Введение в VisualDSP Kernel

Вадим ТОРГАНОВ
Vadim.torganov@analog.com.ru

Эта статья посвящена ядру операционной системы VisualDSP Kernel (VDK) компании Analog Devices и его использованию совместно с процессорами Blackfin. Мы исследуем возможности VDK, проведем обзор API, которые составляют его основу, и оценим его требовательность к производительности процессора и объему памяти.

Введение

VDK является составной частью пакета VisualDSP++. В этой статье мы обсудим VisualDSP Kernel, некоторые концепции, лежащие в его основе, и возможности, которыми оно обладает. Для лучшего понимания материала читателю необходимо иметь, по меньшей мере, представление о концепциях разработки программного обеспечения, а также о базовых концепциях операционных систем, либо на примере другой коммерческой RTOS, с которой он работал ранее, либо на примере операционной системы собственной разработки. Специалистам, не знакомым с устройством операционных систем, можно порекомендовать статью [1].

Сначала мы дадим вводный обзор имеющихся вариантов операционных систем для процессоров Blackfin. Затем мы поговорим от возможностях VDK и некоторых API, составляющих его основу. И, наконец, для иллюстрации влияния VDK на производительность системы мы дадим оценки требований к производительности и объему памяти для типового приложения.

Операционные системы для процессоров Blackfin

Для начала следует отметить, что для процессоров Blackfin существует целый ряд операционных систем (ОС), выпускаемых третьими фирмами: uClinux; INTEGRITY; VelOSity; µC/OS-II; ThreadX; RTXC Quadros; Nucleus Plus.

Полный список ОС по состоянию на текущий момент времени можно найти на сайте Analog Devices [2].

Перечисленные ОС различаются свойствами и возможностями. Кроме того, они отличаются друг от друга в плане поддержки, стоимости и отчислений с продаж. С точки зрения свойств доступные ОС варьируются от «облегченных» ОС малой сложности, таких как VDK, до обладающих исчерпывающим набором возможностей ОС, таких как uClinux. В некоторых ОС, наподобие INTEGRITY, делается упор на высокой надежности, в других — на потребностях конкретного сегмента рынка (например, RTA-OSEK ориентирована на нужды рынка автомобильной электроники). С точки зрения поддержки, стоимости и отчислений продаж потенциальные варианты можно сравнить схожим образом. Например, uClinux распространяется бесплатно, но единственный источник поддержки по данной ОС — это поддержка, оказываемая сообществом uClinux. В отличие от нее такие ОС, как RXC Quadros, требуют лицензирования и/или отчислений с продаж, но имеют обширную поддержку. Более подробную информацию по этой теме можно найти в статье [3].

При выборе операционной системы можно прийти к выводу, что свойства и/или финансовые затраты на использование одной из этих ОС идеально подходят для вашего конкретного приложения. А возможно, что вы сочтете ядро ОС VDK оптимальным вариантом для вас. Поэтому перейдем к более подробному рассмотрению свойств VDK.

Введение в VDK

Интегрированная среда разработки VisualDSP содержит собственное ядро ОС, которое называется VisualDSP Kernel, или VDK. Это небольшое, устойчивое ядро, которое поставляется со средой VisualDSP и интегрировано в нее. Для использования VDK не нужны дополнительные денежные затраты, лицензионные отчисления и какие-либо другие вложения средств. VDK поддерживает все выпускаемые на данный момент процессоры Blackfin.

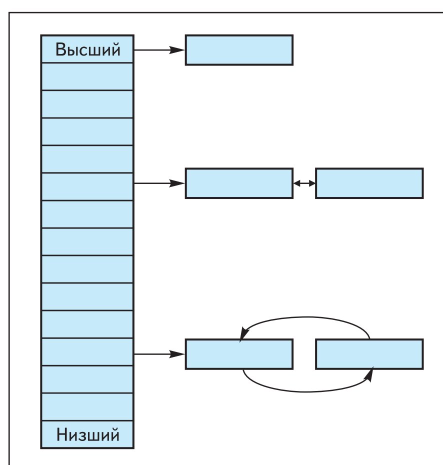

Рисунок. Работа нескольких потоков с одинаковым уровнем приоритета

В этой статье мы сосредоточимся на таких аспектах VDK, как:
- потоки;
- назначение приоритетов и планирование;
- критические и непланируемые области;
- семафоры (включая периодические семафоры);
- сообщения;
- требуемый объем памяти;
- требуемая производительность.

Потоки и приоритеты

При работе с процессорами Blackfin потоки могут исполняться на любом из 31 уровня приоритетов. Количество потоков, выполняемых в системе в любой момент времени, абсолютно произвольно. Их может быть больше, чем 31, и в большинстве приложений обязательно будет. Планирование выполнения задач с разным приоритетом решается довольно очевидно. То, какая задача будет выполняться системой, диктуется приоритетом. Поток с наибольшим приоритетом всегда получает полный контроль над процессором, пока не случится что-то, из-за чего выполнение этого потока необходимо будет отложить.

Принятие решения усложняется и становится более интересным, когда несколько потоков имеют одинаковый уровень приоритета. На рисунке эта ситуация показана при помощи двух смежных блоков в середине диаграммы. Один из потоков имеет полный контроль над процессором, пока он не закончит выполнять свою задачу и не передаст управление другому потоку. Такая схема известна под названием кооперативной параллельной обработки, или кооперативной многозадачности.

В нижней части диаграммы на рисунке можно увидеть сценарий с «квантованием времени», который также часто именуют пла-

нированием с циклической сменой приоритета. При квантовании времени операционная система автоматически передает контроль всем активным потокам, имеющим одинаковый приоритет, по кругу.

Период времени, в течение которого каждый поток управляет процессором, определяется программистом. Приоритеты потокам могут назначаться статически или динамически. Статически назначаемые приоритеты объявляются при компоновке приложения. Динамическое назначение приоритетов осуществляется на этапе исполнения приложения, когда поток работает или когда создается экземпляр потока. Чтобы система начала работать, необходимо создать на этапе загрузки хотя бы один поток. В противном случае системе будет нечего делать. Поэтому в VisualDSP принудительно создается, по меньшей мере, один загрузочный поток. Ограничений на количество потоков не накладывается, но на практике оно будет ограничено объемом памяти системы.

Каждый поток получает собственный стек, куда компилятор языка C будет помещать локальные переменные и стеки вызовов. Это означает, что программисту не нужно будет беспокоиться о том, что два потока случайно будут обращаться к одной локальной переменной или другим структурам.

На простейшем уровне поток представляет собой реализацию четырех функций: create (создания), destroy (уничтожения), run (исполнения) и error (ошибки). Если вы знакомы с языком C++, то можно провести аналогию между функциями create и destroy, с одной стороны, и конструкторами и деструкторами C++ — с другой. Create — это обычно небольшие функции, которые выполняются непосредственно в момент создания потока для статической инициализации. Аналогичным образом функции destroy, или деструкторы, выполняют небольшой объем работы по очистке памяти, необходимый для уничтожения потока. Основную часть времени поток проводит в функции run. В действительности очень часто возврата из функции run не происходит. Например, никогда не выполняется возврат из цикла “while(1)”. Если из выполняемой функции никогда не происходит возврата, то поток затем автоматически уничтожается.

Приведем перечень API-функций, которые используются для управления потоками:

```
VDK_ClearThreadError()
VDK_CreateThread()
VDK_CreateThreadEx()
VDK_DestroyThread()
VDK_FreeDestroyedThreads()
VDK_GetLastThreadError()
VDK_GetLastThreadErrorValue()
VDK_GetPriority()
VDK_GetThreadID()
VDK_GetThreadStackUsage()
VDK_GetThreadStatus()
VDK_ResePriority()
VDK_SetPriority()
VDK_SetThreadError()
VDK_Sleep()
VDK_Yield()
```

Некоторые из этих функций, например, ClearThread() и DestroyThread(), используются часто. Последние две из перечисленных API-функций — Sleep() и Yield() — используются для помещения приложения в состояние сна на определенное количество времени, после чего VDK автоматически выводит приложение из этого состояния и его работа возобновляется. Функция Yield() может использоваться в упоминавшейся ранее схеме кооперативной многозадачности, когда два потока могут передавать управление приложением друг другу. Функции, связанные с обработкой ошибок, предназначены для целей отладки и обычно в окончательном варианте приложения не используются.

Критические/непланируемые области

Любой, кто работал с операционными системами реального времени (ОСРВ), сталкивался с ситуациями, когда какую-то небольшую операцию ни в коем случае нельзя прерывать. Для защиты этих операций VDK предоставляет два механизма: критические и непланируемые области.

Критические области отключают планировщик и все прерывания, предоставляя процессу полный и безоговорочный контроль над процессором, независимо от того, что еще происходит в системе. Например, предположим, что ваше приложение запрашивает значение глобальной переменной, принимает на его основе какое-то решение и затем изменяет значение этой глобальной переменной. Если важно, чтобы во время этого процесса систему не прерывали другие задачи, вы можете поместить соответствующий код в критическую область.

Критические области нужно использовать осторожно, поскольку они могут иметь серьезные побочные эффекты. Если вы оставите процессор в критической области на длительное время, то он может не отработать поступившее на него прерывание. Кроме того, при работе с критическими областями также возрастает задержка обслуживания прерываний.

Непланируемые области имеют назначение, аналогичное критическим областям, но являются менее радикальным средством. В них отключается возможность контекстного переключения, однако прерывания по-прежнему могут быть обслужены. Набор API-функций для работы с критическими и непланируемыми областями:

```
VDK_PopCriticalRegion()
VDK_PopNestedCriticalRegion()
VDK_PopNestedUnscheduledRegions()
VDK_PopUnscheduledRegion()
VDK_PushCriticalRegion()
VDK_PusUnscheduledRegion()
```

Обратите внимание на то, что при работе с критическими областями используется такой же механизм, как и при работе со стеками. То есть вы помещаете критическую область в некоторое подобие стека и затем извлекаете ее оттуда. Это позволяет использовать критические области более чем в одной функции. При написании функции вы работаете с критической областью, не заботясь о том, что эта функция могла быть вызвана другой функцией, которая, в свою очередь, также находится в критической области.

Семафоры

Это, вероятно, наиболее естественный механизм синхронизации событий между несколькими потоками или между потоками и процедурами обслуживания прерываний. Семафоры обычно используются, когда потокам необходимо обратиться к какому-нибудь совместно используемому ресурсу. Классический пример такого ресурса — это аудиобуфер, который заполняется одним потоком, а опустошается другим. Мы должны гарантировать, что поток, осуществляющий запись, не перезапишет данные раньше, чем предыдущие будут прочитаны потоком, осуществляющим чтение. Поток, осуществляющий чтение, (опустошающий аудиобуфер) должен в таком случае «сказать»: «Пожалуйста, не выполняйте запись в это время». Перед началом записи потоку, который заполняет буфер, необходимо будет дождаться, пока семафор не будет снят. При такой организации взаимодействия конфликтов между различными процессами, пытающимися обратиться к одному и тому же буферу, периферийному модулю или другому ресурсу, возникать не будет.

Большинство семафоров имеют булев тип: они либо сняты, либо нет. Например, аудиобуфер либо доступен, либо нет. Такой тип семафора называется мьютексом (взаимоисключающим). Возможна ситуация, когда вам захочется совместно использовать в нескольких процессах пул, состоящий из нескольких отдельных ресурсов. В этом случае вы можете воспользоваться так называемым вычислительным семафором. Вычислительные семафоры реализуются точно так же, как и булевы семафоры, за тем исключением, что количество совместно используемых ресурсов здесь больше одного.

И, наконец, необходимо отметить, что семафоры можно сделать периодическими. То есть ОС будет автоматически выставлять семафор с интервалом времени, заданным в приложении. Например, предположим, что у вас есть анимационное изображение, которое нужно обновлять 24 раза в секунду. Возможно, у вас возникнет желание реализовать это при помощи периодического семафора, которые выставляется каждую 1/24 секунды. Этот подход намного изящнее, чем, например, помещение потока в состояние сна.

Набор API-функций для работы с семафорами:

```
VDK_CreateSemaphore()
VDK_DestroySemaphore()
VDK_MakePeriodic()
VDK_PendSemaphore()
VDK_PostSemaphore()
VDK_RemovePeriodic()
```

Используя эти API-функции, вы можете динамически создавать и уничтожать семафоры. Их можно также создавать статически на этапе компоновки приложения.

Сообщения

Они представляют собой несколько более сложный способ координирования потоков. Сообщение — это направленная передача от одного потока к другому. Сообщения имеют поток-отправитель и поток-получатель. Поток-отправитель формирует сообщение какого-либо рода и затем передает его потоку-получателю. Поток-получатель распаковывает сообщение и совершает то действие, которое требуется от него в приложении.

Сообщения реализуются при помощи указателя на объект неизвестного типа (void), как это часто делается в языке C. Такой подход позволяет брать данные произвольной длины и типа и приводить их к указателю типа void. После этого можно передавать данные внутри системы произвольным образом. Это очень удобно, поскольку можно пересылать между потоками, по сути, что угодно. Оборотная сторона медали заключается в том, что приемник в данном случае должен иметь представление о содержимом сообщения, чтобы привести его к надлежащему типу данных.

Приведем набор API-функций для работы с сообщениями:

```
VDK_CreateMessage()
VDK_DestroyMessage()
VDK_ForwardMessage()
VDK_FreeMessagePayload()
VDK_GetMessageDetails()
VDK_GetMessagePayload()
VDK_GetMessageReceiveInfo()
VDK_MessageAvailable()
VDK_PendMessage()
VDK_PostMessage()
VDK_SetMessagePayload()
```

Итак, существуют API-функции для получения информативной составляющей сообщения, а также различной информации о сообщении.

Несмотря на то, что это выходит за рамки данной статьи, стоит отметить, что сообщения можно использовать для организации взаимодействия в многоядерных или многопроцессорных системах. Сообщения можно передавать, например, по последовательному порту или при помощи совместно используемой памяти. Второй вариант характерен для случая работы с двуядерным процессором Blackfin ADSP-BF561.

Требования к объему памяти

Для начала оценим, каков будет размер кода VDK. Как правило, он сильно зависит от характера вашего приложения.

В таблице 1 показан объем памяти данных и памяти программ (в байтах), занимаемый VDK на этапе компоновки. Эти значения соответствуют только части исполняемого файла, отвечающей за VDK, а не всему приложению. Измерения производились в VisualDSP 4.5 при включенной функции удаления «мертвого» кода и данных.

Таблица 1. Используемая VDK память (в байтах) для типового приложения

Приложение	Код	Данные
Один поток на языке C, без вызовов API	5384	1120
Два потока на языке C, без вызовов API	5584	1208
Два потока, с использованием статических семафоров	6916	1252
Добавление критических областей	7068	1252
Добавление передачи сообщений	9260	1292
Добавление окна истории на 512 событий и средств контроля (сценарий с возможностью отладки)	13 304	9536

Первая строка соответствует полученному искусственным образом наилучшему сценарию, в котором имеется только один поток, написанный на языке C, и отсутствуют вызовы API-функций. В данном случае VDK занимает около 5 кбайт кода и 1 кбайт данных. В каждой последующей строке таблицы 1 функциональные возможности наращиваются. При наличии двух потоков и их координировании с помощью статических семафоров вы получаете около 7 кбайт кода. При этом количество данных увеличивается не более чем на 10%. Добавьте передачу сообщений, и вы получите порядка 9 кбайт.

Последняя строка соответствует сценарию с возможностью отладки. В нем реализовано окно истории, которое обеспечивает доступ к расширенным возможностям отладки VisualDSP. В данном случае объем кода достигает приблизительно 13 кбайт, а данных — 10 кбайт. Заметьте, что в окончательном отлаженном варианте приложения требования к объему памяти будут не столь велики.

Быстродействие

Теперь давайте определим количество тактов, которое затрачивается на реализацию наиболее критических с точки зрения производительности аспектов VDK. Производительность будет оцениваться для случая, когда тестовое приложение целиком размещается в памяти L1. Если отдельные части приложения размещаются в L3, то производительность, вероятно, уменьшится. Однако если правильно работать с кэшем, то вы сможете достичь чисел, близких к показанным здесь. Приведенные оценки производительности были получены на процессоре Blackfin ADSP-BF533.

В тестовом приложении имеется пять потоков, работающих на двух разных уровнях приоритета. Время загрузки, как можно увидеть в таблице 2, составляет 15 311 тактов. Если смены потоков в приложении не происходит, то временные метки занимают всего 67 тактов. При смене потока временная метка составляет 722 такта. Аналогичным образом, установка семафора занимает всего 76 тактов в приложениях без смены потока. При смене потока количество тактов увеличивается до 286. Помещение в стек критической области, инкремент глобальной переменной и извлечение критической области из стека занимает порядка 200 циклов. И, наконец, на создание нового потока затрачивается около 2300 циклов.

Таблица 2. Количество циклов, затрачиваемых на выполнение различных задач в VDK

Событие	Циклы
Загрузка (от перехода по вектору сброса до выполнения первой команды в функции выполнения потока с наивысшим приоритетом)	15 311
Tick, без смены потока	67
Tick, со сменой потока	722
Установка семафора, без смены потока	76
Установка семафора, со сменой потока	286
Помещение критической области в стек, инкремент глобальной переменной, извлечение критической области из стека	199
Создание нового потока, без смены потока	2352

Заключение

VDK — это бесплатное небольшое ядро операционной системы с поддержкой многозадачности и приоритетов, которое отлично подходит для приложений, где требуются высокая производительность и малая стоимость. Для ядра VDK необходим небольшой объем памяти, оно незначительно влияет на производительность процессора. VDK поставляется с таким дополнительным связующим программным обеспечением, как стек TCP/IP, стеки USB и несколько драйверов устройств, что упрощает применение этого ядра в самых разнообразных приложениях. Оно хорошо подходит для таких приложений, как телекоммуникационное оборудование (например, VoIP-телефоны), промышленные системы (например, управление двигателями) и автомобильная электроника.

Ядро VDK меньше годится для систем, где необходимы, например, сложный графический интерфейс, а также высокая надежность или специализированные функции. К счастью, пользователи процессоров Blackfin имеют возможность выбора из целого ряда других ОС, среди которых наверняка будут те, которые смогут удовлетворить потребности таких систем. ■

Литература

1. Intro to Real-Time Operating Systems. http://www.dspdesignline.com/showArticle.jhtml?articleID=210600038
2. http://www.analog.com/en/embedded-processing-dsp/blackfin/content/blackfin_operating_systems/fca.html
3. Understanding and selecting real-time operating systems. http://www.dspdesignline.com/showArticle.jhtml?articleID=193003884