часов, ну, в крайнем случае, на неделю.

Когда вы получите портированную микроСи/ОС-II на ваш процессор, вам понадобится проверить эту операцию. Тестирование такого многозадачного ядра реально времени, как микроСи/ОС-II, не так просто как вы можете подумать. Вам нужно тестировать портацию без кода самого приложения. Другими словами, тестируйте работоспособность ядра самого по себе. На это есть две причины. Во-первых, вы не захотите усложнять вещи более, чем это нужно. Во-вторых, если что-то не заработает, вы знаете, что ошибка в портации, а не в вашем приложении. Начинайте с пары простых задач и только сервисной процедуры прерывания по таймеру. Когда получите многозадачный режим, будет легко добавить к вашему приложению другие задачи.

## *8.00 Инструменты разработки*

Как замечено ранее, вам нужен компилятор С для процессора, на который вы хотите портировать микроСи/ОС-II. Так как микроСи/ОС-II это вытесняющее ядро, вы можете использовать только С компилятор, который генерирует реентерабельный код. Ваш С компилятор должен также поддерживать программирование на ассемблере. Большинство компиляторов, разработанных для внедряемых систем, будут, фактически, также содержать ассемблер, сборщик и локатор. Сборщик используется для объединения объектных файлов (откомпилированные и собранные файлы) из различных модулей, в то время как локатор позволит вам расположить код и данные где-нибудь на карте памяти целевого процессора. Ваш С компилятор также должен обеспечивать механизм запрета и разрешения прерываний из С. Некоторые компиляторы позволят вам вставлять ассемблерные выражения прямо в исходный текст на С. Это облегчит вставку соответствующих процессорных команд для разрешения и запрета прерываний. Другие компиляторы будут содержать расширения языка, позволяющие разрешать и запрещать прерывания прямо из С.

## *8.00 Каталоги и файлы*

Установочная программа, представленная на дистрибутивной дискете установит микроСи/ОС-II и портацию для Intel 80x86 (реальный режим, модель Large) на ваш жёсткий диск. Я создал непротиворечивую структуру каталогов, чтобы позволить вам легко искать файлы для желаемого целевого процессора. Если вы добавляете порт для другого процессора, вы должны придерживаться тех же самых соглашений.
Все портации располагаются в директории \SOFTWARE\uCOS-II на вашем жёстком диске. Исходные тексты для каждой портации на микропроцессор или микроконтроллер ДОЛЖНЫ быть найдены всего в двух или трёх файлах: **OS_CPU.H**, **OS_CPU_C.C** и, не обязательно, **OS_CPU_A.ASM**. Файл с ассемблерным текстом не обязателен, потому что некоторые компиляторы позволят вам иметь ассемблерные вставки в тексте на С и, таким образом, вы можете располагать нужный код на ассемблере непосредственно в OS_CPU_C.C. Каталог, в котором расположена портация, определяется тем, какой процессор вы используете. Ниже приведены примеры каталогов, где вы бы могли хранить различные портации. Отметьте, что все файлы имеют одинаковые имена, хотя предназначены для совершенно разных целевых процессоров.

Intel/AMD 80186: **\SOFTWARE\uCOS-II\Ix86S:**
**OS_CPU.H**
**OS_CPU_A.ASM**
**OS_CPU_C.C**
**\SOFTWARE\uCOS-II\Ix86L:**
**OS_CPU.H**
**OS_CPU_A.ASM**
**OS_CPU_C.C**

Motorola 68HC11: **\SOFTWARE\uCOS-II\68HC11:**
**OS_CPU.H**
**OS_CPU_A.ASM**
**OS_CPU_C.C**

## *8.02 INCLUDES.H*

Как я упоминал в главе 1, **INCLUDES.H** это главный заголовочный файл и он находится вверху всех .C файлов в следующем виде:

```
#include "includes.h"
```

**INCLUDES.H** позволяет писать любой .C файл проекта, не беспокоясь о том, какие заголовочные файлы действительно нужны. Единственный недостаток главного заголовочного файла в том, что **INCLUDES.H** может включать заголовочные файлы, не нужные для компилируемого в данный момент .C файла. Это означает, что каждому файлу будет требоваться больше времени для компиляции. Это неудобство оправдывается переносимостью кода. Вы можете подправить **INCLUDES.H** для добавления своих собственных заголовочных файлов, но они должны добавляться в конец списка файлов.

## *8.03 OS_CPU.H*

OS_CPU.H содержит процессорно-зависимые специальные **#defines** константы, макросы и определения типов typedefs. Общий вид файла **OS_CPU.H** показан в листинге 8.1

```
#ifdef OS_CPU_GLOBALS
#define OS_CPU_EXT
#else
#define OS_CPU_EXT extern
#endif
/*
*********************************************************************************************
* ТИПЫ ДАННЫХ
* (зависят от компилятора)
*********************************************************************************************
*/
typedef unsigned char BOOLEAN;
typedef unsigned char INT8U; /* Беззнаковое 8-битное целое */ (1)
typedef signed char INT8S; /* 8-битное целое со знаком */
typedef unsigned int INT16U; /* Беззнаковое 16-битное целое */
typedef signed int INT16S; /* 16-битное целое со знаком */
typedef unsigned long INT32U; /* Беззнаковое 32-битное целое */
typedef signed long INT32S; /* 32-битное целое со знаком */
typedef float FP32; /* Число с плавающей точкой нормальной точности */ (2)
typedef double FP64; /* Число с плавающей точкой двойной точности */
typedef unsigned int OS_STK; /* Каждый элемент стека имеет длину 16 бит */
/*
*********************************************************************************************
* Определяемые процессором
*********************************************************************************************
*/
#define OS_ENTER_CRITICAL() ??? /* Выключение прерываний */ (3)
#define OS_EXIT_CRITICAL() ??? /* Включение прерываний */
#define OS_STK_GROWTH 1 /* Определение направления стека: 1 = Вниз, 0 = Вверх */ (4)
#define OS_TASK_SW() ??? (5)
```

**Листинг 8.1, Содержимое файла OS_CPU.H**

### *8.03.01 OS_CPU.H, зависимые от компилятора типы данных*

Поскольку различные микропроцессоры имеют различную длину слова, портация микроСи/ОС-II включает ряд определений типов для гарантии переносимости. В особенности, коды микроСи/ОС-II никогда не используют типы данных С **short**, **int** и **long**, потому что они не переносимые. Вместо этого, я определил целочисленные типы данных, которые и переносимы, и интуитивно понятны (см. листинг 8.1(1)). Также, для удобства, я включил типы данных с плавающей точкой (см. листинг8.1(2)), даже при том, что микроСи/ОС-II не использует такие числа.

Тип данных **INT16U**, к примеру, всегда представляет 16-битное целое число без знака. микроСи/ОС-II и ваше приложение может теперь предполагать, что диапазон значений для переменных этого типа — от 0 до 65535. В микроСи/ОС-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 микроСи/ОС-II, однако, всё равно будет работать с **INT16U**.

Вы должны сообщить микроСи/ОС-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.

Всё, что вам надо сделать, это почитать руководство к компилятору и найти стандартные типы данных, соответствующие типам, ожидаемым микроСи/ОС-II.

### *8.03.02 OS_CPU.H, OS_ENTER_CRITICAL() и OS_EXIT_CRITICAL()*

МикроСи/ОС-II, подобно всем ядрам реального времени, должна отключать прерывания, когда обращается к критическим участкам кода, и затем включать прерывания по завершении. Это позволяет микроСи/ОС-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.
МикроСи/ОС-II пытается удержать длительность отключения прерываний по минимуму, но в микроСи/ОС-II длительность отключения прерываний в большей степени определяется архитектурой процессора и качеством генерации кода компилятором. Каждый процессор имеет инструкции для разрешения/запрета прерываний и ваш С компилятор должен иметь механизм проведения этих операций непосредственно из С. Некоторые компиляторы позволят вам вставить ассемблерные команды в ваш исходник на С. Это значительно облегчит вставку команды разрешения/запрета прерываний. Другие компиляторы будут, фактически, содержать языковые расширения для разрешения/запрета прерываний непосредственно из С. Для скрытия метода выполнения этих операций, выбранного производителем компилятора, в микроСи/ОС-II определены два макроса для запрета и разрешения прерываний: **OS_ENTER_CRITICAL()** и **OS_EXIT_CRITICAL()**, см. листинг 8.1(3).

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

```
Сервисная функция микроСи/ОС-II
{
OS_ENTER_CRITICAL();
/* Участок критического кода микроСи/ОС-II */
OS_EXIT_CRITICAL();
}
```

**Метод #1:**
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 бы вы вызвали микроСи/ОС-II функцию с отключенными прерываниями, тогда, после возврата из микроСи/ОС-II, прерывания бы разрешились! Если вы отключили прерывания, вы, возможно, хотели бы, чтобы они были отключены и после возврата из функции микроСи/ОС-II. В этом случае, приведенный выше сценарий не адекватен.

**Метод #2:**
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 вы вызвали сервисную функцию микроСи/ОС-II с разрешенными или запрещёнными прерываниями, тогда это состояние было бы защищено на время вызова. Если вы вызываете микроСи/ОС-II сервис с заблокированными прерываниями, вы потенциально продлеваете латентность прерываний вашего приложения. Ваше приложение может использовать **OS_ENTER_CRITICAL()** и **OS_EXIT_CRITICAL()** чтобы также защитить критические участки кода. Будьте внимательным, однако, потому что ваше приложение зависнет, если вы запретите прерывания перед вызовом сервиса типа **OSTimeDly()**. Это произойдёт, потому что задача будет приостановлена на некоторый интервал времени, но так как прерывания запрещены, таймер никогда не сможет отсчитать нужное время! Очевидно, все вызовы PEND также относятся к этой проблеме, так что будьте внимательны. Как общее правило, вы должны всегда вызывать сервисы микроСи/ОС-II с разрешёнными прерываниями!
Вопрос в том, какой метод лучше? Однако все зависит от того, чем вы готовы пожертвовать. Если в вашем приложении без разницы, разрешены ли прерывания после возврата из сервиса микроСи/ОС-II, вы должны выбрать первый метод. Если вы хотите сохранить состояние прерывания после возврата из сервиса микроСи/ОС-II, тогда, очевидно, второй метод - для вас.

Просто для примера: запрет прерываний в Inttel 80186 выполняется по команде **CLI** и разрешение — по команде **STI**. В этом случае вы можете определить эти макросы так:

```
#define OS_ENTER_CRITICAL() asm CLI
#define OS_EXIT_CRITICAL asm STI
```

Обе команды **STI** и **CLI** выполняются на этом процессоре менее чем за два цикла каждая (т.е. всего за 4 цикла). Для защиты состояния прерываний вам нужно определить макросы по-другому:

```
#define OS_ENTER_CRITICAL() asm PUSHF; CLI
#define OS_EXIT_CRITICAL asm POPF
```

В этом случае, **OS_ENTER_CRITICAL()** будет длиться 12 циклов, тогда как **OS_EXIT_CRITICAL()** — 8 циклов (т.е. всего 20 циклов). Защита состояния прерываний займёт на 16 циклов больше, чем просто запрет/разрешение прерываний (по крайней мере на 80186). Очевидно, если вы имеете процессор быстрее чем Intel Pentium-II, эта разница будет меньше.

### *8.03.03 OS_CPU.H, OS_STK_GROWTH*

Стек большинства микропроцессоров и микроконтроллеров растёт в памяти с верху вниз. Есть, однако, некоторые процессоры, работающие по-другому. МикроСи/ОС-II была разработана для любых разновидностей процессоров. Это обеспечивается тем, что вы указываете микроСи/ОС-II, в каком направлении растёт стек, изменяя значение конфигурационной константы **OS_STK_GROWTH** (листинг 8.1(4)):

установите **OS_STK_GROWTH** в 0 если стек растёт снизу вверх, или в 1, если сверху вниз.

### *8.03.04 OS_CPU.H, OS_TASK_SW()*

**OS_TASK_SW()** (листинг 8.1(5)) это макрос, который вызывается когда микроСи/ОС-II переключается от низкоприоритетного задания к высокоприоритетному. **OS_TASK_SW()** всегда вызывается из кода уровня задачи. Другой механизм, **OSIntExit()**, используется для выполнения переключения контекста, когда программа обработки прерывания делает высокоприоритетную задачу готовой выполнению.
Переключение контекста состоит из сохранения регистров процессора в стек задачи, которая будет приостановлена, и восстановления регистров высокоприоритетной задачи из её стека.

В микроСи/ОС-II, фрейм стека для готовой задачи всегда выглядит так, как будто прерывание только что произошло, и все процессорные регистры были сохранены в него. Другими словами, все, что должна сделать микроСи/ОС-II, для запуска готовой задачи, это восстановить регистры из стека и выполнить возврат из прерывания. Для переключения контекста вы выполняете **OS_TASK_SW()** так, как будто моделируете прерывание. Большинство процессоров обеспечивают программное прерывание или инструкцию **TRAP** для выполнения этого. Программа обработки прерывания или указатель trap (также называемый «отлов исключений») ДОЛЖНЫ указывать вектором на функцию **OSCtxSw()** написанную на ассемблере (см. секцию 8.04.02).

Для примера, портация для Intel или AMD 80x86 процессора будет испольовать команду **INT**. Вектор прерывания должен указывать на **OSCtxSw()**. В портации для процессора Motorola 68HC11 более удобно использовать команду **SWI**.
Опять же, указатель для **SWI** должен указывать на функцию **OSCtxSw()**. Наконец, портация для Motorola 680x0/CPU32 процессора вероятно использовала бы одну из 16 команд **TRAP**. Конечно, указатель прерывания для выбранного **TRAP** должен указывать ни на что иное, как на **OSCtxSw()**.

Есть несколько процессоров, подобных Zilog Z80, которые не обеспечивают механизм программных прерываний. В этом случае, вы должны моделировать фрейм стека так близко к фрейму стека прерывания, насколько это возможно.

В этом случае **OS_TASK_SW()** будет просто вызывать **OSCtxSw()** вместо вектора к нему. Таким образом, Z80 — это процессор, который совместим с микроСи/ОС-II и она может быть на него портирована.

### *8.04 OS_CPU_A.ASM*

Для портации микроСи/ОС-II требуется, чтобы вы записали четыре довольно простые функции на языке ассемблера:

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

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

### *8.04.01 OS_CPU_A.ASM, OSStartHighRdy()*

Эта функция вызывается из **OSStart()** для запуска высокоприоритетной задачи, готовой к запуску. Однако, перед вызовом **OSStart()** вы ДОЛЖНЫ создать как минимум одну задачу (см. **OSTaskCreate()** и **OSTaskCreateExt()**). **OSStartHighRdy()** предполагает, что **OSTCBHighRdy** указывает на управляющий блок задачи с наивысшим приоритетом. Как отмечено ранее, в микроСи/ОС-II фрейм стека готовой к запуску задачи всегда выглядит так, будто прерывание только что произошло и все регистры процессора сохранились во фрейме. Для запуска задачи с наивысшим приоритетом все что нужно — это восстановить все регистры из стека задачи в правильном порядке и выполнить возврат из прерывания. Дабы упростить это, указатель стека всегда хранится в начале контрольного блока задачи (т.е. её **OS_TCB**). Другими словами, указатель стека для продолжаемой задачи всегда хранится в смещении 0 от начала её **OS_TCB**.

Заметьте, что **OSStartHighRdy()** должна вызывать **OSTaskSwHook()**, потому что мы в основном выполняем половину переключения контекста — мы восстанавливаем регистры задачи с наивысшим приоритетом. **OSTaskSwHook()** может может проверить переменную **OSRunning** чтобы сообщить, что **OSTaskSwHook()** была вызвана или из **OSStartHignRdy()** (т.е. если **OSRunning** = **FALSE**) или из регулярного переключателя контекста (т.е. **OSRunning** = **TRUE**).

**OSStartHighRdy()** ДОЛЖНА также установить **OSRunning** в **TRUE** перед восстановлением высокоприоритетной задачи (но после вызова **OSTaskSwHook()**).

### *8.04.02 OS_CPU_A.ASM, OSCtxSw()*

Как упомянуто выше, переключатель контекста задачного уровня выполняется командой программного прерывания или, в зависимости от процессора, выполнением команды **TRAP**. Программа обработки прерывания, TRAP или указатель обработки исключений ДОЛЖНЫ указывать на **OSCtxSw()**.

Последовательность событий, которые приводят микроСи/ОС-II к вектору на **OSCtxSw()** следующая. Текущая задача вызывает предоставляемый микроСи/ОС-II сервис, который переходит на более высокоприоритетную задачу, готовую к выполнению. В конце сервисной программы микроСи/ОС-II вызывает функцию OSSched(), которая решает что текущая задача уже не самая приоритетная и её на приостановить. **OSSched()** загружает адрес задачи с наивысшим приоритетом в **OSTCBHighRdy()** и затем выполняет программное прерывание или команду **TRAP**, вызывая макрокоманду **OS_TASK_SW()**. Отметьте, что переменная **OSTCBCur** уже содержит указатель на **OS_TCB** текущей задачи (контрольный блок задачи). Команда программного прерывания (или TRAP) вынуждает некоторые из регистров процессора (наиболее вероятно адрес возврата и слово состояния процессора) положить в стек текущей задачи и затем проццеор перенаправляется на **OSCtxSw()**.

Псевдокод, который необходимо выполнить в **OSCtxSw()** показан в листинге 8.2. Этот код должен быть написан на ассемблере, так как у вас нет доступа к регистрам процессора непосредственно из С.

```
void OSCtxSw(void)
{
Сохранить регистры процессора;
Сохранить текущи указатель стека в OS_TCB текущей задачи:
OSTCBCur->OSTCBStkPtr = Указатель стека;
Вызов пользовательской функции OSTaskSwHook();
OSTCBCur = OSTCBHighRdy;
OSPrioCur = OSPrioHighRdy;
Получение указателя стека новой задачи:
Указатель стека = OSTCBHighRdy->OSTCBStkPtr;
Восстановление всех регистров процессора из стека новой задачи;
Выполнение команды возврата из прерывания;
}
```

**Листинг 8.2, Псевдокод для OSCtxSw()**

Вы должны обратить внимание на то, что прерывания заблокированы во время работы **OSCtxSw()** и также во время выполнения определяемой пользователем функции **OSTaskSwHook()**.

### *8.04.03 OS_CPU_A.ASM, OSIntCtxSw()*

**OSIntCtxSw()** это функция вызываемая из **OSIntExit()** для выполнения переключения контекста из программы обработки прерывания. Так как **OSIntCtxSw()** вызывается из программы обработки прерывания, предполагается, что все регистры процессора правильно сохранены в стеке прерванной задачи. Фактически, в фрейме стека есть больше вещей, чем нам нужно. Таким образом **OSIntCtxSw()** должна будет очистить стек так, чтобы прерванная задача была оставлена только с правильным содержимым фрейма стека.

Чтобы понять, что надо сделать в **OSIntCtxSw()**, давайте взглянем на последовательность событий, приводящих микроСи/ОС-II к вызову **OSIntCtxSw()**. Вы можете обращаться к рисунку 8-2 чтобы понять следующее описание. Предположим, что прерывания не вложенные (т.е. программа обработки прерывания сама не прерывается), прерывания разрешены, и процессор выполняет код задачного уровня. Когда прерывание возникает, процессор завершает текущую команду, распознаёт прерывание и инициирует переход по вектору прерывания. Это обычно состоит из помещения регистров состояния процессора и возвратного адреса прерванной задачи в стек, рисунок 8-2(1). Порядок, в котором регистры помещаются в стек – несоответствующий.

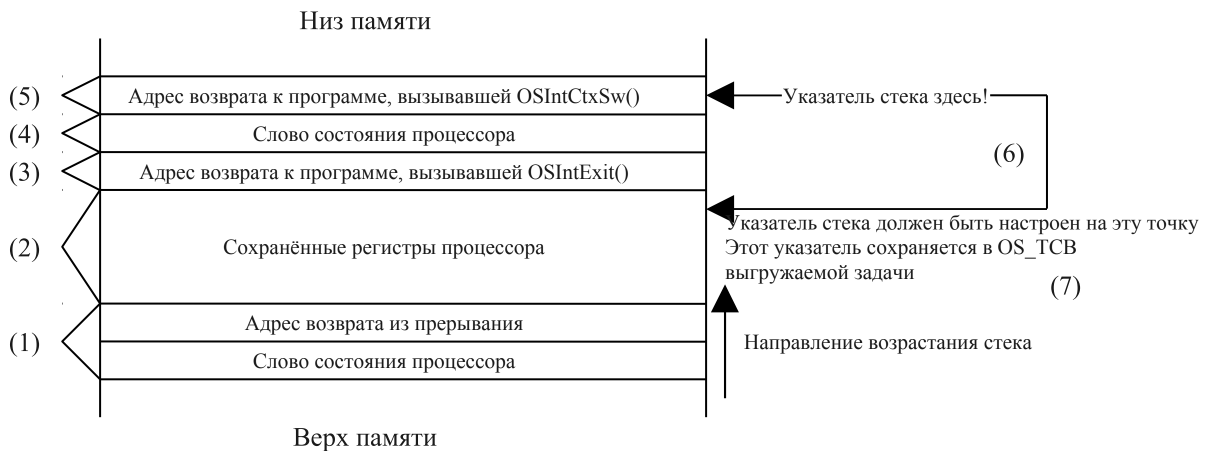


**Рисунок 8-2, Содержимое стека во время выполнения программы обработки прерывания**

Центральный процессор перенаправляется к нужной программе обработки прерывания. МикроСи/ОС-II требует, чтобы ваша программа обработки прерывания начиналась с сохранения оставшихся регистров процессора (рис. 8-2(2)). Как только регистры сохранены, МикроСи/ОС-II требует чтобы вы или вызвали **OSIntEnter()**, или чтобы вы инкрементировали глобальную переменную **OSIntNesting**. В этой точке, фрейм стека прерванной задачи содержит только содержимое регистров прерванной задачи. Программа обработки прерывания теперь может начать обслуживание прерывающего устройства и, возможно, сделать высокоприоритетную задачу готовой. Это произойдёт, если программа обработки прерывания отправит сообщение задаче (вызовом **OSMBoxPost()** или **OSQPost()**), продолжит задачу (вызовом **OSTaskResume()**), вызовет **OSTimeTick()** или **OSTimeDlyResume()**. Теперь предположим, что более приоритетная задача подготовлена к запуску.

МикроСи/ОС-II требует, чтобы ваша программа обработки прерывания вызывала **OSIntExit()** когда завершит обслуживание прервавшего устройства. **OSIntExit()** сообщает системе микроСи/ОС-II что настало время вернуться назад на код задачного уровня. Вызов **OSIntExit()** приводит к тому, что адрес возврата в вызывающую программу будет помещён в стек прерванной задачи (рис. 8-2(3)).

**OSIntExit()** начинается с запрета прерываний, потому что требуется выполнить критический код. В зависимости от реализации **OS_ENTER_CRITCAL()** (см. секцию 8.03.02) регистры состояния процессора помещаются в стек прерванной задачи (рис. 8-2(4)). **OSIntExit()** отмечает, что прерванная задача приостановится, потому что более приоритетная задача теперь готова к запуску. В этом случае в **OSTCBGighRdy** сохраняют указатель на **OS_TCB** новой задачи, затем функция **OSIntExit()** вызывает **OSIntCtxSw()** для выполнения переключения контекста. Вызов **OSIntCtxSw()** приводит к помещению адреса возврата в стек прерванной задачи (рис. 8-2(5)).

Поскольку мы переключаем контекст, мы хотим оставить только объекты 8-2(1) и 8-2(2) в стеке и проигнорировать 8-2(3), 8-2(4) и 8-2(5).Это выполняется прибавлением константы к указателю стека 8-2(6). Точная величина регулировки стека должна быть известна и это значение сильно зависит от целевого процессора (адрес может быть 8, 16, 32 или 64-битным), используемого компилятора, его настроек, модели памяти, и.т.д. Также слово состояния процессора может быть 8, 16, 32 или 64-битным, **OSIntExit()** может создавать локальные переменные. Одни процессоры позволяют вам непосредственно добавлять константу к указателю стека, другие – нет. В последнем случае вы просто выполняете нужное число команд POP для считывания в один из регистров процессора для получения того же результата. Как только стек настроен, новый указатель стека может быть сохранён в **OS_TCB** задачи, из которой переключаются (рис. 8-2(7)).

**OSIntCtxSw()** единственная функция в микроСи/ОС-II (и в микроСи/ОС), зависимая от компилятора и генерирует больше почтовых сообщений, чем любая другая часть микроСи/ОС. Если ваша портация виснет после нескольких переключений контекста, тогда вам следует подозревать, что стек не настраивается правильно в функции **OSIntCtxSw()**.

Псевдокод в листинге 8.3 показывает, что должна делать **OSIntCtxSw()**. Этот код должен быть написан на языке ассемблера, так как у вас нет доступа к регистрам процессора непосредственно из С. Если ваш компилятор С поддерживает встраивание ассемблерных кодов в С, вы можете расположить код для **OSIntCtxSw()** в **OS_CPI_C.C** вместо **OS_CPU_A.ASM**. Как вы можете видеть, код идентичен функции **OSCtxSw()** не считая первой строчки. Таким образом вы можете уменьшить число строк кода в портации перепрыгивая в соответствующий участок кода в **OSCtxSw()**.

```
void OSIntCtxSw(void)
{
настройка указателя стека для удаления вызова к :
OSIntExit(),
OSIntCtxSw() и возможно помещённого в стек слова состояния процессора;
Сохранение указателя стека текущей задачи в структуру OS_TCB текущей же задачи:
OSTCBCur->OSTCBStkPtr = Указатель стека;
Вызов определённой пользователем OSTaskSwHook();
OSTCBCur = OSTCBHighRdy;
OSPrioCur = OSPrioHighRdy;
Получение указателя стека продолжаемой задачи:
Указатель стека = OSTCBHighRdy->OSTCBStkPtr;
Восстановление всех регистров из стека новой задачи;
Выполене команды возврата из прерывания;
}
```

**Листинг 8.3, Псевдокод для OSIntCtxSw()**

### *8.04.04 OS_CPU_A.ASM, OSTickISR()*

МикроСи/ОС-II требует, чтобы вы обеспечили периодический временной источник чтобы следить за временными задержками и паузами. Импульс временного сигнала должен возникать с частотой от 10 до 100 Гц. Для этого вы можете выделить аппаратный таймер или использовать 50/60 Гц от цепей питания переменного тока.

Вы должны разрешить прерывания таймера ПОСЛЕ запуска мультизадачности, т.е. после вызова **OSStart()**. Другими словами, вы должны инициализировать прерывания и таймер в первой задаче, следующей за вызовом **OSStart()**. Обычная ошибка — разрешение прерывания по таймеру между вызовами **OSInit()** и **OSStart()**, как показано в листинге 8.4:

```
void main(void)
{
.
.
OSInit(); /* Инициализация  МикроСи/ОС-II */
.
.
/* Код инициализации приложения ... */
/* ... Создание как минимум одной задачи через вызов OSTaskCreate() */
.
.
Разрешение прерываний по ТАЙМЕРУ; /* НЕ ДЕЛАЙТЕ ЭТО ЗДЕСЬ!!! */
.
.
OSStart(); /* Запуск многозадачности */
}
```

**Листинг 8.4, Неправильное место для запуска прерываний по таймеру**

Что будет (а это обязательно будет), если прерывания разрешить до того, как микроСи/ОС-II запустит первую задачу? В этой точке микроСи/ОС-II находится в неизвестном состоянии и ваше приложение повиснет.

Псевдокод программы прерывания по таймеру приведён в листинге 8.5. Этот код должен быть написан на языке ассемблера, так как у вас нет доступа к регистрам процессора непосредственно из С. Если ваш процессор может увеличить значение **OSIntNesting** одной командой, вам не нужно вызывать **OSIntEnter()**. Приращение **OSIntNesting** происходит намного быстрее, чем вызов функции и возврат из неё. **OSIntEnter()** только увеличивает **OSIntNesting**, защищая это приращение в критической секции.

```
void OSTickISR(void)
{
Сохранить регистры процессора;
Вызов OSIntEnter() или непосредственное увеличение OSIntNesting;
Вызов OSTimeTick();
Вызов OSIntExit();
Восстановление регистров процессора;
Выполнение команды возврата из прерывания;
}
```

**Листинг 8.5, Псевдокод для программы обработки прерывания по таймеру.**

## *8.05 OS_CPU_C.C*

Портация микроСи/ОС-II требует, чтобы вы написали шесть довольно простых функций на С:

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

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

### *8.05.01 OS_CPU_C.C, 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. Рисунок 8-3 показывает, что **OSTaskStkInit()** помещает в стек создаваемой задачи. Отметьте, что на рисунке 8-3 принят стек растущий вниз. Дальнейшее обсуждение также относится и к стеку, растущему вверх.

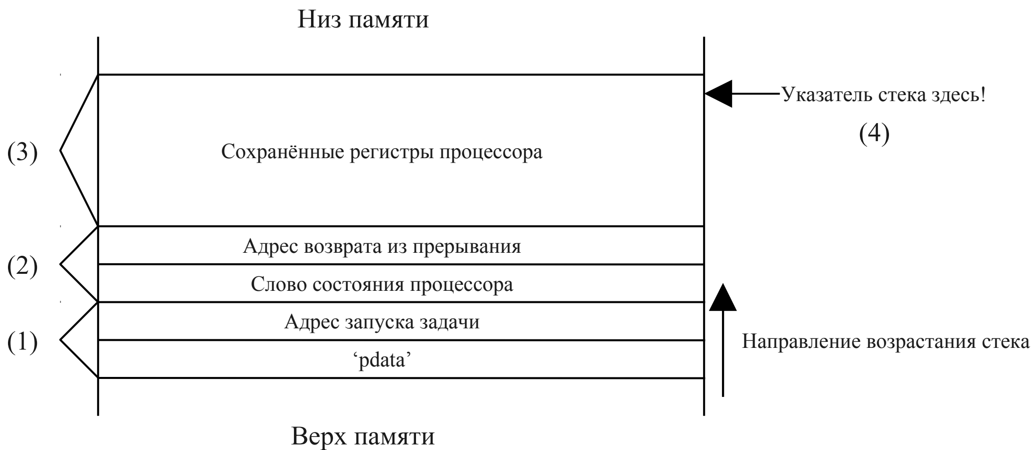


**Рисунок 8-3, Инициализация фрейма стека параметром 'pdata', передаваемым в стек.**

Когда вы создаёте задачу, вы определяете в **OSTaskCreate()** или **OSTaskCreateExt()** начальный адрес задачи, вы передаёте указатель **pdata**, верхнюю границу стека задачи и приоритет задачи. **OSTaskCreate()** требует дополнительных аргументов, но здесь их обсуждать не будем. Для правильной инициализации фрейма стека, **OSTaskStkInit()** требует только три первых аргумента, а также упомянем дополнительное значение ‘option’, которое доступно только в **OSTaskCreateExt()**.

Повторный вызов под мискроC/OC-II, задача написана как показано ниже. Задача есть бесконечный цикл, но с другой стороны, так же выглядит любая другая функция C. Когда задача запущена из мискроC/OC-II, она принимает аргумент как если бы она была вызвана другой функцией.

```
void MyTask (void *pdata)
{
/* Сделать что-нибудь с аргументом ‘pdata’ */
for (;;) {
/* Код задачи */
}
}
```

Если я хочу вызвать **MyTask()** из другой функции, компилятор C положит аргументы в стек вслед за адресом возврата в функцию, вызвавшую **MyTask()**. Некоторые компиляторы могут фактически передать параметр **pdata** в одном или нескольких регистрах. Эту ситуацию обсудим позже. Предположим **pdata** помещёно в стек, **OSTaskStkInit()** просто моделирует этот сценарий и загружает стек соответственно рисунку 8-3(1). Показано, что в отличие от вызова функций C, однако, мы на самом деле не знаем адрес возврата в вызвавшую программу. Всё, что мы имеем, это адрес начала задачи, а адреса возврата в функцию, которая вызвала эту функцию (т.е. задачу) у нас нет! Отсюда видно, что мы об этом действительно не заботимся, так как предполагается, что задача никогда не вернётся в другую функцию так или иначе.

В этой точке мы должны положить в стек регистры, которые автоматически кладутся процессором, когда он распознает и начинает обрабатывать прерывания. Некоторые процессоры кладут в стек все свои регистры, тогда как другие — только несколько. В общем, процессор поместит в стек, по крайней мере, значение счётчика команд для команды, на которую он должен будет вернуться после прерывания, и слово состояния процессора (рис. 8-3(3)). Вы должны точно следовать порядку операций процессора.

Затем, вам нужно положить в стек остальные регистры процессора. Их порядок определяется тем, даёт вам выбор ваш процессор или нет. Некоторые процессоры имеют одну или несколько инструкций, которые помещают в стек несколько регистров за раз. В этом случае вы должны подражать этому порядку. Например, Intel 80x86 имеет команду **PUSHA**, которая помещает восемь регистров в стек. У процессора Motorola 68HC11, все регистры автоматически помещаются в стек в течение реакции на прерывание, так что вы должны следовать тому же порядку.

Теперь настало время вернуться назад к проблеме того, что делать если ваш компилятор C передаёт аргумент **pdata** в регистрах, а не через стек. Вам нужно найти в документации на компилятор, в каких регистрах будет передаваться **pdata**. Фрейм стека будет выглядеть как показано на рисунке 8-4. **pdata** будет просто размещён в области, где вы сохраните соответствующие регистры.

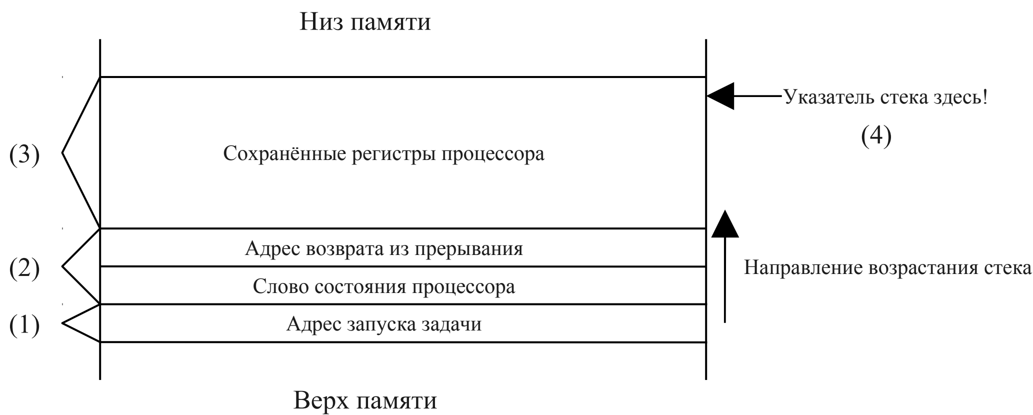


**Рисунок 8-4, Инициализация фрейма стека параметром 'pdata' переданным в регистрах**

Как только вы завершите инициализацию стека, **OSTaskStkInit()** должна будет вернуть адрес, где будет находится указатель стека после завершения заполнения стека. **OSTaskCreate()** или **OSTaskCreateExt()** возьмет этот адрес и сохранит его в блоке контроля задачи (**OS_TCB**). Документация на процессор подскажет вам, где будет находиться указатель стека: на следующем пустом месте, либо на последнем сохранённом значении. Для примера, на процессоре Intel 80x86 указатель стоит на последнем сохранённом значении, тогда как в Motorola 68HC11 он указывает на следующее пустое место.

### *8.05.02 OS_CPU_C.C, OSTaskCreateHook()*

**OSTaskCreateHook()** вызывается всякий раз, когда создаётся задача через **OSTaskCreate()** или **OSTaskCreateExt()**. Это позволяет вам или потребителю вашей портации расширять функциональность мискроC/OC-II. **OSTaskCreateHook()** вызывается когда мискроC/OC-II завершает настройку внутренней структуры, но до вызова планировщика. Прерывания блокируются на время работы этой функции, поэтому вы должны 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()** вы можете 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.

Код для **OSTaskCreateHook()** генерируется только если **OS_CPU_HOOKS_EN** установлено в 1 в файле **OS_CFG.H**. Это позволяет пользователю вашей портации переопределять все функции ловушки (hook) в отдельном файле. Очевидно, пользователям вашей портации понадобятся исходники системы для установления **OS_CPU_HOOKS_EN** в 1 во избежание множественного переопределения символов во время линковки.

### *8.05.03 OS_CPU_C.C, OSTaskDelHook()*

**OSTaskDelHook()** вызывается при каждом удалении задачи. **OSTaskDelHook()** вызывается перед отделением задачи от внутреннего списка активных задачи мискроС/ОС-II. При вызове **OSTaskDelHook()** получает указатель на **OS_TCB** подлежащей удалению задачи и, таким образом, может 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**.

### *8.05.04 OS_CPU_C.C, 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, вы должны 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**.

### *8.05.05 OS_CPU_C.C, OSTaskStatHook()*

**OSTaskStatHook()** вызывется раз в секунду из **OSTaskStat()**. С помощью **OSTaskStatHook()** вы можете расширить возможности статистики. Например, вы могли контролировать и отображать время выполнения каждой задачи, процент загрузки процессора каждой задачей, частоту выполнения задачи и др. **OSTaskStatHook()** не имеет параметров и ничего не возвращает.

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

### *8.05.06 OS_CPU_C.C, OSTimeTickHook()*

**OSTimeTickHook()** вызывается из **OSTimeTick()** в каждом системном тике. Фактически, **OSTimeTickHook()** вызывается перед тем, как мискроС/ОС-II действительно начнёт обработку тика, чтобы дать вашему приложению вперёд прореагировать на тик. **OSTimeTickHook()** не имеет параметров и ничего не возвращает.

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

## *OSTaskCreateHook()*

**`void OSTaskCreateHook(OS_TCB *ptcb)`**

| Файл | Откуда вызывается | Чем разрешается генерация кода |
|---|---|---|
| OS_CPU_C.C | OSTaskCreate() и OSTaskCreateExt() | OS_CPU_HOOKS_EN |

Эта функция вызывается при каждом создании задачи. **OSTaskCreateHook()** вызывается после того, как TCB был выделен и инициализирован, и инициализирован фрейм стека. **OSTaskCreateHook()** позволяет вам расширять функциональность функции создания задач своими собственными наворотами (features) Для примера, вы можете инициализировать и сохранить содержимое регистров с плавающей точкой, регистров управления памятью (MMU) или что-нибудь ещё, связанное с задачами. Вы, однако, обычно сохраняли бы эту дополнительную информацию в памяти, распределенной вашим приложением. Вы можете также использовать **OSTaskCreateHook()** для запуска осциллографа, логического анализатора, или установки точки останова.

**Аргументы**

**ptcb** —указатель на OS_TCB создаваемой задачи.

**Возвращаемое значение**

Нет.

**Замечания/предупреждения**

При вызове этой функции прерывания запрещаются. Поэтому вы должны минимизировать код функции для уменьшения латентности прерываний.

**Пример**

Этот пример предполагает, что вы создали задачу через **OSTaskCreateExt()**, так как это требуется для наличия поля **.OSTCBExtPtr** в блоке **OS_TCB** задачи, содержащем указатель на хранилище для регистров с плавающей точкой.

```
void OSTaskCreateHook (OS_TCB *ptcb)
{
if (ptcb->OSTCBExtPtr != (void *)0) {
/* Сохранение регистров с плавающей точкой в ... */
/* .. расширении TCB */
}
}
```

## *OSTaskDelHook()*

**`void OSTaskDelHook(OS_TCB *ptcb)`**

| Файл | Откуда вызывается | Чем разрешается генерация кода |
|---|---|---|
| OS_CPU_C.C | OSTaskDel() | OS_CPU_HOOKS_EN |

Эта функция вызывается при каждом удалении задачи функцией **OSTaskDel()**. Таким образом вы могли бы распорядиться памятью, которая была выделена в ловушке (hook) создания задачи **OSTaskCreateHook()**. **OSTaskDelHook()** вызывается перед исключением TCB удаляемой задачи из списка TCB системы. Вы можете также использовать **OSTaskDelHook()** для запуска осциллографа, логического анализатора, или установки точки останова.

**Аргументы**

**ptcb** — указатель на структуру TCB удаляемой задачи.

**Возвращаемое значение**

Нет.

**Замечания/предупреждения**

При вызове этой функции прерывания запрещаются. Поэтому вы должны минимизировать код функции для уменьшения латентности прерываний.

**Пример**

```
void OSTaskDelHook (OS_TCB *ptcb)
{
/* Выходной сигнал на триггер или осциллограф */
}
```

## *OSTaskSwHook()*

**void OSTaskSwHook(void)**

| Файл | Откуда вызывается | Чем разрешается генерация кода |
|---|---|---|
| OS_CPU_C.C | OSCtxSw() и OSIntCtxSw() | OS_CPU_HOOKS_EN |

Эта функция вызывается при каждом переключении контекста. Глобальная переменная **OSTCBHighRdy** указывает на OS_TCB задачи, в распоряжение которой перейдёт процессор, в то время как **OSTCBCur** указывает на OS_TCB выключаемой задачи. **OSTaskSwHook()** вызывается только после сохранения регистров задачи и сохранения указателя стека в OS_TCB текущей задачи. Вы можете использовать эту функцию для сохранения содержимого регистров с плавающей точкой, регистров MMU, трассировки задачи во время выполнения, слежения за количеством переключений задачи и др.

**Аргументы**

Нет

**Возвращаемое значение**

Нет

**Замечания/предупреждения**

При вызове этой функции прерывания запрещаются. Поэтому вы должны минимизировать код функции для уменьшения латентности прерываний.

**Пример**

```
void OSTaskSwHook (void)
{
/* Сохранить регистры с плавающей точкой в расширение OS_TCB текущей задачи. */
/* Восстановить регистры с плавающей точкой из расширения OS_TCB текущей
задачи. ??? */
}
```

## *OSTaskStatHook()*

**void OSTaskStatHook(void)**

| Файл | Откуда вызывается | Чем разрешается генерация кода |
|---|---|---|
| OS_CPU_C.C | OSTaskStat() | OS_CPU_HOOKS_EN |

Эта функция вызывается каждую секунду задачей статистики мискроС/ОС-II и позволяет вам добавлять свои параметры статистики.

**Аргументы**

Нет

**Возвращаемое значение**

Нет

**Замечания/предупреждения**

Задача статистики стартует через 5 секунд после вызова **OSStart()**. Отметьте, что эта функция не вызывается, если **OS_TASK_STAT_EN** или **OS_TASK_CREATE_EXT_EN** установлены в 0.

**Пример**

```
void OSTaskStatHook (void)
{
/* Вычисление общего времени выполнения всех задач */
/* Вычисление процента выполнения каждой задачи */
}
```

## *OSTimeTickHook()*

**void OSTimeTickHook(void)**

| Файл | Откуда вызывается | Чем разрешается генерация кода |
|---|---|---|
| OS_CPU_C.C | OSTimeTick() | OS_CPU_HOOKS_EN |

Эта функция вызывается из **OSTimeTick()**, которая, в свою очередь, вызывается при каждом тике системных часов. **OSTimeTickHook()** вызывается немедленно после входа в **OSTimeTick()**, чтобы позволить выполнить критический код вашего приложения.

Вы можете также использовать эту функцию для запуска осциллографа для отладки, запуска логического анализатора, настройки точки останова для эмулятора, и т.п.

**Аргументы**

Нет

**Возвращаемое значение**

Нет

**Замечания/предупреждения**

**OSTimeTick()** в основном вызывается из прерываний и, таким образом, время обработки прерывания по тику увеличивается вашим кодом, находящимся в этой функции. Прерывания могут быть разрешены или запрещены перед вызовом **OSTimeTickHook()** в зависимости от реализации портации. Если прерывания отключены, это увеличивает их латентность.

Пример

```
Void OSTimeTickHook(void)
{
 /* Запуск осциллографа */
}
```