Операционная система реального времени FX-RTOS

Руководство по эксплуатации

Версия 3.X

Введение

Настоящий документ описывает общую архитектуру ОСРВ FX-RTOS, а также дает краткое описание основных компонентов, входящих в состав ОСРВ. Описание предназначено для разработчиков встраиваемого ПО, использующих ОСРВ FX-RTOS.

Содержание

Терминология

В документе используются следующие термины и сокращения:

ОСРВ – Операционная система реального времени.
Модуль, компонент – набор заголовочных файлов и связанных с ними исходных текстов, которые реализуют некоторую законченную функциональную единицу (например, примитив синхронизации «семафор»).

Интерфейс - совокупность типов данных и прототипов функций, описанных в заголовочном файле, которые могут использоваться в файлах исходных текстов.
Реализация- набор функций и данных, реализующие определенный интерфейс, описанный в соответствующем заголовочном файле.

API – Application program interface. Интерфейс прикладного программирования.

Асинхронное событие – изменение счетчика инструкций, произошедшее не в результате действий приложения (например, аппаратное прерывание).

Поток - последовательность инструкций, которая выполняется независимо от других. Каждый поток имеет свой контекст процессора, включающий счетчик инструкций и указатель стека.

Использование FX-RTOS

Встраиваемые приложения

Встраиваемое приложение - приложение, которое работает внутри какого-либо устройства. В качестве примеров можно привести сотовые телефоны, бытовую технику, устройства для промышленной автоматизации и прочие. Во всех этих устройствах пользователь не взаимодействует напрямую с графическим или текстовым интерфейсом приложения, как это происходит в настольных компьютерах, встраиваемое приложение используется для реализации определенного функционала устройства, в котором оно работает.

Усложнение устройств вызывает соответствующее усложнение встраиваемого ПО: обработка прерываний, многопоточность и прочее. Кроме того, во встраиваемых системах большое разнообразие применяемых процессоров и аппаратных решений, которые препятствуют переносимости уже написанного кода с одной аппаратной платформы на другую.

Для решения этих задач применяются встраиваемые ОСРВ. Использование ОС позволяет абстрагировать используемое аппаратное окружение от приложений, добиться переносимости кода, а также разрабатывать многопоточные встраиваемые приложения. FX-RTOS предназначена для использования во встроенных системах, работающих в реальном времени (то есть, на ПО таких систем налагаются также ограничения по задержкам и времени работы). Также возможно использование отдельных компонентов в системах, которые не нуждаются в ОС.

Поддерживаемые архитектуры процессоров

Все компоненты в FX-RTOS, кроме самых низкоуровневых, являются кроссплатформенными и написаны на языке Си. Поскольку интерфейсы платформы просты и определены, портирование на новую процессорную архитектуру может быть выполнено максимально быстро. Таким образом, ОС может быть легко использована в нестандартных процессорах или контроллерах (например, реализованных в ПЛИС) с разной шириной шины и порядком байтов.

Список поддерживаемых архитектур включает ARM Cortex-M, ARM Cortex-A, RISC-V, Эльбрус 2000, x86_64, MIPS, AVR32, MSP430.

Комплект поставки FX-RTOS

ОСРВ реализована в виде статической библиотеки С. В результате сборки получаются два файла:

1. Общий заголовочный файл, который должен быть включен во все пользовательские файлы, использующие функции FX-RTOS.
2. Статическая библиотека, в которой находится реализация функций ОС.

Библиотека должна быть статически скомпонована с программой, для получения загружаемого образа ОС, как показано на Рис. 1:

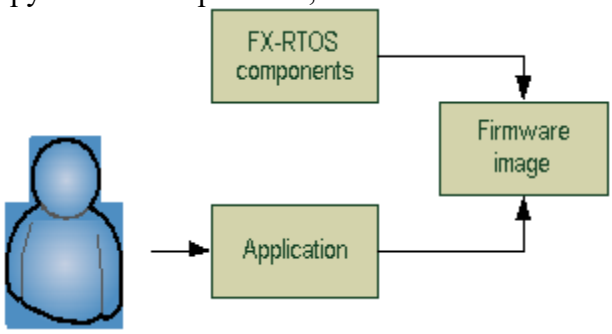


Рис. 1 Использование FX-RTOS.

FX-RTOS поставляется в виде архива (zip-файла), который содержит:

- Скомпилированную библиотеку, содержащую ОС - **fxrtos.a** (расширение файла может различаться для разных платформ и инструментов)
- Заголовочный файл библиотеки - **FXRTOS.h**, исходные тексты ОС для данной конфигурации
- Сценарий сборки **build.bat**, предназначенный для получения библиотеки из исходных текстов с использованием инструмента, для которого предназначен данный вариант ОС (например, IAR Embedded Workbench, Keil uVision, GNU toolchain и другие).

В комплект также могут входить демо-проект с использованием указанного инструмента и файл **readme.txt**, в котором содержатся указания для разработки приложений, настройки аппаратного обеспечения и т.д.

В большинстве случаев ОС используется в виде библиотеки, использование FX-RTOS в виде исходных текстов предпочтительно в тех случаях, когда требуется изменение параметров компиляции либо внесения изменений в FX-RTOS.

Конфигурационные опции

Из конфигурируемости системы (и наличия, возможно, нескольких реализаций какой-либо функциональности) следует возможность построения разных ОС с разными свойствами. При изменении конфигурации архитектура системы может варьироваться. Набор опций доступных для конфигурирования также может различаться для различных конфигураций ОС.

Опции содержатся в конфигурационном заголовочном файле **CFG_OPTIONS.h**. При изменении опций, библиотеку ОС требуется перекомпилировать.

Примеры опций, настраиваемых в большинстве конфигураций FX-RTOS:

- LANG_ASSERT_ERROR_CHECKING_TYPE - Выбор механизма обработки ошибок. Возможные значения перечислены в разделе "Обработка ошибок".
- FX_SCHED_ALG_PRIO_NUM - Количество доступных уровней приоритетов потоков.
- FX_TIMER_THREAD_PRIO - Приоритет потока, обрабатывающего программные таймеры.
- FX_TIMER_THREAD_STACK_SIZE - Размер стека потока, обрабатывающего таймеры.

Начало работы

Встраиваемое ПО обычно разрабатывается с участием т.н. инструментального ПК. Это компьютер, на котором разрабатывается и отлаживается ПО, которое затем загружается в целевое устройство, которым оно должно управлять. Требования к инструментальному ПК обычно указываются в документации к используемому инструменту для разработки встраиваемого ПО. После распаковки архива содержащего ОС, следует добавить библиотеку ОС и заголовочный файл в проект используемого для разработки ПО инструмента, либо использовать поставляемый демо-проект в качестве основы для разработки нового приложения. Для информации о том, как создать новый проект и изменять его параметры, обратитесь к документации производителя инструмента.

Обработка ошибок

Все функции API FX-RTOS поддерживают два способа обработки ошибок (определяемые с помощью конфигурационных опций):

- Классическая (опция LANG_ASSERT_ERROR_CHECKING_TYPE == 1)
- Централизованная (опция LANG_ASSERT_ERROR_CHECKING_TYPE == 2)

Классическая обработка ошибок работает по обычным правилам языка C: функции возвращают статус операции, который должен быть проанализирован внешним кодом. Данный подход является стандартом де-факто и используется по умолчанию.

Альтернативный метод – централизованная обработка ошибок, которая имеет некоторые общие черты с механизмом исключений, имеющемся в высокоуровневых языках. В этом случае, вместо возврата статуса операции, в случае возникновения ошибки вызывается пользовательская функция **fx_error_catch**, в которую передается информация о возникшей ошибке. Функция имеет следующий прототип:

```
void fx_error_catch(const char* func, const char* err_str, int err_code);
```

В нее передается имя функции, в которой возникла ошибка, код ошибки в строковом представлении, а также числовой код ошибки.

При этом возврат из этой функции-обработчика не предполагается (то есть ОС при обнаружении ошибки прекращает свою работу). Данный метод удобно использовать при отладке и прототипировании, т.к. не требуется писать код анализа статусов возврата всех вызываемых функций, а также (в отличие от классического метода) возникшие ошибки не могут быть проигнорированы.

Наконец, проверку ошибок можно отключить вовсе, что увеличивает производительность (опция LANG_ASSERT_ERROR_CHECKING_TYPE == 0).

Рекомендуется использовать централизованную обработку ошибок при прототипировании и начальной отладке, классическую схему для надежных систем, некритичных к производительности, и отключение проверок для полностью отлаженных систем, для которых не требуется проверка на ошибки во время выполнения.

Важно отметить, что численное значение кода ошибки может меняться в зависимости от конфигурации ОС, поэтому приложения должны использовать для анализа статуса операции только символьные значения, определенные в документации.

Программирование приложений

В зависимости от конфигурации, могут использоваться различные модели инициализации и запуска ОС. Типичным, для основанных на микроконтроллерах встроенных систем, является случай, когда вся первичная инициализация и настройка аппаратного обеспечения выполняется пользователем, затем создаются исполняемые сущности ОС (например, потоки), после чего управление передается ОСРВ.

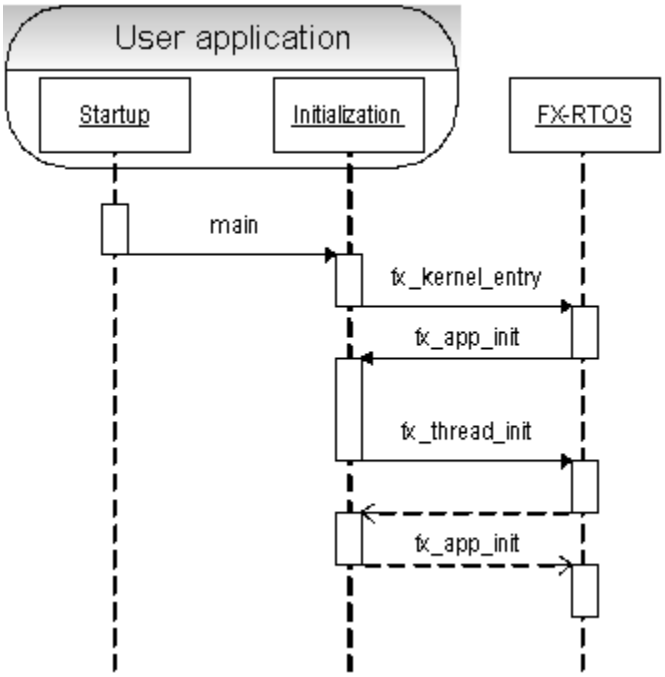


Рис. 2 Процесс инициализации приложения FX-RTOS

На Рис. 2 показан типичный процесс инициализации. Первым получает управление модуль startup, обычно он предоставляется производителем оборудования и содержит таблицу векторов прерываний, код поддержки времени исполнения (инициализация статических данных, перенос изменяемых данных из энергонезависимой памяти в оперативную и т.д.). После выполнения этих действий получает управление код, написанный пользователем (функция main или ее аналог). Затем, после настройки оборудования, указателя стека и прочих действий по инициализации приложения (но до использования сервисов ОС) пользователь вызывает функцию fx_kernel_entry и управление получает ОС. Ядро инициализирует свои структуры данных, после чего вызывает функцию fx_app_init, которую также должен предоставить пользователь. Эта функция вызывается, когда ОС полностью инициализирована и возможно использование ее API. Пользователь внутри fx_app_init создает все необходимые для работы объекты, потоки, примитивы синхронизации и т.д., после чего функция fx_app_init возвращает управление обратно в библиотеку FX-RTOS. После этого происходит запуск планировщика и управление получает наиболее приоритетный поток созданный пользователем.

Таким образом, процесс создания приложения включает в себя следующие шаги:

- Создание функции main и вызов в ней функции fx_kernel_entry
- Создание функции fx_app_init
- Создание в fx_app_init всех необходимых приложению объектов

Следует обратить особое внимание на то, что функция fx_kernel_entry не возвращает управление! Поэтому вся необходимая инициализация оборудования должна быть добавлена в функцию main до вызова fx_kernel_entry.

Поскольку функция fx_app_init выполняется в контексте потока простоя с наименьшим приоритетом, этот поток не может блокироваться, поэтому запрещается использование блокирующих функций в контексте инициализации. Также следует обратить внимание на то, что при создании потоков в процессе инициализации, если эти потоки создаются не в приостановленном состоянии, они сразу же после создания вытесняют поток инициализации. При необходимости создания нескольких потоков за одну неделимую операцию, следует воспользоваться функциями блокирования планировщика.

```
fx_sched_state_t prev_sched_state;
fx_sched_lock(&prev_sched_state);
// Создание нескольких потоков готовых к выполнению...
fx_sched_unlock(prev_sched_state);
```

Сборка ОС из исходных текстов

Несмотря на то, что FX-RTOS рекомендуется использовать в виде библиотеки (это сокращает время компиляции, не требует указания директорий для заголовочных файлов, и т.д.), в отдельных случаях может потребоваться сборка ОС из исходных текстов, например, при необходимости изменения настроек компилятора, либо внесения изменений во внутренние компоненты FX-RTOS.

Возможны различные сценарии использования ОС. При использовании интегрированной среды разработки (IDE), исходные тексты ОС могут быть добавлены непосредственно в IDE и компилироваться вместе с приложением. В этом случае необходимо внести изменения в настройки среды, для возможности компиляции ОС: FX-RTOS использует макрос FX_INTERFACE для включения заголовочных файлов, в типичной поставке (в виде плоского набора файлов без подпапок) этот макрос должен быть определен как:

```
#define FX_INTERFACE( i ) <i.h>
```

Если используемая IDE позволяет указывать макросы с помощью командной строки компилятора (например, IAR Embedded Workbench), этот макрос может быть определен непосредственно в настройках проекта. В случае, если среда не позволяет определять подобные макросы с помощью командной строки должен использоваться входящий в комплект исходных текстов файл includes.inc, который должен быть принудительно (с помощью ключа компилятора) включен во все компилируемые файлы ОС.

FX-RTOS использует также макрос FX_METADATA для хранения внутренних метаданных в исходных текстах. Эти данные не оказывают никакого влияния на процесс компиляции и должны игнорироваться: для этого необходимо определить макрос FX_METADATA(x) с единственным аргументом как пустой макрос.

```
#define FX_METADATA( x )
```

Если ОС необходимо перекомпилировать один раз (для получения библиотеки), и нет необходимости добавлять ее исходные тексты в используемую IDE, могут использоваться сценарии сборки, которые входят в комплект поставки.

Обзор архитектуры

Компоненты FX-RTOS

Как и многие ОС, архитектура FX-RTOS состоит из двух основных компонентов: слоя абстракции оборудования (Hardware abstraction layer, HAL) и собственно ядра, которое реализует высокоуровневые объекты и функции для приложений, пользуясь универсальным интерфейсом HAL. Логическая структура ОС показана на на Рис. 3.

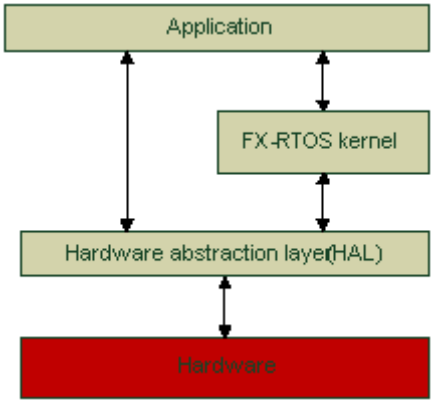


Рис. 3 Логическая структура FX-RTOS

В то же время, в отличие от других ОС, FX-RTOS имеет компонентную архитектуру и внутри самого ядра и HAL, что позволяет добиться идеального соответствия функционала ОС требованиям конкретного встраиваемого приложения.

Поскольку интерфейс, предоставляемый HAL является универсальным и общим для всех поддерживаемых процессоров, некоторые приложения могут быть написаны с использованием только HAL, особенно, если они должны работать в условиях ограниченных ресурсов, либо если интерфейсы ядра ОС обладают избыточной функциональностью.

Объекты ядра

Потоки и объекты синхронизации

После завершения инициализации, рассмотренной в разделе Программирование приложений, ОС начинает выполнять наиболее приоритетный поток, созданный пользователем (если он готов к выполнению). Это значит, что в процедуре инициализации должен быть создан хотя бы один пользовательский поток.

Потоки синхронизируются для доступа к разделяемым данным и областям памяти с помощью предоставляемых объектов синхронизации. Если поток пытается получить доступ к разделяемому ресурсу, который заблокирован другим потоком, он переходит в

состояние ожидания: связанный с ней контекст процессора (состояние его регистров) сохраняется, а процессорное время отдается другому готовому к выполнению потоку.

Кроме того, потоки применяются как обработчики аппаратных прерываний: первичная процедура-обработчик (рассматриваются далее в этом разделе) переводит указанный поток в состояние готовности, после чего дальнейшая обработка прерываний происходит в этом потоке.

Для потоков доступны различные алгоритмы планировщика (алгоритмы выбора для исполнения), они рассматриваются подробно в соответствующем разделе.

Прерывания

Обработка прерываний является одной из важейших задач RTOS, без прерываний трудно обеспечить приемлемое время реакции на внешние события (по отношению к встраиваемой системе).

Приложение FX-RTOS обрабатывает прерывания с помощью установки пользовательских обработчиков, для этого предоставляется специальное API, которое подробно рассматривается в разделе посвященном прерываниям. В большинстве конфигураций ОС, прерывания могут быть прерваны другими (более высокоприоритетными) прерываниями.

В обработчиках запрещено использование блокирующих функций, поэтому обычно обработка прерывания включает в себя несколько этапов. После возникновения прерывания, его обработчик активирует поток либо обработчик событий, для выполнения там дальнейшей работы.

Отложенные процедуры

Во время работы обработчика прерываний, все менее приоритетные обработчики запрещены, поэтому время, проводимое в обработчике прерываний должно быть сведено к минимуму. Для этого используются т.н. "отложенные процедуры". Как следует из названия, эти процедуры используются для того, чтобы дать возможность обработчику прерывания "отложить" некоторую работу, после чего обработчик должен быть завершен, а все менее приоритетные прерывания разрешены.

Отложенные процедуры используются только в сегментированной схеме синхронизации (обсуждается в разделе Схема синхронизации).

Таймеры

Аппаратное обеспечение имеет ограниченное количество таймеров, которых может быть недостаточно для нужд приложения. Помимо этого, таймеры используются также для реализации таймаутов, квантов и т.д. Поэтому ОС реализует программные таймеры, на основе аппаратных. Таймер представлен объектом (структурой) и функцией, которая должна вызываться как результат срабатывания таймера (которы может быть как одиночным, так и периодическим). Эта функция похожа на обработчик прерывания (так же не допускает блокирующих вызовов внутри себя), хотя ее фактическое окружение может

меняться в зависимости от конфигурации. При программировании приложений, следует считать, что окружение обработчика таймера такое же, как окружение DPC-процедур.

Удаление объектов ядра

Ядро ОС не имеет механизма подсчета ссылок на объекты, поэтому управление памятью, предоставляемой под системные объекты (потоки, объекты синхронизации и прочие) полностью управляется пользователем. При удалении примитивов синхронизации, все ожидатели оповещаются с соответствующим статусом ошибки, необходимо гарантировать, что объект не будет использоваться позднее (до его повторной инициализации). Для достижения этих гарантий можно либо использовать логику приложения, либо предоставляемые ядром сервисы DPC (для вызова функции на других процессорах) и барьеров, для синхронизации процесса удаления.

Описание файлов исходных текстов

В зависимости от конфигурации, количество и названия файлов исходных текстов, входящих в состав ядра, может меняться. Название файла, как правило, состоит из трех частей: префикса, имени компонента и опционального суффикса, используемого, если компонент состоит из нескольких файлов исходных текстов (например, имеет часть написанную на C и ассемблере).

<префикс><*компонент*><суффикс>.h (.c, .S, .asm)
Используются следующие основные префиксы файлов:

Префикс	Описание
hw	Низкоуровневые функции для работы с аппаратурой. Различаются для разных аппаратных платформ.
hal	Компоненты HAL, являются наиболее низкоуровневым кроссплатформенным интерфейсом.
lang	Компоненты, расширяющие синтаксис языка C, общеупотребительные макросы, типы и т.д.

fx	Кроссплатформенные высокоуровневые компоненты ядра FX-RTOS.
trace	Компоненты для сбора статистики и анализа производительности. Если эта функциональность отключена, присутствуют в виде файлов-заглушек.
rtl	Библиотека поддержки времени исполнения (runtime library). Реализация структур данных, функций работы с памятью и т.д.
cfg	Конфигурационные компоненты.
ex	Компоненты исполнительной подсистемы (executive subsystem).

Ниже приведено соответствие файлов и реализуемых ими компонентов, которые входят в большинство конфигураций.

Имя файла	**Описание**
hw_pic.c	Функции для работы с контроллером прерываний (Programmable Interrupt controller, PIC).
hw_pit.c	Функции для работы с таймером (Programmable Interval Timer, PIT).

hw_cpu.c	Реализация атомарных операций и специфических для процессора функций, таких как управление энергопотреблением.
hal_cpu_intr.c	Реализация программных и аппаратных прерываний.
hal_cpu_context.c	Управление контекстом процессора.
hal_async.c	Функции для синхронизации и управления SPL (system priority level).
hal_clock.c	Таймер и часы в HAL.
hal_init.c	Модуль первичной инициализации.
lang_asm.h	Определения, используемые в исходных текстах на ассемблере.
lang_types.h	Стандартные типы, используемые ядром и HAL.
fx_rtp.h	Проверки типов объектов во время исполнения (runtime protection).
fx_dpc.c	Реализация отложенных процедур (Deferred procedure call, DPC).
fx_spl.h	Схема синхронизации ядра (унифицированная, сегментированная или упрощенная).
fx_sched.c	Планировщик.

fx_sync.c	Базовые абстракции "ожидатель"-"ожидаемый" для реализации примитивов синхронизации.
fx_thread.c	Реализация потоков. Обычно включает в себя несколько файлов, различающихся суффиксом.
fx_timer.c	Пользовательские таймеры.
fx_timer_internal.c	Реализация таймеров, использующихся системой.
fx_sem.c	Компонент, реализующий примитив синхронизации "семафор".
fx_event.c	Компонент, реализующий примитив синхронизации "событие".
fx_mutex.c	Компонент, реализующий примитив синхронизации "мьютекс".
fx_msgq.c	Компонент, реализующий примитив синхронизации "очередь".
fx_block_pool.c	Компонент, реализующий примитив синхронизации "пул блоков памяти".

Поддержка многопроцессорных систем

В отличие от настольных и серверных систем, во встроенных системах редко требуется балансировка нагрузки на процессоры. Дополнительные процессоры как правило используются для выполнения выделенной им работы, более того, они могут быть лишь частично совместимы с кодом, выполняемым на другом процессоре (например, не иметь доступа к периферии, доступной другому процессору). Поэтому FX-RTOS *не распределяет* пользовательские потоки по процессорам автоматически. Поток всегда по-умолчанию работает на том процессоре, на котором он был создан. При этом, если целевая система позволяет такое использование, балансировка нагрузки может быть реализована пользователем, путем привязки потоков к определенным процессорам с помощью API (если поток выполнялся на одном процессоре, привязка его к другому процессору вызовет миграцию).

Схема синхронизации

Любой код, исполняющийся в FX-RTOS, имеет определенный приоритет, по отношению к асинхронным событиям. Этот приоритет называется «уровень SPL» (system priority level). Количество этих уровней может быть различно для разных систем и конфигураций, главным свойством SPL является то, что код с более высоким SPL может прерывать (в любой момент, асинхронно) код с более низким SPL, но не наоборот. Если во время выполнения кода с высоким SPL возникает асинхронная активность более низкого (либо того же) уровня, она откладывается до того момента, пока SPL не будет понижен. Это накладывает ограничения на возможности использования API в различных контекстах.

Для некоторых асинхронных событий (например, аппаратных прерываний) семантика SPL реализуется аппаратно (низкоприоритетное прерывание не может прерывать высокоприоритетное), для некоторых (например, программных прерываний) – программно.

ОС предоставляет API для управления текущим уровнем SPL. Оно используется для синхронизации в рамках одного процессора (каждый процессор имеет в каждый момент времени свой SPL). С помощью временного повышения уровня, можно запретить определенные асинхронные события. Например, поскольку планировщик FX-RTOS реализован как обработчик программного прерывания, повысив SPL до уровня планировщика можно временно запретить планирование потоков на текущем процессоре.

Среди множества уровней SPL, определяемых HAL (в зависимости от количества прерываний в целевой системе) три уровня имеют специальные названия:

- SPL_SYNC (на этом уровне все прерывания на данном процессоре запрещены).
- SPL_DISPATCH (все аппаратные прерывания разрешены, все программные - запрещены).
- SPL_LOW (на этом уровне все асинхронные активности разрешены).

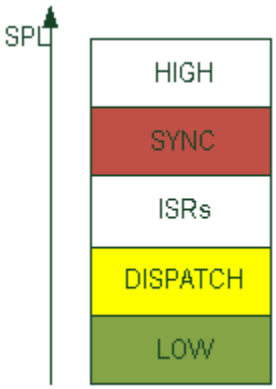


Рис. 4 Уровни SPL

Уровень SPL, закрепленный за каждой исполняемой сущностью ОС (такой как прерывания, DPC, потоки и т.д.), определяет накладываемые на эт сущность ограничения. Например, если сервисы ОС работают на уровне SPL_DISPATCH, то прерывания не могут напрямую пользоваться этими сервисами, т.к. между ними отсутствует синхронизация - прерывания могут прерывать выполнение системных сервисов, и в случае использования этих же сервисов обработчиками, получается, что сервис может прерывать выполнение самого себя.

В большинстве ОС распределение уровней SPL (и определение того, что и на каком уровне может исполняться) является фиксированным и называется "архитектурой прерываний", это исторически сложившееся название. В настоящем документе используется термин "схема синхронизации", т.к. это распределение влияет на только на прерывания, но и на механизмы синхронизации ОС в целом. Каждая схема обладает своими достоинствами и недостатками, поэтому FX-RTOS позволяет конфигурировать используемую схему синхронизации, для оптимального соответствия ОС требованиям приложения.

Из-за возможности конфигурирования схемы синхронизации, может меняться окружение и/или ограничения, распространяющиеся в том числе и на пользовательский код, содержащий обработчики прерываний и DPC, ниже в данном разделе обсуждается каждая из поддерживаемых схем, а также способы написания универсальных приложений, которые не требуется изменять, в случае изменения схемы синхронизации.

Альтернативный подход заключается в том, чтобы выбрать заранее оптимальную для конкретного приложения схему и разрабатывать систему только в соответствии с ней; возможности конфигурирования ОС в этом случае будут ограничены.

Унифицированная схема

Относительно простой и в то же время эффективной является унифицированная схема. Ее название происходит из того, что сервисы ОС можно использовать из обработчиков прерываний так же, как и из пользовательских потоков. За счет этой «унификации» ОС содержит меньшее количество функций и более проста в освоении.

Так как из прерываний могут быть вызваны любые сервисы ОС, в том числе и те, которые работают со структурами данных планировщика, из этого автоматически следует необходимость синхронизации с прерываниями и самого планировщика, то есть планировщик должен выполняться с запрещенными прерываниями, на уровне SPL_SYNC. Сервисы ОС, синхронно вызываемые из пользовательских потоков также должны повышать уровень до SYNC.

Важно отметить, что вызов планировщика происходит с уровня SPL_DISPATCH: при вложении обработчиков, каждый из них может изменить состояние планировщика, при этом нежелательна отработка планировщика после каждого из вложенных прерываний, поэтому планировщик должен инициироваться как программное прерывание уровня SPL_DISPATCH (наименее приоритетное), а после получения управления, повышать уровень до SPL_SYNC (иными словами, запрещать прерывания) и после этого выполнять планирование.

На Рис. 5 показан вызов сервиса ОС (показан как OS) из обработчика аппаратного прерывания (показан как ISR).

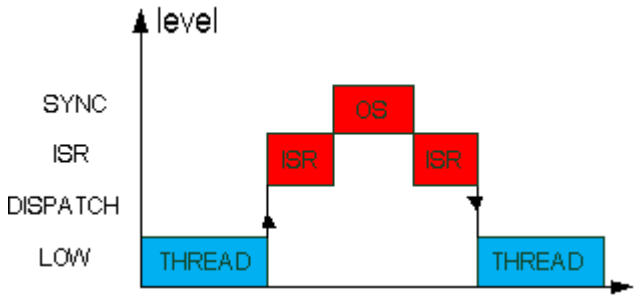


Рис. 5 Вызов сервиса ОС из обработчика прерываний

После передачи управления в обработчик прерывания, он вызывает нужную функцию ОС, которая повышает уровень до SPL_SYNC. Вызов системного сервиса из пользовательского потока выглядит аналогичным образом:

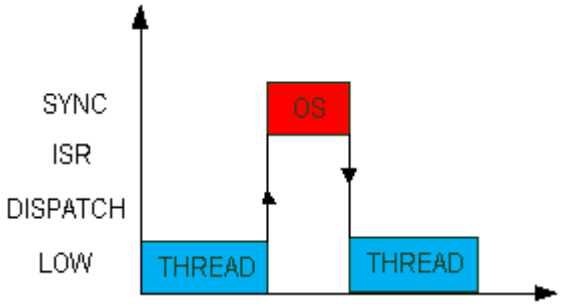


Рис. 6 Вызов сервиса ОС из пользовательского потока

Несмотря на свою простоту, унифицированная схема обладает и недостатками, среди которых – латентность. Выполнение системного сервиса внутри обработчика прерывания может занять относительно много времени, при этом все прерывания, имеющие меньший приоритет остаются запрещенными все это время, то есть увеличивается их латентность и при высокой частоте прерываний, некоторые из них могут быть пропущены.

В то же время эта схема обладает наилучшей производительностью, и если латентность прерываний не является главным приоритетом, то при прочих равных унифицированная схема является предпочитаемой – она обеспечивает как простоту и универсальность сервисов ОС, так и наилучшую производительность.При использовании унифицированной схемы, функции для работы с DPC отображаются на прямой вызов DPC-функции внутри обработчика. Так как все системные сервисы повышают уровень SPL до SPL_SYNC, любой сервис (кроме функций ожидания, вызов которых запрещен везде, кроме потоков) может быть вызван из обработчика. При этом следует учитывать, что DPC-функция (а также обработчик таймера) могут работать на уровне SPL выше SPL_DISPATCH.

Сегментированная схема

Сегментированная схема является более сложной и менее детерминированной, чем унифицированная. Ее название происходит от разделения обработчика прерывания на две части (два «сегмента»), одна из них работает на уровне приоритета, соответствующего аппаратным прерываниям (и может блокировать другие, менее приоритетные, прерывания на время своего выполнения), вторая часть работает на уровне приоритета выше, чем пользовательские потоки, но ниже, чем аппаратные прерывания. В целях упрощения синхронизации, эти «части» обработчиков, называемые «отложенными процедурами» работают на одном уровне SPL с планировщиком и сервисами ОС - на уровне SPL_DISPATCH. При этом, поскольку этот уровень приоритета расположен ниже, чем уровни приоритета аппаратных прерываний, последние остаются разрешенными как во время выполнения планировщика, так и во время выполнения большинства сервисов ОС. За счет этого обеспечивается низкая латентность обработчиков прерываний.

В отличие от унифицированной схемы, в сегментированной схеме не всякий код может быть выполнен синхронно с помощью повышения уровня SPL. Т.к. обработчик прерывания разделен на две части, и вторая часть менее приоритетна, чем первая, этот второй обаботчик не может быть синхронно вызван из первого, и необходимы дополнительные структуры данных, для того чтобы обеспечить выполнение низкоприоритетного обработчика только после выхода из всех высокоприоритетных.

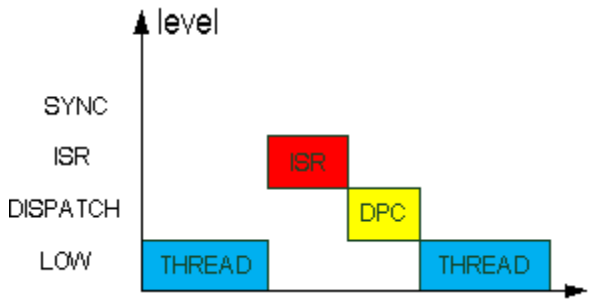


Рис. 7 Разделение обработчика прерываний на два сегмента имеющих разный приоритет

Обработчики прерываний в сегментированной архитектуре являются исключительно простыми и как правило выполняют всего одно действие – запрос отложенной обработки (запрос DPC).

Основную работу по обработке прерывания должна выполнить отложенная процедура. ОС должна обеспечить выполнение этой процедуры на уровне SPL_DISPATCH и только из нее можно пользоваться сервисами ОС (а не напрямую из прерывания, как в унифицированной или упрощенной схеме).

Так как все сервисы ОС в сегментированной схеме выполняются на уровне SPL_DISPATCH, и не пересекаются по данным с обработчиками (которые могут только запрашивать отложенную обработку на уровне SPL_DISPATCH), для синхронизации при выполнении системного сервиса достаточно только повышать уровень до DISPATCH, не запрещая прерывания полностью.

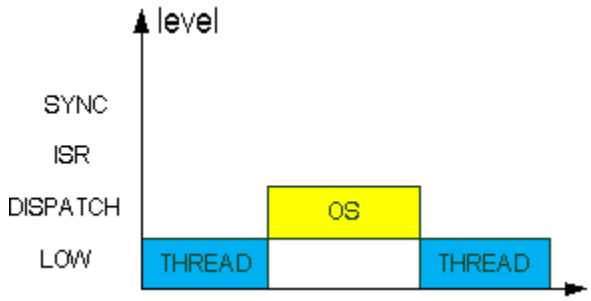


Рис. 8 Вызов системного сервиса из пользовательского потока

Обработчики прерываний могут прерывать выполнение системного сервиса:

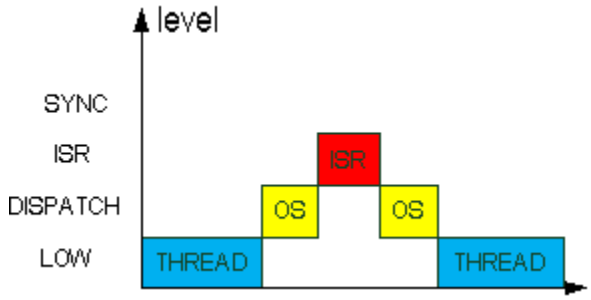


Рис. 9 Прерывание системного сервиса обработчиком прерывания

В целом, сегментированная архитектура является более сложной в реализации и в типичных случаях обладает меньшей производительностью, однако, за счет того, что прерывания остаются разрешенными во время работы почти всех сервисов ОС, латентность прерываний (от возникновения прерывания до входа в обработчик) может оказаться существенно ниже, чем при использовании унифицированной схемы. Помимо возможности обработки прерывания традиционным способом – в потоках, существует также возможность обработки прерывания внутри отложенной процедуры, которая вызывается сразу после выхода из всех обработчиков аппаратных прерываний и не требует участия планировщика, то есть, при определенных условиях, сегментированная архитектура может обработать больше прерываний в единицу времени, чем унифицированная, хотя у данного подхода также есть свои минусы. Отложенные процедуры выполняются в порядке очереди и не имеют приоритета, поэтому их использование для обработки прерываний может быть неприемлемо, а обработка прерываний в потоках может быть более медленной, чем в унифицированной архитектуре за счет наличия дополнительного промежуточного слоя.

Наконец, в случае если обработчик прерываний не требует взаимодействия в сервисами ОС, он может быть обработан полностью в процедуре ISR.

Сегментированная архитектура является более предпочтительной для реализации многопроцессорных систем, поскольку очередь отложенных процедур на каждом процессоре синхронизирована с планировщиком, это дает простой и эффективный асинхронный способ общения с планировщиками остальных процессоров, не сопряженный с защитой глобальных структур данных, поэтому ОС предназначенные для работы на многоядерных и многопроцессорных систем как правило имеют сегментированную схему синхронизации.

В FX-RTOS сегментированная схема также предназначена для использования в конфигурациях, предназначенных для работы в многопроцессорных системах, хотя возможно ее применение и в однопроцессорных. В этом случае, внутри пользовательских обработчиков прерываний запрещены вызовы сервисов ОС, для этого следует использовать DPC, которые выполняются на уровне SPL_DISPATCH. Обработчик таймера также работает на уровне SPL_DISPATCH.

Разработка приложений

Несмотря на то, что функционирование ОС с разными схемами синхронизации принципиально различается, возможно написание встраиваемых приложений, которые без изменений могут работать с любой схемой синхронизации.

Код приложений должен быть написан так, как если бы использовалась сегментированная схема (которая является самой сложной из рассматриваемых, все остальные могут быть получены путем ее упрощения).

Типичный случай использования FX-RTOS с обсуждением особенностей его работы с различными схемами синхронизации приведен ниже (прототипы функций упрощены, возвращаемые значения и аргументы могут отличаться от используемых в FX-RTOS):

```
fx_dpc_t dpc;
fx_sem_t semaphore;
void DPC_func(void *arg);

void ISR_func(void)
{
        fx_dpc_request(&dpc, DPC_func, &semaphore);
}

void DPC_func(void *arg)
{
        fx_sem_t* sem = (fx_sem_t*) arg;
        fx_sem_post(sem);
}

void THREAD_func(void *arg)
{
        ...
        device_request(...);
        status = fx_sem_timedwait(&semaphore, 10);
}
```

В рассматриваемом примере пользовательский поток THREAD_func обращается к некоторому аппаратному устройству с помощью функции device_request и ожидает ответа от устройства в виде ожидания семафора, который устанавливается в активное состояние обработчиком прерывания.

В случае сегментированной схемы, обработчик прерывания, представленный функцией ISR_func работает на некотором уровне ISR (между SPL_SYNC и SPL_DISPATCH), напрямую вызывать функцию fx_sem_post на этом уровне нельзя, поэтому используется очередь DPC: обработчик прерывания запрашивает отложенный вызов функции DPC_func. После выхода из обработчика (и всех вложенных обработчиков), управление получает функция DPC_func, которая и выполняет фактический инкремент семафора. После выполнения DPC, отрабатывает планировщик и пользовательский поток продолжает выполнение.

В случае унифицированной схемы, вызов fx_dpc_request отображается в прямой вызов DPC_func внутри обработчика прерываний, т.к. выполнение fx_sem_post уже допустимо в контексте обработчиков. После этого, по аналогии с сегментированной схемой, также выполняется планировщик и пользовательский поток продолжает выполнение.

Как видно из этого примера, изменение схемы синхронизации происходит прозрачно для приложения, необходимо только учитывать, что окружение DPC-функции может изменяться от прямого вызова в контексте ISR до отложенной обработки на уровне SPL_DISPATCH (поэтому там нельзя использовать функции, которые, например, явно требуют исполнения только на уровне SPL_DISPATCH).

Если изменение схемы синхронизации не требуется, можно явно использовать определенную схему: например, используя только унифицированную схему синхронизации, можно переписать пример следующим образом:

```
fx_sem_t semaphore;

void ISR_func(void)
{
        fx_sem_post(&semaphore);
}

void THREAD_func(void *arg)
{
        ...
        device_request(...);
        status = fx_sem_timedwait(&semaphore, 10);
}
```

При попытке изменения схемы синхронизации на сегментированную, это вызовет ошибку времени выполнения из-за недопустимости вызова fx_sem_post в контексте обработчиков.

Чтобы узнать, какой вариант используется в ваше системе, обратитесь к документации используемой версии FX-RTOS.

Потоки

Поддержка потоков, наряду с прерываниями, также является одной из обязанностей ОСРВ. После завершения инициализации должна быть создана хотя бы один пользовательский поток, в дальнейшем ОС обеспечивает выполнение наиболее приоритетного, готового к выполнению потока.

Каждый поток в каждый момент времени находится в определенном состоянии. Диаграмма состояний и переходов между ними приведена на Рис. 10:

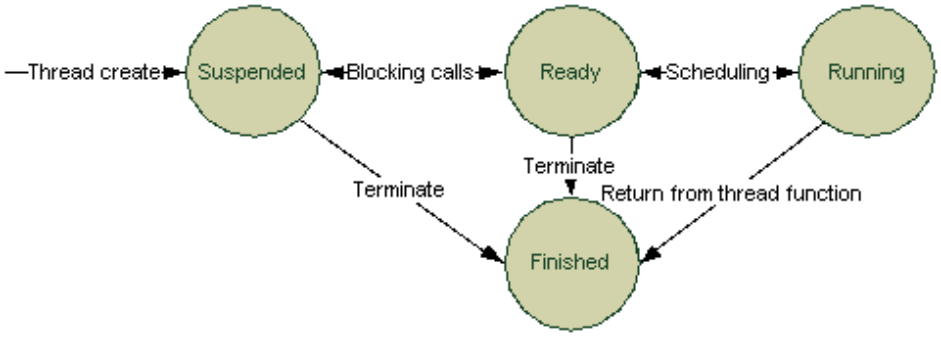


Рис. 10 Диаграмма состояний потока

Приостановленное (suspended) состояние: поток не готов к выполнению из-за ожидания некоторого события, таймаута, либо вследствие явной приостановки с помощью функций API.

Готов к выполнению (ready): поток готов к выполнению, но не выполняется в данный момент из-за вытеснения более приоритетным потоком. Из этого состояния поток может быть выбран планировщиком для исполнения и перейти в состояние выполнения (running).

После завершения потока по любой причине, он переходит в завершенное состояние (finished). Когда поток находится в этом состоянии, он не может использоваться приложением. Для его повторного использования он должна быть удален, а затем инициализирована заново с помощью функции **fx_thread_init**.

Каждый поток FX-RTOS имеет отдельный стек, указываемый при инициализации, который должен иметь размер, достаточный для того чтобы вместить контекст, а также обеспечить выполнение потока в дальнейшем. Недостаточный размер стека приводит к неопределённому поведению.

Объекты синхронизации

Для синхронизации потоков и обмена данными между ними FX-RTOS предоставляет набор объектов синхронизации. Также FX-RTOS расширяет стандартное API ожидания с помощью т.н. отменяющих событий: выход из функции ожидания может происходить как в результате выполнения условия ожидания или таймаута, так и в результате установки в активное состояние отменяющего события, которое передается в

большинство функций ожидания в качестве дополнительного аргумента. Это позволяет легко реализовывать различные сценарии синхронизации (например, завершение всех ожиданий при завершении работы системы) и упрощает программирование.

Объект «событие»

Объект «событие» является широковещательным объектом без побочных эффектов ожидания. Событие может быть в активном или неактивном состоянии. Доступны только «уведомительные» события, то есть их состояние не меняется при активации ожидающих события потоков. Для создания «синхронизирующего события» (называемого также «событием с автоматическим сбросом») следует использовать бинарный семафор.

Поскольку нотификация ожидающих потоков происходит с заблокированным планировщиком, в случае необходимости нотификации большого количества потоков это может вызвать задержки планирования, кратные количеству ожидающих потоков.

Мьютексы и инверсия приоритетов

Как и многие ОСРВ, FX-RTOS содержит реализацию мьютексов, для реализации взаимного исключения. Мьютексы FX-RTOS являются рекурсивными, то есть поток, владеющий мьютексом, может захватить его повторно; для освобождения мьютекса он должна освободить его столько раз, сколько раз мьютекс был захвачен.

Для противодействия инверсии приоритетов, мьютексы FX-RTOS поддерживают потолок приоритета – при захвате мьютекса, поток получает параметры планирования, соответствующие мьютексу (указанные при его создании). Ответственность за их правильную установку несет пользователь. Если параметры при создании мьютекса не были указаны, потолок приоритета отключается и захват мьютекса не влечет изменения приоритета потока, его захватившего.

Семафоры

FX-RTOS поддерживает также стандартный интерфейс для семафоров, с некоторыми расширениями. В частности, допустимо указание максимального значения счетчика семафора, что позволяет в рамках одного API создавать как бинарные, так и обычные семафоры. Семафор поддерживает две основные операции: post и wait, первая увеличивает внутренний счетчик семафора, а вторая пытается его уменьшить. Если счетчик был не равен 0 - поток, вызвавший wait продолжает выполнение. Если же счетчик равен 0, поток блокируется до тех пор, пока другой поток, либо обработчик прерывания или DPC не увеличат счетчик, используя post.

В отличие от мьютексов, которые могут быть переведены в активное состояние только потоком-владельцем, семафор может быть установлен в активное состояние любым потоком, поэтому может использоваться в качестве объекта, сообщающего о некотором событии (например, прерывании).

Пул блоков памяти

В приложениях часто возникает необходимость выделения памяти. К сожалению, выделение блоков произвольного размера сопряжено с большими накладными расходами. FX-RTOS предоставляет сервисы для работы с блоками памяти фиксированного размера. В статически выделенной памяти приложения создается пул блоков, память из которого может быть выделена и освобождена путем использования API.

Память, используемая под пул должна быть выровнена как минимум на размер машинного слова или указателя!

Пул блоков памяти может использоваться совместно с очередью, для реализации обмена сообщениями между потоками, либо потоками и обработчиками прерываний.

Очереди сообщений

Очереди сообщений обычно используются для синхронизации передачи некоторой информации между потоками или потоками и обработчиками прерываний. Размер очереди указывается при инициализации, при этом размер одного сообщения фиксирован и равен размеру указателя (который обычно равен размеру машинного слова процессора). Сообщения, которые меньше либо равны по размеру указателю могут быть переданы по значению, сообщения большего размера следует передавать косвенным образом через пересылку указателя. Сообщения могут помещаться как в начало так и в конец очереди. Как и в случае с пулом блоков памяти, буфер, используемый для хранения сообщений должен быть правильным образом выровнен (на размер указателя).

Планирование очереди ожидания

Каждый нешироковещательный объект (который переводит в активное состояние не все ожидающие его потоки при активации) в FX-RTOS имеет дополнительный параметр – политику планирования очереди ожидания. В настоящее время поддерживаются две основные политики: FIFO и priority.

В первом случае, при наличии нескольких ожидателей и необходимости активировать одного или нескольких из них (не всех) используется дисциплина FIFO: первым будет переведен в активное состояние ожидатель, который раньше начал ожидание. Во втором случае, первым получит управление наиболее приоритетный из ожидателей (внутри группы ожидателей с максимальным приоритетом по-прежнему работает дисциплина FIFO). Политика планирования очереди указывается для каждого объекта индивидуально (при инициализации). Следует также учитывать, что в некоторых реализациях, поиск потока с максимальным приоритетом требует анализа всех ожидающих данного объекта потоков, что может повлечь задержки, зависящие от количества ожидающих потоков.

Примитивы POSIX

FX-RTOS также поддерживает стандартные примитивы POSIX. Среди этих примитивов: барьер, условная переменная, RW-блокировка. Так как многие из этих объектов имеют недетерминированное время выполнения (требуют уведомления более чем одного потока), их применение не рекомендуется в системах жесткого реального времени. Следует использовать эти примитивы для облегчения портирования уже написанных приложений.

Планировщик

Текущая версия FX-RTOS поддерживает три вида планирования потоков: FIFO, Round-robin и Timeslicing. Для хранения готовых к выполнению потоков используется структура, известная как «многоуровневая приоритетная очередь» (priority-based multi-level queue).

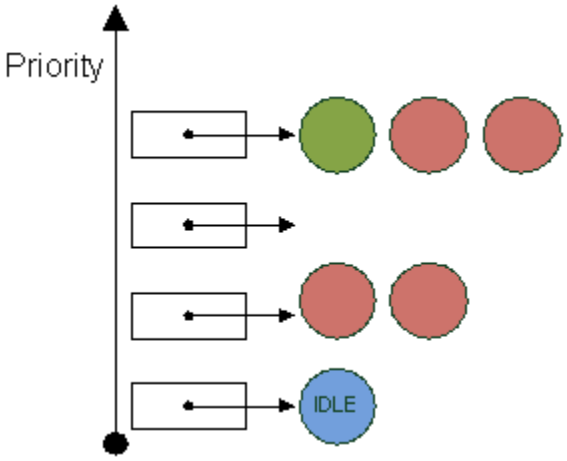


Рис. 11 Структуры данных планировщика.

Для каждого приоритета определена очередь, элемент, находящийся в голове самой приоритетной очереди является активным (показан зеленым цветом). Самая низкоприоритетная очередь содержит один элемент, соответствующий потоку простоя (idle, показан синим цветом). Требование его единственности вытекает из того, что этот поток никогда не блокируется, поэтому при FIFO-политике планирования другие потоки с таким же приоритетом никогда не получат управления.

Для систем с ограниченными ресурсами возможен также вариант планировщика, который не использует списков и позволяет существовать только одному потому заданного приоритета (т.н. bitmap-контейнер). Если не используется кооперативная многозадачность и в системе никогда не бывает двух и более активных потоков заданного приоритета, этот планировщик обеспечивает более высокую производительность.

При FIFO-политике планирования, единожды начав выполнение, поток продолжает работу до тех пор, пока явно не отдаст процессорное время по своей инициативе с помощью вызова блокирующей функции ОС.

Round-robin-планировщик реализует кооперативную многозадачность, в дополнение к FIFO политике: поток может явно отдать процессор другому потоку, имеющему тот же приоритет.

Timeslicing -планирование предполагает, что каждому потоку выделяется квант времени, в течение которого он может занимать процессор, после этого ОС принудительно отдает процессорное время другому потоку, имеющему тот же приоритет.

Round-robin и Timeslicing планирование могут применяться только в конфигурациях, допускающих существование более одного потока каждого приоритета!

Квант обновляется всякий раз, когда поток получает управление после работы планировщика.

Блокировка планировщика

При необходимости выполнения некоторого кода атомарно относительно планирования потоков (с запрещенным вытеснением) следует использовать специальное API для блокировки, предоставляемое планировщиком. В коде это выглядит следующим образом:

```
fx_sched_state_t prev_sched_state;
fx_sched_lock(&prev_sched_state);
//
// Планирование запрещено, выполнение критического кода, не допускающего
// вытеснение.
//
fx_sched_unlock(prev_sched_state);
```

В зависимости от реализации планировщика и используемой схемы синхронизации, код в критическом регионе может быть выполнен на SPL=SPL_DISPATCH или SPL=SPL_SYNC. Определяется также константа **FX_SPL_SCHED_LEVEL**, равная уровню SPL, на котором работает планировщик, что позволяет блокировать его также с помощью функций управления SPL.

Приоритет

Другим важным свойством потока является его приоритет. Он указывается при инициализации потока и может быть впоследствии изменен с помощью вызовов **fx_thread_set_param**. Количество приоритетов определяется конфигурацией ОС и обычно лежит в пределах от 16 до 64. Кроме того, от конфигурации зависит также возможность создания нескольких потоков одного приоритета. Поток имеющая приоритет 0 является наиболее приоритетным. Количество имеющихся приоритетов определяется константой **FX_SCHED_ALG_PRIO_NUM**, при этом значение **FX_SCHED_ALG_PRIO_NUM-1**, является зарезервированным для системного потока простоя. Таким образом, наименее приоритетный поток, которая может быть создан пользовательским приложением имеет приоритет **FX_SCHED_ALG_PRIO_NUM-2**.

Прерывания

В качестве первичных обработчиков прерываний в пакете поддержки (board support package, BSP) устанавливаются обработчики HAL, которые специфичны для конкретного процессора или платформы, обычно они называются как **hal_intr_entry**, но их количество может меняться в зависимости от требований процессора.

После установки обработчиков HAL на аппаратные векторы прерываний, в ответ на любое аппаратное прерывание будет вызываться функция **fx_intr_handler** (функция не имеет аргументов и возвращаемого значения), которая должна быть реализована в пользовательском приложении. Внутри этой функции может быть вызвана функция **hal_intr_get_current_vect**, которая возвращает текущий вектор прерывания, что дает возможность обрабатывать прерывания с произвольными векторами. Данный способ обработки прерываний является рекомендуемым, поскольку обеспечивает кроссплатформенность, хорошее время реакции, а также не требует никаких дополнительных структур данных, кроме таблицы прерываний, требуемой процессором.

Вложение ISR

Во время выполнения обработчика устанавливается соответствующий ему уровень SPL, что автоматически блокирует все прерывания того же, либо более низкого приоритета, а также программные прерывания (т.к. приоритет аппаратных прерываний всегда выше, чем программных), однако более приоритетные прерывания могут прерывать текущий обработчик. Для временного запрета всех прерываний в целях синхронизации следует использовать повышение SPL до уровня SPL_SYNC.

Синхронизация обработчиков прерываний с приложением

Поскольку обработчики работают на повышенном уровне SPL, в них запрещено использование блокирующих функций, а также функций с недетерминированным временем работы. Кроме того, поскольку на некоторых платформах прерывания используют отдельный стек, не рекомендуется выделять на стеке большие объемы данных. В случае необходимости использования каких-либо буферов, рекомендуется использовать выделенный заранее буфер, не связанный со стеком.

Для синхронизации с обработчиками прерываний из приложения следует использовать функции управления SPL, предоставляемые FX-RTOS.

Таймеры

В FX-RTOS поддерживается множество вариантов реализации таймеров, поэтому, как и для прочих компонентов, можно выбрать реализацию, наболее подходящую для требований приложения. В FX-RTOS поддерживаются два типа таймеров: одиночные (one-shot) и периодические. Как следует из названия, одиночный таймер обеспечивает выполнение функции один раз, по истечению указанного интервала времени (абсолютного, отсчитывая от старта системы, либо относительного, от момента вызова функции). Периодический таймер обеспечивает вызов указанной функции многократно, с указанным периодом. Таймаут таймера, а также период, указывается в **тиках**. Тик - интервал времени между прерываниями аппаратного таймера. Фактическая частота таймера зависит от аппаратной платформы. Ниже приведены возможные варианты с кратким описанием.

Таймер считается сработавшим в момент, когда системный счетчик тиков стал равен таймауту таймера. Таким образом, при установке таймера, например, на 1 тик, фактическая задержка может составить от 0 до 1 тика (в случае, если следующее таймерное прерывание возникнет сразу после установки таймера, задержка будет равна 0).

Реализации таймеров

Легковесный таймер

Легковесные таймеры используют единый, упорядоченный по таймауту, список всех таймеров в системе, обработка которого происходит в прерывании таймера. Поскольку работа с таким списком порождает задержки порядка O(N) при вставке нового таймера в список, использовать эту реализацию в системах жёсткого реального времени не рекомендуется. Кроме того, легковесные таймеры не могут использоваться с сегментированной архитектурой прерываний (подразумевающей использование DPC), а также в многопроцессорных системах. При небольшом количестве активных таймеров, эта реализация обеспечивает наилучшую производительность.

Обработка таймеров в потоке

В данной реализации используется скрытый системный поток, который обрабатывает таймеры (его приоритет указывается пользователем). Данная реализация использует списки с хэшированием по таймауту. Существует, однако, несколько недостатков, присущих данной архитектуре: среди них можно отметить то, что использование таймеров в потоках, имеющих приоритет выше, чем у потока, обслуживающей таймеры, может привести к инверсии приоритета за счет вытеснения таймерного потока, кроме того, для таймерного потока требуется также отдельный стек, что влечет дополнительный расход памяти. Данная реализация доступна как для однопроцессорных так и для многопроцессорных систем.

Изоляция и защита памяти

В дополнение к описанным возможностям, FX-RTOS может поддерживать механизмы защиты памяти и изоляции приложений от ядра ОС и друг от друга. В этом случае приложение состоит из привилегированной части (называемой kernel application) и опциональных непривилегированных частей, которые компилируются отдельно от ядра, и размещаются в отдельных бинарных файлах. Использование этих возможностей не является обязательным. Так, приложение может состоять только из привилегированной части, которая разрабатывается по правилам описанным выше: компилируется и статически компонуется с ядром и работает в привилегированном режиме процессора. Привилегированное приложение может инициализировать и запустить один или более непривилегированных приложений, для чего используется расширенное API.

Использование защиты памяти предполагает использование специальных апапратных средств: устройства управления памятью (Memory Management Unit, MMU) или устройства защиты памяти (Memory Protection Unit, MPU). FX-RTOS может работать с обоими типами устройств.

Интерфейс прикладного программирования

Для программирования непривилегированных приложения используется подмножество основного API FX-RTOS, с исключением из него потенциально опасных функций, которые могут повлиять на стабильность работы системы.

Приложения обращаются к библиотеке, которая содержит функции API. В отличие от статической компоновки с ядром, эта библиотека содержит обертки для системных вызовов (Рис. 12).

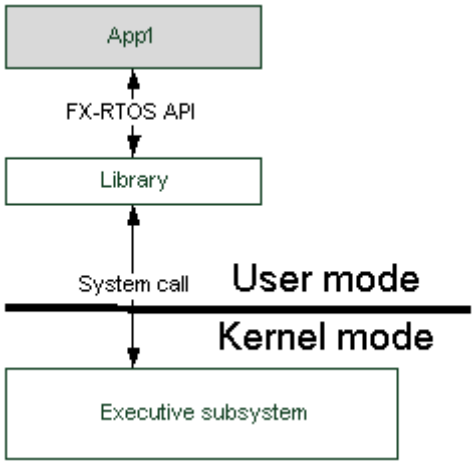


Рис. 12 Механизм вызова функций API из непривилегированного приложения

Эта библиотека должна быть скомпонована с каждым приложением.

Исполнительная подсистема

Для реализации этого функционала используется компонент ядра, называемый исполнительной подсистемой. Этот компонент обслуживает непривилегированные

приложения, а также предоставляет API для управления ими, которое используется привилегированным приложением в режиме ядра (Рис. 13).

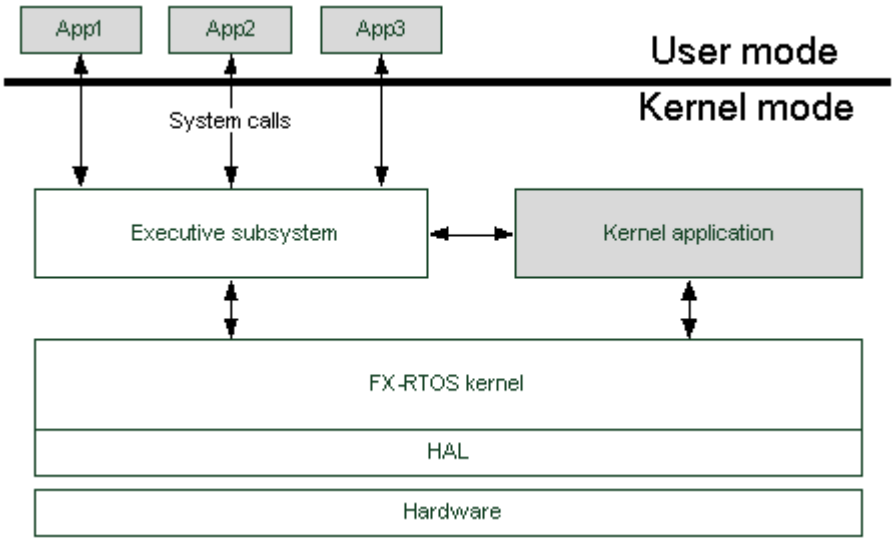


Рис. 13 Структура системы с поддержкой защиты памяти

Непривилегированные приложения используют механизм системных вызовов, через который они запрашивают выполнение функций ядра. При этом производится контроль прав доступа, подсчет ссылок на объекты и т.д. Это позволяет гарантировать, что в результате любых, даже потенциально злонамеренных действий со стороны непривилегированных приложений само ядро, привилегированное приложение, а также другие модули продолжат свою работу и стабильность системы, а также ее временные характеристики не пострадают.

Непривилегированные приложения

В отличие от систем общего назначения, которые динамически распределяют имеющиеся аппаратные ресурсы между приложениями, в FX-RTOS используется статическая модель, для достижения большей предсказуемости работы а также снижения сложности.

На этапе проектирования системы разработчик должен распределить имеющуюся физическую память между всеми приложениями, причем впоследствии во время работы это распределение не меняется. Перед запуском, все непривилегированные приложения, также как и ядро ОС, должны быть загружены в память. Ядро не предоставляет API для загрузки приложений и не накладывает никаких ограничений на то, каким образом происходит загрузка. Как правило, загрузка как ядра так и приложений происходит с участием первичного загрузчика.

Запуск привилегированного приложения происходит аналогично тому, как описывалось выше: путем вызова функции **fx_app_init**, после завершения инициализации ядра в привилегированном режиме процессора. Запуск непривилегированного модуля со стороны привилегированного приложения состоит из следующих основных этапов:

1. Инициализация объекта "непривилегированное приложение"
2. Создание адресного пространства приложения
3. Установка источника ресурсов для приложения
4. Создание первого потока приложения

Объект "непривилегированное приложение" или "модуль" это системный объект, который должен быть выделен привилегированным приложением. Этот объект ничем не отличается от всех прочих системных объектов, таких как потоки или примитивы синхронизации и работа с ним происходит по тем же принципам.

По умолчанию объект инициализируется с пустым адресным простанством, то есть не может обращаться ни по какому адресу памяти, что , разумеется, означает невозможность его выполнения. После инциализации объекта, привилегированное приложение, используя API исполнительной подсистемы, должно установить права доступа к регионам памяти в адресном пространстве модуля. Если используемая аппаратное обеспечение содержит MMU, возможно создание произвольного адресного пространства, то есть отображение любых виртуальных адресов на любые физические. Если же используется MPU, то допускается только установка прав доступа к определенным адресам, без возможности трансляции адресов.

Приложениям для работы, как правило, требуются различные объекты. Поскольку ядро ОС работает с такими объектами непосредственно, разрушение внутренней структуры объектов ставит под угрозу функционирование ядра, поэтому непривилегированным приложениям нельзя доверить использование таких объектов напрямую. Вместо этого, память под такие объекты управляется ядром ОС, а приложение использует их косвенно, через механизм идентификаторов, которые подсчитывают количество ссылок на объекты и не дают использовать их некорректно. Каждому модулю ставится в соответствие (с помощью функций API) пул ресурсов, из которых происходит выделение памяти под объекты, запрашиваемые модулем. Это исключает исчерпание системных ресурсов из-за действий модуля, поскольку каждый модуль может использовать только те ресурсы, которые были ему явно предоставлены.

После завершения инициализации, создается поток, выполняемый в адресном пространстве модуля. Его запуск возлагается не привилегированное приложение. Впоследствии модуль может создавать дополнительные потоки и прочие необходимые ему объекты, если он имеет для этого возможности (имеет ресурсы для выделения памяти).

Модуль трассировки

Обзор функциональности

Модуль трассировки предназначен для упрощения отладки при разработке программ для FX-RTOS, а также для большей наглядности того, как работает система.

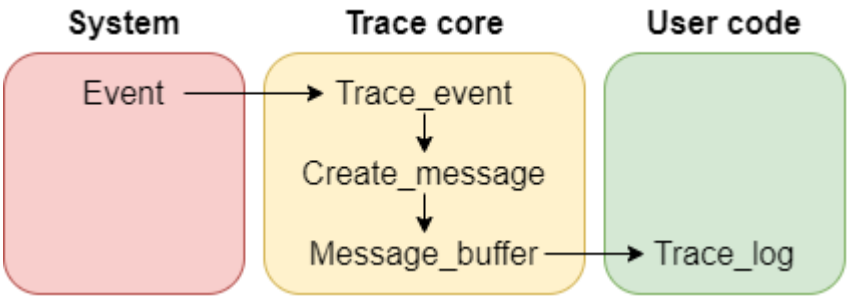
В модуле трассировки реализован инструментарий для логирования данных о следующих объектах системы:

- Потоки
- Idle-потоки
- Мьютексы
- Семафоры
- Очереди сообщений

Логирование данных о перечисленных выше объектах происходит в контексте системных событий, происходящих с ними (например, переключений контекста для потоков), которые также могут логироваться. Также в модуле трассировки предусмотрена возможность логирования для следующих событий:

- Прерывания
- Достижение потоком выполнения пользовательских меток

Архитектура модуля



При возникновении системного события, которое может логироваться, вызывается соответствующая этому событию функция из модуля трассировки. Если настройки логирования действительно требуют записи какой-либо информации про возникшее событие, то во внутреннем для модуля трассировки статическом буфере формируется соответствующее сообщение. Далее вызывается функция trace_log, в которую передается указатель на начало сообщения и его длина. Функция trace_log должна быть реализована пользователем в соответствии с его потребностями.

Модуль трассировки разбит на 3 части: TRACE_CORE, TRACE_ISR, TRACE_SHARED. В первой реализована трассировка событий системных объектов и пользовательских меток, во второй - трассировка прерываний, в третьей объявлены и имплементированы общие для первых двух частей вещи.

Формат логирования

Каждое событие, требующее трассировки, соотносится с определенным объектом трассировки (не путать с системными объектами, перечисленными выше. Строго говоря, отображение множества трассируемых системных объектов во множество объектов трассировки инъективно). Существуют следующие объекты трассировки:

- Idle-поток
- Поток
- Мьютекс
- Семафор
- Очередь сообщений
- Пользовательская метка
- Пользовательское сообщение
- Пользовательское сообщение об ошибке
- Прерывание

Каждому событию соответствует один лог. Лог - информация о событии, записанная в соответствии с настройками логирования в буфер, который затем передается в trace_log.

В первом байте каждого лога обозначено, к какому объекту трассировки относится данный лог.

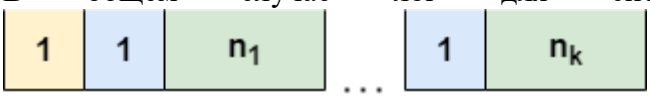
Формат остальной части лога отличается для системных объектов и для всего остального.

Для системных объектов

Для системных объектов существует возможность гибкой настройки логируемых данных. Например, можно выбрать, логировать ли физический адрес объекта или нет. Так как множество логируемых данных для системных полностью определяется пользователем (в рамках реализации), то для их записи выбран следующий формат:

| ID поля | | Значение поля |. (ID поля всегда занимает 1 байт. Значение поля может занимать разное количество байт (обычно 1 или 4) в зависимости от того, что это за поле (например, адрес объекта занимает 4 байта, а ID произошедшего с объектов события занимает 1 байт). После значения поля может следовать ID следующего поля.

В общем случае лог для системных объектов выглядит так:



Желтый байт - ID объекта трассировки
Голубые байты - ID полей
Зеленые байты - значения полей
n_i - количество байт для значения i-го поля

Для всего остального

Так как для прерываний, пользовательских меток и пользовательских сообщений не нужны гибкие настройки, формат их логирования проще. После первого байта лога в строго определенном порядке записывается вся необходимая информация (например, для пользовательской метки лог всегда состоит из 9 байт: 1 байт - обозначает объект трассировки, 4 байта - временная метка, 4 байта - ID метки).

API

Функции, реализуемые пользователем

uint32_t trace_get_time() - функция, используемая для получения временной метки при логировании. Желательно привязать к какому-нибудь таймеру.

void trace_log(uint8_t* msg, size_t size) - функция, в которую передаются все логи. Здесь должна быть какая-либо обработка логов. Например, сохранение в буфер.

Функции для настройки логирования

Для каждого объекта, поддерживающего трассирование, перед его инициализацией должна быть единожды вызвана функция установки настроек логирования.

void trace_set_thread_log_lvl(trace_thread_handle_t*, log_lvl_t) - функция для настройки логирования потоков. В качестве второго аргумента должна быть передана битовая дизъюнкция следующих значений:

- TRACE_THREAD_TIMESTAMP_KEY - логирование временной метки
- TRACE_THREAD_ADDR_KEY - логирование адреса потока
- TRACE_THREAD_ID_KEY - логирование ID потока
- TRACE_THREAD_EVENT_KEY - логирование ID события
- TRACE_THREAD_PRIO_KEY - логирование приоритета потока
- TRACE_THREAD_CS_ONLY_KEY - логирование только при context_switch
- THREAD_LOG_LVL_DEFAULT - логирование временной метки, ID потока и ID события

либо LOG_LVL_SUPPRESSED, если логирование для данного потока не требуется.

void trace_set_mutex_log_lvl(trace_mutex_handle_t*, log_lvl_t) - функция для настройки логирования мьютексов. В качестве второго аргумента должна быть передана битовая дизъюнкция следующих значений:

- TRACE_MUTEX_TIMESTAMP_KEY - логирование временной метки
- TRACE_MUTEX_ADDR_KEY - логирование адреса мьютекса
- TRACE_MUTEX_ID_KEY - логирование ID мьютекса
- TRACE_MUTEX_EVENT_KEY - логирование ID события
- TRACE_MUTEX_OWNER_KEY - логирование владельца мьютекса

либо LOG_LVL_SUPPRESSED, если логирование для данного мьютекса не требуется.

void trace_set_sem_log_lvl(trace_sem_handle_t*, log_lvl_t) - функция для настройки логирования семафоров. В качестве второго аргумента должна быть передана битовая дизъюнкция следующих значений:

- TRACE_SEM_TIMESTAMP_KEY - логирование временной метки
- TRACE_SEM_ADDR_KEY - логирование адреса семафора
- TRACE_SEM_ID_KEY - логирование ID семафора
- TRACE_SEM_EVENT_KEY - логирование ID события
- TRACE_SEM_VAL_KEY - логирование текущего значения семафора
- TRACE_SEM_MAX_VAL_KEY - логирование максимального значения семафора

либо LOG_LVL_SUPPRESSED, если логирование для данного семафора не требуется.

void trace_set_queue_log_lvl(trace_queue_handle_t*, log_lvl_t) - функция для настройки логирования очередей сообщений. В качестве второго аргумента должна быть передана битовая дизъюнкция следующих значений:

- TRACE_QUEUE_TIMESTAMP_KEY - логирование временной метки
- TRACE_QUEUE_ADDR_KEY - логирование адреса очереди сообщений
- TRACE_QUEUE_ID_KEY - логирование ID очереди сообщений
- TRACE_QUEUE_EVENT_KEY - логирование ID события

либо LOG_LVL_SUPPRESSED, если логирование для данной очереди сообщений не требуется.

Особые функции для настройки логирования

Данные настройки обязательно должны быть совершены до вызова fx_kernel_entry().

Для настройки логирования idle-потоков должна быть объявлена глобальная переменная trace_idle_thread_log_lvl типа log_lvl_t. Ей должно быть присвоено значение, сформированное аналогично настройке логирования обычных потоков.

Для настройки логирования прерываний существуют следующие функции:

- trace_turn_isr_on(id) - включить логирование прерывания с номером id (id ∈ [0;512))
- trace_turn_isr_off(id) - выключить логирование прерывания с номером id (id ∈ [0;512))
- trace_turn_all_isr_on() - включить логирование всех прерываний
- trace_turn_all_isr_off() - выключить логирование всех прерываний

Данные функции можно применять последовательно. Например, можно выключить логирование всех прерываний, а после включить логирование только каких-то определенных из них.

Функции для пользовательского логирования

void trace_put_usermark(uint32_t) - при достижении потоком выполнения данной функции в trace_log будет передан лог с текущей временной меткой и ID метки. Аргументом функции является ID метки.

void trace_put_text(char*, uint32_t) - при достижении потоком выполнения данной функции в trace_log будет передан лог с текущей временной меткой и переданным текстом. Аргументами являются указатель на начало текста и его длина. Длина не должна быть больше 250 символов.

void trace_put_error(char*, uint32_t) - функция аналогичная предыдущей, но с другим ID объекта трассировки.