FX-RTOS: компонентная ОС для встраиваемых систем

Вслед за развитием электроники, встраиваемые системы получили большое распространение в современном мире. Большое разнообразие задач и требований, предъявляемых к этим системам, определило также и большой выбор встраиваемого ПО. В отличие от серверов и настольных компьютеров, в мире встраиваемых систем требования для разных устройств могут изменяться вплоть до противоположных: одним устройствам требуется работа в жестком реальном времени, другим - как можно меньшее энергопотребление (возможно, в ущерб всем остальным характеристикам), третьим - как можно меньший объем потребляемых ресурсов, для возможности использования более дешевых процессоров.

Важнейшим компонентом встраиваемой системы является ОС. В большинстве случаев она имеет вид статической библиотеки и реализует вытесняющую многозадачность и обработку прерываний. Было написано немало литературы на тему "необходимо ли вообще использовать какую-либо ОС во встраиваемых системах?", поэтому подробно этот вопрос в настоящей статье рассматриваться не будет, однако можно вкратце перечислить преимущества использования ОС: абстракция оборудования (перенос приложения на другие аппаратные платформы без изменений, что обеспечивает гибкость в выборе аппаратного обеспечения и независимость от конкретной архитектуры/производителя), использование проверенного решения сокращает сроки вывода продукта на рынок и менее подвержено ошибкам, и т.д.

В отличие от настольных и серверных ОС, которые определяют интерфейс для приложений и требуют, чтобы приложение работало в соответствии с требованиями ОС, в мире встраиваемых систем дело обстоит с точностью до наоборот: т.к. устройство обычно предназначено для выполнения какой-то определенной функции, и его встраиваемое ПО изначально рассчитано на реализацию этой функции, то в данном случае не приложение должно соответствовать ОС, а наоборот, ОС должна идеально соответствовать требованиям именно данного конкретного

приложения. Это означает, что ОС не должна содержать никакого избыточного функционала, который не будет использоваться в данном устройстве, не должна содержать дополнительных накладных расходов, и в то же время соответствовать тем же требованиям, которым соответствует приложение, например требованиям жесткого реального времени, а также использоваться совместно с теми инструментами (компилятор, IDE, отладчик и т.д.), с помощью которых разрабатывается приложение.

Так же как и приложение, ОС является обычной программой, поэтому достаточно трудно разработать ее так, чтобы она могла соответствовать всем требованиям, включая те, которые могут быть даже неизвестны на этапе ее разработки. Поэтому в настоящее время существуют десятки и сотни встраиваемых ОС, каждая из которых предназначена для использования в определенном классе устройств с определенными требованиями. Многие из них реализуют т.н. "компонентную модель": программная платформа для приложений состоит из нескольких компонентов, таких как, собственно ядро ОС, стеки коммуникационных протоколов, реализация файловой системы и т.д., что позволяет задавать конфигурацию системы на уровне этих компонентов, исключая неиспользуемые и задавая параметры используемым.

Разрабатываемая компанией Eremex ОС FX-RTOS является дальнейшим развитием идеи компонентно-ориентированного ПО с распространением принципов "компонентности" на само ядро ОС. Если в таких системах как eCos или MQX ядро является неделимым компонентом, среди множества других компонентов, в FX-RTOS нет четко определенного понятия "ядра": необходимый функционал составляется из более мелких "кирпичиков".

Нельзя сказать, что эта идея нова, уже не один десяток лет существуют библиотеки, и ничто не мешает разделить на отдельные компоненты любой код, в том числе и ядро ОС, однако такой подход до сих пор во встраиваемых системах применялся редко, потому что использование баблиотек содержит накладных расходы времени выполнения. Подход FX-RTOS отличается тем, что используются компоненты уровня исходных текстов. То есть "компонентность" как таковая не несет никаких накладных

расходов, а конфигурирование выполняется путем "внедрения зависимостей".

Компоненты FX-RTOS ничего не знают ни о структуре проекта, ни друг о друге, не привязаны к системе сборке и к структуре директорий исходных текстов.
Такая модель конфигурирования позволяет масштабировать ОС в очень широких пределах. Например, FX-RTOS позволяет запустить 3-5 потоков на 16-битном контроллере TI MSP430 FR5739, который содержит всего 1Кб оперативной памяти. В этом объеме содержатся данные ОС, данные приложения, стеки потоков и стек прерываний, стек одного потока имеет размер порядка 150-200 байт. В то же время существует версия FX-RTOS работающая на многоядерных процессорах семейства ARM Cortex-A. При этом используются все вычислительные ядра, возможна миграция потоков между ядрами по запросу, балансировка нагрузки и т.д. ОС при этом не содержит никаких глобальных блокировок, то есть производительность не деградирует с ростом числа ядер и приложения работающие на разных ядрах не влияют друг на друга. Эти системы принципиально различаются по своей внутренней структуре и механизмам синхронизации, однако API, предоставляемое пользователю, остается неизменным и подобная масштабируемость достигается путем замены компонентов ядра ОС с сохранением внутренних интерфейсов.
Отдельным вопросом является вопрос производительности. Хотя компонентно-ориентированная модель разработки ПО содержит некоторые накладные расходы из-за небходимости поддерживать постоянными интерфейсы внутри ядра ОС, принцип замены частей ОС на те, которые идеально соответствуют решаемой задаче, позволяет добиться лучшей производительности, чем в случае "статического" ядра. Можно привести следующий пример: алгоритмы имеют определенную асимптотическую сложность, как правило, улучшение асимптотических характеристик связано с усложнением алгоритма. Например, некоторые операции работы со списком (вроде поиска или вставки элемента в упорядоченный список) имеют сложность O(n): при росте количества элементов, время, затрачиваемое на работу алгоритма растет линейно, и при большом количестве элементов список становится неэффективной структурой. Можно использовать дерево,

например красно-черное или AVL, которые имеют сложность O(log n), однако преимущество дерева над списком начнет проявляться только с некоторого N, так как сбалансированные деревья - довольно сложные структуры данных требущие относительно большого (по сравнению со списком) объема кода. Поэтому ни один алгоритм не может обеспечить идеальную производительность для всех возможных случаев, решить данную проблему можно только с помощью конфигурирования и дополнительного знания о том, в каких условиях системе предстоит работать.

Для оценки производительности ОС использовался тест Preemptive scheduling test или пакета Thread Metrics, разработанного компанией Express logic (www.rtos.com. Этот тест создает 5 нитей, имющих разный приоритет. Затем наименее приоритетная нить увеличивает свой счетчик итераций, после чего активирует более высокоприоритетную нить, которая делает то же самое с еще более приоритетной нитью и т.д.

```
while(1)
{
  /* Resume thread N. */
  tm_thread_resume(N+1);
  /* Increment this thread's counter. */
  tm_preemptive_thread_N_counter++;
  /* Suspend self! */
  tm_thread_suspend(N);
}
```

Процесс продолжается 30 секунд, после чего рассчитывается общее количество итераций каждой нити, а также время, затрачиваемое на одну итерацию. При использовании ядра, сконфигурированного для максимальной производительности, одна итерация теста происходит примерно за 5 мкс (процессор ARM Cortex M3 100MHz, около 6 млн итераций за 30с), что является одним из лучших результатов, среди конкурирующих решений.

Если задача сохранения единого API не является определяющей, можно использовать компоненты FX-RTOS для построения собственных ОС, которые будут идеально соответствовать используемому приложению, вплоть до учета особенностей

конкретного устройства и приложения в планировщике или во внутренних механизмах синхронизации ОС. Вместе с FX-RTOS возможно также использование ее инфраструктуры, позволяющей распространить компонентный подход и на приложение, включение компонентов RTOS в результирующий образ при этом будет происходит автоматически, в зависимости от нужд приложения.

ОС поддерживает работу в жестком реальном времени, при использовании соответствующей конфигурации, все критические секции внутри ОС имеют фиксированную задержку и не зависят от параметров времени выполнения: независимо от числа созданных пользователем и активных объектов синхронизации, потоков, таймеров и т.д., время реакции системы остается фиксированным и предсказуемым. С помощью профилирования можно также определить максимальное время нахождения внутри критических секций.

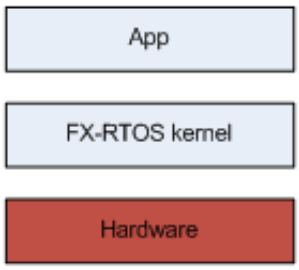
Как и другие подобные ОС, поддерживаются все основные инструменты и средства разработки, в число предоставляемых пробных версий ОС входят примеры с использованием GNU-инструментария, IAR Embedded Workbench (для AVR32, MSP430 и ARM), а также Keil uVision и Microchip MPLAB X. Среди поддерживаемых процессоров: Atmel AVR32, ARM Cortex-M0, ARM Cortex-M3, ARM Cortex-A9 MPCore, Texas Instruments MSP430, Microchip PIC32, а также x86. Платформенно-зависимый код имеет относительно малый объем, поэтому портирование на новые процессоры не занимает много времени, в пределах 2-4 недель. Как следует из приведенного списка процессоров ОС является полностью независимой от порядка байт в слове (big/little endian), а также от разрядности процессора.

Для удобства отладки предоставляется также порт системы на Win32 API, при этом ФС и стек TCP/IP также имеют возможность работы в "программном" окружении, что позволяет производить отладку с использованием Microsoft Visual Studio и продвинутых средств отладки Windows, без наличия физического устройства. Кроме того, это позволяет также разрабатывать автоматические тесты, "прошивка" компилируется в исполняемый файл windows который может тестироваться так же, как тестируются настольные приложения, после завершения тестирования, ОС заменяется на предназначенную для аппаратного решения. От

приложения требуется только замена драйверов на "настоящие" драйверы, работающие с реальными устройствами (win32 симулятор пользуется средствами Windows для доступа к сети, накопителям и т.д.).

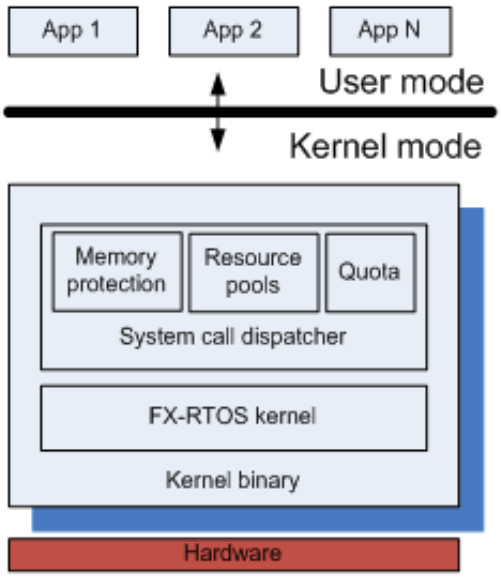
Кроме того, для удобства работы с системами, которые управляются событиями, FX-RTOS предоставляет специальный профиль, предназначенный для асинхронной обработки событий. Реализация нитей из системы удаляется, вместо них используются обработчики событий, называемые ESR (event service routine), похожие на программные прерывания. При использовании ESR достигается сокращение размера кода, с одновременным увеличением производительности и сокращением размера занимаемой памяти (т.к. в отличие от нитей, для каждого обработчика не требуется выделять собственный стек), упрощается синхронизация и программирование приложений, что, в конечном итоге, сокращает сроки разработки.

Наконец, еще одним важным вопросом является безопасность. Встраиваемые ОС обычно реализованы в виде библиотеки, которая должна компоноваться вместе с приложением, формируя исполняемый образ, который затем должен быть загружен в целевое устройство. FX-RTOS также поддерживает такую модель взаимодействия ОС и приложения.



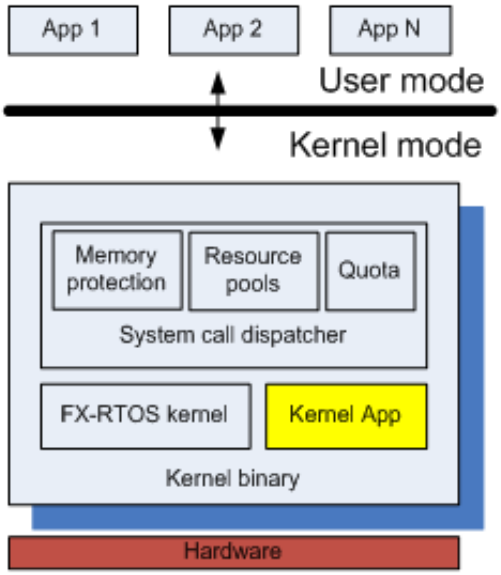
Компоновка ОС и приложения в единый исполняемый модуль позволяет достичь максимальной возможной производительности (вызов сервиса ОС не отличается от вызова функции внутри приложения и не содержит никаких накладных расходов), наименьшего размера конечного образа и т.д. Вместе с тем, такая архитектура имеет и серьезные недостатки: драйверы устройств, все приложения и сервисы, ядро ОС - всё это работает в едином домене защиты и поэтому потенциально уязвимо извне (например, если устройство имеет сетевой интерфейс).

Зачастую, устройство выполняет несколько функций: промышленный контроллер, управляющий технологическими процессами, может иметь возможности для обновления прошивки, содержать сервер статистики и прочий функционал, который не является критически важным для работы устройства и может использоваться достаточно редко. Однако потенциальные уязвимости в реализации сетевого стека (который в случае TCP/IP является достаточно сложным) и второстепенного функционала самого устройства дают злоумышленнику возможности для получения полного контроля над устройством и всеми его функциями, что может иметь крайне негативные последствия. Использование же полнофункциональных ОС, обладающих продвинутыми средствами безопасности может быть неприемлемо по экономическим причинам: все эти ОС требуют гораздо более производительных (и дорогих) процессоров.
Для защиты в том числе от этих угроз, существует специальная версия FX-RTOS, которая частично реализует возможности "больших" ОС, с учетом специфики микроконтроллеров. Приложение и ОС разделяются на отдельные модули, а API FX-RTOS доступно через интерфейс системных вызовов, по аналогии с тем, как это сделано в "больших" ОС, таких как VxWorks, QNX или Windows.



Модуль приложения содержит одну или более нитей, выполняющихся в непривилегированном режиме процессора. Возможна также загрузка более чем одного модуля приложения.

Каждый модуль имеет свои настраиваемые права доступа к регионам памяти и устройствам, а также имеет ограничения на возможности создания системных объектов (например, высокоприоритетных нитей). При попытке модуля выполнить действие, которое не разрешено (например, доступ к памяти ядра ОС, доступ к памяти другого модуля, попытка использования устройства, доступ к которому для данного модуля запрещен и т.п.) выполнение всех нитей данного модуля принудительно завершается. При этом остальные приложения продолжают работу. Кроме того, доступна также традиционная модель встраиваемого ПО (статическая компоновка приложения и ядра ОС). Таким образом, критичные к производительности приложения могут быть по прежнему скомпонованы с ядром и использование защиты не вносит никаких дополнительных накладных расходов там, где эта защита не требуется.



Прикладной интерфейс FX-RTOS остается неизменным как для привилегированных, так и для непривилегированных приложений (в непривилигированном режиме используется подмножество основного API). Таким образом, одно и то же приложение может использоваться по-разному на разных устройствах без внесения в него изменений.

Наконец, использование защиты памяти и ограничения доступа позволяет существенно упростить отладку приложений: попытки доступа к чужой памяти или к устройствам могут быть перехвачены и проанализированы. В отличие от

полнофункциональных ОС, все ресурсы, а также карта памяти, используемые приложением выделяются статически, количество динамически создаваемых объектов сведено к минимуму, что обеспечивает максимальную производительность и компактность кода, которые недостижимы при использовании "больших" ОС: ядро FX-RTOS с подержкой семафоров, мьютексов и очередей сообщений имеет размер всего в 11K кода, и может, как и все подобные ОС, быть расположено в read-only памяти.

Как и в случае статической компоновки с приложением, набор сервисов, предоставляемых ОС непривигированным приложениям является настраиваемым, и в результирующий образ ядра попадают только те функции, которые необходимы. В отличие от большинства реализаций защищенных ОС для контроллеров, память, которая явно не была выделена для использования приложением, недоступна даже для чтения, что обеспечивает дополнительную защиту, т.к. в случае компрометации недоверенных модулей, они не только не смогут получить контроль над устройством, но также не смогут прочитать ни код, ни данные защищенных приложений, а также ОС.

В качестве примера возможного применения можно рассмотреть задачу создания контроллера, который получает команды извне (по сети) и управляет при этом некоторым устройством и поддерживает внутреннюю машину состояний. Стек протоколов TCP/IP является достаточно тяжеловесным (например открытый стек lwIP - около 40000 строк кода), поэтому даже если протокол управления связанного с контроллером устройства является простым, вся система является потенциально уязвимой через возможные ошибки реализации сетевого стека, объем которого может в несколько раз превышать код, реализующий основной функционал. Можно рассмотреть вариант использования "большой" ОС или даже выделенного компьютера, который общался бы с контроллером (при этом предполагается что любой, кто может посылать команды контроллеру, является доверенным). Однако все эти решения требуют дополнительного оборудования, ПО, а главное - не снижают сложности решения. Уязвимости в любом месте такой системы приводят к потенциальной возможности захватить полный контроль над устройством, которым призван управлять контроллер, причем это может быть осуществлено удаленно.

Если поставить во главу угла снижение сложности, тогда логично ограничить доверенный код несколькими тысячами строк кода, а весь остальной код, включая реализацию сетевого стека объявить недоверенным. В этом случае граница между доверенной и недоверенной средой находится уже внутри самого микроконтроллера. Получив команды по сети, сетевой стек может размещать их в особой области памяти и нотифицировать каким-либо образом основное доверенное приложение. Либо это приложение может само анализировать некоторый разделяемый буфер с определенной периодичностью. В этом случае даже получив возможность выполнять произвольный код на контроллере, программа управления устройством остается корректной даже в случае полной неработоспособности сетевого стека в результате ошибки (обнаружив завершение работы сетевого стека основное приложение может безопасно остановить работу устройства либо перезапустить контроллер).

Накладные расходы на организацию защищенной среды относительно невысоки, при отсутствии оптимизаций компилятора дополнительный объем кода (по сравнению с аналогичным ядром, но без защиты) составляет всего 5-6K флеш-памяти.