Планировщик задач для ARM Cortex-M3:
пример реализации

Многие разработчики встраиваемых систем применяют микроконтроллеры на базе ARM Cortex-M3, создавая исходный код с применением операционных систем реального времени, при этом не многие из новичков до конца представляют реализацию многозадачности в конкретной используемой RTOS из-за большого объема кода самой операционной системы и трудностей сопоставления кода с аппаратной платформой.
Для многих реализация многозадачности с контекстным переключением является сложной, поэтому в статье речь пойдет о практическом применении механизма переключения контекста в Cortex-M3 с примером программной реализации планировщика задач с вытесняющей многозадачностью. Разработанный планировщик обладает небольшим объемом исходного кода, что позволяет без особых трудностей разобраться с переключением контекста, модифицировать либо переписать исходный код под свою задачу, разработать несложную собственную операционную систему реального времени.
В статье кратко рассмотрена архитектура ARM Cortex-M3, детально исследован процесс переключения контекста, описаны особенности и возможности созданного планировщика.

Дмитрий ГЛАЗКОВ
glazkov-d@yandex.ru

Введение

Проект планировщика создан в двух исполнениях — для отладочной платы STM32L-DISCOVERY (на базе микроконтроллера STM32L152RBT6) и симулятора KEIL семейства STM32F103X. У разработанного планировщика имеются следующие особенности:

- Небольшой размер текста исходного кода.
- Вытесняющая циклическая многозадачность.
- Максимальное число задач зависит лишь от объема оперативной памяти микроконтроллера.
- 255 уровней приоритета.
- Минимальный расход памяти данных на стек задачи — 16 32-разрядных слов, максимальный — ограничен объемом памяти микроконтроллера.
- Для синхронизации выполнения задач используются: критические секции, мьютексы, приоритеты.
- Размер памяти кода программы — 1500 байт.
- Время переключения между задачами — 25 мкс при частоте ядра 16 МГц.

Проект создан с использованием языков: ANSI C, Assembler в среде разработки Keil uVision V4.53.0.0 (Toolchain: MDK-ARM Standard Version 4.53.0.0).

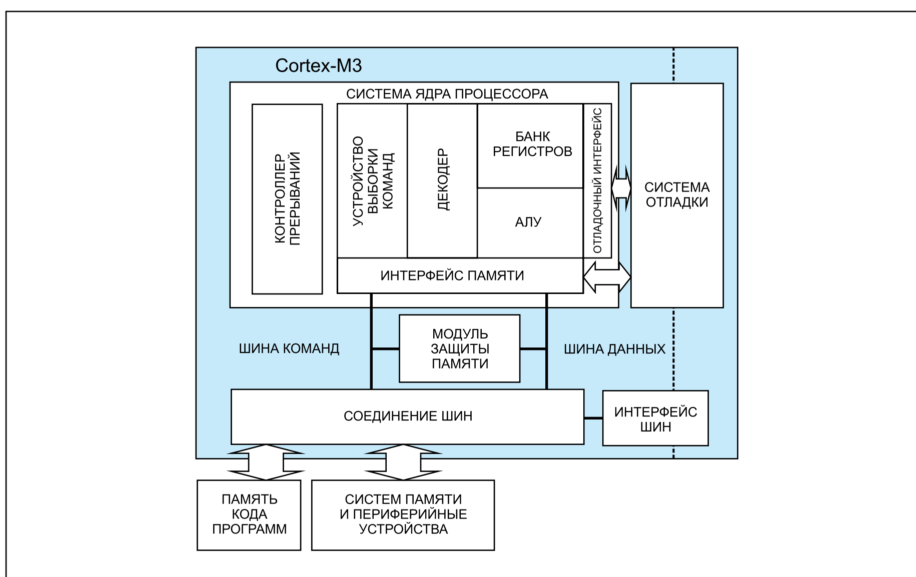

Рис. 1. Структура ядра ARM Cortex-M3

Краткие сведения об архитектуре ARM Cortex-M3

Ядро Cortex-M3 базируется на гарвардской архитектуре, имеет раздельные шины для команд и для данных в отличие от стандартных ARM-процессоров. Упрощенная структура ядра представлена на рис. 1.

Ядро процессора Cortex-M3 содержит декодер для традиционной системы команд Thumb и для новой системы Thumb-2, усовершенствованное АЛУ с поддержкой аппаратного умножения и деления, управляющую логику и интерфейсы к другим компонентам системы. Процессор на базе архитектуры

Cortex-M3 представляет собой 32-разрядный процессор с 32-разрядными шиной данных, банком регистров и интерфейсом памяти. Он содержит 13 регистров общего назначения, два указателя стека, регистр связей, счетчик команд, регистр статуса и множество специальных регистров.

Для выполнения операций обработки данных вначале необходимо поместить операнды из памяти в центральный регистровый файл (табл. 1), затем выполнить требуемую операцию над данными в регистрах и, наконец, перезаписать результат обратно в память.

Таблица 1. Структура регистрового файла

Регистр	Описание
R0	Регистр общего назначения
R1	Регистр общего назначения
R2	Регистр общего назначения
R3	Регистр общего назначения
R4	Регистр общего назначения
R5	Регистр общего назначения
R6	Регистр общего назначения
R7	Регистр общего назначения
R8	Регистр общего назначения
R9	Регистр общего назначения
R10	Регистр общего назначения
R11	Регистр общего назначения
R12	Регистр общего назначения
R13(SP)	Указатель стека
R14(LR)	Регистр связи
R15(PC)	Счетчик команд

Вся активность программы фокусируется вокруг регистрового файла процессора. Этот регистровый файл образуют шестнадцать 32-битных регистров. Регистры R0–R12 — обычные регистры, которые можно использовать для хранения программных переменных. У регистров R13–R15 имеются особые функции. Регистр R13 выступает в роли указателя стека. Этот регистр является банковым, что делает возможной работу в двух режимах, в каждом из которых используется свое собственное пространство стека. Такая возможность обычно используется операционными системами реального времени (ОСРВ), которые могут выполнять свой «системный» код в защищенном режиме. У двух стеков имеются собственные наименования: основной стек и стек процесса. Следующий регистр R14 называется регистром связи. Он используется для выбора способа возврата из подпрограммы или прерывания. Благодаря ему процессор быстро переходит к подпрограмме или прерыванию и выходит из них. Если же в программе используется несколько уровней вложений подпрограмм, то компилятор будет автоматически сохранять R14 в стек. Последний регистр R15 — счетчик команд; поскольку он является частью центрального регистрового файла, его чтение и обработка могут выполняться аналогично любым другим регистрам.

Помимо регистрового файла, имеется отдельный регистр статуса (PSR), который хранит флаги АЛУ (нуля, переноса, статус исключения).

Переключения контекста

Переключение контекста в многозадачных средах — это процесс прекращения выполнения процессором одной задачи с сохранением всей необходимой информации и состояния, необходимых для последующего продолжения с прерванного места, и восстановления и загрузки состояния задачи, к выполнению которой переходит процессор.

В ядре Cortex-M3 при обработке прерывания происходит автоматическое сохранение в стек регистров R0–R3, R12, LR, PC и PSR до момента входа в программу-обработчик. Делается это с целью предоставления свободных регистров обработчику, с которыми можно работать, так как если не сохранить эти регистры, то возврат из прерывания приведет к повреждению памяти программы. Адрес ячейки памяти, где находится начало сохраненных регистров, записывается в регистре указателя стека. У многих может возникнуть вопрос, почему сохраняются именно вышеуказанные регистры, ведь в R4–R11 также могут храниться важные данные, которые могут быть повреждены обработчиком, если обработчик реализован в виде СИ функции. Дело в том, что согласно СИ стандартам (C/C++ standard Procedure Call Standard for the ARM Architecture, AAPCS, Ref 5) именно автоматически сохраняемые регистры могут быть изменены обработчиком, что позволяет коду обработчика быть обычной СИ функцией. Поэтому в обработчике прерывания можно не бояться повреждения регистров R4–R11, так как они не могут быть использованы компилятором в коде обработчика. При выходе из обработчика прерывания регистры R0–R3, R12, LR, PC и PSR автоматически восстанавливаются. Все изложенное объясняет, как войти в обработчик и произвести в нем необходимые действия, а затем вернуться в прерванную программу без повреждения данных. Здесь был рассмотрен процесс входа в обычное прерывание и выход из него.

Однако определение переключения контекста говорит о необходимости вернуться не в ту задачу, которую прервали, а в другую. Для этого у Cortex-M3 имеется специальное прерывание PendSV_Handler, которое инициируется программно. В основном именно в нем происходит процесс выхода из прерывания на другой участок кода. Схематично этот процесс показан на рис. 2.

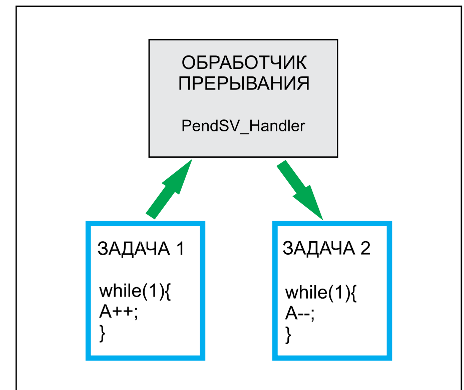

Рис. 2. Переключение контекста

На рис. 2 видно, что в момент выхода из обработчика прерывания необходимо восстановить регистры не для задачи № 1, а для задачи № 2. Для этого следует подменить значение указателя стека на тот адрес, который ссылается на место, где лежит начало сохраненных регистров R0–R3, R12, LR, PC, PSR для задачи № 2.

Могут возникнуть два вопроса: как можно с помощью одного указателя стека сделать возврат из обработчика в другую задачу и что же является стеком задачи. На первый вопрос для мало знакомых с архитектурой ответ окажется неожиданным — указателей стека два, но об этом далее. По второму вопросу: стек задачи — это некоторая определенная область памяти, куда сохраняются промежуточные значения работы задачи в момент ее прерывания. В рассматриваемом в статье планировщике стек задачи представлен на рис. 3.

Как видно на рис. 3, массив хранит все рабочие регистры, указатель стека хранит адрес памяти, где находится начало необходимых регистров. Так как при выходе из обработчика прерывания аппаратно восстанавливаются лишь R0–R3, R12, LR, PC и PSR, указатель стека будет содержать адрес восьмого элемента массива на момент выхода из прерывания. Работа с регистрами R4–R11 осуществляется программно.

В кратком обзоре архитектуры ARM Cortex-M3 было сказано, что регистр указателя стека R13(SP) один, а указателей стека два. Как было отмечено в кратком обзоре архитектуры, регистр R13(SP) является банковым, то есть в одном случае он указывает на один стек, а в другом — на другой.

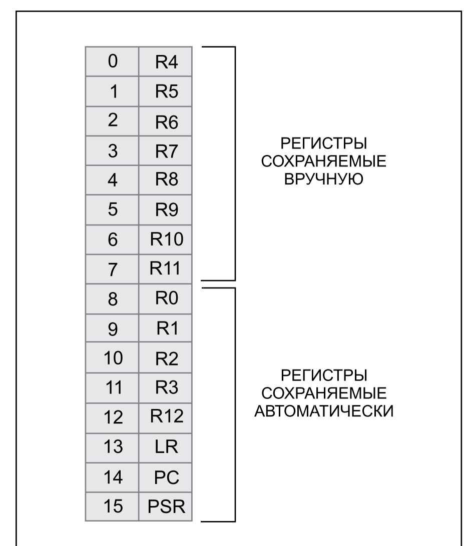

Рис. 3. Структура стека задачи

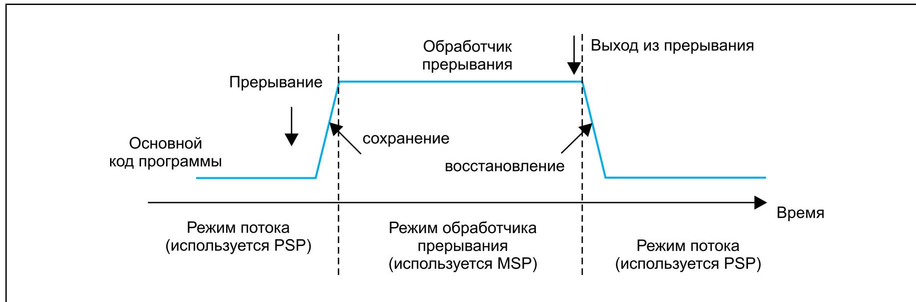

Рис. 4. Переключение рабочего указателя стека при переключении контекста

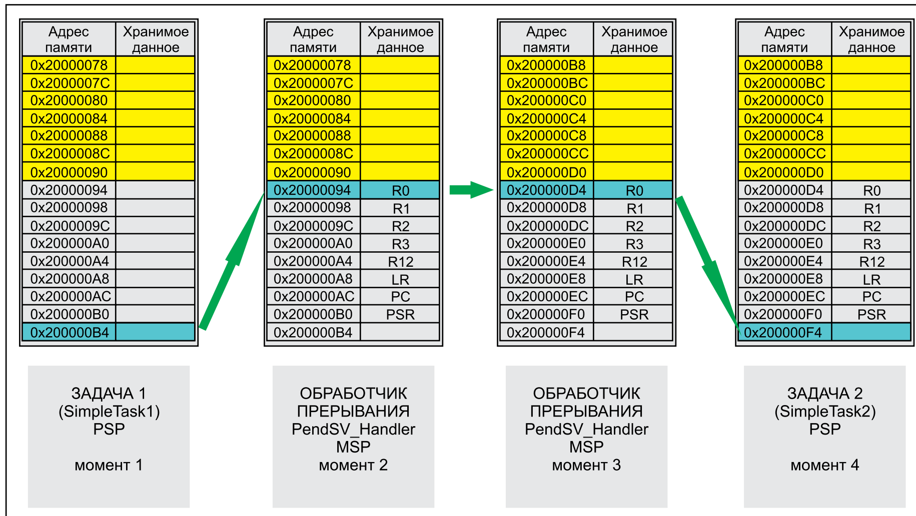

Рис. 5. Подмена указателя PSP при переключении контекста

В Cortex-M3 имеется два указателя стека со следующими предназначениями:

1) MSP (Main Stack Pointer) — используется в обработчиках прерываний либо во всей программе.
2) PSP (Process Stack Pointer) — используется только в задачах (процессах) и не может использоваться в обработчиках прерываний.

Хотя регистр R13 (указатель стека) один, коды команд (0xF3EF8009) и (0xF3EF8008) различны, хотя и работают с одним регистром:

```
    10:         mrs r0,psp
0x080001C0 F3EF8009  MRS      r0,PSP
    11:         mrs r0,msp
    12:
0x080001C4 F3EF8008  MRS      r0,MSP
```

Эти различия в кодах команд и говорят, с каким указателем стека работать через регистр R13, то есть из какого банка его выбрать (значение MSP либо PSP). Раз у обработчика прерывания свой указатель стека, а у задачи свой, проиллюстрируем более подробно процесс работы со стеком во время переключения контекста с помощью рис. 4 и 5.

Как видно на рис. 4, в режиме потока (задачи) используется указатель PSP, в момент системного вызова происходит сохранение R0–R3, R12, LR, PC и PSR с адреса памяти, на который указывает PSP. Обработчик прерывания работает с указателем MSP, поэтому не влияет на PSP, при этом указатель PSP доступен в обработчике прерываний. Здесь его и можно подменить на значение, ссылающееся на контекст второй задачи, для которой будет произведено восстановление. На рис. 5 зеленой стрелкой показано изменение значения указателя PSP.

В момент времени № 1 процессор выполняет задачу № 1 — SimpleTask1, голубым цветом указано значение указателя стека PSP. В момент возникновения прерывания, начиная с адреса (PSP-0x00000004), происходит автоматическое сохранение регистров R0–R3, R12, LR, PC, PSR. Наступает момент времени № 2, в нем используется указатель стека MSP для собственной работы обработчика. В момент времени № 3 обработчик изменяет указатель PSP со значения 0x20000094 на 0x200000D4. Во время выхода из прерывания возврат производится по новому указателю стека PSP=0x200000D4, начиная с которого процессор восстанавливает регистры R0–R3, R12, LR, PC, PSR. Поскольку это стек задачи SimpleTask2, в нем будет храниться значение счетчика команд (PC) для задачи № 2 и возврат произойдет уже в задачу № 2. Это и будет конечный момент времени № 4. Желтым цветом выделено место для сохранения вручную регистров R4–R11 в обработчике прерывания.

Для упрощения на рис. 5 были изображены стеки задач минимальной длины, но так как задача может содержать вызовы функций, глубина стека может быть больше, ведь необходимо хранить информацию, куда вернуться после обработки вызванной функции. Для установки глубины стека в планировщике предусмотрена специальная константа, пока назовем ее «размер_стека». Она будет определять размер стека для каждой из задач. Так как каждой задаче необходим стек, число стеков будет равно числу задач. Было рассмотрено, что обработчик подменил указатель PSP на значение, соответствующее задаче SimpleTask2. Помимо стеков задач необходимо хранить текущие значения указателей PSP, так как указатель PSP может изменять свое значение в пределах адресного пространства стека задачи, ведь нужно знать, где именно остановилась каждая задача в момент ее прерывания и переключения на другую задачу.

Для хранения стеков задач организован массив стеков, для хранения указателей — массив указателей стеков, взаимоотношения между массивами отображены на рис. 6.

Как видно на рис. 6, в массиве указателей стеков хранятся значения PSP на момент входа в обработчик прерывания. Именно обра-

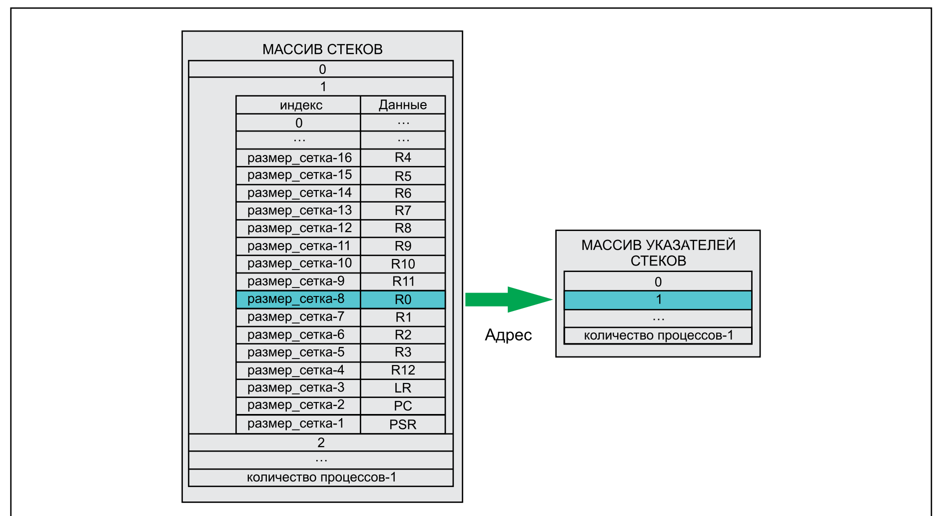

Рис. 6. Взаимоотношение массива стеков и массива указателей стеков

Таблица 2. Описание файлов проекта

Файл	Содержимое
main.c	Тело программы пользователя
OS_Typedef.h OS_defvar.c	Объявления постоянных выражений
OS_external.c	Объявления прототипов переменных и функций
OS_mutex.c	Функции для работы с мьютексами
OS_ARMCortexM3_port.c	Функции переключения контекста
OS_config.h	Константы для планировщика, задаваемые пользователем
OS_Init_funct.c	Функции инициализации планировщика, стеков задач, назначения приоритетов

ботчик сохраняет значение PSP (например, для задачи № 1) в массив указателей, берет новое значение PSP из массива указателей (например, для задачи № 2), завершает прерывание с восстановлением нового контекста. Такая структура хранения информации о задачах используется в планировщике.

Реализация переключения контекста в планировщике

Основным файлом планировщика является *OS_ARMCortexM3_port.c*, где находится обработчик прерывания по переключению контекста PendSV_Handler. Код этого обработчика рассмотрим, так как это основная функция планировщика, производящая замену контекста:

```
__asm void PendSV_Handler() {
   extern OS_PSPs
   extern OS_PSPsOffset
   extern OS_PSPsOffset_OLD
   LDR r0,=OS_PSPsOffset_OLD  R0←адрес(OS_PSPsOffset_
OLD)
   LDR r1,=OS_PSPs            R1←адрес(OS_PSPs)
   LDR r2,[r0,#0]             R2←значение(OS_PSPsOffset_
OLD)
   MRS r3,psp                 R3←значение PSP(регистра R12)
   STR r3,[r1,r2]             OS_PSPs(OS_PSPsOffset_
OLD)←R3
   SUBS r3,r3,#32             R3←R3-32
   STMIA R3!,{R4-R11}         Сохранять R4…R11 начиная с
адреса R3 с инкрементом R3
   LDR r0,=OS_PSPsOffset      R0←адрес(OS_PSPsOffset)
   LDR r2,[r0,#0];            R2←значение(OS_PSPsOffset)
   LDR r3,[r1,r2];            R3←OS_PSPs[OS_PSPsOffset]
   SUBS R3,R3,#32;            R3←R3-32
   LDMIA R3!,{R4-R11}         Загрузить R4…R11 значениями,
начиная с адреса R3
   MSR PSP, R3                PSP(R12)←R3
   LDR lr,=0xFFFFFFFD         LR←0xFFFFFFFD
   BX lr
Выход из прерывания по LR
   ALIGN
}
```

В коде планировщика массив указателей стеков, о котором говорили ранее, называется OS_PSPs, переменная ***OS_PSPsOffset*** — смещение внутри массива для следующей задачи, запуск которой будет произведен, ***OS_PSPsOffset_OLD*** — смещение внутри массива для текущей задачи, которую прервал обработчик.

В обработчике PendSV_Handler сначала сохраняется текущее значение PSP в ячейку PSPs[(OS_PSPsOffset_OLD], затем отдельно вычисляется смещение указателя стека (PSP-32) для хранения R4–R11. Эти регистры хранятся выше, согласно рис. 3.

Таблица 3. API-функции планировщика

API-функция	Прототип	Описание
OS_Init()	void OS_Init(void)	Инициализирует начальные значения переменных планировщика. Пример использования: OS_Init();
OS_Task_Create	unsigned int OS_Task_Create(FunctionPointer F)	Создает задачу, устанавливает уровень приоритета «1» для этой задачи и возвращает дескриптор. В качестве параметра передается адрес задачи. Пример использования: unsigned int Task_Descriptor1; Task_Descriptor1=OS_Task_Create(&SimpleTask1);
OS_Set_Task_Priority	void OS_Set_Task_Priority(unsigned int TaskDescriptor, unsigned char PriorityLevel)	Устанавливает приоритет задачи. В качестве параметров принимает дескриптор задачи, уровень приоритета. Пример использования: OS_Set_Task_Priority(Task_Descriptor1,2)
OS_Create_Mutex	unsigned char OS_Create_Mutex(void)	Создает мьютекс и возвращает его дескриптор. unsigned char Mutex_Descriptor_1 Mutex_Descriptor_1=OS_Create_Mutex();
OS_Take_Mutex	void OS_Take_Mutex(unsigned char MutexDescriptor)	Занимает мьютекс. В случае если мьютекс занят другой задачей, прерывает выполнение задачи до момента его освобождения. Пример использования: unsigned char Mutex_Descriptor_1 OS_Take_Mutex(Mutex_Descriptor_1);
OS_Return_Mutex	void OS_Return_Mutex(unsigned char MutexDescriptor)	Возвращает мьютекс. Пример использования: unsigned char Mutex_Descriptor_1 OS_Return_Mutex(Mutex_Descriptor_1);
OS_Switch_Task	void OS_Switch_Task(void)	Прерывает выполнение задачи по ее требованию

Сохранение ведется с помощью команды ***STMIA R3!,{R4-R11}***, где в R3 изначально находится значение (PSP-32). По мере сохранения каждого из регистров R3 увеличивается до значения PSP.

После сохранения старого контекста производится чтение указателя стека для следующей задачи OS_PSPs[OS_PSPsOffset], модификация указателя путем вычитания числа 32 для чтения R4…R11 (аналогично выполненной ранее операции смещения). Чтение производится с помощью ***LDMIA R3!,{R4… R11}***. По завершении чтения R3 возвращается в начальное состояние (до вычитания числа 32). Регистр R3 передается в PSP и производится выход из прерывания с помощью команды BX LR, где равенство LR=0xFFFFFFFD означает возврат в режим потока с использованием стека процесса (PSP).

Обработчик PendSV_Handler вызывается с помощью функции вызова программного прерывания ***void CallPendSV(void)***, которая, в свою очередь, вызывается из функции void ***OS_Switch_Task(void)***. Последняя функция определяет, какая задача должна быть запущена следующей. Функция ***OS_Switch_Task*** может быть вызвана как из SysTick_Handler, так и из кода задачи.

Описание проекта планировщика и пример его использования

Разработанный планировщик предоставляет возможность программисту на несложном примере кода изучить процесс переключения контекста и понять принципы работы ОСРВ. Этот планировщик может быть встроен в относительно несложные проекты, где требуется параллельная работа задач и нет необходимости в использовании сложно настраиваемой RTOS, для применения которой нужно изучить немало документации.

Большое преимущество этого планировщика заключается в легкости портирования на любой микроконтроллер на базе ядра Cortex-M3, так как в планировщике не используются связи с какой-либо периферией конкретного микроконтроллера, только средства ядра Cortex-M3.

Основными недостатками планировщика являются:

- фиксированный размер стека для всех задач;
- малое количество предоставляемых функциональных возможностей.

Описание файлов проекта приведено в таблице 2. API-функции, предоставляемые планировщиком, представлены в таблице 3.

Также имеется флаг установки критической секции ***OS_Active_Critical_Section***. При нахождении флага в значении «1» — планировщику запрещено переключение задач, при нахождении в значении «0» — разрешено.

Планировщик выполняет задачи с наиболее высоким приоритетом, задачи с более низким приоритетом не выполняются. Если необходимо временно приостановить задачу, нужно установить ей уровень приоритета «0». По умолчанию функция ***OS_Task_Create*** создает каждую задачу с приоритетом, равным единице, и задача с приоритетом 0 будет иметь уровень ниже остальных задач и выполняться не будет. Если все задачи имеют одинаковый приоритет, то они выполняются в последовательности их создания

Таблица 4. Файл конфигурации OS_config.h

Параметр	Минимальное значение	Описание
OS_ProcessCount	1	Задает количество задач, которые будут созданы функцией OS_Task_Create
cProcStackSize	16	Задает размер стека (число 32-разрядных слов) для каждой из задач
c_OS_Mutex_Count	1	Задает количество мьютексов, которые будут созданы с помощью функции OS_Create_Mutex

функцией ***OS_Task_Create*** вне зависимости от численного значения приоритета.

Настройка планировщика осуществляется в файле *OS_config.h*, параметры настройки приведены в таблице 4.

Размер стека не регулируется для каждой отдельной задачи и является постоянной величиной, определяемой ***cProcStackSize***. Поэтому данную величину необходимо определять исходя из потребностей в глубине стека задачи, содержащей максимальное число вложенных вызовов функций.

Пример использования

В файле *main.c* проекта находится следующий код программы:

```
unsigned int Task1_descriptor,Task2_descriptor,Task3_descriptor,
Task4_descriptor,Task5_descriptor,Task6_descriptor;

unsigned char MutexA;

unsigned int WorkingVariable1,WorkingVariable2,WorkingVariable3;

void SampleTask1(void){
  while(1){
     OS_Take_Mutex(MutexA);
     while(WorkingVariable1<200000){
        WorkingVariable1++;
     }
     OS_Return_Mutex(MutexA);
     while(WorkingVariable1>199999){};
  }
}

void SampleTask2(void){
  while(1){
     OS_Take_Mutex(MutexA);
     while(WorkingVariable1>0){
        WorkingVariable1--;
     }
     OS_Return_Mutex(MutexA);
     while(WorkingVariable1<1){};
  }
}

void SampleTask3(void){
  while(1){
     OS_Set_Task_Priority(Task4_descriptor,0);
     while(WorkingVariable2<200000){
        WorkingVariable2++;
     }
     OS_Set_Task_Priority(Task4_descriptor,1);
     while(WorkingVariable2>0){
     }

  }
}

void SampleTask4(void){
  while(1){
     OS_Set_Task_Priority(Task3_descriptor,0);
     while(WorkingVariable2>0){
        WorkingVariable2--;
     }
     OS_Set_Task_Priority(Task3_descriptor,1);
     while(WorkingVariable2<200000){
     }
  }
}

void SampleTask5(void){
  while(1){
     WorkingVariable3=1;
  }
}
void SampleTask6(void){
  while(1){
     WorkingVariable3=0;
  }
}

int main(void)
{
  WorkingVariable1=0;
  WorkingVariable2=0;
  OS_Init();
  MutexA=OS_Create_Mutex();
  Task1_descriptor=OS_Task_Create(&SampleTask1);
  Task2_descriptor=OS_Task_Create(&SampleTask2);
  Task3_descriptor=OS_Task_Create(&SampleTask3);
  Task4_descriptor=OS_Task_Create(&SampleTask4);
  Task5_descriptor=OS_Task_Create(&SampleTask5);
  Task6_descriptor=OS_Task_Create(&SampleTask6);

  SysTick_Config(1600);
  while(1){
  }
}
```

Этот пример демонстрирует работу с мьютексами и приоритетами и переключением задач. Сначала инициализируются рабочие объекты (переменные) ***WorkingVariable***. Затем с помощью функции ***OS_Init()*** устанавливаются начальные значения переменных планировщика. После этого создается дескриптор мьютекса MutexA. Следующим этапом создаются задачи SampleTask1… SampleTask6 с присвоением дескрипторов этих задач. Только после настройки системного таймера и возникновения первого прерывания SysTick_Handler микроконтроллер переключается из цикла ***while(1)*** в ***int main(void)*** на выполнение SimpleTask1. Эта задача забирает ***MutexA*** для работы с переменной ***WorkingVariable1***. В данном примере мьютекс занят, пока задача не насчитает число 200 000. Пока этого не произойдет, задача SimpleTask2 не сможет захватить мьютекс и уменьшать число до нуля. Как только задача 1 завершит счет до 200 000, она вернет мьютекс и уменьшением значения до нуля займется задача SimpleTask2, при этом захваченный мьютекс будет блокировать задачу SimpleTask1. Задача SimpleTask3 блокирует с помощью приоритета задачу SimpleTask4, пока не увеличит переменную ***WorkingVariable2*** до 200 000. Как только это произойдет, задача SimpleTask3 разблокирует задачу SimpleTask4, которая займется уменьшением значения ***WorkingVariable2*** до нуля. На момент уменьшения значения ***WorkingVariable2*** задача SimpleTask4 блокирует задачу SimpleTask3 с помощью приоритета. Задачи SimpleTask5 и SimpleTask6 не производят взаимной блокировки и выполняются последовательно друг за другом.

Заключение

Разработанный планировщик позволит заинтересованным программистам изучить на практическом примере процесс переключения контекста, а при желании — создать собственную операционную систему реального времени. Рассмотренные вопросы позволят понять, что происходит в существующих операционных системах реального времени.

Литература

1. Yiu J. The Definitive Guide to the ARM Cortex-M3 // Elsevier. 2007.
2. http://www.gaw.ru/html.cgi/txt/doc/micros/arm/cortex_arh/2_3_2.htm
3. www.arm.com
4. http://ru.osdev.wikia.com/wiki/Обработка_прерываний_в_архитектурах_ARMv6-M_и_ARMv7-M
5. http://www.coocox.org/CoOS.htm
6. http://www.kit-e.ru/articles/STM32L-Discovery.zip