Использование алгоритма планирования RMS в микроконтроллерах

Николай Баландин, Александр Крапивный (Москва)

В статье рассмотрены различные подходы к написанию многозадачных приложений для микроконтроллеров, в том числе расширение жёсткого реального времени RMEX для микроконтроллеров Microchip PIC18, которое использует алгоритм монотонной частоты для планирования задач и обеспечивает гарантированное время отклика.

ОСРВ для микроконтроллера

Все начинающие программисты знакомы со структурой программы для микроконтроллера, имеющей основной бесконечный цикл (loop) в теле функции *main* и выполняющей в ней все действия. При этом прерывания выполняются на переднем плане, а суперцикл – в фоновом режиме. Такая структура удобна для выполнения очень небольшого количества задач, и при необходимости увеличения числа задач программа становится довольно громоздкой и трудночитаемой. Также начинают возникать проблемы взаимодействия и синхронизации между задачами и процедурами обработки прерываний. Поэтому на сегодняшний день всё большую популярность приобретает применение операционных систем реального времени (ОСРВ) в микроконтроллерах. Такие ОС должны быть компактными, чтобы работать в микроконтроллере с объёмом программной памяти до нескольких килобайт и объёмом ОЗУ от нескольких сотен байт. Использование ОСРВ позволяет использовать больше задач, программа становится легко расширяемой, её легче читать и отлаживать.

ОСРВ для микроконтроллеров Microchip

Главная задача планировщика в ОСРВ – обеспечить, чтобы текущая выполняемая задача имела наивысший приоритет из всех задач, готовых к выполнению. В кооперативных ОСРВ программист должен явно переключать контекст внутри своих задач. Вследствие этого «зависание» одной из задач сказывается на системе в целом. При вытесняющем планировании операционная система использует свой таймер и прерывание, т.е. «зависание» какой-либо задачи не должно повредить работе остальных задач, если они независимы друг от друга и задача с высшим приоритетом не попала в бесконечный цикл.

Приоритетный планировщик задач требует наличия в процессоре программно доступного стека задач для его сохранения при переключении. Такой планировщик труднореализуем и неэффективен в 8-битных микроконтроллерах, которые имеют программно доступный стек (например, Atmel), а на микроконтроллерах Microchip PIC16, имеющих только аппаратный стек (инструкции *call-return*), вообще нельзя реализовать (аппаратно) сохранение контекста задачи.

Наиболее известными ОС реального времени для микроконтроллеров Microchip PIC16 являются Salvo [1], OSA [2] и PICos18.

Salvo – кооперативная многозадачная ОСРВ с поддержкой приоритетов, семафоров, очередей сообщений и функций таймеров. Salvo не использует стек общего назначения (на уровне инструкций *PUSH/POP*). Переключение контекста производится только самим пользователем и на уровне задачи. Переключение задачи из подпрограммы внутри задачи приведёт к непредсказуемым последствиям.

OSA – это кооперативная многозадачная ОСРВ для микроконтроллеров. Задачами в OSA являются обычные функции. Тело функции должно содержать бесконечный цикл, внутри которого должен быть хотя бы один вызов сервиса переключения задач (иначе остальные задачи не получат управления).

PICos18 – вытесняющая, основанная на приоритетах ОСРВ для микроконтроллеров Microchip PIC18, использующая стандарт OSEK и стек для сохранения контекстов задач. Такая ОСРВ безопаснее в плане переключения задач, но даже здесь существует вероятность того, что попавшая в бесконечный цикл задача с высшим приоритетом заблокирует выполнение всех остальных задач в системе.

Таким образом, многозадачные приложения для микроконтроллера, работающие в режиме жёсткого реального времени, требуют наличия планировщика задач, минимального по объёму кода и максимального по быстродействию, устойчивого к зависанию задачи и гарантирующего предсказуемое время отклика даже при наивысшей загрузке системы.

Теория алгоритма RMS

Алгоритм RMS (Rate Monotonic Scheduling – планирование монотонной частоты) разработан в 1973 г. [4] для применения в системах жёсткого реального времени. Алгоритм использует статические приоритеты для задач (static priorities), которые назначаются каждой задаче на этапе компиляции приложения. RMS позволяет определить, может ли вообще управляться планировщиком данный набор задач, т.е. каждая задача из набора будет выполнена в свой срок (deadline) даже при наихудших условиях. Таким образом, система будет иметь предсказуемое поведение. Данный алгоритм применяется в таких операционных системах жёсткого реального времени, как RTEMS и Deos.

Каждая задача в системе имеет три характеристики: время выполнения (execution time), период (period) и конечный срок (deadline). Период задачи – интервал времени между двумя её успешными выполнениями. Приоритет назначается каждой задаче в зависимости от длины её периода: чем короче период, тем выше приоритет задачи. Время выполнения – время, в течение которого задача получает доступ к процессору за один период. Конечный срок – время, к которому задача должна быть выполнена обязательно. Если задача не выполняется к конечному сроку, то планировщик дол-

жен прервать её, сообщить об ошибке и переключиться на другую задачу, чтобы остальные задачи были выполнены к своим конечным срокам. Обычно конечный срок у задачи совпадает с её периодом. Описанные выше временные характеристики задачи показаны на рисунке 1.

Показано [4], что RMS является оптимальным алгоритмом для планирования независимых периодических задач на одном процессоре; оптимальным в том смысле, что если имеется набор задач, каждая из которых может быть выполнена в свой срок, RMS осуществит планирование этого набора задач. Естественно, не все наборы задач могут быть выполнены в свой срок. Простой пример: задача 1 с периодом 20 и временем выполнения 10 и задача 2 с периодом 10 и временем выполнения 6 потребуют более чем 100-% использования процессора. Ключевым параметром для RMS является использование процессора – сумма времени выполнения задач, поделенная на период для каждой задачи:

$$U = \sum_i \frac{c_i}{p_i}.$$

Очевидно, никакой алгоритм не обеспечит успешное планирование для $U > 1$. Главное правило алгоритма RMS состоит в том, что он может успешно планировать набор из n задач, если

$$U < n(2^{1/n} - 1).$$

Для одной задачи ($n = 1$) граница для U равна 100%, для двух задач 83%, и т.д. При неограниченном увеличении числа задач верхняя граница для U стремится к 69%:

$$\lim_{n\to\infty} n\left(2^{1/n} - 1\right) = \log 2 = 0{,}69.$$

Таким образом, если набор задач использует процессор менее чем на 69%, RMS гарантирует успешное планирование этого набора. Если же уровень использования процессора превышает 69%, то RMS также может планировать данный набор задач, но без гарантии.

Некоторые приложения имеют задачи как с жёсткими (hard deadlines), так и с мягкими сроками выполнения (soft deadlines). Набор задач с жёсткими сроками выполнения является критичным и может планироваться с помощью RMS, при этом некритичные задачи не будут выполняться при полной загрузке системы. Это достигается тем, что задача мягкого реального времени с наивысшим приоритетом будет иметь приоритет ниже, чем задача жёсткого реального времени с самым низким приоритетом. Хотя RMS может быть использован и для того, чтобы назначить периоды и приоритеты задачам мягкого времени, это не является необходимым. По сути в этом случае RMS может гарантировать только выполнение задач жёсткого времени. Для них RMS позволяет определить гарантированное время ответа, – даже при наихудших условиях оно будет меньше конечного срока для этой задачи.

Ядро расширения монотонной частоты RMEX

В качестве примера использования алгоритма RMS для планирования в системе жёсткого реального времени рассмотрим работу расширения монотонной частоты RMEX (Rate Monotonic EXtension) для микроконтроллера Microchip PIC18F2550 [6]. Расширение монотонной частоты RMEX берёт на себя функции планирования задач жёсткого реального времени, управления прерываниями и возможности подключения задачи мягкого реального времени. Микроконтроллер имеет раздельную память программы (флэш) 32 Кб и данных (ОЗУ) 2048 байт, 31-уровневый стек, один 8-битный таймер и три 16-битных таймера.

Приложение будет работать следующим образом. Все задачи разделяются на задачи, работающие в жёстком реальном времени (планируются при помощи алгоритма RMS), и задачи мягкого времени (планируются при помощи кооперативной многозадачности без использования таймера в фоновом режиме, прерывания выполняются на переднем плане). Ядро RMEX написано на языке С и состоит из исходного и заголовочного файлов (*rmex.c*, *rmex.h*), которые включаются в компоновку вместе с программой пользователя. Задачи пользователя представляют собой простые функции. Написание приложения сводится к разделению приложения на задачи, деление этих задач на критичные и некритичные по времени наборы, созданию таблицы конфигурации планировщика.

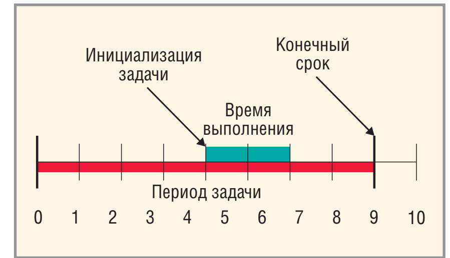

Рис. 1. Временные характеристики задачи

В отдельном заголовочном файле конфигурации планировщика указатели на задачи жёсткого реального времени заносятся в таблицу *RMEX_TASKS*, некритичные по времени задачи вызываются в функции *SOFT_TASK*. В простейшем варианте некритичные по времени задачи выполняются в суперцикле. Для задач реального времени информация о периоде и времени выполнения для каждой задачи заносится в таблицу *RMEX_TCB*. Структура *RMEX_TCB* состоит из следующих полей: период задачи; время, прошедшее с начала периода; максимальное время выполнения задачи; текущее время выполнения задачи:

```
struct {
unsigned char period_time;
unsigned char period_elapsed;
unsigned char exec_time;
unsigned char exec_elapsed;
} RMEX_TCB[CONFIGURE_RMEX_TASKS]
```

Если конечное время у задачи больше её периода, то необходимо взять за период задачи срок её выполнения, а если время выполнения задачи варьируется, то следует взять время при наихудших условиях.

Алгоритм RMS предполагает вытеснение задач, т.е. необходимо сохранять контекст задач (регистры *WREG, STATUS, BSR, FSR1, FSR2, STKPTR* и стек) при вытеснении задачи. Один адрес в стеке микроконтроллера занимает 3 байта памяти, так что для сохранения, например, 10 адресов возврата в стеке понадобится 30 байт плюс 6 байт для регистров. При этом вполне вероятно, что задача мягкого реального времени будет постоянно вытесняться и работа планировщика будет замедляться сохранением стека фоновой задачи.

Для ускорения работы планировщика кооперативные задачи и задачи монотонной частоты используют разные области стека микроконтроллера, благодаря чему планировщику не прихо-

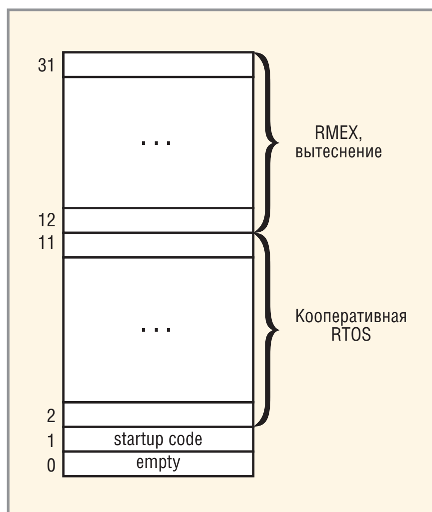

Рис. 2. Использование стека задачами реального времени и кооперативной ОС

дится постоянно тратить время на сохранение стека кооперативных задач. Отведём для кооперативных задач адреса в стеке с 2 по 11, а для задач монотонной частоты – от 12 и выше. Таким образом, при вытеснении кооперативной задачи задачей монотонной частоты с новым периодом планировщик должен сохранить только её регистры, установить указатель стека на 12-й адрес в стеке и начать выполнение новой задачи. Схема использования стека задачами приведена на рисунке 2.

Рассмотрим алгоритм работы данного расширения в микроконтроллере. В начале функции *main* выполняется инициализация всех данных программы пользователя и модулей микроконтроллера, после чего устанавливается прерывание для системного таймера, освобождается место в стеке для фоновой задачи и происходит переключение на ядро RMEX, т.е. запускается многозадачность. При получении управления планировщик перебирает все задачи монотонной частоты, начиная с нулевой (задачи размещены в таблице по возрастанию приоритетов для ускорения поиска), и передаёт управление первой готовой к выполнению задаче. При этом задача может быть начата заново (новый период для задачи) или возобновлена (восстановление контекста). Через заданный квант времени системный таймер сгенерирует прерывание, и вновь произойдёт вызов планировщика. Если текущая задача исчерпала свой лимит времени, то её выполнение прекращается, освобождается место в стеке, и планировщик начинает поиск новой готовой к выполнению задачи монотонной частоты. Если таковая отсутствует, то происходит возврат к кооперативному набору задач. Если же текущая задача не исчерпала лимит времени выполнения, то проверяется наличие готовых к выполнению задач с периодом меньше, чем у текущей задачи, и если таковые находятся, происходит вытеснение. В противном случае прерванная задача продолжает выполняться дальше.

Структура для сохранения контекста задач монотонной частоты имеет следующий вид:

```
struct context_t {
// 3 байта адреса в стеке
unsigned char tosu_b[STACK_DEEP];
unsigned char tosh_b[STACK_DEEP];
unsigned char tosl_b[STACK_DEEP];
// указатель стека STKPTR
unsigned char stk_b;
// регистр статуса
unsigned char status_b;
// рабочий регистр
unsigned char wreg_b;
// регистр банка памяти
unsigned char bsr_b;
// регистры указатели
unsigned char fsr1_b;
unsigned char fsr2_b;
}
_RMEX_CONTEXT[CONFIGURE_RMEX_TASK
S];
```

Параметр *STACK_DEEP* характеризует максимально допустимую глубину стека для задачи, а *CONFIGURE_RMEX_TASKS* – число задач монотонной частоты. Важно, чтобы задача не превышала максимально допустимую глубину стека, иначе при сохранении её контекста произойдёт потеря адреса возврата.

Нерегулярные задачи с коротким временем ответа

Как было сказано выше, в случае использования RMEX программист разделяет свои задачи на периодические с жёстким конечным сроком и непериодические задачи с мягкими сроками выполнения, для которых быстрое время ответа желательно, но не гарантируется. Существуют задачи, которые не являются периодическими (их планирование при помощи алгоритма RMS не будет эффективным), но вместе с тем требующими конечного времени ответа. Как правило, такие задачи зависят от наступления какого-то внешнего события и являются случайными (sporadic). Использование RMS для такой задачи будет неэффективно, так как чем меньшее время ответа требует данная задачи, тем чаще эта задача будет выполняться, причём до наступления нужного события вообще нет смысла передавать ей управление. Другой вариант – сделать её кооперативной задачей – не подходит по причине потери гарантированного времени отклика. Выходом из данной ситуации является создание ещё одного типа задач жесткого реального времени – нерегулярных задач, создающихся в системе динамически и управляемых прерываниями. Таким образом, нерегулярная задача не будет вхолостую потреблять ресурсы процессора и обеспечит гарантированный быстрый отклик.

Выбор системного таймера

Решение о том, какой конкретно таймер из четырёх возможных использовать в качестве системного таймера, определяется требованиями приложения по использованию этих таймеров.

Таймер может использоваться в качестве модуля захвата/сравнения или модуля ШИМ и, таким образом, не может быть системным. Вторым параметром является требуемая точность кванта времени для задачи. Прерывание системного таймера также должно иметь высокий приоритет (вектор прерывания 0x08), т.е. соответствующий этому прерыванию бит в регистре *IPR1, IPR2* или *INTCON2* должен быть установлен в 1. Большинство инструкций в микроконтроллерах PIC18 выполняются за четыре такта генератора или за один такт инструкции (за исключением инструкций переходов). Таким образом, при использовании внешнего генератора с тактовой частотой 40 МГц микроконтроллер будет работать со скоростью 10 миллионов инструкций в секунду.

Значение таймеров увеличивается при выполнении очередного такта инструкции; для них могут быть выбраны предделитель и постделитель частоты. Например, если предделитель установлен в соотношение 1 : 8, то значение соответствующего ему таймера будет увеличиваться на единицу каждые 8 тактов инструкции. Прерывание у 8-битного таймера бу-

дет сгенерировано при его переполнении с 0xFF в 0x00, у 16-битного – при переполнении с 0xFFFF в 0x0000. Любой таймер, имеющий регистр периода, может генерировать прерывание при совпадении его значения со значением регистра периода, чтобы при его использовании получить более точное значение кванта времени для задачи.

При использовании таймера 0 без предделителя в 16-битном режиме прерывание будет генерироваться примерно раз в 65 535/10 000 000 = = 6,5 мс. При использовании таймера 0 в 8-битном режиме с предделителем 1 : 64 прерывание будет наступать примерно раз в 1,6 мс. Для получения кратного 10 значения кванта времени, например, 1 мс, можно использовать таймер 2 в режиме сравнения с регистром периода *PR2* = 39, предделителем и постделителем, установленными в соотношение1 : 16.

Использование кооперативной ОС как задачи мягкого времени в RMEX

Расширение RMEX представляет собой микроядро, выполняющее функции многозадачности и вытеснения задач. В качестве задачи, получающей управление, когда не активна ни одна из задач жёсткого реального времени, можно использовать ядро кооперативной ОС. При этом программист освобождается от написания кооперативного планировщика и других полезных функций, реализованных в кооперативных ОС. При таком подходе кооперативные задачи сами обрабатывают прерывания, не используемые планировщиком RMEX или его задачами, и на кооперативную ОС накладываются следующие ограничения:

- кооперативная ОС не может блокировать прерывание, используемое системным таймером, или прерывания, используемые задачами жёсткого реального времени. Кооперативная ОС не должна предохранять себя от вытеснения задачами жёсткого времени;
- ядро RMEX и его задачи используют высокий приоритет прерываний, задачи кооперативного ядра – низкий приоритет, таким образом, процедуры обработки прерываний кооперативных задач также будут вытесняться прерываниями и задачами RMEX;
- при большой загруженности системы задачами монотонной частоты будет наблюдаться существенное запаздывание в работе кооперативной ОС.

В качестве примера использования кооперативной ОС возьмем ОСРВ OSA, которая обеспечивает переключение задач, отсчёт времени ожидания, выбор готовой к управлению задачи с наивысшим приоритетом и передачу ей управления, а также функции синхронизации задач (очереди сообщений, семафоры).

В начале функции *main* выполняется вызов функции *OS_Init()* для инициализации OSA, инициализация всех данных программы пользователя и модулей микроконтроллера, создание кооперативных задач для OSA, после чего устанавливается прерывание для системного таймера, освобождается место в стеке для фоновой задачи и происходит переключение на ядро RMEX. Следующей командой после передачи управления ядру RMEX и команд возврата по стеку является вызов функции *OS_Run()*, т.е. планировщика OSA, который получит управление сразу же, как только ни одна из RMEX-задач не останется активной. Последовательность вызова функций приведена ниже:

```
// инициализация OSA
OS_Init() ;
// инициализация данных
…
// создание задач для OSA
OS_Task_Create(0, Task_1);
OS_Task_Create(0, Task_2);
// инициализация системного тай-
мера
RMEX_Start_Timer();
// освобождение места в стеке и
вызов ядра RMEX
RMEX_Start_Shed();
// передача управления ядру OSA
OS_Run();
```

Заключение

Алгоритмы планирования задач эволюционируют от простых (суперцикл, кооперативная многозадачность) к сложным (вытесняющая многозадачность, монотонная частота, монотонная частота с кооперативной многозадачностью и нерегулярными задачами), использующим математические модели. Цель алгоритмов – гарантировать время реакции на события, являющиеся критичными для данного приложения. Для небольших приложений подойдёт и суперцикл; если требуется их дальнейшее расширение, то подойдёт кооперативная ОСРВ OSA или Salvo. Для приложений жёсткого реального времени удобно использовать расширения RMEX, позволяющее интегрировать уже написанные приложения для OSA или Salvo. Выбор – за программистом.

Литература

1. Salvo RTOS User's Manual, v. 3.2.2.
2. OSA, http://wiki.pic24.ru/doku.php/osa/ref/ /intro.
3. MPLAB C18 C Compiler User's Guide.
4. *Liu C.L., Layland J.W.* Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment. J. of the Association of Computing Machinery. January 1973.
5. *Баландин Н., Крапивный А.* Отладка приложений и настройка параметров конфигурации операционной системы реального времени RTEMS. Современная электроника. 2010. № 5.
6. Microchip PIC18F2550 Data Sheet.

Реклама