scmRTOS

ОПЕРАЦИОННАЯ СИСТЕМА РЕАЛЬНОГО ВРЕМЕНИ

для однокристальных микроконтроллеров

Версия 5

Оглавление

Листинги

Термины и сокращения

C

Процедурный низкоуровневый язык программирования общего назначения.

C++

Язык программирования общего назначения, поддерживающий процедурную, объектную и объектно-ориентированную парадигмы программирования.

EC++

Embedded C++, является подмножеством C++. Не поддерживает пространства имён, шаблоны, множественное наследование, RTTI, обработку исключений, новый синтаксис явного преобразования типов.

ISR

Interrupt Service Routine – обработчик прерываний.

TOS

Top Of Stack, вершина стека. Адрес элемента[1] стека, на который указывает аппаратный указатель стека процессора.

Вытеснение

Совокупность действий элементов операционной системы, направленных на принудительную передачу управления от одного процесса другому.

Исполняемая функция процесса

Статическая функция-член класса[2] процесса, реализующая самостоятельный асинхронный поток выполнения программы в виде бесконечного цикла.

[1] Ячейки памяти.
[2] Экземпляра шаблона.

Карта процессов

Объект операционной системы, содержащий один или несколько тегов процессов. Физически реализуется на основе целочисленной переменной. Каждый бит в карте процессов соответствует одному процессу и однозначно отображается на приоритет процесса.

Кольцевой буфер

Объект данных, представляющий собой очередь. Имеет два порта данных (функции доступа) – входной для записи и выходной для чтения. Реализуется на основе массива и двух индексов (указателей), обозначающих начало и конец очереди. По достижении физического конца массива, запись/чтение начинается с начала, т.е. индексы перемещаются по кольцу, отчего объект и получил своё название.

Контекст процесса

Программно-аппаратное окружение исполняемого кода, включающее в себя регистры процессора, указатели стеков и другие ресурсы, необходимые для выполнения программы. Т.к. передача управления от одного процесса другому в вытесняющей ОС может производиться в непредсказуемый момент, контекст процесса должен быть сохранён до следующего получения этим процессом управления. Поскольку каждый процесс выполняется самостоятельно и асинхронно по отношению к другим процессам, для обеспечения правильности работы вытесняющей ОС каждый процесс должен иметь собственный контекст.

Конфигурация ОС

Совокупность макросов, типов, других определений и объявлений, задающих количественные и качественные характеристики и свойства операционной системы в конкретном проекте. Конфигурация осуществляется путём определения содержимого специальных заголовочных конфигурационных файлов, а также некоторым пользовательским кодом, выполняемым до запуска ОС.

Критическая секция

Фрагмент кода, при выполнении которого запрещена передача управления. В ***scmRTOS*** в настоящее время реализуется простейшим[1] способом путём общего запрещения прерываний.

МК

Микроконтроллер.

ОЗУ

Оперативное запоминающее устройство – память.

ОС

Операционная система.

ОСРВ

Операционная система реального времени.

Планировщик

Элемент ядра ОС, осуществляющий функции по управлению очерёдностью выполнения процессов.

Пользовательский хук

Функция, вызываемая из кода ОС, тело которой должно быть определено пользователем. Это позволяет исполнять код, определяемый пользователем, непосредственно из внутренних функций операционной системы, не модифицируя её код.

Чтобы не вынуждать пользователя определять тело хуков, которые он не использует[2], вызов хука осуществляется только в том случае, если это разрешено при конфигурации. Т.е., если пользователь желает использовать тот или иной хук, он должен разрешить использование этого хука и определить его тело.

Порт ОС

Совокупность общего и платформеннозависимого кода ОС, настроенного на конкретную программно-аппаратную платформу.

[1] Как следствие, самым быстрым и дешёвым в смысле накладных расходов.
[2] А также избежать неоправданных накладных расходов.

Приоритет процесса

Свойство процесса (объект целочисленного типа), определяющее очерёдность выбора процесса при операциях в планировщике и других элементах ОС. Является уникальным идентификатором процесса.

Процесс операционной системы

Объект, реализующий выполнение целостного самостоятельного асинхронного по отношению к другим фрагмента программы, включая поддержку передачи управления как на уровне процессов, так и на уровне прерываний.

Профилировщик

Объект, измеряющий тем или иным способом распределение процессорного времени между процессами и имеющий средства предоставления этой информации для пользователя.

Расширения ОС

Программные объекты, расширяющие функциональность операционной системы, но не в входящие в основой[1] состав ОС. Примером расширения может служить профилировщик работы процессов системы.

Системный таймер

Аппаратный таймер целевого процессора, выбранный в качестве источника генерации прерываний с заданным периодом, а также функция ОС, вызываемая из ISR таймера и реализующая логику обработки таймаутов процессов.

Средства межпроцессного взаимодействия

Объекты и/или расширения ОС, предназначенные для безопасного взаимодействия (синхронизации работы и обмена данными) разных процессов, а также организации работы программы по событиям (event-driven execution), возникающим в прерываниях и процессах.

[1]Основной состав ОС — компоненты системы, необходимые для обеспечения работоспособности всех основных средств.

Стек прерываний

Специально выделенная область ОЗУ, предназначенная для использования в качестве стека при выполнении кода обработчиков прерываний. Если в программе используется стек прерываний, то при входе в обработчик прерываний указатель стека процессора переключается на стек прерываний, а при выходе – обратно на стек процесса.

Стек процесса

Область памяти в виде массива, являющегося членом-данным объекта процесса, используемая в качестве стека в исполняемой функции процесса. Является также местом, в котором сохраняется контекст процесса при передаче управления.

Стековый кадр

Stack Frame. Представляет собой совокупность данных, размещённых в стеке процесса так, как это имеет место при сохранении контекста процесса при передаче управления[1].

Таймаут

Промежуток времени, заданный объектом целочисленного типа, используемый для организации условного или безусловного[2] ожидания событий процессами.

Тег процесса

Маска двоичного числа, содержащая только один ненулевой бит, позиция которого однозначно связана с номером приоритета процесса. Как и приоритет процесса, тег является уникальным идентификатором, имеющим иное, нежели приоритет процесса, представление. Каждое представление (приоритет или тег) используется там, где оно более уместно с точки зрения эффективности работы программы.

Фоновый процесс

`IdleProc`. Является системным процессом, получающим управление, когда все пользовательские процессы находятся в состоянии ожидания со-

[1] Существует другой смысл этого термина – это область памяти в стеке, которая выделяется компилятором для размещения в ней локальных объектов выполняемой функции. В настоящем документе используется трактовка, приведённая в пояснении термина.
[2] Например, при использовании функции **`sleep()`**.

бытий. Этот процесс не может переходить в состояние ожидания[1], может выполнять вызов пользовательского хука, если это разрешено при конфигурации.

Ядро

Важнейшая и центральная часть операционной системы, осуществляющая функции по организации процессов, планировку их выполнения, поддержку межпроцессного взаимодействия, системного времени и расширений ОС.

[1] Что вполне понятно – управление отдавать больше некому.

Предисловие

Все приведённые ниже рассуждения, оценки, выводы основаны на личном опыте и знаниях автора документа и вследствие этого обладают известной долей субъективизма, что обуславливает наличие как неточностей, так и ошибок. Поэтому просьба рассматривать нижеизложенное с учётом этого обстоятельства.

* * *

Главными причинами, вызвавшими появление ***scmRTOS***, были:

- появление в последнее время недорогих однокристальных микроконтроллеров (МК) с достаточно приличными (для использования операционной системы (ОС)) ресурсами (1 и более килобайт ОЗУ);
- осознание того факта, что и на мелких процессорах с очень ограниченными ресурсами вполне возможно организовать event-driven поток управления с приоритетным вытеснением, не теряя при этом возможности реализовать всю требуемую функциональность;
- недоступность (на момент начала разработки) аналогичных ОС, способных работать уже на кристаллах с размером ОЗУ от 512 байт.

Немного истории. Работа началась с того, что было написано простое ядро с вытесняющим планированием процессов для микроконтроллера **MSP430F149** фирмы **Texas Instruments**. Так получилось, что тот момент совпал по времени с бета-тестированием EC++ компилятора фирмы **IAR Systems**, отчасти поэтому базовые механизмы были реализованы с использованием EC++. Дальнейший опыт показал, что выбор в качестве языка разработки C++ (даже в урезанном варианте EC++) имеет ряд преимуществ перед C.

Основные преимущества: более защищённая модель программных объектов и, как следствие, более простое и безопасное использование; более строгий контроль типов; встроенные в язык средства автоматизации рутинных операций — инициализация объектов, повторное использование кода через наследование, механизм виртуальных функций, шаблоны и др.

Основной недостаток C++ – большая по сравнению с C сложность языка. На момент начала разработки ***scmRTOS*** были ещё некоторые трудности как с распространённостью компиляторов, так и с поддерживаемыми ими средствами языка — в частности, первые версии компиляторов отвечали требованиям специ-

фикации EC++ (Embedded C++), которая является подмножеством C++ и не включает в себя такие мощные средства языка, как, например, шаблоны C++. С тех пор прогресс не стоял на месте, разработчики компиляторов для embedded процессоров «перешагнули» границу EC++ и расширили арсенал средств почти до полного, исключая обработку исключений (exception handling) и определение типов во время выполнения (RTTI – Real Time Type Identification), что является разумным, т.к. эти два средства требуют для реализации довольно много ресурсов при неудовлетворительном для многих МК быстродействии.

К настоящей редакции документа проблем с доступностью хороших компиляторов C++ для процессоров, пригодных для эффективного использования на них ***scmRTOS***, не существует. Сегодня уже сложнее найти платформу, для которой нет компилятора C++, нежели ту, для которой есть.

Следующим шагом был перенос ОС на другую платформу – микроконтроллеры семейства **AVR** фирмы **Atmel**. Поскольку архитектура AVR весьма отличается от архитектуры **MSP430** (первый построен по Гарвардской, а второй – по фон Неймановской, в **AVR** при использовании пакетов **IAR EWAVR** реализованы два стека: один для данных, другой для адресов возвратов), процесс переноса помог выявить ряд серьёзных изъянов в структуре ОС, что привело к вводу дополнительной функциональности, позволяющей более тонко учесть особенности как самого микроконтроллера, так и компилятора под него – в частности, наличие отдельного стека для адресов возвратов.

На портах под обе архитектуры было реализовано несколько реальных проектов, что явилось хорошим тестом на прочность для ***scmRTOS***.

С появлением поддержки компиляторами механизма шаблонов была выпущена версия ***scmRTOS*** следующего поколения (версии 2.xx), где объекты процессов и ряд средств межпроцессного взаимодействия были реализованы на основе шаблонов C++. При этом изменения коснулись и других частей ОС – например, класс `OS`, выполнявший функцию пространства имён для составных частей операционной системы заменён на настоящее пространство имён (***namespace***), часть средств, входивших в версии 1.xx, удалена. В частности, удалены Mailboxes и Memory Manager. Mailboxes удалены в силу того, что в версии 2.xx введены более мощные и безопасные средства на основе шаблонов, выполняющие те же функции – это arbitrary-type channels и messages. Диспетчер памяти удалён из состава ОС, т.к. он не имеет прямого отношения к собственно ОС и ранее нужен был только для поддержки механизма передачи сообщений через Mailboxes. Поскольку теперь Mailboxes заменены на более мощные и универсальные

средства, необходимость в диспетчере памяти для нужд операционной системы отпала. Если пользователю может понадобиться подобный или другой диспетчер памяти, то требуемый диспетчер памяти может быть свободно добавлен к проекту безо всяких конфликтов с кодом ОС – ОС и диспетчер памяти есть сущности ортогональные.

Чуть позже уже в версии 2 появился ещё один порт — для процессора **Blackfin** фирмы **Analog Devices**.

Приблизительно в это время к проекту проявили интерес несколько высококвалифицированных специалистов-разработчиков, которые выпустили ряд портов для процессоров, с которыми им доводилось работать. Так появились порты для процессоров на ядре **ARM7**/**IAR Systems** с примерами для **AT91SAM7** (**Atmel**), **ADuC70xx** (**Analog Devices**), **LPC2xxx** (**NXP**) и **STR71x** (**STMicroelectronics**), **AVR**/**GCC**, **FR** (**Fujitsu**)/**Softune**, **MSP430**/**GCC**.

Проект заметно вырос по объёму и по сути стал открытым (публичным), что подразумевало наличие средств для организации командной работы. Одним из разработчиков был предложен вариант с организацией одноимённого с ОС проекта на известном Интернет-ресурсе **SourceForge** (www.sourceforge.net), что позволило использовать общедоступный репозиторий системы управления версиями для работы над исходными кодами операционной системы, а также систему файлового хостинга для размещения архивов выпущенных релизов проекта.

Все эти события привели к выходу в свет версии 3. Технические изменения в ней преимущественно были связаны с расширением аппаратной базы (появлением новых портов) и повышением удобства и гибкости использования на уровне пользовательского проекта — в частности, выбор и настройка системного таймера, а также выбор прерывания переключения контекстов (см. далее) были вынесены на уровень исходных файлов пользовательского проекта.

Использование ***scmRTOS*** версии 3 продолжалось несколько лет, было выпущено несколько релизов, которые преимущественно были нацелены на устранение обнаруженных ошибок. В течение этого периода на основе опыта реальной эксплуатации были намечены некоторые тенденции дальнейшего развития проекта, а также выявлен ряд аспектов, реализация которых могла бы сделать ***scmRTOS*** более удобной и безопасной в использовании.

В частности, несколько раз со стороны пользователей возникали вопросы о расширении набора средств межпроцессного взаимодействия для более полного удовлетворения потребностей целевых проектов. Такой подход не представляется

верным — очевидно, что расширение базового набора сервисов для удовлетворения частных потребностей того или иного пользовательского проекта легко приведёт к неконтролируемому росту числа сервисных типов, что породит сложности с документированием, сопровождением и т.д. Поэтому одним из разработчиков была предложена идея разработать механизм расширения для средств межпроцессного взаимодействия, хорошо документировать его и предоставить пользователю, который нуждается в каком-либо специализированном классе-сервисе, разработать конкретную реализацию, наилучшим образом удовлетворяющую потребности пользовательского проекта, самостоятельно.

Это привело к разработке механизма расширений функциональных возможностей ОС, позволяющего добавлять новые средства, не требуя модификации кода операционной системы.

Кроме того, для повышения удобства и безопасности использования был разработан и реализован ряд средств для отладки и тестирования поведения операционной системы в составе целевого проекта. Сюда вошли возможности по контролю за достаточностью объёма памяти (запаса), выделенной под стеки процессов, фиксации адреса объекта-сервиса, в ожидании которого находится процесс, не готовый в выполнению, а также возможность прервать работу любого процесса в произвольный момент и запустить процесс с самого начала и средство измерения относительного времени активной работы процессов (профилировка), выполненное в качестве расширения.

Кроме этих достаточно крупных изменений была проделана работа по приведению исходного кода к единообразному состоянию путём выработки единого стиля кодирования, необходимого для успешной работы над коллективными проектами. Это привело к переименованию функций, что повлекло за собой некоторую несовместимость с предыдущими версиями 3.xx. Для облегчения перехода на новую версию 4.0 предусмотрены специальные средства, которые носят временный характер и в следующем релизе ***scmRTOS*** будут удалены[1].

Все вышеперечисленное в совокупности составило суть новой — 4-й — версии, релиз которой состоялся в 2012 году. Использование этой версии ***scmRTOS*** на протяжении нескольких лет показало на практике стабильность и целостность системы.

[1]Это сделано для того, чтобы не делать перевод на новую версию текущих проектов слишком резким – и так новшеств немало. Пользователю предоставляется время постепенно привести свой рабочий код в соответствие с новой системой кодирования. В принципе, вся работа в основном сведётся к корректировке имён функций, что представляется несложной задачей.

Нужно отметить, что на протяжении всего описанного выше периода процесс разработки вёлся с использованием системы управления версиями[1] **Subversion** (svn). Процесс развития средств разработки показал, что существует более эффективная система управления версиями программных продуктов — **Git**, и было принято решение перевести проект на эту VCS одновременно с переносом проекта на более удобный для данной системы хостинг github.com.

Перенос проекта с **Subversion** на **Git** объективно повлёк за собой изменения в структуре файлов и директорий, что, в свою очередь, породило несовместимость и обусловило необходимость в правке конфигурационных файлов, отвечающих за сборку. Пользуясь этим обстоятельством, была дополнительно проведена работа по переименованию части директорий и файлов с целью приведения их к более консистентному виду с целью более удобного использования.

Помимо этого сама ОС получила некоторые изменения, расширяющие её функциональность. В частности, в код процесса были добавлены средства, позволяющие осуществлять старт процессов в неактивном (suspended) состоянии — это оказалось удобно в некоторых случаях, когда возникает необходимость начать выполнение собственно кода процесса по команде извне.

Порты **CortexM0-M3-M4/GCC** были сведены в один унифицированный порт **CortexM/GCC**, а также добавлен порт **Blackfin/CCES**. Позднее ОС была дополнена протом **Blackfin/GCC**.

Также было добавлено расширение, реализующее карусельный режим планировки для отдельных процессов. Из кода ОС была удалена поддержка совместимости со устаревшими именами, а также некоторые другие менее значимые изменения.

Всё это привело к выпуску 5-й версии ***scmRTOS***, которой и посвящён настоящий документ.

* * *

Что нужно для того, чтобы использовать ***scmRTOS***?

Нужен, в первую очередь, соответствующий компилятор C++. В настоящее время, как уже говорилось выше, доступность таких компиляторов не представляет проблем.

[1] Version Control System (VCS).

Второе. Нужно, чтобы используемый микроконтроллер имел необходимый минимум ОЗУ. Эта величина определена как 512[1] байт. На МК с меньшим количеством ОЗУ ***scmRTOS*** **работать** не будет. То есть, можно, конечно, попытаться запустить её и на 256 байтах – один или два крохотных процесса, возможно, влезут (не проверялось), но здравый смысл подсказывает, что реальную задачу такая конфигурация решать вряд ли сможет.

Ну, и третье. Нужно хоть немножко знать C++, хотя бы основы, базовые концепции и синтаксис. Это совсем не так сложно, как может показаться вначале, тем более, что эти знания, учитывая темпы, с которыми C++ входит в embedded мир, почти наверняка пригодятся в будущем. Сам язык не требует, чтобы его сразу весь выучили – для начала достаточно освоить понятие класса (оно ключевое), механизмы наследования и шаблонов. В остальном можно вполне пользоваться привычными «сишными» конструкциями.

* * *

scmRTOS – маленькая ОС. В ней использованы очень простые[2] механизмы реализации требуемых возможностей, потребление ОЗУ для служебных целей минимизировано. Благодаря простоте и малому размеру процесс освоения значительно упрощается. Главное в освоении – понять основные аспекты (планировка процессов, параллельный доступ к ресурсам и межпроцессное взаимодействие), а разобраться во внутренних механизмах, на самом деле, будет несложно. И в этом всегда можно обратиться к разработчикам проекта, которые охотно и оперативно окажут квалифицированную помощь в решении возникших вопросов.

* * *

Эта ОС разрабатывалась для себя и для всех желающих её использовать. Любой изъявивший желание может использовать и/или распространять её совершенно бесплатно как в образовательных целях, так и в некоммерческих и коммерческих проектах. ***scmRTOS*** поставляется «как есть» (“as is”), никаких гарантий, естественно, не предоставляется.

[1] На время появления первой версии ***scmRTOS*** (и до момента написания данного документа) градация по объему ОЗУ среди доступных МК, на которые была ориентирована ОС, была такой: 128 байт, 256 байт, 512 байт, 1024 байта и т.д. Т.е. отсутствовали, в частности, варианты с 384 и 768 байтами ОЗУ, поэтому нижней границей указано значение в 512 байт. На 384 байтах уже можно было бы попытаться нечто соорудить, хотя это представляется не слишком разумным.

[2] И, как следствие, быстрые.

Если подходить строго, то проект ***scmRTOS*** существует под вполне конкретной лицензией, которая не является чем-то новым и представляет собой широко известную лицензию **MIT**:

> **Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:**
>
> **The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.**
>
> **THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.**

Глава 1 Введение

Многие вещи нам непонятны не потому, что понятия наши слабы, но потому, что сии вещи не входят в круг наших понятий.

К. Прутков

Тот, кто хорошо знаком с проблемами и принципами построения ОС для малых процессоров, может пропустить настоящий раздел, хотя желательно все же его прочитать (он небольшой), чтобы более чётко понять контекст дальнейшего описания, рассуждений и выводов.

* * *

Что есть операционная система (ОС) вообще? Вопрос в достаточной мере абстрактен. Исходя из разных точек зрения, можно давать совершенно разные ответы, с разной степенью детализации, приводя совершенно не похожие друг на друга примеры. Очевидно, что ОС для больших машин настолько отличается от ОС для 8-разрядного процессора какой-нибудь embedded системы, что найти там общие черты – задача непростая и вдобавок бессмысленная.

Поскольку в нашем случае речь идёт о микроконтроллерах, то и аспекты ОС рассматриваются соответствующие.

Итак, в контексте текущего рассмотрения, операционная система – совокупность программного обеспечения (ПО), дающего возможность разбить поток выполнения программы на несколько независимых, асинхронных по отношению друг к другу процессов и организовать взаимодействие между ними. Т.е. внимание обращено на базовые функции, оставляя в стороне такие вещи, присущие ОС для больших машин, как файловые системы (т.к. и файлов-то никаких обычно нет), драйверы устройств (которые вынесены на уровень пользовательского ПО) и т.п.

Таким образом, исходя из того, что основная функция ОС – поддержка параллельного асинхронного исполнения разных процессов и взаимодействия

между ними, встаёт вопрос о планировании (Scheduling) процессов, т.е. когда какой процесс должен получить управление, когда отдать управление другому процессу и т.д. Эта задача возлагается (хоть и не полностью) на часть ядра ОС, называемой планировщиком (Scheduler). По способу организации работы планировщики бывают:

- с приоритетным вытеснением (preemptive, когда при возникновении «работы» для более приоритетного процесса, он вытесняет менее приоритетный (т.е. более приоритетный при необходимости отбирает управление у менее приоритетного). Примерами ОС с такими планировщиками являются, например, широко известная и популярная коммерческая ОС реального времени **uC/OC-II** (www.micrium.com) и бесплатная **proc** (www.nilsenelektronikk.no);
- с вытеснением без приоритетов (round-robin или «карусельного» типа), когда каждый процесс получает квант времени, по истечении которого управление у данного процесса отбирается операционной системой и передаётся следующему в очереди процессу;
- без вытеснения (cooperative), когда процессы выполняются последовательно, и для того, чтобы управление от одного процесса перешло к другому, нужно, чтобы текущий процесс сам отдал управление системе. Кооперативные планировщики также могут быть приоритетными и неприоритетными. Примером кооперативной ОС с приоритетным планированием является **Salvo** (www.pumpkininc.com). Оставшийся вариант – кооперативный планировщик без приоритетов – это, например, всем хорошо известный бесконечный цикл в функции main, откуда по очереди вызываются различные функции, являющиеся «процессами».

Это лишь некоторые (базовые) типы, реально встречаются различные комбинации из упомянутых вариантов, что вносит значительное разнообразие в алгоритмы планирования процессов.

Операционная система реального времени (ОСРВ, Real-Time Operating System - RTOS) – ОС, обладающая одним важным свойством: время реакции на события в такой ОС *в известной степени* детерминировано — другими словами, имеется возможность оценить, через сколько времени с момента поступления события оно будет обработано. Конечно, это достаточно приблизительная оценка, т.к. на момент возникновения события система может находиться в прерывании, которое не может быть прервано, или процесс, который должен обработать событие, не имеет в данный момент контроля над процессором (например, имеет низкий приоритет и вытеснен другим, более приоритетным процессом, выполняющим свою работу, при использовании вытесняющей ОС). Т.е. в случае большой загрузки время реакции на событие может варьироваться в значительных пределах. Но основные нити управления в руках пользователя — ведь он решает, какой код использовать в прерывании или насколько критично занятие вычислитель-

ного ресурса приоритетным процессом, и в соответствии с заданными целями и имеющимися возможностями пользователь может определить конфигурацию целевого проекта так, чтобы *программа работала управляемо и предсказуемо в масштабе реального времени*. И операционная система тут не помеха, а наоборот – подспорье в достижении этих целей.

Вот эта возможность держать под контролем динамические характеристики программы, работающей в окружении средств операционной системы, и является ключевым аспектом, отличающим операционные системы реального времени от остальных операционных систем. Следствием вышесказанного является то обстоятельство, что ОСРВ – это, как правило, небольшие по объёму системы, с достаточно простыми и понятными внутренними связями, что даёт предсказуемое поведение кода ОС во всех ситуациях при целевом использовании.

Большие же операционные системы, являющиеся, как правило, сложными комплексами программного обеспечения, рассчитанными на работу в разнообразном программно-аппаратном окружении и ориентированными на широкий спектр решаемых задач, содержат в себе нетривиальные механизмы взаимодействия внутренних объектов, что порождает трудности с обеспечением предсказуемости поведения в смысле времени реакции на то или иное событие. Т.е. такие ОС сами по себе вносят в цепочки обработки событий слабоконтролируемые задержки, что делает их малопригодными для построения систем, ориентированных на обработку событий в масштабе реального времени.

Очевидно, что способность ОС реагировать на события определяется в первую очередь типом планировщика. Из упомянутых выше наиболее «быстрыми» в смысле реакции на события являются ОС с приоритетными вытесняющими планировщиками – у них время реакции (при прочих равных условиях) минимально, т.к. в такой ОСРВ при возникновении события планировщик стремится сразу отдать управление коду, который ожидает это событие.

По детерминированности (но не по скорости) времени отклика из оставшихся на первом месте - ОС с вытесняющими round-robin планировщиками, в этом случае событие будет обработано тогда, когда дойдёт очередь до процесса, ответственного за обработку. Время реакции, очевидно, тут далеко не оптимальное.

В операционных системах с кооперативными планировщиками время реакции на события в большей степени определяется не столько самой ОС, сколько прикладной программой, которая должна быть организована так, чтобы не занимать надолго процессор внутри процесса. Хорошим решением является так

называемая **FSMOS** (Finite State Machine Operating System, http://www.nilsenelektronikk.no/), где каждый процесс организован как конечный автомат и пребывание внутри каждого состояния делается как можно более коротким. Это позволяет повысить скорость реакции на события, но по-прежнему она (скорость реакции) определяется в первую очередь прикладной программой: если пользовательский процесс не позаботится о том, чтобы отдать управление, весь остальной код, включая код самой ОС, окажется неработоспособным.

Напрашивается вопрос: если самые быстрые (в смысле скорости реакции на события) ОСРВ – это ОС с вытесняющим приоритетным планированием, то зачем тогда остальные? Ответ может выглядеть так: вытесняющие ОС имеют серьёзный недостаток – они значительно более требовательны к ОЗУ, чем невытесняющие. Это принципиальный аспект: причина этого кроется как раз в способности вытеснять процессы, т.е. в силу того, что любой процесс может быть прерван в любой момент времени (непредсказуемый для процесса), его – процесса – аппаратное окружение – контекст, куда относится содержимое регистров процессора, состояние стека, должно быть сохранено соответствующим образом, чтобы при следующем получении управления этим процессом он смог продолжить свою работу как ни в чем не бывало. Контекст сохраняется обычно в стеке, который у каждого процесса свой. Таким образом, потребность в ОЗУ резко возрастает: если у программы без ОС (или с кооперативной ОС) потребность в ОЗУ определяется количеством статических[1] переменных + размер стека, который является общим для всех, то при использовании ОС с вытесняющий многозадачностью каждый процесс требует стек размером равным: размер контекста + глубина вложенности вызова функций (включая вызовы обработчиков прерываний, если не используется отдельный стек для прерываний) + переменные процесса.

Например, в **MSP430**[2] размер контекста равен порядка 30 байт (12 регистров + SR + SP = 28 байт), при вложенности вызовов функций до 10 (и это немного) – это ещё 20 байт, уже получаем порядка 50 байт только для обеспечения работы процесса ОС в самом минимально возможном варианте, т.е. накладные расходы по памяти. И так для каждого процесса. Если процессов штук пять-шесть, то одних накладных получается порядка 250-300 байт, что соизмеримо с общим объёмом ОЗУ многих однокристальных микроконтроллеров. Поэтому ис-

[1] Имеется в виду статический класс памяти (static storage duration), а не тип связывания (internal linkage).

[2] Для других МК картина похожая – контекст может быть больше или меньше, но принцип не меняется: у каждого процесса свое отдельное окружение, занимающее ресурсы (ОЗУ) которые, в силу асинхронности работы процессов, не могут быть использованы другими процессами программы.

пользование ОС с вытесняющим планированием натыкается на принципиальные трудности при использовании МК с объёмом ОЗУ менее полукилобайтн. Такие МК – ниша для кооперативных ОС.

Ещё одно принципиальное ограничение – аппаратный стек в некоторых МК. По этой причине вытесняющая ОС, видимо, никогда не сможет работать на микроконтроллерах, например, семейства **PIC16** (**Microchip**), поэтому МК этого семейства также кандидаты на использование в них кооперативных ОС.

Процессы в кооперативных ОС тоже имеют контексты, которые переключаются при передаче управления от одного процесса к другому, но размер этих контекстов очень мал (по сравнению с суммарным размером контекстов процессов в вытесняющих ОС). Сохранять все рабочие регистры процессора нет необходимости, т.к. управление передаётся не в произвольный момент времени, а во вполне определённый, поэтому сохранение регистров происходит таким же образом, как при вызове любой функции.

Что касается ОС с чистым вытесняющим планировщиком round-robin (т.е. не приоритетным), то глубокого смысла использовать их для решения реальных задач в сегменте малых ОСРВ не просматривается в силу объективных недостатков такого типа планировщика: вытеснение (т.е. асинхронное отбирание управления и передача его другому процессу) реализовано, соответственно, ресурсы для отдельного стека каждому процессу имеются (и расходуются), таким образом, главное ограничение преодолено — и вполне можно сделать приоритетный планировщик, он не сложнее карусельного (round-robin). Справедливости ради надо отметить, что реально ОС с такими планировщиками (по крайней мере, в сегменте малых процессоров) не видно ни одной.

Ещё встречаются комбинированные планировщики: например, приоритетное планирование используется наряду с карусельным: если ОС допускает несколько (два и более) процессов с одинаковым приоритетом, то запуск процессов на уровне данного приоритета в такой ОС происходит по схеме round-robin. Такое планирование несколько сложнее простого приоритетного, что отражается как в быстродействии при передаче управления между процессами, так и в размере кода, а *объективные* преимущества не особенно заметны.

* * *

Операционная система реального времени для однокристальных микроконтроллеров ***scmRTOS*** (Single-Chip Microcontroller Real-Time Operating Sys-

tem) использует приоритетное вытесняющее планирование процессов. Как видно из названия, ключевой особенностью данной ОС является то обстоятельство, что она ориентирована именно на применение в однокристальных микроконтроллерах. Что это означает? Что такого характерно для однокристальных МК в контексте операционных систем реального времени с вытесняющим планированием? Ключевой особенностью таких МК является ограниченное и, как правило, небольшое количество доступной оперативной памяти (ОЗУ), объем которой невозможно расширить, не заменяя сам МК на более мощный. К моменту создания ***scmRTOS*** у подавляющего большинства микроконтроллеров эта величина не превышала 2-4 килобайт, и это были уже приличные по остальным ресурсам кристаллы – с объёмом ПЗУ 16 и более (исчисляется десятками) килобайт, большим количеством периферийных устройств (таймеров, АЦП, портов ввода-вывода, последовательных портов, как синхронных (SPI, I^2C), так и асинхронных (UART)).

В настоящее время модельный ряд однокристальных микроконтроллеров значительно расширился, появилось много недорогих и весьма ресурсоёмких его представителей, способных работать на приличных (для «однокристаллок») частотах – в 50 и выше мегагерц. Но и спектр задач, решение которых возлагается на эти приборы, тоже изменился, что выдвинуло новые требования к ресурсам, особенно к тем, которые не поддаются расширению — в первую очередь это внутренняя оперативная память.

Таким образом, внутреннее ОЗУ является одним из наиболее дефицитных ресурсов МК для реализации ОС с вытесняющим планированием.

Как показано выше, существуют принципиальные ограничения на использование вытесняющего механизма – главным образом из-за требований к объёму оперативной памяти. При разработке описываемой ОС ставилась цель получить минимально ресурсоёмкое решение, чтобы его можно было реализовать на однокристальных МК с объёмом ОЗУ от 512 байт. Основные характеристики ОС закладывались на основе этой главной цели. Именно этим обусловлено ограниченное количество процессов, отказ от механизмов динамического создания/удаления процессов, изменения приоритетов процессов на этапе выполнения программы и т.д, словом, всего того, что может повлечь за собой дополнительные накладные расходы как по размеру памяти, так и по времени выполнения.

Благодаря исходной ориентации на небольшие МК, что позволило применить упрощённые и облегчённые решения, удалось добиться сравнительно неплохого результата. Выигрыш по быстродействию достигнут, главным образом, благодаря очень простому механизму планировки (и вычисления приоритетов),

возможность использования которого обеспечена ограниченным количеством процессов в ***scmRTOS***.

Упрощённая функциональность также даёт выигрыш по ресурсоёмкости: ядро в ***scmRTOS*** занимает порядка 8-12 + 2*(кол-во процессов) байт, данные процесса (без стека) – 5 байт.

* * *

Как уже говорилось, в качестве языка разработки ОС выбран C++. Реально используются не все средства языка C++. В состав неиспользуемых средств входят обработка исключений, множественное наследование и RTTI. Обработка исключений и RTTI «тяжелы» и во многом избыточны для однокристальных МК. К тому же к настоящему моменту почти не существуют компиляторов, разработанных для использования их в embedded области, поддерживающих оба этих механизма.

В настоящее время достойные внимания компиляторы C++ выпускает шведская фирма **IAR Systems**. Существуют компиляторы, например, для следующих аппаратных платформ:

- **ARM7**;
- **AVR (Atmel);**
- **Cortex-M**;
- **MSP430** (**Texas Instruments**) и др.

Кроме компиляторов, выпускаемых специализированными фирмами-производителями компиляторов, такими, как **IAR Systems**, существуют неплохие компиляторы, выпускаемые фирмами-производителями самих процессоров – например, **Analog Devices** (пакет **VisualDSP++**) и **Texas Instruments** (пакет **Code Composer Studio**).

Кроме коммерческих (хороших, но весьма недешёвых) компиляторов существует семейство бесплатных, но достаточно гибких и мощных компиляторов **GCC** (**GNU** Compiler Collection), способных составить достойную альтернативу высококачественным коммерческим продуктам. Спектр поддерживаемых аппаратных платформ компиляторами семейства **GCC** так же весьма представителен (все вышеперечисленные процессоры в него входят), а по широте реализованных возможностей языка программирования C++ и по строгости следования Стандарту C++ **GCC** является одним из лидеров.

Существуют компиляторы и для других платформ. Процесс этот развивается, и тенденция такова, что C++ постепенно занимает нишу C, т.к. он покрывает все возможности C и добавляет к этому принципиально новые средства, позволяющие вести разработку ПО на качественно новом уровне, в частности, сместить акцент в разработке ПО на этап проектирования и даже в известной степени формализовать его. Это ещё одна причина, обуславливающая выбор C++ в качестве языка разработки ОС.

scmRTOS исходно была разработана с использованием компилятора EC++ для **MSP430** от **IAR Systems** и на текущий момент имеет несколько портов под разные программные (IAR Systems, GCC, а также специализированные фирменные программные пакеты) и аппаратные платформы (**MSP430**, **AVR**, **ARM7**, **Cortex-M**, **Blackfin**[1]) — см. оперативную информацию, размещённую на сайте проекта.

* * *

При написании исходного кода ***scmRTOS*** принят следующий стандарт кодирования.

Отступ (indent) принят размером в 4 символа.

Открывающая фигурная скобка блока '`{`' начинается на пустой строке первым символом вровень с первым символом (т.е. под ним) ключевого слова оператора блока или квалификатора функции.

[1] Появление этого процессора в списке поддерживаемых платформ может вызвать удивление – ведь Blackfin является достаточно мощным процессором даже для того, чтобы запускать на нем такие операционные системы, как uCLinux. На самом деле ничего удивительного нет – Blackfin является очень многоплановым процессором – он может быть чисто DSP процессором, может быть центральным процессором системы, на которой запускают сторонние приложения, где интенсивно используется внешняя SDRAM под память кода и данных, кэширование и т.д., а может быть однокристальным (в известной степени) микроконтроллером, у которого весь исполняемый код и все данные размещены во внутренней памяти, интенсивно используется внутренняя периферия и решается широкий класс задач от сбора данных и управления до обработки сигналов (и все эти задачи Blackfin может решать достаточно эффективно).
При этом требуются минимальные накладные расходы как по памяти (размер внутренней памяти Blackfin'а вполне соизмерим с размерами памяти многих других однокристальных МК), так и по быстродействию. Т.е. нужна удобная среда программирования с простой, прозрачной и формализованной организацией потока управления при минимальных требованиях от аппаратной части процессора и возможностью приоритетного вытеснения. Именно в этом ключе и следует рассматривать совместное использование ***scmRTOS*** и Blackfin.
Эти же соображения в полной мере относятся и к другим достаточно мощным процессорам, куда входит большинство представителей с 32-разрядным ядром.

Имена функций выглядят: **`like_this()`**.

Имена переменных выглядят: **`LikeThis`**.

Имена типов, определяемых пользователем, выглядят: **`TLikeThis`**.

Имена псевдонимов встроенных типов выглядят: **`like_this_t`**.

* * *

О принятой терминологии. В ***scmRTOS*** используется термин «процесс» (process) для обозначения части программы, выполняемой циклически, самостоятельно и асинхронно по отношению к остальным частям программы. В литературе и в терминологии, принятой в других ОСРВ для обозначения этого, часто используются термины «задача» (task) и «тред» (thread – «нить», поток). Термин «процесс» был выбран сознательно, т.к. представляется, что он более удачно подчёркивает смысл обозначаемого.

Действительно, «задача» — понятие весьма широкое и может обозначать широкий спектр от школьной задачки по алгебре до боевой задачи роте спецназа. «Тред» — это дословно «нить», т.е., как видно из названия, это нечто такое, что имеет характеристику линейности, а не цикличности. Учитывая вышеприведённые доводы, термин «процесс» представляется более удачным вариантом — действительно, смысл этого слова в явном виде отражает *действие*[1], протяжённое во времени, с возможностью *цикличности*[2] как отдельных составных частей процесса, так и всего процесса в целом.

[1] Коего смысла нет, например, в термине «задача».
[2] Что не отражается термином «тред» - у «нити», как правило, есть начало и конец.

Глава 2 Обзор операционной системы

— Гоги, ты знаешь, что такое «ОС»?

— Канэшна знаю: «ОС» - эта балщой паласатый мух!

— Нэт! Балщой паласатый мух – эта шмел, а «ОС» - эта то, на чём вэртится наша Земла!

Анекдот

2.1. Общие сведения

scmRTOS является операционной системой реального времени с приоритетной вытесняющей многозадачностью. ОС поддерживает до 32 процессов (включая системный процесс `IdleProc`, т.е. до 31 пользовательского процесса), каждый из которых имеет уникальный приоритет. Все процессы статические, т.е. их количество определяется на этапе сборки проекта и они не могут быть добавлены или удалены во время исполнения.

Отказ от динамического создания процессов обусловлен соображениями экономии ресурсов, которые в однокристальных МК весьма ограничены. Динамическое удаление процессов также не реализовано, т.к. в этом немного смысла – память программ, используемая процессом, при этом не освобождается, а ОЗУ для последующего использования должно иметь возможность быть выделяемым/освобождаемым с помощью диспетчера памяти, который сам по себе достаточно непростая вещь, требует приличного количества ресурсов и, как правило, не используется в проектах на однокристальных МК[1].

[1] Имеется в виду стандартный менеджер памяти, поставляемый в составе средств разработки. Существуют ситуации, когда для работы программы необходимо хранить данные между вызовами функций (т.е. автоматический класс хранения данных — на стеке/в регистрах процессора — не подходит), и при этом на этапе компиляции программы неизвестно, сколько всего будет этих дан-

В текущей версии приоритеты процессов также статические, т.е. каждый процесс получает приоритет на этапе сборки проекта и приоритет не может быть изменён во время выполнения программы. Такой подход также обусловлен стремлением сделать систему как можно более лёгкой в части требований к ресурсам и динамичной, т.к. изменение приоритетов в процессе функционирования системы - совсем нетривиальный механизм, который для корректной работы требует анализа состояния всей системы (ядра, сервисов) с последующей модификацией составляющих ядра и остальных частей ОС (семафоров, флагов событий и проч.), что неизбежно порождает длительные периоды работы при блокированных прерываниях и, как следствие, значительно ухудшает динамические характеристики системы.

2.2. Структура ОС

Система состоит из трёх основных составных частей: ядра (Kernel), процессов и средств межпроцессного взаимодействия.

2.2.1. Ядро

Ядро осуществляет:

- функции по организации процессов;
- планирование (Scheduling) как на уровне процессов, так и на уровне прерываний;
- поддержку межпроцессного взаимодействия;
- поддержку системного времени (системный таймер);
- поддержку расширений.

ных — их появление и время жизни определяются событиями, возникающими на этапе выполнения программы. Для хранения таких данных наилучшим образом подходит размещение их в т. н. свободной памяти — в «куче». Эти действия, как правило, возлагаются на менеджер памяти. Поэтому в ряде приложений без такого средства не обойтись, но учитывая потребление ресурсов стандартным менеджером памяти, его применение оказывается неприемлемым. В этой ситуации нередко используют специализированный менеджер памяти, спроектированный специально для удовлетворения требований прикладной задачи оптимальным образом. Принимая во внимание вышесказанное, становится очевидным, что создание универсального менеджера памяти, в равной степени хорошо удовлетворяющего потребностям разнообразных проектов, малореально, что обусловило отсутствие менеджера памяти в составе ***scmRTOS.***

Подробнее о структуре, составе, функциях и механизмах ядра см. «**Глава 3 Ядро ОС**»

2.2.2. Процессы

Процессы предоставляют возможность создавать отдельный (асинхронный по отношению к остальным) поток управления программы. Каждый процесс для этого предоставляет функцию, которая должна содержать бесконечный цикл, являющийся главным циклом процесса – пример см. «**Листинг 2.1 Исполняемая функция процесса**».

```
{1} template<> void TSlon::exec()
{2} {
{3}     ... // Declarations
{4}     ... // Init process's data
{5}     for(;;)
{6}     {
{7}         ... // process's main loop
{8}     }
{9} }
```

Листинг 2.1 Исполняемая функция процесса

При старте системы управление передаётся в функцию процесса, где на входе могут быть размещены объявления используемых данных {3} и код инициализации {4}, за которыми следует главный цикл процесса {5}-{8}. Пользовательский код должен быть написан так, чтобы исключить выход из функции процесса. Например, войдя в главный цикл, не покидать его (основной подход), либо, если выйти из главного цикла, то попасть или в другой цикл (пусть даже пустой), или в бесконечную «спячку», вызвав функцию **`sleep()`**[1] без параметров (или с параметром «0»), — подробнее об этом см. «**4.1.6 Функция sleep()**». В коде процесса не должно также быть операторов возврата из функции **`return`**.

[1] При этом никакой другой процесс не должен «будить» этот спящий перед выходом процесс, иначе возникнет неопределенное поведение и система, скорее всего, «упадет». Единственным безопасным действием, которое может быть применено к процессу в этой ситуации — это прекращение работы процесса (с возможностью дальнейшего запуска его с начала), см. «**4.3 Перезапуск процесса**».

2.2.3. Межпроцессное взаимодействие

Так как процессы в системе выполняются параллельно и асинхронно по отношению друг к другу, то простое использование глобальных данных для обмена между ними некорректно и опасно: во время обращения к тому или иному объекту (который может быть переменной встроенного типа, массивом, структурой, объектом класса и проч.) со стороны одного процесса может произойти прерывание его работы другим (более приоритетным) процессом, который также производит обращение к тому же объекту, и, в силу неатомарности операций обращения (чтение/запись), второй процесс может как нарушить правильность действий первого процесса, так и просто считать некорректные данные.

Для предотвращения таких ситуаций нужно принимать специальные меры: производить обращение внутри так называемых критических секций (Critical Section), когда передача управления между процессами запрещена, или использовать специальные средства для межпроцессного взаимодействия. К таким средствам в ***scmRTOS*** относятся:

- флаги событий (OS::TEventFlag);
- семафоры взаимоисключения (OS::TMutex);
- каналы для передачи данных в виде очереди из байт или объектов произвольного типа (OS::channel);
- сообщения (OS::message).

Какое из средств (или их совокупность) применить в каждом конкретном случае, должен решать разработчик, исходя из требований задачи, доступных ресурсов и личных предпочтений.

Начиная с ***scmRTOS v4***, средства межпроцессного взаимодействия (сервисы) выполнены на основе общего специализированного класса **`TService`**, который предоставляет все необходимые базовые средства для реализации сервисных классов/шаблонов. Интерфейс этого класса документирован и предназначен для расширения набора сервисов самим пользователем, который при необходимости может спроектировать и реализовать своё собственное средство межпроцессного взаимодействия, наилучшим образом отвечающее требованиям конкретного целевого проекта.

2.3. Программная модель

2.3.1. Состав и организация

Исходный код ***scmRTOS*** в любом проекте состоит из трёх частей: общая (core) и платформеннозависимая (target), и проектнозависимая (project).

Общая часть содержит объявления и определения функций ядра, процессов, системных сервисов, а также небольшую библиотеку поддержки, содержащую некоторый полезный код, часть которого непосредственно используется ОС.

Платформеннозависимая часть – объявления и определения, отвечающие за реализацию функций, присущих данной целевой платформе, расширения языка для используемого компилятора и т.п. К платформеннозависимой части относятся ассемблерный код переключения контекста и старта системы, функция формирования структуры стекового кадра (stack frame), определение класса-«обёртки» (wrapper) критической секции, а также обработчик прерывания от аппаратного таймера данной платформы, используемого в качестве системного, и другое платформеннозависимое поведение.

Проектнозависимая часть — три заголовочных файла с определениями конфигурационных макросов, подключениями расширений и необходимого в ряде случаев кода для тонкой настройки операционной системы под конкретный целевой проект — в частности, сюда входят определения псевдонимов типов для задания разрядности переменных таймаутов, выбор источника прерывания переключения контекстов и другие необходимые для оптимального функционирования системы средства.

Рекомендуемое размещение исходных файлов системы: общая часть — в отдельной директории core, платформеннозависимая часть — в своей директории <target>, где target — название целевого порта системы, проектнозависимая часть — непосредственно в исходных файлах проекта. Такое размещение предлагается из соображений удобства хранения, переноса и сопровождения проекта, а также более простого и безопасного процесса обновления системы при переходе на новые версии.

Исходные тексты общей части содержатся в восьми файлах:

- scmRTOS.h – главный заголовочный файл, включает в себя всю иерархию заголовочных файлов системы.
- os_kernel.h – основные объявления и определения типов ядра ОС.
- os_kernel.cpp– объявления объектов и определения функций ядра.
- scmRTOS_defs.h – вспомогательные объявления и макросы.
- os_services.h – определения типов и шаблонов сервисов.
- os_services.cpp – определения функций сервисов.
- usrlib.h – определения типов и шаблонов библиотеки поддержки.
- usrlib.cpp – определения функций библиотеки поддержки.

Как видно из вышеприведённого списка, в состав ***scmRTOS*** входит ещё небольшая библиотека поддержки, где находится код, используемый средствами ОС[1]. Поскольку сама по себе эта библиотека по сути не является частью ОС, то внимания её (библиотеки) рассмотрению в текущем документе уделено не будет.

Исходный код платформеннозависимой части находится в трёх файлах:

- os_target.h – платформеннозависимые объявления и макросы.
- os_target_asm.ext[2] - низкоуровневый код, функции переключения контекста, старта ОС.
- os_target.cpp – определения функции инициализации стекового кадра процесса и функции обработчика прерывания от таймера, используемого в качестве системного, корневая функция фонового (idle) процесса.

Проектнозависимая часть состоит из трёх заголовочных файлов:

- scmRTOS_config.h – конфигурационные макросы и псевдонимы некоторых типов, в частности, типа, задающего разрядность объектов таймаутов.
- scmRTOS_target_cfg.h – код для настройки механизмов ОС под нужды конкретного проекта; сюда, например, может входить задание вектора прерываний для обработчика прерываний от аппаратного таймера, выбранного в качестве системного, макросы управления системным таймером, определение функции активации прерывания переключения контекстов и др.
- scmRTOS_extensions.h – управление подключением расширений. Более подробно см. «**3.3 TKernelAgent и расширения**»

[1] В частности, класс/шаблон кольцевого буфера.
[2] Расширение ассемблерного файла для целевого процессора.

2.3.2. Внутренняя структура

Все, что относится к ***scmRTOS***, за исключением нескольких функций, реализованных на ассемблере и имеющих спецификацию связывания **`extern "C"`**, помещено внутрь пространства имён **`OS`** – таким способом реализовано отдельное пространство имён для составных частей операционной системы.

Внутри этого пространства имён объявлены следующие классы[1]:

- **`TKernel`**. Поскольку ядро в системе может быть представлено только в одном экземпляре, то существует только один объект этого класса. Пользователь не должен создавать объекты этого класса. Подробнее см. с. 49;
- **`TBaseProcess`**. Реализует тип объекта, являющегося основой для построения шаблона **`process`**, на основе которого реализуется любой (пользовательский или системный) процесс. Подробнее см. с. 70;
- **`process`**. Шаблон, на основе которого создаётся тип любого процесса ОС.
- **`TISRW`**. Это класс-«обёртка» для облегчения и автоматизации процедуры создания кода обработчиков прерываний. Его конструктор выполняет действия при входе в обработчик прерывания, а деструктор – соответствующие действия при выходе.
- **`TKernelAgent`**. Специальный служебный класс, предназначенный для предоставления доступа к необходимым ресурсам ядра для расширения возможностей ОС. На основе этого класса построены класс TService, являющийся базой для всех средств межпроцессного взаимодействия, а также класс профилировщика.

[1] Почти все классы ОС объявлены как друзья (friend) друг для друга. Это сделано для того, чтобы обеспечить доступ для составных частей ОС к представлению других составных частей, не открывая интерфейс наружу, чтобы пользовательский код не мог напрямую использовать внутренние переменные и механизмы ОС, что повышает безопасность использования.

В перечень сервисных классов входят:

- **`TService.`** Базовый класс для построения всех типов и шаблонов средств межпроцессного взаимодействия. Содержит общий функционал и определяет интерфейс прикладного программирования — API (Application Programmable Interface) для всех типов-потомков. Является основой для расширения набора средств межпроцессного взаимодействия.
- **`TEventFlag`**. Предназначен для межпрограммных взаимодействий путём передачи бинарного семафора (флага события). Подробнее см. с. 87;
- **`TMutex`**. Бинарный семафор, предназначенный для организации взаимного исключения доступа к совместно используемым ресурсам. Подробнее см. с. 90;
- **`message`**. Шаблон для создания объектов-сообщений Сообщение «похоже» на **`EventFlag`**, но вдобавок может ещё содержать объект произвольного типа (обычно это структура), представляющий собой тело сообщения. Подробнее см. с. 96;
- **`channel`** . Шаблон для создания канала передачи данных произвольного типа. Служит основой для построения очередей сообщений. Более подробно см. с. 99.

Как видно из приведённого выше списка, отсутствуют счётные семафоры. Причина этого в том, что при всем желании не удалось увидеть острой необходимости в них. Ресурсы, которые нуждаются в контроле с помощью счётных семафоров, находятся в остром дефиците в однокристальных МК, это прежде всего —оперативная память. А ситуации, где все же необходимо контролировать доступное количество, обходятся с помощью объектов, созданных на основе шаблона **`channel`**, внутри которых в том или ином виде уже реализован соответствующий механизм. При необходимости в таком сервисе пользователь может самостоятельно добавить его к базовому набору путём создания своей реализации в виде расширения, см. «**3.3 TKernelAgent и расширения**».

scmRTOS предоставляет пользователю несколько функций для контроля:

- **`run()`**. Предназначена для запуска ОС. При вызове этой функции начинается собственно функционирование операционной системы – управление передаётся процессам, работа которых и взаимное взаимодействие определяется пользовательской программой. Передав управление коду ядра ОС, функция уже не получает его (управление) обратно и, следовательно, возврата из функции не предусмотрено;
- **`lock_system_timer()`**. Блокирует прерывания от системного таймера. Поскольку выбор и обслуживание аппаратной части системного таймера находятся в компетенции проекта, то определить содержимое этой функции должен пользователь. То же самое касается и парной функции **`unlock_system_timer()`**;
- **`unlock_system_timer()`**. Разблокирует прерывания от системного таймера;
- **`get_tick_count()`**. Возвращает количество тиков системного таймера. Счётчик тиков системного таймера должен быть разрешён при конфигурировании системы;
- **`get_proc()`**. Возвращает указатель на константный объект процесса по индексу, переданному в функцию в качестве аргумента. Индекс фактически является значением приоритета процесса.

2.3.3. Критические секции

В силу вытесняющего характера работы процессов, любой из них может быть прерван в произвольный момент времени. С другой стороны, существует ряд случаев[1], когда необходимо исключить возможность прервать процесс во время выполнения определённого фрагмента кода. Это достигается запрещением передачи управления[2] на период выполнения упомянутого фрагмента. Т.е. этот фрагмент является как бы непрерываемой секцией. В терминах ОС такая секция называется критической.

Для упрощения организации критической секции используется специальный класс-«обёртка» **`TCritSect`**. В конструкторе этого класса запоминается состояние процессорного ресурса, управляющего общим разрешением/запрещением прерываний, и прерывания запрещаются. В деструкторе этот процессорный ресурс приводится к тому состоянию, в котором он пребывал перед запрещением прерываний. Таким образом, если прерывания были запрещены, то они и останутся запрещёнными. Если были разрешены, то будут разрешены. Реализация этого класса платформеннозависима, поэтому её определение содержится в соответствующем файле os_target.h.

[1] Например, обращение к переменным ядра ОС или представлению средств межпроцессного взаимодействия.
[2] В ***scmRTOS*** в настоящее время это достигается путём общего запрещения прерываний.

Использование **TCritSect** тривиально: в точке, которая соответствует началу критической секции, достаточно объявить объект этого типа, и от места объявления до конца блока прерывания будут запрещены[1].

2.3.4. Синонимы встроенных типов

Для облегчения работы с исходным текстом, а также для переносимости введены следующие синонимы:

- **TProcessMap** – тип для определения переменной, выполняющий функцию карты процессов. Её размер зависит от количества процессов в системе. Каждому процессу соответствует уникальный тег – маска, содержащая только один ненулевой бит, расположенный в соответствии с приоритетом этого процесса. Процессу с наибольшим приоритетом соответствует младший бит (позиция 0)[2]. При количестве пользовательских процессов менее 8 размер карты процессов – 8 бит. При количестве от 8 до 15 размер – 16 бит, при 16 и более пользовательских процессов — 32 бита.
- **stack_item_t** – тип элемента стека. Зависит от целевой архитектуры. Например, на 8-разрядном AVR этот тип определён как **uint8_t**, на 16-разрядном MSP430 – **uint16_t**, а на 32-разрядных платформах, как правило, – **uint32_t**.

2.3.5. Использование ОС

Как уже отмечалось выше, для достижения максимальной эффективности везде, где возможно, использовались статические механизмы, т.е. вся функциональность определяется на этапе компиляции.

В первую очередь это касается процессов. Перед использованием каждого процесса должен быть определён его тип[3], где указывается имя типа процесса, его приоритет и размер области ОЗУ, отведённой под стек процесса[4]. Например:

```
OS::process<OS::pr2, 200> MainProc;
```

[1] При выходе из блока автоматически будет вызван деструктор, который восстановит состояние, предшествовавшее входу в критическую секцию. Т.е. при таком способе отсутствует возможность «забыть» разрешить прерывания при выходе из критической секции.

[2] Такой порядок принят по умолчанию. Если **scmRTOS_PRIORITY_ORDER** определён как 1, то порядок расположения бит в карте процессов обратный — т.е. старший бит соответствует наиболее приоритетному процессу, младший бит — наименее приоритетному. Обратный порядок приоритетов может оказаться полезным для процессоров, у которых есть аппаратные средства поиска первого ненулевого бита в двоичном слове, — например для процессоров семейства Blackfin.

[3] Каждый процесс – это объект отдельного типа (класса), производного от общего базового класса **TBaseProcess**.

[4] Более подробно см. «**4.1.3 Стек**».

Здесь определён процесс с приоритетом **pr2** и размером стека в 200 байт. Такое объявление может показаться несколько неудобным из-за некоторой многословности, т.к. при необходимости ссылаться на тип процесса придётся писать полное объявление – например, при определении исполняемой функции процесса:

```
template<>[1] void OS::process<OS::pr2, 200>::exec() { ... }
```

т.к. типом является именно выражение

```
OS::process<OS::pr2, 200>
```

Аналогичная ситуация возникнет и в других случаях, когда понадобится ссылаться на тип процесса. Для устранения этого неудобства можно пользоваться синонимами типов, вводимыми через **typedef**. Это рекомендуемый стиль кодирования: сначала определить псевдонимы типов процессов (лучше всего где-нибудь в заголовочном файле в одном месте, чтобы сразу было видно, сколько в проекте процессов и какие они), а потом уже по месту в исходных файлах объявлять сами объекты процессов. При этом приведённый выше пример выглядит так:

```
// В заголовочном файле
typedef OS::process<OS::pr2, 200> TMainProc;
...
template<> void TMainProc::exec()[2];

// В исходном файле
TMainProc MainProc;
...
template<> void TMainProc::exec()
{
    ...
}
...
```

В этой последовательности действий нет ничего особенного – это обычный способ описания псевдонима типа и создания объекта этого типа, принятый в языках программирования C и C++.

[1] Исполняемая функция *конкретного* процесса технически является полной специализацией функции-члена шаблона **OS::process::exec()**, поэтому в её определении используется синтаксис определения специализации **template<>**.

[2] Рекомендуется объявлять прототип специализации исполняемой функции процесса до первого использования экземпляра шаблона – это позволит компилятору видеть, что существует полная специализация функции для данного экземпляра, поэтому нет необходимости пытаться сгенерировать общую реализацию этой функции шаблона. В ряде случаев это позволяет избежать ошибок компиляции.

ЗАМЕЧАНИЕ. При конфигурации системы должно быть указано количество процессов. И это количество должно точно совпадать с количеством описанных процессов в проекте, иначе система работать не будет. Следует иметь в виду, что для задания приоритетов введён специальный перечислимый тип `TPriority`, который описывает допустимые значения приоритетов[1].

Кроме того, приоритеты всех процессов должны идти подряд, пропусков не допускается, например, если в системе 4 процесса, то приоритеты процессов должны иметь значения `pr0, pr1, pr2, pr3`. Не допускаются также одинаковые значения приоритетов, т.е. каждый процесс должен иметь уникальное значение приоритета. Например, если в системе 4 пользовательских процесса (т.е. всего 5 процессов – один системный процесс `IdleProc`), то значения приоритетов должны быть `pr0, pr1, pr2, pr3` (`prIDLE` – для `IdleProc`), где `pr0` – самый высокоприоритетный процесс, а `pr3` – самый низкоприоритетный из пользовательских процессов. Вообще самым низкоприоритетным процессом является `IdleProc`. Этот процесс существует в системе всегда, его описывать не нужно. Именно этот процесс получает управление, когда все пользовательские процессы находятся в неактивном состоянии.

За пропусками в нумерации приоритетов процессов, а также за уникальностью значений приоритетов процессов компилятор не следит, т.к., придерживаясь принципа раздельной компиляции, не видно эффективного пути сделать автоматизированный контроль за целостностью конфигурации языковыми средствами.

В настоящее время существует специальное инструментальное средство, которое выполняет всю работу по проверке целостности конфигурации. Утилита называется scmIC (IC – Integrity Checker), и позволяет обнаружить подавляющее большинство типовых ошибок конфигурирования ОС, подробнее об этом см. «Приложение B.1 Утилита проверки целостности конфигурации системы стр 139».

Как уже было сказано, определение типов процессов удобно разместить в заголовочном файле, чтобы была возможность легко сделать любой процесс видимым в другой единице компиляции.

Пример типового использования процессов – см. «**Листинг 2.2 Определение типов процессов в заголовочном файле**» и «**Листинг 2.3 Объявление процессов в исходном файле и запуск ОС**»

[1] Это сделано для повышения безопасности использования – нельзя просто указать любое целое значение, подходят только те значения, которые описаны в `TPriority`. А описанные в `TPriority` значения связаны с количеством процессов, указанном при задании конфигурационного макроса `scmRTOS_PROCESS_COUNT`. Таким образом, можно только выбрать из ограниченного количества. Значения приоритетов процессов имеют вид: `pr0`, `pr1` и т.д., где число обозначает уровень приоритета. Системный процесс `IdleProc` имеет отдельное обозначение приоритета `prIDLE`.

```
{1}  //-------------------------------------
{2}  //
{3}  //      Process types definition
{4}  //
{5}  //
{6}  typedef OS::process<OS::pr0, 200> TUARTDrv;
{7}  typedef OS::process<OS::pr1, 100> TLCDProc;
{8}  typedef OS::process<OS::pr2, 200> TMainProc;
{9}  typedef OS::process<OS::pr3, 200> TFPGA_Proc;
{10} //-------------------------------------
```

Листинг 2.2 Определение типов процессов в заголовочном файле

```
{1}  //-------------------------------------
{2}  //
{3}  //  Processes declarations
{4}  //
{5}  //
{6}  TUartDrv   UartDrv;
{7}  TLCDProc   LCDProc;
{8}  TMainProc  MainProc;
{9}  TFPGAProc  FPGAProc;
{10} //-------------------------------------
{11}
{12} //-------------------------------------
{13} void main()
{14} {
{15}     ... // system timer and other stuff initialization
{16}     OS::run();
{17} }
{18} //-------------------------------------
```

Листинг 2.3 Объявление процессов в исходном файле и запуск ОС

Каждый процесс, как уже упоминалось выше, имеет исполняемую функцию. При использовании приведённой выше схемы исполняемая функция процесса называется **`exec`** и выглядит как показано на «**Листинг 2.1 Исполняемая функция процесса**».

Конфигурационная информация задаётся в специальном заголовочном файле scmRTOS_config.h. Состав и значения[1] конфигурационных макросов — см. «**Таблица 2.1 Конфигурационные макросы**».

[1] В таблице приведены примеры значений. В каждом проекте значения задаются индивидуально, исходя из требований проекта.

Название	Знач.	Описание
`scmRTOS_PROCESS_COUNT`	`n`	Количество процессов в системе
`scmRTOS_SYSTIMER_NEST_INTS_ENABLE`[1]	`0/1`	Разрешает вложенные прерывания в обработчике прерывания от системного таймера.
`scmRTOS_SYSTEM_TICKS_ENABLE`	`0/1`	Включает использование счётчика тиков системного таймера.
`scmRTOS_SYSTIMER_HOOK_ENABLE`	`0/1`	Включает в обработчике прерывания системного таймера вызов функции **`system_timer_user_hook()`**. В этом случае указанная функция должна быть определена в пользовательском коде.
`scmRTOS_IDLE_HOOK_ENABLE`	`0/1`	Включает в системном процессе `IdleProc` вызов функции `idle_process_user_hook()`. В этом случае указанная функция должна быть определена в пользовательском коде.
`scmRTOS_ISRW_TYPE`	`TISRW/ TISRW_SS`	Позволяет выбрать тип класса-«обёртки» для обработчика прерывания системного таймера - обычный или с переключением на отдельный стек прерываний. Суффикс `_ss` означает Separate Stack
`scmRTOS_CONTEXT_SWITCH_SCHEME`	`0/1`	Задаёт способ переключения контекстов (передачи управления). Подробнее см. с. 52
`scmRTOS_PRIORITY_ORDER`	`0/1`	Задаёт порядок старшинства приоритетов в карте процессов. Значение 0 соответствует варианту, когда наиболее приоритетный процесс (`pr0`[2]) соответствует младшему биту в карте процессов (`TProcessMap`), значение 1 соответствует варианту, когда наиболее приоритетному процессу соответствует старший бит (из значимых) в карте процессов.
`scmRTOS_IDLE_PROCESS_STACK_SIZE`	`N`	Задаёт размер стека фонового процесса `IdleProc`.
`scmRTOS_CONTEXT_SWITCH_USER_HOOK_ENABLE`	`0/1`	Разрешает вызов пользовательского хука `context_switch_user_hook()` во время переключения контекстов. В этом случае функция должна быть определена в пользовательском коде.

[1] Если портом поддерживается только один вариант, то соответствующее значение макроса определено в порте. Это же самое касается и всех остальных макросов

[2] Для задания приоритета конкретного процесса определён специальный тип **`TPriority`**, содержащий значения вида **`pr0, pr1...prN`**. Самый высокий приоритет, независимо от значения макроса **`scmRTOS_PROIRITY_ORDER`**, *всегда* соответствует **`pr0`**, и далее, по мере увеличения порядкового номера приоритет уменьшается.

scmRTOS_DEBUG_ENABLE	0/1	Включает отладочные средства...
scmRTOS_PROCESS_RESTART_ENABLE	0/1	Позволяет прерывать работу любого процесса в произвольный момент и запустить этот процесс заново.

Таблица 2.1 Конфигурационные макросы

Глава 3 Ядро ОС

3.1. Общие сведения

Ядро операционной системы выполняет:

- функции по организации процессов;
- планирование (Scheduling) как на уровне процессов, так и на уровне прерываний;
- поддержку межпроцессного взаимодействия;
- поддержку системного времени (системный таймер);
- поддержку расширений.

Основу ядра системы составляет класс **`TKernel`**, который включает в себя весь необходимый набор функций и данных. Объект этого класса существует, по понятным причинам, в единственном экземпляре. Почти всё его представление является закрытым и для доступа к нему со стороны некоторых частей ОС, которым требуется доступ к ресурсам этого класса, использован механизм «друзей» (friend) C++ – функции и классы, которым предоставлен такой доступ, объявлены с ключевым словом **`friend`**.

Необходимо отметить, что под ядром в данном контексте понимается не только объект **`TKernel`**, но и средство расширения функциональных возможностей ОС, реализованное в виде класса **`TKernelAgent`**. Этот класс специально введён в состав операционной системы с целью предоставить базу для построения расширений. Забегая вперёд, можно отметить, что в ***scmRTOS*** все средства межпроцессного взаимодействия реализованы на основе такого расширения. Класс **`TKernelAgent`** объявлен «другом» (**`friend`**) класса **`TKernel`** и содержит минимально необходимый набор защищённых (**`protected`**) функций для предоставления потомкам доступа к ресурсам ядра. Расширения строятся путём наследова-

ния от класса **TKernelAgent**. Более подробно см. «**3.3 TKernelAgent и расширения**» с. 67.

3.2. TKernel. Состав и функционирование

3.2.1. Состав

Класс **TKernel** содержит следующие члены-данные[1]:

- **CurProcPriority** – переменная, содержащая номер приоритета текущего активного процесса. Служит для оперативного доступа к ресурсам текущего процесса, а также для манипуляций со статусом процесса (как по отношению к ядру, так и к средствам межпроцессного взаимодействия)[2];
- **ReadyProcessMap** – карта процессов, готовых к выполнению. Содержит теги процессов, готовых к выполнению: каждый бит этой переменной соответствует тому или иному процессу, лог. 1 указывает на то, что процесс готов к выполнению[3], лог. 0 – на то, что процесс не готов;
- **ProcessTable** – массив указателей на процессы, зарегистрированные в системе;
- **ISR_NestCount** – переменная-счётчик входов в прерывания. При каждом входе она инкрементируется, при каждом выходе декрементируется;
- **SysTickCount** – переменная-счётчик тиков (переполнений) системного таймера. Присутствует только если эта функция разрешена (с помощью определения соответствующего макроса в конфигурационном файле);
- **SchedProcPriority*** – переменная для хранения значения приоритета процесса, запланированного для передачи ему управления.

3.2.2. Организация процессов

Функция по организации процессов сводится к регистрации созданных процессов. При этом в конструкторе каждого процесса вызывается функция ядра **register_process(TBaseProcess *)**, которая помещает значение указателя на процесс, переданного в качестве аргумента, в таблицу **ProcessTable** (см. ниже)

[1] Объекты, помеченные '*', присутствуют только в варианте с использованием передачи управления на основе программного прерывания.
[2] Возможно, идеологически более правильным было бы для этих целей использовать указатель на процесс, но анализ показал, что выигрыша по производительности тут не достигается, а размер указателя, как правило, больше, чем размер переменной целого типа для хранения приоритета.
[3] При этом процесс может быть активным, т.е. выполняться, а может быть и неактивным, т.е. находиться в ожидании получения управления – такая ситуация возникает, когда в системе есть другой готовый к выполнению процесс, у которого приоритет выше.

процессов системы. Местоположение этого указателя в таблице определяется в соответствии с приоритетом данного процесса, который фактически является индексом при обращении к таблице.

Код функции регистрации процессов – см. «**Листинг 3.1 Функция регистрации процессов**».

```
{1}  void OS::TKernel::register_process(OS::TBaseProcess * const p)
{2}  {
{3}      ProcessTable[p->Priority] = p;
{4}  }
```

Листинг 3.1 Функция регистрации процессов

Следующая системная функция – это собственно запуск ОС. Код функции запуска системы – см. «**Листинг 3.2 Функция запуска ОС**».

```
{1}  INLINE void OS::run()
{2}  {
{3}      stack_item_t *sp = Kernel.ProcessTable[pr0]->StackPointer;
{4}      os_start(sp);
{5}  }
```

Листинг 3.2 Функция запуска ОС

Как видно, действия предельно просты: из таблицы процессов извлекается указатель на стек самого приоритетного процесса {3} и производится собственно старт системы {4} путём запуска низкоуровневой функции **`os_start()`** с передачей ей в качестве аргумента извлечённого указателя на стек самого приоритетного процесса.

С этого момента начинается работа ОС в основном режиме, т.е. передача управления от процесса к процессу в соответствии с их приоритетами, событиями и пользовательской программой.

3.2.3. Передача управления

Передача управления может происходить двумя способами:

- процесс сам отдаёт управление, когда ему (пока) нечего больше делать, или в результате своей работы процесс должен войти в межпроцессное взаимодействие с другими процессами (захватить семафор взаимоисключения (`OS::TMutex`), или, «просигналив» флаг события (`OS::TEventFlag`), сообщить об этом ядру, которое должно будет произвести (при необходимости) перепланирование процессов;
- управление у процесса отбирается ядром в результате возникновения прерывания по какому-либо событию, и если это событие ожидал процесс с более высоким приоритетом, то управление будет отдано этому процессу, а прерванный процесс будет ждать, пока тот, более приоритетный, не отработает своё задание и не отдаст управление[1].

В первом случае перепланирование процессов производится синхронно по отношению к потоку выполнения программы – в коде планировщика. Во втором случае перепланировка производится асинхронно по возникновению события.

Собственно передачу управления можно организовать несколькими способами. Один из способов – прямая передача управления путём вызова из планировщика[2] низкоуровневой[3] функции переключателя контекста. Другой способ – передача управления путём активации специального программного прерывания, где и происходит переключение контекста. ***scmRTOS*** поддерживает оба способа. И тот, и другой способы имеют свои достоинства и недостатки, которые будут подробно рассмотрены ниже.

3.2.4. Планировщик

Исходный код собственно планировщика представлен в функции `sched()` – см. «**Листинг 3.3 Планировщик**».

Здесь присутствуют два варианта – один для случая прямой передачи управления (`scmRTOS_CONTEXT_SWITCH_SCHEME == 0`), другой – для случая передачи управления с помощью программного прерывания.

[1] Этот более приоритетный процесс может быть прерван, в свою очередь, еще более приоритетным процессом, и так до тех пор, пока дело не дойдет до самого приоритетного процесса, который может быть прерван (на время) только прерыванием, возврат из которого произойдет все равно в этот самый приоритетный процесс. Т.е. самый высокоприоритетный процесс не может быть прерван никаким другим процессом. При выходе из обработчика прерывания управление всегда передается самому приоритетному процессу из готовых к выполнению.

[2] Или при выходе из обработчика прерывания – в зависимости от того, синхронная передача управления или асинхронная.

[3] Обычно реализуемой на ассемблере.

Нужно отметить, что вызов планировки с уровня основной программы производится с помощью функции **scheduler()**, которая вызывает собственно планировщик только, если вызов производится не из прерывания:

```
INLINE void scheduler() { if(ISR_NestCount) return; else  sched(); }
```

При правильном использовании средств ОС такой ситуации не должно происходить, т.к. вызов планировки с уровня прерываний должен осуществляться через специализированные версии соответствующих функций (их имена имеют суффикс **_isr**), которые специально предназначены для работы с уровня прерываний. Например, при необходимости просигналить из прерывания флаг события пользователь должен использовать функцию **signal_isr()**[1] вместо **signal()**. Но в случае использования последней фатальной ошибки при работе программы не произойдёт, просто планировщик реально не будет вызван, и, несмотря на возможно поступившее в прерывании событие, передачи управления не произойдёт, даже если её черёд в этот момент уже наступил. Передача управления произойдёт только при следующем вызове перепланировки, которая произойдёт при выполнении деструктора объекта типа **TISRW/TISRW_SS**. Таким образом, в функции **scheduler()** просто присутствует защита от краха работы программы при неаккуратном использовании сервисов, а также при использовании сервисов, в которых не предусмотрены соответствующие **_isr** функции – например, **channel::push()**.

[1] Нелишне напомнить, что все обработчики прерываний в программе, которые используют средства межпроцессного взаимодействия, должны содержать объявление объекта **TISRW**, размещённое до любого вызова функции-сервиса (т.е. где имеет место вызов планировщика). Этот объект должен быть объявлен до первого использования сервисов ОС.

```
{1}    #if scmRTOS_CONTEXT_SWITCH_SCHEME == 0
{2}    void TKernel::sched()
{3}    {
{4}        uint_fast8_t NextPrty = highest_priority(ReadyProcessMap);
{5}        if(NextPrty != CurProcPriority)
{6}        {
{7}            stack_item_t* Next_SP =
{8}                       ProcessTable[NextPrty]->StackPointer;
{9}            stack_item_t** Curr_SP_addr =
{10}                      &(ProcessTable[CurProcPriority]->StackPointer);
{11}           CurProcPriority = NextPrty;
{12}           os_context_switcher(Curr_SP_addr, Next_SP);
{13}       }
{14}   }
{15}   #else
{16}   void TKernel::sched()
{17}   {
{18}       uint_fast8_t NextPrty = highest_priority(ReadyProcessMap);
{19}       if(NextPrty != CurProcPriority)
{20}       {
{21}           SchedProcPriority = NextPrty;
{22}
{23}           raise_context_switch();
{24}           do
{25}           {
{26}               enable_context_switch();
{27}               DUMMY_INSTR();
{28}               disable_context_switch();
{29}           }
{30}           while(CurrProcPriority != SchedProcPriority);
{31}       }
{32}   }
{33}   #endif // scmRTOS_CONTEXT_SWITCH_SCHEME
```

Листинг 3.3 Планировщик

3.2.4.1. Планировщик с прямой передачей управления

Все действия, выполняемые внутри планировщика, не должны быть прерываемы, поэтому код этой функции выполняется в критической секции. Но т.к. планировщик всегда вызывается при запрещённых прерываниях, то использовать в его коде критическую секцию нет необходимости.

Первым делом вычисляется приоритет самого высокоприоритетного процесса, готового к выполнению (путём анализа карты процессов, готовых к выполнению, **`ReadyProcessMap`**).

Далее найденный приоритет сравнивается с текущим, и если они совпадают, то текущий процесс является как раз самым приоритетным из готовых к выполнению и передачи управления другому процессу не требуется, т.е. поток управления остаётся в текущем процессе.

Если найденный приоритет не совпадает с текущим, то это означает, что появился более приоритетный по сравнению с текущим процесс, готовый к выполнению, и управление должно быть передано ему. Это достигается путём переключения контекстов процессов. Контекст текущего процесса сохраняется в стеке текущего процесса, а контекст следующего процесса извлекается из его стека. Эти действия платформеннозависимы и производятся в низкоуровневой функции (реализованной на ассемблере) **`os_context_switcher()`**, которая вызывается из планировщика {12}. Этой функции передаются в качестве аргументов два параметра:

- адрес указателя стека текущего процесса, куда будет помещён сам указатель по окончании сохранения контекста текущего процесса. {9};
- указатель стека следующего процесса {7}.

При реализации низкоуровневой функции-переключателя контекстов следует обратить внимание на соглашения о вызове функций и передаче параметров для данной платформы и компилятора.

3.2.4.2. Планировщик с передачей управления на основе программного прерывания

В этом варианте планировщик весьма отличается от вышеописанного. Главное отличие – это то, что собственно переключение контекста происходит не путём непосредственного вызова функции-переключателя контекстов, а путём активации специального прерывания, в котором и происходит переключение контекстов. Такой способ таит в себе ряд нюансов и требует специальных мер по предотвращению нарушения целостности работы системы. Ниже ситуация будет рассмотрена подробнее.

Основная трудность, возникающая при реализации этого способа передачи управления, состоит в том, что собственно код планировщика и код функции обработчика прерываний программного прерывания не являются строго непрерывными, «атомарными», между ними может возникнуть прерывание, которое также может инициировать перепланировку, что вызовет своего рода «наложение» результатов текущей перепланировки и нарушить целостность процесса передачи управления. Для того чтобы избежать этой коллизии, процесс «перепланировка-передача управления» разбит на две «атомарные» операции, которые можно безопасно разделять между собой.

Первая операция – это, как и прежде, вычисление приоритета самого приоритетного процесса из готовых к выполнению и проверка необходимости пере-

планировки. Если такая необходимость имеется, то происходит фиксация значения приоритета следующего процесса в промежуточной переменной **`SchedProcPriority`** {21} и активация программного прерывания переключения контекстов.

Далее программа входит в цикл ожидания переключения контекстов {24}. Здесь кроется довольно тонкий момент. Ведь почему бы, например, было просто не сделать зону разрешённых прерываний в виде пары пустых (dummy) команд (чтобы аппаратура процессора успела осуществить собственно само прерывание)? Такая реализация таит в себе трудноуловимую ошибку, состоящую в следующем.

Если на момент разрешения переключения контекстов, которое в данной версии ОС реализуется путём общего разрешения прерываний {26}, кроме программного прерывания были активированы одно или несколько других прерываний, причём приоритет некоторых из них выше, чем приоритет программного прерывания, то при этом, естественно, поток управления будет передан в обработчик соответствующего прерывания, по окончании которого будет произведён возврат в прерванную программу. Теперь в основной программе (т.е. внутри функции-планировщика) процессор может выполнить одну или несколько инструкций[1], прежде чем может быть выполнено следующее прерывание. При этом программа сможет дойти до кода, запрещающего переключение контекстов, что приведёт к тому, что прерывания глобально будут запрещены и программное прерывание, где производится переключение контекста, выполнено не будет. Это означает, что далее поток управления останется в текущем процессе, в то время как должен был бы быть передан системе (и другим процессам) до тех пор, пока не возникнет событие, которое ожидает текущий процесс. Это есть не что иное как нарушение целостности системы и может приводить к самым разнообразным и труднопредсказуемым негативным последствиям.

Очевидно, что такой ситуации возникать не должно, поэтому вместо нескольких пустых команд в зоне разрешённых прерываний используется цикл ожидания переключения контекстов. Т.е. сколько бы прерываний не стояло в очереди, пока реального переключения контекстов не произойдёт, поток управления программы дальше этого цикла не пойдёт.

Для обеспечения работоспособности описанного необходим критерий того, что перепланирование реально произошло. Таким критерием может выступать равенство переменных ядра **`CurProcPriority`** и **`SchedProcPriority`**.

[1] Это обычное свойство многих процессоров – после возврата из прерывания переход на обработчик следующего прерывания становится возможен не сразу в том же машинном цикле, а только лишь через один или более циклов.

Эти переменные становятся равными друг другу (т.е. значение текущего приоритета становится равным запланированному значению) только после выполнения переключения контекстов.

Как видно, никаких обновлений переменных, содержащих указатели стеков и значения текущего приоритета, тут нет. Все эти действия производятся позднее при непосредственном переключении контекстов путём вызова специальной функции ядра **os_context_switch_hook()**.

Может возникнуть вопрос: зачем такие сложности? Чтобы ответить на этот вопрос, можно представить ситуацию: пусть в случае с переключением контекста с помощью программного прерывания реализация планировщика осталась такой же, как и в случае прямого вызова переключателя контекстов. Только вместо вызова:

```
os_context_switcher(Curr_SP_addr, Next_SP);
```

присутствует:

```
raise_context_switch();
<wait_for_context_switch_done>[1];
```

Теперь можно рассмотреть ситуацию, при которой в момент, когда разрешаются прерывания, на очереди стоит ещё одно или несколько прерываний, причём хотя бы одно из них более приоритетное, чем программное прерывание переключения контекстов, и в обработчике этого стоящего на очереди более приоритетного прерывания вызывается какая-либо из функций сервисов (средств межпроцессного взаимодействия). Что при этом получится? При этом будет тоже вызван планировщик и произойдёт ещё одно перепланирование процессов. Но т.к. предыдущая перепланировка не была завершена – т.е. процессы реально не переключились, контексты не были физически сохранены и восстановлены, то новая перепланировка просто перезапишет переменные, содержащие указатели текущего и следующего процессов. Кроме того, при определении необходимости в перепланировке будет использоваться значение **CurProcPriority**, которое фактически ошибочно, т.к. это значение приоритета следующего процесса, запланированного при прошлом вызове планировщика. Словом, произойдёт «наложение» планировок и нарушение целостности работы системы.

Поэтому крайне важно, чтобы фактическое обновление значения **CurProcPriority** и переключение контекстов процессов были «атомарны» –

[1] Под **<wait_for_context_switch_done>** предполагается весь код, обеспечивающий переключение контекстов, начиная от разрешения прерываний.

неразрывны, не прерывались другим кодом, имеющим отношение к планировке процессов. В варианте с прямым вызовом переключателя контекстов это правило выполняется само по себе – вся работа планировщика происходит в критической секции, и прямо оттуда же и вызывается переключатель контекстов.

В варианте с программным прерыванием планировка и переключение контекстов могут быть «разнесены» во времени. Поэтому само переключение и изменение текущего приоритета происходят непосредственно во время выполнения обработчика программного прерывания[1]. В нем сразу после сохранения контекста текущего процесса вызывается функция **`os_context_switch_hook()`** (где и производится непосредственно обновление значения **`CurProcPriority`**), а также указатель стека текущего процесса передаётся в **`os_context_switch_hook()`**, где он сохраняется в объекте текущего процесса, и извлекается и возвращается из функции указатель стека следующего процесса, необходимого для восстановления контекста этого процесса и последующей передачи ему управления.

Для того, чтобы не ухудшить характеристик быстродействия в обработчиках прерываний, существует специальная облегчённая встраиваемая версия планировщика, используемая некоторыми функциями-членами объектов-сервисов, оптимизированными для применения их в ISR. Код этой версии планировщика см. «**Листинг 3.4 Вариант планировщика, оптимизированный для использования в ISR**».

```
{1}void OS::TKernel::sched_isr()
{2}{
{3}    uint_fast8_t NextPrty = highest_priority(ReadyProcessMap);
{4}    if(NextPrty != CurProcPriority)
{5}    {
{6}        SchedProcPriority = NextPrty;
{7}        raise_context_switch();
{8}    }
{9}}
```

Листинг 3.4 Вариант планировщика, оптимизированный для использования в ISR

При выборе обработчика прерываний переключения контекстов следует отдавать предпочтение такому, у которого самый низкий приоритет (в случае приоритетного контроллера прерываний). Это позволит избежать лишних перепланировок и переключений контекстов в случае возникновения нескольких прерываний подряд.

[1] Этот обработчик программного прерывания всегда реализуется на ассемблере и, кроме того, является платформеннозависимым, поэтому здесь его код не приводится.

3.2.5. Достоинства и недостатки способов передачи управления

Оба способа имеют свои достоинства и недостатки. Достоинства одного способа передачи управления являются недостатками другого и наоборот. Подробнее об этом сказано ниже.

3.2.5.1. Прямая передача управления

К достоинствам прямой передачи управления относится, главным образом, то, что для реализации этого варианта не требуется наличия в целевом МК специального программного прерывания – далеко не во всех МК имеется такая аппаратная возможность. Вторым небольшим преимуществом является немного большее быстродействие по сравнению с вариантом программного прерывания, т.к. в последнем случае имеются дополнительные накладные расходы на активацию обработчика прерываний переключения контекстов, цикл ожидания переключения контекста и на вызов **`os_context_switch_hook()`**.

У варианта с прямой передачей управления существует серьёзный недостаток – при вызове планировщика из обработчика прерываний, компилятор вынужден сохранять «локальный контекст» (scratch регистры процессора) из-за вызова невстраиваемой функции переключения контекста, а это накладные расходы, которые могут оказаться весьма немалыми по сравнению с остальным кодом ISR. Негативный момент тут состоит в том, что сохранение этих регистров может оказаться совершенно ненужным – ведь в той функции[1], из-за которой они сохраняются, эти регистры не используются, поэтому если больше нет вызовов невстраиваемых функций, то код сохранения и восстановления этой группы регистров оказывается излишним.

3.2.5.2. Передача управления на основе программного прерывания

Этот вариант лишён вышеописанного недостатка. Благодаря тому, что сам по себе ISR выполняется обычным образом и никакой перепланировки из него не делается, сохранение «локального контекста» также не производится, что значительно сокращает накладные расходы и повышает производительность системы. Чтобы не испортить картину вызовом невстраиваемой функции-члена сервисного объекта межпроцессного взаимодействия рекомендуется пользоваться специаль-

[1] **`os_context_switcher(stack_item_t** Curr_SP, stack_item_t* Next_SP)`**

ными, облечёнными, встраиваемыми версиями таких функций – об этом подробнее см. «**Глава 5 Средства межпроцессного взаимодействия**».

Главным недостатком передачи управления с помощью программного прерывания является то, что не во всех аппаратных платформах имеется поддержка программного прерывания. В этом случае в качестве такого программного прерывания можно использовать одно из незанятых аппаратных прерываний. К сожалению, тут возникает некоторое отсутствие универсальности – заранее неизвестно, потребуется ли то или иное аппаратное прерывание в том или ином проекте, поэтому, если процессор специально не предоставляет подходящего прерывания, то выбор прерывания переключения контекстов передаётся (с уровня порта) на уровень проекта, и пользователь должен сам написать соответствующий код[1].

При использовании передачи управления с помощью программного прерывания в полной мере отражает ситуацию выражение: «Ядро отбирает управление у процессов».

3.2.5.3. Выводы

Учитывая вышеприведённый анализ достоинств и недостатков обоих способов передачи управления, общая рекомендация такова: если целевая платформа предоставляет подходящее прерывание для реализации переключения контекстов, то имеет смысл использовать этот вариант, особенно, если размер «локального контекста» достаточно велик.

Использование прямой передачи управления оправдано реальной невозможностью использовать программное прерывание – например, когда такое прерывание целевая платформа не поддерживает, а использование аппаратного прерывания в качестве программного невозможно по тем или иным причинам, либо если характеристики быстродействия с этим вариантом передачи управления оказываются лучше в силу меньших накладных расходов на организацию переключения контекстов, а сохранение/восстановление «локального контекста» не несёт заметных накладных расходов в силу небольшого размера[2] этого «контекста».

[1] ***scmRTOS*** предлагается ко вниманию в виде нескольких рабочих примеров использования, где весь этот код по организации и настройке программного прерывания присутствует, поэтому пользователю достаточно просто модифицировать этот код под потребности своего проекта или использовать как есть, если всё устраивает.

[2] Например, у MSP430/IAR «локальный контекст» составляет всего 4 регистра.

3.2.6. Поддержка межпроцессного взаимодействия

Поддержка межпроцессного взаимодействия сводится к предоставлению ряда функций для контроля за состояниями процессов, а также в предоставлении доступа к механизмам перепланирования составным частям ОС – средствам межпроцессного взаимодействия. Более подробно об этом см. «**Глава 5 Средства межпроцессного взаимодействия**».

3.2.7. Прерывания

3.2.7.1. Особенности использования с ОСРВ и реализация

Возникшее прерывание может быть источником события, которое нуждается в обработке тем или иным процессом, поэтому для минимизации (и детерминированности) времени отклика на событие используется (при необходимости) перепланирование процессов и передача управления наиболее приоритетному из готовых к выполнению.

Код любого обработчика прерывания, который использует сервисы межпроцессного взаимодействия, должен на входе вызвать функцию **`isr_enter()`**, которая проинкрементирует переменную **`ISR_NestCount`**, и на выходе вызвать функцию **`isr_exit()`**, которая декрементирует **`ISR_NestCount`** и по её значению определяет уровень вложенности прерываний (в случае вложенных). Когда величина **`ISR_NestCount`** становится равной 0, это означает, что имеет место выход из обработчика прерывания в основную программу, и **`isr_exit()`** производит перепланирование (при необходимости) процессов путём вызова планировщика уровня прерываний.

Для упрощения использования и переносимости код, выполняемый на входе и выходе обработчиков прерываний, помещён соответственно в конструктор и деструктор специального класса-«обёртки» - **`TISRW`**, объект которого необходимо использовать в обработчике прерываний[1]. Достаточно создать объект этого типа в коде обработчика прерываний, всё остальное компилятор сделает самостоятельно. Важно, чтобы объявление этого объекта было до первого использования функций сервисов.

[1] Упомянутые выше функции **`isr_enter()`** и **`isr_exit()`** являются функциями-членами этого класса-«обёртки».

Следует иметь в виду, что если в обработчике прерываний имеет место вызов невстраиваемой функции, то компилятор сохранит «локальный контекст» — scratch[1] регистры[2]. Поэтому желательно избегать вызовов невстраиваемых функций из обработчиков прерываний, т.к. даже частичное сохранение контекста ухудшает характеристики и по скорости, и по коду[3]. В связи с этим в текущей версии ***scmRTOS*** у некоторых объектов межпроцессного взаимодействия появились специальные дополнительные облегчённые функции для использования их в обработчиках прерываний. Функции эти являются встраиваемыми и используют облегчённую версию планировщика, которая также является встраиваемой. Подробнее об этом см. «**Глава 5 Средства межпроцессного взаимодействия**».

3.2.7.2. Отдельный стек прерываний и вложенные прерывания

С прерываниями связан ещё один аспект использования ОСРВ вытесняющего типа. Как известно, при возникновении прерывания и передаче управления обработчику прерываний программа для работы использует стек прерванного процесса, который должен иметь размер, достаточный для удовлетворения потребностей как самого процесса, так и любого обработчика прерываний. Причём суммарных потребностей и по самому наихудшему варианту – например, выполнение кода процесса занимает пространство в стеке пиковое значение, и в этот момент возникает прерывание, обработчик которого тоже займёт часть стека. Размер стека должен быть таким, чтобы и в этом случае не возникло переполнения стека.

Очевидно, что вышеприведённые обстоятельства касаются всех процессов системы, и в случае наличия обработчиков прерываний, потребляющих значительный объем стекового пространства, размеры стеков *всех* процессов должны быть увеличены на определённую величину. Это приводит к повышенным накладным расходам по памяти. В случае же вложенных прерываний ситуация драматически усугубляется.

---

[1] Как правило, компилятор делит регистры процессора на две группы: scratch и preserved. Scratch регистры – это те, которые любая функция может использовать без предварительного сохранения. Preserved – регистры, значения которых в случае необходимости должны быть сохранены. Т.е. если функции потребовался регистр из группы preserved, то она должна сначала сохранить значение регистра, а после использования восстановить.

[2] На разных платформах доля (в общем количестве) этих регистров разная, например, при использовании EWAVR они занимают примерно половину от общего количества, при использовании EW430 - меньше половины. В случае с VisualDSP++/Blackfin доля этих регистров велика, но на этой платформе и размеры стеков, как правило, достаточно большие, чтобы беспокоиться об этом.

[3] К сожалению, при использовании схемы с прямой передачей управления имеет место вызов невстраиваемой функции переключения контекстов, поэтому избежать накладных расходов на сохранение scratch регистров тут не удаётся.

Для борьбы с этим эффектом применяется переключение указателя стека процессора на специализированный стек обработчиков прерываний, в случае возникновения последних. Таким образом, стеки процессов и стек прерываний оказываются «развязанными» относительно друг друга, и не возникает необходимости резервировать в стеке каждого процесса дополнительный объем памяти для обеспечения работы обработчиков прерываний.

Реализация отдельного стека прерываний выполняется на уровне порта. Некоторые процессоры имеют аппаратную поддержку переключения указателя стека на стек прерываний, это позволяет сделать использование этой возможности эффективным и безопасным[1].

Вложенные прерывания – т.е. такие, обработчики которых могут прерывать не только работу основной программы, но и работу обработчиков прерываний также имеют особенности применения, понимание которых важно для эффективного и безопасного использование этого механизма. В случае наличия у процессора контроллера прерываний с поддержкой многоуровневых прерываний с приоритетами, ситуация с использованием вложенных прерываний оказывается достаточно проста – возникновение опасных ситуаций при разрешении вложенных прерываний, как правило, учтено разработчиками процессора, и контроллер прерываний не позволяет случаться неприятностям, например, таким, как описано ниже.

В случае, когда процессор имеет одноуровневую систему прерываний, его реализация, как правило, такова, что при возникновении любого прерывания автоматически происходит общее запрещение прерываний. Это делается из соображений простоты и безопасности. Т.е. вложенные прерывания в такой системе не поддерживаются. Для того, чтобы разрешить вложенные прерывания, достаточно сделать общее разрешение прерываний, которое на процессорах с одноуровневой системой прерываний, как правило, выключается аппаратно при передаче управления обработчику прерываний. При этом возможна ситуация, когда уже выполняющийся обработчик прерываний будет вызван ещё раз – в случае, если «висит» запрос на обработку этого же прерывания[2].

Как правило, это является ошибочной ситуацией, которую необходимо избегать. Для того, чтобы не оказаться в таком положении, нужно чётко понимать,

[1] В этом случае такой механизм является единственным реализованным в порте, и нет необходимости в отдельной реализации класса-«обёртки» **`TISRW_SS`**.

[2] Это может быть связано, например, со слишком частым возникновением событий, инициирующих прерывание, либо несброшенным флагом прерывания, который инициирует запрос на обработку прерывания.

как особенности работы процессора, так и его «контекст»[1], и весьма аккуратно писать код: перед общим разрешением прерываний запретить активизацию прерывания, обработчик которого уже выполняется, дабы избежать вторичного входа в этот же обработчик, а по окончании его работы не забыть вернуть управляющие ресурсы процессора в исходное состояние, которое было до манипуляций с разрешением вложенных прерываний.

Исходя из вышесказанного, можно дать следующую рекомендацию.

ПРЕДУПРЕЖДЕНИЕ. Несмотря на видимое преимущество схемы с отдельным стеком прерываний, не рекомендуется использовать этот вариант на процессорах, которые не имеют аппаратных средств переключения указателя стека на стек прерываний. Это связано с дополнительными накладными расходами по переключению стека, плохой переносимостью – любые нестандартные расширения являются источником проблем, а также тем, что прямое вмешательство в процесс управления указателем стека может так или иначе вызвать коллизии с адресацией локальных объектов – например, компилятор, видя тело обработчика прерываний, выделяет[2] память под локальные объекты в стеке. Причём делает это до вызова[3] конструктора «обёртки» – таким образом, после переключения указателя стека на стек прерываний память, которая была выделена ранее, физически окажется в другом месте, и программа будет работать неправильно, а компилятор не сможет выявить эту ситуацию.

Аналогично не рекомендуется использовать вложенные прерывания на процессорах, которые не поддерживают такую возможность аппаратно. Такие прерывания требуют аккуратного использования и, как правило, дополнительного обслуживания – например, блокировки источника прерывания, чтобы при разрешении прерываний не возник ещё один вызов этого же обработчика.

Краткий вывод. Мотивация использования переключения указателя стека на стек прерываний коррелирует с использованием вложенных прерываний – ведь в случае вложенных прерываний потребление стека (в прерываниях) весьма возрастает, что накладывает – в случае отсутствия переключения на отдельный стек прерываний – дополнительные требования на размеры стеков процессов[4]. В слу-

[1] Под «контекстом» в данном случае подразумевается логическое и смысловое окружение, в котором выполняется данная часть программы.
[2] Точнее – резервирует. Обычно это делается путём модификации указателя стека.
[3] Имеет на это полное право.
[4] Причём каждый процесс должен иметь такой размер стека, чтобы покрыть как потребности самого процесса, так и потребление стека обработчиками прерываний, включая всю иерархию вложенности.

чае использования ОСРВ вытесняющего типа имеется возможность построить программу так, чтобы обработчики прерываний были только источниками событий, а всю обработку событий вынести на уровень процессов. Это позволяет сделать обработчики прерываний маленькими и быстрыми, что, в свою очередь, нивелирует необходимость и в переключении на стек прерываний, и в разрешении вложенных прерываний. В этом случае тело обработчика прерываний может быть соизмеримым с накладными расходами на переключение указателя стека на стек прерываний и разрешение вложенных прерываний.

Именно так рекомендуется поступать в случае, когда процессор не поддерживает аппаратного переключения указателя стека на стек прерываний и не имеет контроллера прерываний с аппаратной поддержкой вложенных прерываний. Следует заметить, что ОСРВ с приоритетным вытеснением является в некотором роде аналогом многоуровневого приоритетного контроллера прерываний, т.е. предоставляет возможность распределить выполнение кода в соответствии с важностью/срочностью. В связи с этим, в большинстве случаев не возникает необходимости размещать код обработки событий на уровне прерываний даже при наличии такого аппаратного контроллера, а использовать прерывания только как источники событий[1], поместив их обработку на уровень процессов. Это рекомендуемый стиль построения программы.

3.2.8. Системный таймер

Системный таймер служит для формирования определённых временных интервалов, необходимых при работе процессов. Сюда относится поддержка таймаутов.

В качестве системного таймера используется обычно один из аппаратных таймеров процессора[2].

Функциональность системного таймера реализуется в функции ядра **`system_timer()`**. Код этой функции представлен – см. **«Листинг 3.5 Системный таймер».**

[1] Сделав обработчики прерываний максимально простыми, короткими и быстрыми.

[2] Для этого подходит самый простой (без «наворотов») таймер. Единственное принципиальное требование к нему – он должен быть способен генерировать периодические прерывания через равные промежутки времени – например, прерывание по переполнению. Желательно, также, чтобы имелась возможность управлять величиной периода переполнения, чтобы подобрать подходящую частоту системных тиков.

```
{1}  void OS::TKernel::system_timer()
{2}  {
{3}      SYS_TIMER_CRIT_SECT();
{4}  #if scmRTOS_SYSTEM_TICKS_ENABLE == 1
{5}      SysTickCount++;
{6}  #endif
{7}
{8}  #if scmRTOS_PRIORITY_ORDER == 0
{9}      const uint_fast8_t BaseIndex = 0;
{10} #else
{11}     const uint_fast8_t BaseIndex = 1;
{12} #endif
{13}
{14}     for(uint_fast8_t i = BaseIndex; i < (PROCESS_COUNT-1 + BaseIndex); i++)
{15}     {
{16}         TBaseProcess* p = ProcessTable[i];
{17}
{18}         if(p->Timeout > 0)
{19}         {
{20}             if(--p->Timeout == 0)
{21}             {
{22}                 set_process_ready(p->Priority);
{23}             }
{24}         }
{25}     }
{26} }
```

Листинг 3.5 Системный таймер

Как видно из исходного кода, действия очень простые:

1. если разрешён счётчик тиков, то переменная счётчика инкрементируется {5};
2. далее в цикле проверяются значения таймаутов всех зарегистрированных процессов, и если значение проверяемой переменной не равно 0[1], тогда значение декрементируется и проверяется на 0. При равенстве (после декремента) 0 (т.е. таймаут данного процесса истёк) данный процесс переводится в состояние готового к выполнению.

Т.к. эта функция вызывается внутри обработчика прерываний от таймера, то при выходе в основную программу, как описано выше, управление будет передано наиболее приоритетному процессу из готовых к выполнению. Т.е. если таймаут какого-то (более приоритетного, чем прерванный) процесса истёк, то по выходе из прерывания он получит управление. Это реализуется с помощью планировщика (см. выше)[2].

[1] Это означает, что процесс находится в ожидании с таймаутом.
[2] При наличии возможности задавать приоритет прерывания системного таймера

ЗАМЕЧАНИЕ. В некоторых ОС есть рекомендации по установке величины длительности системного тика. Чаще всего называется диапазон 10[1] – 100 мс. Возможно, применительно к тем ОС это и правильно. Баланс тут определяется желанием получить наименьшие накладные расходы на прерывания от системного таймера и желанием получить большее разрешение по времени.

Исходя из ориентации *scmRTOS* на малые МК, работающие в реальном времени, а также принимая во внимание тот факт, что накладные расходы (по времени выполнения)[2] невелики, рекомендуемое значение системного тика равно 1 – 10 мс.

Здесь можно провести аналогию с другими областями, где малые объекты являются обычно более высокочастотными: например, сердцебиение у мыши намного чаще, чем у человека, а у человека чаще, чем у слона. При этом «поворотливость» как раз обратная. В технике есть аналогичная тенденция, поэтому разумно ожидать, что для малых процессоров период системного тика меньше, чем для больших – в больших системах и накладные расходы больше ввиду, как правило, большей загрузки более мощного процессора и, как следствие, меньшей его «поворотливости».

3.3. TKernelAgent и расширения

3.3.1. Агент ядра

Класс **`TKernelAgent`** является специальным средством для предоставления доступа к ресурсам ядра при построении средств расширения функциональности операционной системы.

Замысел в целом таков. Для создания того или иного расширения функциональных средств ОС требуется доступ к определённым ресурсам ядра – в частности, к переменной ядра, содержащей приоритет активного процесса, или к карте процессов системы. Предоставлять прямой доступ к этой части представления было бы не слишком разумно – это является нарушением модели безопасности объектного подхода[3], что влечёт за собой такие негативные последствия, как неработоспособность программы при отсутствии должной дисциплины кодирования и/или потеря совместимости в случае изменения внутреннего представления ядра.

[1] А как, например, организовать динамическую индикацию с таким периодом переключения разрядов, когда известно, что для комфортной работы необходимо, чтобы период переключения (при четырех разрядах) был не более 5 мс?

[2] Ввиду малого количества процессов, а также простого и быстрого планировщика.

[3] Принципов инкапсуляции и абстракции.

Поэтому для решения задачи доступа к ресурсам ядра предложен подход на основе специально созданного класса – агента ядра – ограничивающего доступ через свой интерфейс, который является документированным. Всё это позволяет создавать расширения формализованным путём, что делает этот процесс проще и безопаснее. Код класса агента ядра – см. «**Листинг 3.6 TKernelAgent**».

```
{1} class TKernelAgent
{2} {
{3}     INLINE static TBaseProcess * cur_proc();
{4}
{5} protected:
{6}     TKernelAgent() { }
{7}     INLINE static uint_fast8_t const & cur_proc_priority();
{8}
{9}     INLINE static volatile TProcessMap & ready_process_map();
{10}    INLINE static volatile timeout_t   & cur_proc_timeout();
{11}    INLINE static void reschedule();
{12}
{13}#if scmRTOS_DEBUG_ENABLE == 1
{14}    INLINE static TService * volatile & cur_proc_waiting_for();
{15}#endif
{16}
{17}#if scmRTOS_PROCESS_RESTART_ENABLE == 1
{18}    INLINE static volatile TProcessMap * & cur_proc_waiting_map();
{19}#endif
{20}};
```

Листинг 3.6 TKernelAgent

Как видно из кода, определение класса таково, что невозможно создавать объекты этого класса. Это сделано сознательно, т.к. по замыслу **`TKernelAgent`** является основой для создания расширений: его главная функция - предоставить документированный интерфейс к ресурсам ядра. Поэтому всё использование этого кода становится возможным только в потомках этого класса, которые и являются собственно расширениями. Пример использования **`TKernelAgent`** будет подробно рассмотрен ниже при описании базового класса для создания средств межпроцессного взаимодействия **`TService`**.

Весь интерфейс класса представляет собой встраиваемые функции, что в большинстве случаев позволяет реализовать необходимые расширения без потери эффективности по сравнению с вариантом, когда доступ к ресурсам ядра производится непосредственно.

3.3.2. Расширения

Вышеописанный класс агента ядра позволяет создавать дополнительные средства, расширяющие функциональные возможности ОС. Методология создания таких средств проста – достаточно объявить класс-наследник **`TKernelAgent`** и определить его содержимое. Такие классы называются расширениями операционной системы.

Размещение кода ядра ОС таково, что определения классов и определения ряда функций-членов классов разнесены в заголовочном файле os_kernel.h. Это даёт возможность написать пользовательский класс, которому доступны все определения типов ядра ОС, и в то же время определения этого пользовательского класса оказываются доступны в функциях-членах классов ядра – например, в планировщике и в функции системного таймера[1].

Подключение расширений осуществляется с помощью конфигурационного файла scmRTOS_extensions.h, который включается в os_kernel.h между определениями типов ядра и их функций-членов. Это позволяет определение класса-расширения физически разместить в отдельном пользовательском заголовочном файле и подключить в проект посредством включения этого файла в scmRTOS_extensions.h. После этого расширение готово к использованию в соответствии со своим назначением.

[1] В пользовательских хуках.

Глава 4 Процессы

— Поручик, вы любите детей?

— Детей? Нет-с!.. Но сам процесс…

Анекдот

4.1. Общие сведения и внутреннее представление

4.1.1. Процесс как таковой

Процесс в ***scmRTOS*** – это объект типа, производного от класса **`OS::TBaseProcess`**. Причина, по которой для каждого процесса требуется отдельный тип (ведь почему бы просто не сделать все процессы объектами типа **`OS::TBaseProcess`**), состоит в том, что процессы, несмотря на всю похожесть, всё-таки отличаются – у них разные размеры стеков и разные значения приоритетов (которые, не следует забывать, задаются статически). Для определения типов процессов используется стандартное средство C++ – шаблоны (templates), что позволило получить «компактные» типы процессов, в которых содержатся все необходимые внутренности, включая и непосредственно стек процесса, который у всех процессов имеет разный размер и задаётся индивидуально.

```
{1}  class TBaseProcess
{2}  {
{3}      friend class TKernel;
{4}      friend class TISRW;
{5}      friend class TISRW_SS;
{6}      friend class TKernelAgent;
{7}
{8}      friend void run();
{9}
{10} public:
{11}     TBaseProcess( stack_item_t * StackPoolEnd
{12}                 , TPriority pr
{13}                 , void (*exec)()
{14}             #if scmRTOS_DEBUG_ENABLE == 1
{15}                 , stack_item_t * StackPool
{16}             #endif
{17}                 );
{18} protected:
{19}     INLINE void set_unready() { Kernel.set_process_unready(this->Priority); }
{20}     void init_stack_frame( stack_item_t * StackPoolEnd
{21}                          , void (*exec)()
{22}                      #if scmRTOS_DEBUG_ENABLE == 1
{23}                          , stack_item_t * StackPool
{24}                      #endif
{25}                          );
{26} public:
{27}     static void sleep(timeout_t timeout = 0);
{28}            void wake_up();
{29}            void force_wake_up();
{30}     INLINE void start() { force_wake_up(); }
{31}     INLINE bool is_sleeping() const;
{32}     INLINE bool is_suspended() const;
{33}
{34} #if scmRTOS_DEBUG_ENABLE == 1
{35}     INLINE TService * waiting_for() { return WaitingFor; }
{36} public:
{37}            size_t     stack_slack() const;
{38} #endif // scmRTOS_DEBUG_ENABLE
{39}
{40} #if scmRTOS_PROCESS_RESTART_ENABLE == 1
{41} protected:
{42}            void reset_controls();
{43} #endif
{44}     //-----------------------------------------------------
{45}     //
{46}     //    Data members
{47}     //
{48} protected:
{49}     stack_item_t *     StackPointer;
{50}     volatile timeout_t Timeout;
{51}     const TPriority    Priority;
{52} #if scmRTOS_DEBUG_ENABLE == 1
{53}     TService           * volatile WaitingFor;
{54}     const stack_item_t * const StackPool;
{55} #endif // scmRTOS_DEBUG_ENABLE
{56}
{57} #if scmRTOS_PROCESS_RESTART_ENABLE == 1
{58}     volatile TProcessMap * WaitingProcessMap;
{59} #endif
{60}
{61} #if scmRTOS_SUSPENDED_PROCESS_ENABLE != 0
{62}     static TProcessMap SuspendedProcessMap;
{63} #endif
{64} };
```

Листинг 4.1 TBaseProcess

4.1.2. TBaseProcess

Основная функциональность процесса определена в базовом классе **`OS::TBaseProcess`**, от которого, как уже говорилось выше, и производятся сами процессы на основе шаблона **`OS::process<>`**. Такой метод использован для того, чтобы не множить одинаковый код в экземплярах[1] шаблона при их реализации. Поэтому в самом шаблоне объявлено только то, что относится к различающимся в разных процессах сущностям – стеки и исполняемые функции процесса (**`exec`**). Исходный код класса **`OS::TBaseProcess`** представлен[2] – см. «**Листинг 4.1 TBaseProcess**»).

Несмотря на кажущуюся обширность определения этого класса, на самом деле он очень небольшой и простой. Его представление содержит всего три члена-данных – это указатель стека {49}, счётчик тиков таймаута {50} и значение приоритета {51}. Остальные члены-данные являются вспомогательными и присутствуют только при разрешении дополнительной функциональности – возможность прерывать работу процесса в любой момент с последующим перезапуском, а также средства отладки[3].

Интерфейс класса предоставляет следующие функции:

- **`sleep(timeout_t timeout = 0)`**. Переводит процесс в состояние «спячки»: значение аргумента присваивается внутренней переменной-счётчику таймаута, процесс удаляется из карты процессов, готовых к выполнению, и вызывается планировщик, который передаст управление следующему процессу из готовых к выполнению.
- **`wake_up()`**. Выводит процесс из состояния «спячки». Процесс переводится в состояние готового к выполнению, только если он находился в состоянии ожидания с таймаутом события; при этом если этот процесс имеет приоритет выше текущего, то он сразу получает управление;
- **`force_wake_up()`**. Выводит процесс из состояния «спячки». Процесс переводится в состояние готового к выполнению всегда. При этом если этот процесс имеет приоритет выше текущего, то он сразу получает управление. Этой функцией нужно пользоваться с особой осторожностью, т.к. некорректное использование может привести к неправильной (непредсказуемой) работе программы;
- **`is_sleeping()`**. Проверяет, находится ли процесс в состоянии «спячки», т.е. в состоянии ожидания с таймаутом события;
- **`is_suspended()`**. Проверяет, находится ли процесс в неактивном состоянии.

[1] На жаргоне часто используют термин инстанс – instance.

[2] На самом деле существует два варианта этого класса – обычный (он и показан) и с отдельным стеком для адресов возвратов, код которого тут не приводится для краткости, т.к. никаких принципиальных для понимания и изложения отличий в нём нет.

[3] Это же касается и остального кода – большая часть определения класса занята описанием этих вспомогательных возможностей.

4.1.3. Стек

Стек процесса – это некоторая непрерывная область оперативной памяти, используемая для хранения в ней данных процесса, а также сохранения контекста процесса и адресов возвратов из функций и прерываний.

В силу особенностей некоторых архитектур может быть использовано два раздельных стека – один для данных, другой для адресов возвратов. ***scmRTOS*** поддерживает такую возможность, позволяя размещать в каждом объекте-процессе две области ОЗУ – два стека, размер каждой из которых может быть указан индивидуально, исходя из требований прикладной задачи. Поддержка двух стеков включается с помощью макроса **`SEPARATE_RETURN_STACK`**, определяемого в файле os_target.h.

В защищённой секции объявлена очень важная функция **`init_stack_frame()`**, которая отвечает за формирование стекового кадра (stack frame). Дело в том, что старт исполняемых функций процессов происходит не так, как у обычных функций – исполняемые функции процесса не вызываются традиционным образом. Управление в них попадает тем же способом, что и при передаче управления между процессами (при переключении контекстов), поэтому старт исполняемой функции процесса происходит путём восстановления контекста данного процесса из стека с последующим переходом по адресу, содержащемуся в стеке на месте сохранённого адреса точки прерывания процесса. Для того, чтобы такой старт стал возможным, требуется подготовить стек процесса соответствующим образом – проинициализировать ячейки памяти в стеке по заданным адресам необходимыми значениями – т.е. содержимое стека процесса должно быть таким, как будто у процесса до этого отобрали управление (сохранив, естественно, контекст процесса). Конкретные действия по подготовке стекового кадра являются индивидуальными для каждой платформы, поэтому реализация функции **`init_stack_frame()`** вынесена на уровень портов операционной системы.

4.1.4. Таймауты

Каждый процесс имеет специальную переменную **`Timeout`** для контроля за поведением процесса при ожиданиях событий с таймаутами или при «спячке». По сути эта переменная является счётчиком тиков системного таймера, и если её значение не равно нулю, то в обработчике прерывания системного таймера она декрементируется и сравнивается с нулём, при равенстве которому процесс-владе-

лец этой переменной переводится в готовые к выполнению. Таким образом, если процесс находится в «спячке» с таймаутом, т.е. переведён в неготовые к выполнению путём вызова функции **`sleep(timeout)`** с аргументом, отличным от нуля, то через промежуток времени, равный количеству тиков системного таймера[1], процесс будет «разбужен»[2] в обработчике прерываний системного таймера.

Аналогичная ситуация будет и в случае вызова функции сервиса, которая предполагает ожидание события с таймаутом. В этом случае процесс будет переведён в готовые к выполнению либо при возникновении события, которое он ожидает, вызвав функцию сервиса, либо по истечению таймаута. Значение, возвращаемое функцией сервиса, однозначно указывает на источник «пробуждения» процесса, что позволяет пользовательской программе без проблем принять решение о дальнейших действиях в сложившейся ситуации.

4.1.5. Приоритеты

Каждый процесс имеет также поле данных, содержащее приоритет процесса. Это поле является идентификатором процесса при манипуляции с процессами и их представлением, в частности, приоритет процесса – это индекс в таблице указателей на процессы, находящейся в составе ядра, куда записывается адрес каждого процесса при регистрации, подробнее см. с. 50.

Приоритеты являются уникальными – не может быть двух процессов с одинаковым приоритетом. Внутреннее представление приоритета – переменная целочисленного типа. Для безопасности использования при задании приоритетов используется специальный перечислимый тип **`TPriority`**, более подробно о задании приоритетов см. замечание на с. 44.

4.1.6. Функция `sleep()`

Эта функция служит для перевода текущего процесса из активного состояния в неактивное. При этом если функция вызывается с аргументом, равным 0 (или без указания аргумента – функция объявлена с аргументом по умолчанию, равным 0), то процесс перейдёт в «спячку» до тех пор, пока его не разбудит, например, какой-либо другой процесс с помощью функции

[1] Строго говоря, не точно равный количеству тиков системного таймера, а с точностью до доли этого периода, которая зависит от момента вызова функции **`sleep`** по отношению к моменту возникновения прерывания системного таймера.
[2] Т.е. переведён в готовые к выполнению.

TBaseProcess::force_wake_up(). Если функция вызывается с аргументом, то процесс будет «спать» указанное количество тиков системного таймера, после чего будет «разбужен», т.е. приведён в состояние готового к выполнению. В этом случае «спячка» также может быть прервана другим процессом или обработчиком прерывания с помощью функций **TBaseProcess::wake_up()**, **TBaseProcess::force_wake_up()**.

4.2. Создание и использование процесса

4.2.1. Определение типа процесса

Для создания процесса нужно определить его тип и объявить объект этого типа.

Тип конкретного процесса описывается с помощью шаблона **OS::process**: см. **«Листинг 4.2 Определение шаблона типа процесса»**

```
{1}   template<TPriority pr, size_t stack_size>
{2}   class process : public TBaseProcess
{3}   {
{4}   public:
{5}       INLINE_PROCESS_CTOR process();
{6}
{7}       OS_PROCESS static void exec();
{8}
{9}   #if scmRTOS_PROCESS_RESTART_ENABLE == 1
{10}      INLINE void terminate();
{11}  #endif
{12}
{13}  private:
{14}      stack_item_t Stack[stack_size/sizeof(stack_item_t)];
{15}  };
```

Листинг 4.2 Определение шаблона типа процесса

Как видно, к тому, что предоставляет базовый класс, добавлены две сущности:

- стек процесса **Stack** с размером **stack_size**. Размер задаётся в байтах;
- статическая функция **exec()**, являющаяся собственно той функцией, где размещается пользовательский код процесса.

4.2.2. Объявление объекта процесса и его использование

Теперь достаточно объявить объект этого типа, который и будет собственно процессом, а также определить саму процессную функцию **exec()**.

```
typedef OS::process<OS::prn, 100> TSlon;
TSlon Slon;
```

где **n** – номер приоритета.

«**Листинг 2.1 Исполняемая функция процесса**» иллюстрирует пример типовой процессной функции.

Использование процесса состоит, главным образом, в написании пользовательского кода внутри функции процесса. При этом, как уже говорилось, следует соблюдать ряд простых правил:

- необходимо позаботиться о том, чтобы поток управления программы не покидал процессной функции, в противном случае, в силу того, что эта функция не была вызвана обычным образом, при выходе из неё поток управления попадёт, грубо говоря, в неопределённые адреса, что повлечёт неопределённое поведение программы (хотя на практике поведение, как правило, вполне определённое – программа не работает!);
- использовать функцию **TBaseProcess::wake_up()** нужно с осторожностью и внимательно, а **TBaseProcess::force_wake_up()** – с особой осторожностью, т.к. неаккуратное использование может привести к несвоевременной «побудке» спящего (отложенного) процесса, что может привести к коллизиям в межпроцессном взаимодействии.

4.2.3. Старт процесса в неактивном состоянии

Иногда возникает необходимость в том, чтобы исполняемая функция процесса начинала работу не сразу после старта системы, а по определённому сигналу. Например, есть несколько процессов, которые должны начать свою работу только после инициализации/настройки какого-то (возможно, внешнего по отношению к МК) оборудования, в противном случае могут возникнуть неприятные последствия из-за некорректных действий по отношению к такому оборудованию. В этой ситуации потребуется некоторая диспетчеризация — процессы каким-то образом должны будут организовать свою работу так, чтобы не нарушить логику взаимодействия с этим оборудованием - например, все процессы, кроме одного (диспетчера) встают в самом начале в ожидание события (старта работы), которое будет им просигналено процессом-диспетчером. Процесс-диспетчер выполняет всю необходимую подготовительную работу и затем объявляет старт работы ожидающим процессам. Описанный подход потребует в каждом ожидающем

старта процессе добавление соответствующего кода вручную, что загромождает код, добавляет работы и чревато ошибками.

Кроме того, могут быть иные ситуации, когда требуется, чтобы процесс не сразу начал свою работу. Для обеспечения описанной функциональности процесс имеет возможность стартовать в т.н. неактивном состоянии. Такой процесс ничем не отличается от любого другого кроме того, что в карте процессов, готовых к выполнению (**ReadyProcessMap**), отсутствует его тег.

Объявление такого процесса выглядит так:

```
typedef OS::process<OS::pr1, 300, OS::ssSuspended[1]> TProc2;
...
TProc2 Proc2;
```

В дальнейшем для старта работы этого процесса запускающий код должен будет вызвать функцию **force_wake_up()**:

```
Proc2.force_wake_up();
```

4.3. Перезапуск процесса

Иногда возникает ситуация, когда необходимо прервать выполнение процесса извне и запустить его выполнение сначала. Например, некий процесс производит длительные вычисления, и случается так, что результаты этих вычислений оказываются в какой-то момент уже не нужны, а необходимо запустить новый цикл вычислений с новыми данными. Сделать это можно, завершив выполнение процесса с возможностью последующего его запуска с самого начала.

Для реализации вышесказанного ОС предоставляет пользователю две функции:

- OS::process::terminate();
- OS::TBaseProcess::start().

Функция **terminate()** предназначена для вызова извне останавливаемого процесса. Внутри неё производится приведение всех связанных с данным процессом ресурсов в исходное состояние и процесс переводится в состояние неготовых

[1]Префикс **ss** означает Start State.

к выполнению. При этом, если процесс находился в ожидании какого-либо сервиса, тег процесса удаляется из карты ожидающих процессов этого сервиса.

Запуск процесса производится раздельно – чтобы пользователь имел возможность сделать это в нужный с его точки зрения момент - и осуществляется с помощью функции **`start()`**, которая просто переводит процесс в готовые к выполнению. Процесс начнёт работу в соответствии с очерёдностью, определяемой его приоритетом и загрузкой ОС.

Для того, чтобы прерывание работы процесса и его старт работали правильно, эта функциональность должна быть разрешена при конфигурации – значение макроса **`scmRTOS_PROCESS_RESTART_ENABLE`** должно быть установлено в **`1`**.

Глава 5 Средства межпроцессного взаимодействия

— *Мальчик, тебя как зовут?*
— *Чего?..*
— *Ты что – тормоз?*
— *Вася.*
— *Что «Вася»?*
— *Я не тормоз.*

Анекдот

5.1. Введение

В ***scmRTOS***, начиная с версии 4, разработан и применён принципиально иной, нежели в предыдущих версиях, механизм реализации средств межпроцессного взаимодействия. Ранее каждый класс сервиса был разработан индивидуально и никак не был связан с остальными, а для доступа к ресурсам ядра классы сервисов были объявлены как «друзья» ядра. Такой подход не позволял достичь повторного использования кода[1] и не давал возможности расширять набор сервисов, по каким причинам решено было от него отказаться и спроектировать лишённый обоих недостатков вариант.

В основе реализации лежит концепция расширения функциональности ОС путём определения классов-расширений на основе наследования от **`TKernelAgent`** (см. «**3.3 TKernelAgent и расширения**»).

[1] Поскольку средства межпроцессного взаимодействия производят сходные действия по взаимодействию с ресурсами ядра, они содержат местами почти идентичный код.

Ключевым классом для построения средств межпроцессного взаимодействия является класс **TService**, в котором реализована общая функциональность всех классов-сервисов, и все они являются потомками **TService**. Это касается как штатного набора сервисов, входящих в дистрибутив ОС, так и тех, которые разработаны[1] в качестве расширений стандартного ряда сервисов.

К средствам межпроцессного взаимодействия, входящим в состав ***scmRTOS***, относятся:

- **OS::TEventFlag;**
- **OS::TMutex;**
- **OS::message;**
- **OS::channel;**

5.2. TService

5.2.1. Определение класса

Код базового класса для построения сервисных типов приведён – см. «**Листинг 5.1 TService**».

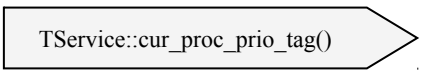
```
{1} class TService : public TKernelAgent
{2} {
{3} protected:
{4}     TService() : TKernelAgent() { }
{5}
{6}     INLINE static TProcessMap  cur_proc_prio_tag();
{7}     INLINE static TProcessMap  highest_prio_tag(TProcessMap map);
{8}
{9}     //--------------------------------------------------------------------------
{10}    //
{11}    //   Base API
{12}    //
{13}    INLINE        void suspend               (TProcessMap volatile & waiters_map);
{14}    INLINE static bool is_timeouted          (TProcessMap volatile & waiters_map);
{15}           static bool resume_all            (TProcessMap volatile & waiters_map);
{16}    INLINE static bool resume_all_isr        (TProcessMap volatile & waiters_map);
{17}           static bool resume_next_ready     (TProcessMap volatile & waiters_map);
{18}    INLINE static bool resume_next_ready_isr (TProcessMap volatile & waiters_map);
{19}};
```

Листинг 5.1 TService

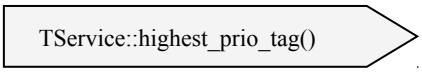
[1] Или могут быть разработаны пользователем под нужды своего проекта.

Как и класс-предок **TKernelAgent**, класс **TService** не позволяет создавать объекты своего типа: его назначение - предоставить базу для построения конкретных типов – средств межпроцессного взаимодействия. Интерфейс этого класса представляет собой набор функций, выражающих общие действия любого класса-сервиса в контексте передачи управления между процессами. Логически эти функции можно разделить на две части: основные и служебные.

К служебным функциям относятся:

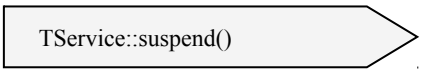


1. Возвращает тег[1], соответствующий текущему активному процессу. Этот тег активно используется основными функциями сервисов для фиксации идентификаторов[2] процессов при постановке текущего процесса в состояние ожидания.

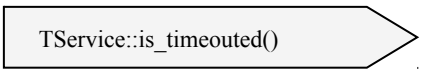


2. Возвращает тег наиболее приоритетного процесса из карты процессов, передаваемой в качестве аргумента. Используется главным образом для получения идентификатора (процесса) из зафиксированных в карте процессов объекта-сервиса, соответствующего процессу, который следует перевести в готовы к выполнению.

Основные функции:



1. Переводит процесс в состояние неготовых к выполнению с фиксацией идентификатора процесса в карте процессов сервиса и вызывает планировщик ОС. Эта функция является основой функций-членов сервисов, которые используются для ожидания события (wait, pop, read) или действий, способных вызвать ожидание освобождения ресурса (lock, push, write).

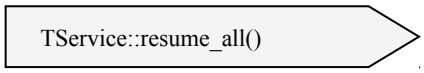


2. Функция возвращает **false**, если процесс был переведён в готовые к выполнению путём вызова функции-члена сервиса; если же процесс был переведён в готовые к выполнению по таймауту[3] или принудительно с помощью функций-членов класса **TBaseProccess** **wake_up()** и **force_wake_up()**, то функция возвращает **true**. Результат этой функции используется для определения, дождался ли процесс события (освобождения ресурса), которого ждал, или нет.

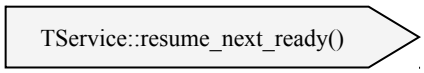
[1] Тег процесса технически является маской типа **TProcessMap**, в которой только один ненулевой бит. Позиция этого бита в маске соответствует приоритету процесса. Теги процессов служат для манипуляции с объектами **TProcessMap**, которые определяют готовность/неготовность процессов к выполнению, а также служат для фиксации тегов процессов.

[2] Наряду с номером приоритета процесса тег тоже может выполнять роль идентификатора процесса – между номером приоритета и тегом процесса существует однозначное соответствие. Каждый из типов идентификаторов имеет преимущества по эффективности использования в конкретной ситуации, поэтому оба типа интенсивно используются в коде ОС.

[3] Иными словами, «разбужен» в обработчике системного таймера.



3. Функция проверяет наличие процессов, «записанных» в карте процессов данного сервиса, но находящихся в состоянии неготовых к выполнению[1]; если таковые имеются, то все они переводятся в состояние готовых к выполнению и вызывается планировщик. При этом функция возвращает **`true`**, в противном случае – **`false`**.



4. Эта функция производит действия, сходные с вышеописанной resume_all(), но с той разницей, что при наличии ожидающих процессов, в готовые к выполнению из них переводятся не все, а только один – самый приоритетный.

Для функций **`resume_all()`** и **`resume_next_ready()`** существуют версии, оптимизированные для использования внутри обработчиков прерываний – это **`resume_all_isr()`** и **`resume_next_ready_isr()`**. По назначению и смыслу они похожи на основные варианты[2], главное отличие состоит в том, что из них не вызывается планировщик.

5.2.2. Использование

5.2.2.1. Предварительные замечания

Любой сервисный класс создаётся из **`TService`** путём наследования. Для примера использования **`TService`** и создания сервисного класса на его основе будет рассмотрено одно из штатных средств межпроцессного взаимодействия – **`TEventFlag`**:

```
class TEventFlag : public TService { ... }
```

Сам сервисный класс **`TEventFlag`** будет подробно описан ниже, в данный момент для целостности повествования следует отметить, что это средство межпроцессного взаимодействия служит для синхронизации работы процессов в соответствии с возникающими событиями. Основная идея использования: один процесс ждёт события, используя для этого функцию-член класса **`TEventFlag::wait()`**, другой процесс[3] при возникновении события, которое

[1] То есть процессов, состояние ожидания которых не было прервано по таймауту и/или принудительно с помощью **`TBaseProccess::wake_up()`** и **`TBaseProccess::force_wake_up()`**.
[2] Поэтому их полноценное описание не приводится.
[3] Или обработчик прерываний – смотря что является источником событий. Для обработчика прерываний существует специальная версия функции, которая сигналит флаг, но в контексте текущего описания это несущественно, поэтому этот нюанс опущен.

должно быть обработано в ожидающем процессе, сигналит флаг с помощью функции-члена **`TEventFlag::signal()`**.

Учитывая вышесказанное, основное внимание при рассмотрении примера использования будет уделено именно этим двум функциям, т.к. именно они несут основную смысловую нагрузку сервисного класса[1] и его разработка сводится, в основном, к разработке таких функций.

5.2.2.2. Требования к функциям разрабатываемого класса

Требования к функции ожидания флага события. Функция должна проверять факт возникновения события на момент вызова функции и при отсутствии такового иметь возможность ожидать[2] событие как безусловно, так и с условием до истечения таймаута. В случае возврата из функции по событию значение возврата **`true`**; в случае возврата из функции по таймауту значение возврата – **`false`**.

Требования к функции посылки флага события. Функция должна перевести все процессы, ожидающие флага события, в состояние готовых к выполнению и передать управление планировщику.

5.2.2.3. Реализация

Внутри функции-члена **`wait()`**, см. «**Листинг 5.2 Функция TEventFlag::wait()**», первым делом производится проверка, не просигналено ли уже событие, и если это имеет место быть, то функция возвращает **`true`**. Если же событие не было просигналено (т.е. нужно его ожидать), то выполняются подготовительные действия – в частности, значение таймаута ожидания записывается в переменную **`Timeout`** текущего процесса и вызывается функция **`suspend()`**, определённая в базовом классе **`TService`**, которая записывает тег текущего процесса в карту процессов объекта-флага события, переданную функции **`suspend()`** в качестве аргумента, переводит данный процесс в неготовые к выполнению и отдаёт управление другим процессам путём вызова планировщика.

При возврате из **`suspend()`**, что означает, что данный процесс был переведён в готовые к выполнению, производится проверка на предмет того, что явилось источником «пробуждения» данного процесса. Это выполняется с помощью вызова функции **`is_timeouted()`**, которая возвращает **`false`**, если процесс был

[1] Остальное его представление носит вспомогательный характер и служит для придания законченности классу и улучшению его пользовательских характеристик.
[2] Т.е. отдать управление ядру системы и остаться в пассивном ожидании.

«разбужен» через вызов функции **`TEventFlag::signal()`** – т.е. ожидаемое событие возникло (и таймаута не произошло), и **`true`**, если «пробуждение» процесса произошло по истечении таймаута, заданного аргументом **`TEventFlag::wait()`**, или принудительно.

Такая логика работы функции-члена **`TEventFlag::wait()`** позволяет эффективно использовать её в пользовательском коде при организации работы процесса, синхронизированной с возникновением требуемых событий[1]. При этом код реализации этой функции простой и прозрачный.

```
{1} bool OS::TEventFlag::wait(timeout_t timeout)
{2} {
{3}     TCritSect cs;
{4}
{5}     if(Value)                          // if flag already signaled
{6}     {
{7}         Value = efOff;                 // clear flag
{8}         return true;
{9}     }
{10}    else
{11}    {
{12}        cur_proc_timeout() = timeout;
{13}
{14}        suspend(ProcessMap);
{15}
{16}        if(is_timeouted(ProcessMap))
{17}            return false;            // waked up by timeout or by externals
{18}
{19}        cur_proc_timeout() = 0;
{20}        return true;            // otherwise waked up by signal() or signal_isr()
{21}    }
{22}}
```

Листинг 5.2 Функция TEventFlag::wait()

```
{1}  void OS::TEventFlag::signal()
{2}  {
{3}      TCritSect cs;
{4}      if(!resume_all(ProcessMap))   // if no one process was waiting for flag
{5}          Value = efOn;
{6}  }
```

Листинг 5.3 Функция TEventFlag::signal()

Код функции **`TEventFlag::signal()`**, см. «**Листинг 5.3 Функция TEventFlag::signal()**», предельно прост: внутри неё все ожидающие данного флага событий процессы переводятся в готовые к выполнению и производится перепланировка. Если таковых не оказалось, то внутренняя переменная флага событий **`efOn`** получает значение **`true`**, что означает, что событие произошло, но его никто

[1] В том числе и при отсутствии возникновения оных в заданный интервал времени.

пока не обработал. Более подробно об особенностях функционирования сервисного класса флага событий см. «**5.3OS::TEventFlag**».

Подобным образом может быть спроектировано и определено любое средство межпроцессного взаимодействия. При его разработке необходимо лишь чётко представлять, что делают функции-члены класса **`TService`**, и использовать их к месту.

5.3. OS::TEventFlag

При работе программы часто возникает необходимость в синхронизации между процессами. Т.е., например, один из процессов для выполнения своей работы должен дождаться события. При этом он может поступать разными способами: может просто в цикле опрашивать глобальный флаг или делать то же самое с некоторым периодом, т.е. опросил — «упал в спячку» с таймаутом — «проснулся» — опросил и т.д. Первый способ плох тем, что при этом все процессы с меньшим приоритетом не получат управления, т.к. в силу своих более низких приоритетов они не смогут вытеснить процесс, опрашивающий в цикле глобальный флаг. Второй способ тоже плох – период опроса получается достаточно большим (т.е. временное разрешение невысокое), и в процессе опроса процесс будет занимать процессор на переключение контекстов, хотя неизвестно, произошло ли событие.

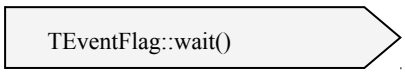
Грамотным решением в этой ситуации является перевод процесса в состояние ожидания события и передача управления процессу только когда событие произойдёт.

Эта функциональность в ***scmRTOS*** реализуется с помощью объектов **`OS::TEventFlag`** (флаг события). Определение класса - см. «**Листинг 5.4 OS::TEventFlag**».

```
{1}  class TEventFlag : public TService
{2}  {
{3}  public:
{4}      enum TValue { efOn = 1, efOff= 0 }; // prefix 'ef' means: "Event Flag"
{5}
{6}  public:
{7}      INLINE TEventFlag(TValue init_val = efOff);
{8}
{9}             bool wait(timeout_t timeout = 0);
{10}     INLINE void signal();
{11}     INLINE void clear()       { TCritSect cs; Value = efOff; }
{12}     INLINE void signal_isr();
{13}     INLINE bool is_signaled() { TCritSect cs; return Value == efOn; }
{14}
{15} private:
{16}     volatile TProcessMap ProcessMap;
{17}     volatile TValue      Value;
{18} };
```

Листинг 5.4 OS::TEventFlag

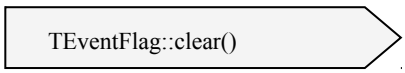
С объектами этого типа можно делать четыре действия:



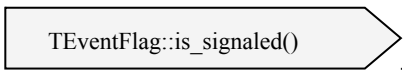
1. ждать. При вызове функции **`wait()`** происходит следующее: проверяется, установлен ли флаг и если установлен, то флаг сбрасывается и функция возвращает **`true`**, т.е. событие на момент опроса уже произошло. Если флаг не установлен (т.е. событие ещё не произошло), то процесс переводится в состояние ожидания этого флага (события) и управление отдаётся ядру, которое, перепланировав процессы, запустит следующий. Если вызов функции был произведён без аргументов (или с аргументом, равным 0), то процесс будет находиться в состоянии ожидания до тех пор, пока флаг события не будет «просигнален» другим процессом или обработчиком прерывания (с помощью функции **`signal()`**) или выведен из неактивного состояния с помощью функции **`TBaseProcess::force_wake_up()`** (в последнем случае нужно проявлять крайнюю осторожность). Если функция **`wait()`** была вызвана без аргумента, то она всегда возвращает **`true`**. Если функция была вызвана с аргументом (целое число от 1 до 2^n-1, где n – разрядность объектов таймаутов, задаётся на уровне пользовательского проекта), который обозначает таймаут на ожидание в тиках системного таймера, то процесс будет ждать события, как и в случае вызова функции **`wait()`** без аргумента, но, если в течение указанного периода флаг события не будет «просигнален», процесс будет «разбужен» таймером и функция **`wait()`** вернёт **`false`**. Таким образом реализуются ожидание безусловное и ожидание с таймаутом;

2. «сигналить». Процесс, который желает сообщить посредством объекта **`TEventFlag`** другим процессам о том, что то или иное событие произошло, должен вызвать функцию **`signal()`**. При этом все процессы, ожидающие указанное событие, будут переведены в состояние готовых к выполнению, а управление получит самый приоритетный из них (остальные в порядке очерёдности приоритетов);

3. вариант вышеописанной функции, оптимизированный для использования в прерываниях. Функция является встраиваемой и использует специальную облегчённую встраиваемую версию планировщика. Этот вариант нельзя использовать вне кода обработчика прерываний;



4. очищать. Иногда для синхронизации нужно дождаться следующего события, а не обрабатывать уже произошедшее. В этом случае необходимо очистить флаг события и только после этого перейти к ожиданию. Для очистки служит функция **`clear`**;



5. проверять. Не всегда нужно ждать события, отдавая управление. Иногда по логике работы программы нужно только проверить факт его свершения. Эту задачу выполняет функция **`is_signaled()`**.

Пример использования флага события – см. «**Листинг 5.5 Использование TEventFlag».**

В этом примере один процесс (**`Proc1`**) ждёт события с таймаутом, равным 10 тиков системного таймера {9}. Второй процесс (**`Proc2`**) при выполнении условия «сигналит» {27}. При этом, если первый процесс был более приоритетным, то он сразу получит управление.

ЗАМЕЧАНИЕ. Когда произошло событие и какой-то процесс «сигналит» флаг, то <u>все</u> процессы, ожидавшие этот флаг, будут переведены в состояние готовых к выполнению. Другими словами, все, кто ждал, дождались. Управление они, конечно, получат в порядке очерёдности их приоритетов, но событие не будет пропущено ни одним процессом, успевшим встать на ожидание, независимо от приоритета процесса. Таким образом, флаг события обладает свойством широковещательности, что весьма полезно для организации оповещений и синхронизации многих процессов по одному событию. Конечно, ничего не мешает использовать флаг событий по схеме «точка-точка», когда есть только один ожидающий события процесс.

```
{1}  OS::TEventFlag EFlag;
{2}  ...
{3}  //--------------------------------------------------------------------
{4}  template<> void Proc1::exec()
{5}  {
{6}      for(;;)
{7}      {
{8}          ...
{9}          if( EFlag.wait(10) ) // wait event for 10 ticks
{10}         {
{11}             ...   // do something
{12}         }
{13}         else
{14}         {
{15}             ...   // do something else
{16}         }
{17}         ...
{18}     }
{19} }
{20} ...
{21} //--------------------------------------------------------------------
{22} template<> void Proc2::exec()
{23} {
{24}     for(;;)
{25}     {
{26}         ...
{27}         if( ... ) EFlag.signal();
{28}         ...
{29}     }
{30} }
{31} //--------------------------------------------------------------------
```

Листинг 5.5 Использование TEventFlag

5.4. OS::TMutex

Семафор Mutex (от Mutual Exclusion – взаимное исключение), как видно из названия, служит для организации взаимного исключения доступа к нему. Т.е. в каждый момент времени не может быть более одного процесса, захватившего этот семафор. Если какой-либо процесс попытается захватить Mutex, который уже занят другим процессом, то пытающийся процесс будет ждать, пока семафор не освободится.

Основное применение семафоров Mutex – организация взаимного исключения при доступе к тому или иному ресурсу: например, некоторый статический массив с глобальной областью видимости[1], и два процесса обмениваются друг с другом данными через этот массив. Во избежание ошибок при обмене нужно исключить возможность иметь доступ к массиву для одного процесса на протяжении

[1] Чтобы к нему имелся доступ различных частей программы.

промежутка времени, пока с массивом работает другой процесс. Использовать для этого критическую секцию не лучший способ, т.к. при этом прерывания будут запрещены на все время обращения процесса к массиву, а это время может быть значительным, и в течение его система будет не способна реагировать на события. В этой ситуации как раз хорошо подходит семафор взаимоисключения: процесс, который планирует работать с совместно используемым ресурсом, должен сначала захватить семафор Mutex. После этого можно спокойно работать с ресурсом. По окончании работы нужно освободить семафор, чтобы другие процессы могли получить к нему доступ. Излишне напоминать, что так вести себя должны все процессы, работающие с общим ресурсом, т.е. производить обращение через семафор[1].

Эти же самые соображения в полной мере относятся к вызову нереентерабельной[2] функции.

ПРЕДУПРЕЖДЕНИЕ. При использовании семафоров взаимоисключения возможно возникновение ситуации, когда один процесс, захватив семафор и работая с соответствующим ресурсом, пытается получить доступ к другому ресурсу, доступ к которому также производится через захват другого семафора, и этот семафор уже захвачен другим процессом, который, в свою очередь, пытается получить доступ к ресурсу, с которым уже работает первый процесс. При этом получается, что что оба процесса ждут, когда каждый из них освободит захваченный ресурс и до этого момента оба они не могут продолжить свою работу. Эта ситуация называется «смертельный замок[3]», в англоязычной литературе она обозначается словом "Deadlock". Во избежание её программист должен внимательно следить за доступом к совместно используемым ресурсам. Хорошим правилом, позволяющим избежать вышеописанной ситуации, является захват не более одного семафора взаимоисключения одновременно. В любом случае, залог успеха тут базируется на внимательности и дисциплине разработчика программы.

Для реализации бинарных семафоров этого типа в ***scmRTOS*** определён класс `OS::TMutex`, см. **«Листинг 5.6 OS::TMutex».**

[1] Общее правило: все процессы, работающие с общим ресурсом, должны вести себя так, то есть производить обращение через семафор.

[2] Функция, которая использует в процессе своей работы объекты с нелокальным классом памяти, поэтому для предотвращения нарушения целостности работы программы такую функцию нельзя вызывать, если уже запущен экземпляр этой же функции.

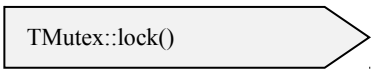
[3] Иногда встречается перевод «смертельные объятия».

```
{1}  class TMutex : public TService
{2}  {
{3}  public:
{4}      INLINE TMutex() : ProcessMap(0), ValueTag(0) { }
{5}             void lock();
{6}             void unlock();
{7}             void unlock_isr();
{8}
{9}      INLINE bool try_lock()        { TCritSect cs; if(ValueTag) return false;
{10}                                                   else lock(); return true; }
{11}     INLINE bool is_locked() const { TCritSect cs; return ValueTag != 0; }
{12}
{13} private:
{14}     volatile TProcessMap ProcessMap;
{15}     volatile TProcessMap ValueTag;
{16}
{17} };
```

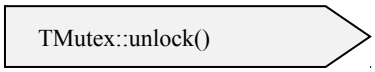
Листинг 5.6 OS::TMutex

Очевидно, что перед тем, как использовать, семафор нужно создать. В силу специфики применения семафор должен иметь класс памяти и область видимости такую же, как и обслуживаемый им ресурс, т.е. должен быть статическим объектом с глобальной областью видимости[1].

Как видно из интерфейса класса, с объектами этого типа можно производить следующие действия:



1. захватывать. Функция **`lock()`** выполняет эту задачу. Если до этого семафор не был захвачен другим процессом, то внутреннее значение будет переведено в состояние, соответствующее захваченному, и поток управления вернётся обратно в вызываемую функцию. Если семафор был захвачен, то процесс будет переведён в ожидание, пока семафор не будет освобождён, а управление отдано ядру;



2. освобождать. Это выполняет функция **`unlock()`**. Она переводит внутреннее значение в состояние, соответствующее освобождённому семафору, и проверяет, не ждёт ли какой-либо другой процесс этого семафора. Если ждёт, то управление будет отдано ядру, которое произведёт перепланирование процессов так, что, если ожидающий процесс был более приоритетным, он тут же получит управление. Если семафора ожидали несколько процессов, то управление получит самый приоритетный из них. Снять блокировку семафора может только тот процесс, который его заблокировал, – т.е. если выполнить описываемую функцию в процессе, который не заблокировал объект-мутекс, то никакого эффекта это не произведёт, объект останется в том же состоянии;

[1] Хотя ничего не мешает размещать Mutex вне области видимости кода процесса и использовать указатель или ссылку как напрямую, так и через классы-«обёртки», позволяющие автоматизировать процесс разблокировки ресурса через автоматический вызов деструктора класса-«обёртки».

TMutex::unlock_isr()

3. иногда возникает ситуация, когда мутекс блокируется в процессе, но работа с соответствующим защищаемым ресурсом производится в обработчике прерываний (при этом и запуск этой работы производится в процессе одновременно с захватом мутекса). В этом случае удобно делать разблокировку прямо в обработчике прерываний по окончанию работы с ресурсом. Для этого в состав **`TMutex`** введена функция **`unlock_isr()`** разблокировки семафора непосредственно в прерывании;

Tmutex::try_lock()

4. «мягко» захватывать. Функция **`try_lock()`**. Разница с обычным захватом семафора состоит в том, что захват будет иметь место только в случае, если семафор свободен. Например, требуется поработать с ресурсом, но кроме этого у процесса ещё есть много другой работы. Пытаясь захватить «жёстко», можно встать на ожидание и стоять там, пока семафор не освободится, хотя можно это время потратить на другую работу, если таковая имеется, а работу с совместно используемым ресурсом производить только тогда, когда доступ к нему не заблокирован. Такой подход может быть актуален в высокоприоритетном процессе: если семафор захвачен низкоприоритетным процессом, то при наличии работы в высокоприоритетном разумно не отдавать управление низкоприоритетному. И только когда уже делать будет больше нечего, имеет смысл пытаться захватить семафор обычным способом – с отдачей управления (ведь низкоприоритетный процесс тоже должен рано или поздно получить управление для того, чтобы закончить свои дела и освободить семафор). Учитывая вышеизложенное, пользоваться этой функцией нужно с осторожностью, т.к. это может привести к тому, что низкоприоритетный процесс вообще не получит управления из-за того, что его (управление) не отдаёт высокоприоритетный;

TMutex::is_locked()

5. проверять. Для этого предназначена функция **`is_locked()`**. Функция просто проверяет значение и возвращает **`true`**, если семафор захвачен, и **`false`** в противном случае. Иногда бывает удобно использовать семафор в качестве флага состояния, когда один процесс выставляет этот флаг (захватив семафор), а другие процессы проверяют его и выполняют действия в соответствии с состоянием того процесса.

Пример использования – см. **«Листинг 5.7 Пример использования OS::TMutex»**

```
{1} OS::TMutex Mutex;
{2} byte buf[16];
{3} ...
{4} template<> void TSlon::exec()
{5} {
{6}     for(;;)
{7}     {
{8}         ...                             // some code
{9}                                         //
{10}        Mutex.lock();                   // resource access lock
{11}        for(byte i = 0; i < 16; i++)    //
{12}        {                               //
{13}            ...                         // do something with buf
{14}        }                               //
{15}        Mutex.unlock();                 // resource access unlock
{16}                                        //
{17}        ...                             // some code
{18}    }
{19}}
```

Листинг 5.7 Пример использования OS::TMutex

Для удобства использования семафора взаимоисключения можно применить уже не раз упоминавшуюся технику классов-«обёрток», которая в данном случае реализуется с помощью класса **`TMutexLocker`**, см. «**Листинг 5.8 Класс-«обёртка» OS::TMutexLocker**», входящего в дистрибутив ОС.

```
{1}class TMutexLocker
{2}{
{3}public:
{4}    TMutexLocker(OS::TMutex & mutex) : Mutex(mutex) { Mutex.lock(); }
{5}    ~TMutexLocker() { Mutex.unlock(); }
{6}private:
{7}    TMutex & Mutex;
{8}};
```

Листинг 5.8 Класс-«обёртка» OS::TMutexLocker

Методология использования объектов этого класса ничем не отличается от методологии использования других классов-«обёрток» – **`TCritSect`**, **`TISRW`**.

* * *

Следует сказать несколько слов о таком связанном с семафорами взаимоисключения механизме, как инверсия приоритетов.

Сама идея инверсии приоритетов возникает из следующей ситуации. Например, в системе есть несколько процессов, и процессы с приоритетами N[1] и N+n, где n>1, используют один и тот же ресурс, разделяя работу посредством семафора взаимоисключения. В какой-то момент процесс с приоритетом N+n за-

[1] Предполагается, что приоритетность выполнения процессов связана с номерами приоритетов в обратной зависимости – т.е. процесс с приоритетом 0 является самым приоритетным, по мере возрастания номеров приоритетов приоритетность процессов уменьшается.

хватил семафор и производит работу с общим ресурсом. Во время этого происходит событие, активизирующее процесс с приоритетом N, который пытается получить доступ к общему ресурсу, и, в попытке захватить семафор, переходит в состояние ожидания, в котором он будет находиться до тех пор, пока процесс с приоритетом N+n не разблокирует семафор. Задержка более приоритетного процесса в этой ситуации является вынужденной, т.к. невозможно отобрать управление у процесса с приоритетом N+n, не нарушив логики его работы и целостности доступа к общему ресурсу. Зная об этом, разработчик, как правило, старается минимизировать время работы с ресурсом, чтобы низкоприоритетный процесс не блокировал работу высокоприоритетного.

Но в этой ситуации существует неприятность, состоящая в том, что если в вышеописанный момент вдруг активизируется процесс с приоритетом, например, N+1, то он вытеснит процесс с приоритетом N+n (т.к. n>1) и тем самым внесёт дополнительную задержку в ожидание более приоритетного процесса с приоритетом N. Ситуация усугубляется тем, что разработчик программы обычно не связывает работу процесса с приоритетом N+1 и манипуляции процессами с приоритетами N и N+n над общим ресурсом, поэтому может не ставить задачу оптимизации работы процесса с приоритетом N+1 в связи с этим, что может вообще блокировать работу процесса с приоритетом N на непредсказуемое время. Это является весьма нежелательной ситуацией.

Чтобы избежать этого, применяется приём под названием «инверсия приоритетов». Суть его состоит в том, что если при попытке захвата семафора взаимоисключения высокоприоритетным процессом семафор уже был захвачен низкоприоритетным процессом, то производится временный обмен приоритетами до разблокировки семафора. При этом по факту получается, что низкоприоритетный процесс работает с приоритетом процесса, который пытался захватить семафор. В этом случае ситуация, описанная выше, когда низкоприоритетный процесс блокирует работу высокоприоритетного, оказывается невозможной.

При всей стройности и элегантности метода инверсии приоритетов он не лишён недостатков. Главный – его техническая реализация порождает накладные расходы, которые сравнимы или превышают затраты на реализацию функциональности собственно семафора взаимоисключения, и может получиться так, что переключение приоритетов процессов и всё связанное с этим – нужно учесть все элементы ОС, связанные с приоритетами процессов, задействованных при инверсии приоритетов, – замедлит работу системы до неприемлемого уровня.

В связи с этим механизм инверсии приоритетов в текущей версии ***scmRTOS*** не используется, а для решения вышеописанной проблемы с блокировкой работы высокоприоритетного процесса низкоприоритетным предлагается механизм делегирования заданий, подробно рассмотренный в «Приложение А

Примеры использования, А.1 Очередь заданий», который представляет собой унифицированный метод перераспределения выполнения связанного по контексту программного кода в процессах с разными приоритетами.

5.5. OS::message

`OS::message` представляет собой C++ шаблон для создания объектов, реализующих обмен между процессами путём передачи структурированных данных. **`OS::message`** похож на **`OS::TEventFlag`** и отличается главным образом тем, что кроме самого флага содержит ещё и объект произвольного типа, составляющий собственно тело сообщения.

Определение шаблона – см. «**Листинг 5.9 OS::message**».

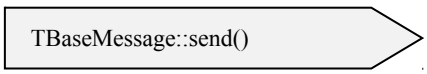
Как видно из листинга, шаблон сообщения построен на основе класса **`TBaseMessage`**. Это сделано из соображений эффективности, чтобы общий код не дублировался в экземплярах шаблона – общий для всех сообщений код вынесен на уровень базового класса[1].

[1] Этот же приём применён и при построении шаблона процесса: связка **`class TBaseProcess – template process<>`**.

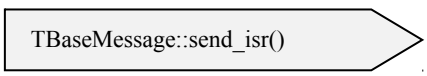
```
{1}  class TBaseMessage : public TService
{2}  {
{3}  public:
{4}      INLINE TBaseMessage() : ProcessMap(0), NonEmpty(false) { }
{5}
{6}      bool wait  (timeout_t timeout = 0);
{7}      INLINE void send();
{8}      INLINE void send_isr();
{9}      INLINE bool is_non_empty() const { TCritSect cs; return NonEmpty;  }
{10}     INLINE void reset       ()       { TCritSect cs; NonEmpty = false; }
{11}
{12} private:
{13}     volatile TProcessMap ProcessMap;
{14}     volatile bool NonEmpty;
{15} };
{16}
{17} template<typename T>
{18} class message : public TBaseMessage
{19} {
{20} public:
{21}     INLINE message() : TBaseMessage()   { }
{22}     INLINE const T& operator= (const T& msg)
{23}     {
{24}         TCritSect cs;
{25}         *(const_cast<T*>(&Msg)) = msg; return const_cast<const T&>(Msg);
{26}     }
{27}     INLINE operator T() const
{28}     {
{29}         TCritSect cs;
{30}         return const_cast<const T&>(Msg);
{31}     }
{32}     INLINE void out(T& msg) { TCritSect cs; msg = const_cast<T&>(Msg); }
{33}
{34} private:
{35}     volatile T Msg;
{36} };
```

Листинг 5.9 `OS::message`

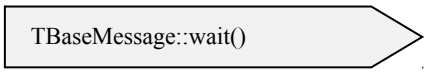
Реализация шаблона достаточна проста. Над объектами, созданными по шаблону, можно производить следующие действия:



1. посылать сообщение[1]. Функция **`void send()`** выполняет эту операцию, которая сводится к переводу процессов, ожидающих сообщение, в состояние готовых к выполнению и вызову планировщика;



2. вариант вышеописанной функции, оптимизированный для использования в прерываниях. Функция является встраиваемой и использует специальную облегчённую встраиваемую версию планировщика. Этот вариант нельзя использовать вне кода обработчика прерываний;



3. ждать сообщение[2]. Для этого предназначена функция **`void wait (timeout_t timeout = 0)`**, которая проверяет, не пустое ли сообщение и если не пустое, то возвращает **`true`**, если пустое,

[1] Аналог функции **`OS::TEventFlag::signal()`**.
[2] Аналог функции **`OS::TEventFlag::wait()`**.

то переводит текущий процесс из состояния готовых к выполнению в состояние ожидания этого сообщения. Если при вызове не было указано аргумента либо аргумент был равен 0, то ожидание будет продолжаться до тех пор, пока какой-нибудь процесс не пошлёт сообщение или текущий процесс не будет «разбужен» с помощью функции **`TBaseProcess::force_wake_up()`**[1]. Если в качестве аргумента было указано целое число в диапазоне от 1 до 2^n-1 (где n – разрядность типа **`timeout_t`**), которое является значением величины таймаута, выраженное в тиках системного таймера, то ожидание сообщения будет происходить с таймаутом, т.е. процесс будет «разбужен» в любом случае. Если это произойдёт до истечения таймаута, что означает приход сообщения до того, как таймаут истёк, то функция вернёт **`true`**. В противном случае, т.е. если таймаут истечёт до того, как сообщение будет послано, функция вернёт **`false`**;

TBaseMessage::is_non_empty()

4. проверить сообщение. Функция **`bool is_non_empty()`** возвращает **`true`**, если сообщение было послано, и в **`false`** противном случае;

TBaseMessage::reset()

5. сбросить сообщение. Функция **`void reset()`** сбрасывает сообщение, т.е. переводит сообщение в состояние empty. При этом тело сообщения остаётся без изменений;

message::operator=()

6. записать в тело объекта-сообщения содержимое собственно сообщения. Штатный способ использования **`OS::message`** – это запись тела сообщения и посылка сообщения с помощью функции **`TBaseMessage::send()`** – см. «**Листинг 5.10 Использование OS::message**»;

message::operator T()

7. возвращает константную ссылку на тело сообщения. Пользоваться этим средством следует с осторожностью, отдавая себе отчёт в том, что во время доступа к телу сообщения по ссылке оно может быть изменено в другом процессе (или обработчике прерывания). Поэтому рекомендуется для чтения тела сообщения использовать функцию **`message::out()`**;

message::out()

8. предназначена для чтения тела сообщения. Для достижения эффективности, чтобы не возникало лишнего копирования тела сообщения, в функцию передаётся ссылка внешний объект-тело сообщения, в который внутри функции message::out() копируется содержимое сообщения.

[1] В последнем случае нужно проявлять крайнюю осторожность.

```
{1}  struct TMamont { ... }           //  data type for sending by message
{2}
{3}  OS::message<TMamont> MamontMsg;  // OS::message object
{4}
{5}  template<> void TProc1::exec()
{6}  {
{7}      for(;;)
{8}      {
{9}          TMamont Mamont;
{10}         MamontMsg.wait();      // wait for message
{11}         MamontMsg.out(Mamont); // read message contents to the external object
{12}         ...                    // usage of the contents of Mamont
{13}     }
{14} }
{15}
{16} template<> void TProc2::exec()
{17} {
{18}     for(;;)
{19}     {
{20}         ...
{21}         TMamont m;         // create message content
{22}
{23}         m...  =            // message body filling
{24}         MamontMsg = m;     // put the content to the OS::message object
{25}         MamontMsg.send();  // send the message
{26}         ...
{27}     }
{28} }
```

Листинг 5.10 Использование `OS::message`

5.6. OS::channel

`OS::channel` представляет собой C++ шаблон для создания объектов, реализующих кольцевые буфера[1] для безопасной с точки зрения вытесняющей ОС передачи данных произвольного типа. **`OS::channel`** также, как и любое другое средство межпроцессного взаимодействия, решает задачи синхронизации. Тип конкретного буфера задаётся на этапе инстанцирования[2] шаблона в пользовательском коде. Шаблон канала **`OS::channel`** основан на шаблоне кольцевого буфера **`usr::ring_buffer<class T, uint16_t size, class S = uint8_t>`**, определённого в библиотеке, входящей в дистрибутив поставки ***scmRTOS*** .

[1] Функционально это FIFO, т.е. объект-очередь для передачи данных по схеме First Input – First Output.
[2] Создания экземпляра.

Нужно заметить, что в прежних версиях ***scmRTOS*** существовал ещё один тип сервиса межпроцессного взаимодействия — **OS::TChannel**, который позволял организовать межпроцессную очередь из объектов типа **uint8_t**. Указанный тип канала является устаревшим и, начиная с версии 5, удалён из состава ОС. Вместо него следует использовать шаблон **OS::channel**.

Использование каналов на основе **OS::channel** по сравнению с **OS::TChannel** имеет следующие преимущества:

- данные канала могут иметь произвольный тип;
- безопасность – компилятор производит полный контроль типов при инстанцировании шаблона;
- при шаблонной реализации нет необходимости вручную выделять память под буфер;
- функциональность расширена – можно записывать данные не только в конец буфера, но и в начало, равно как и читать не только из начала буфера, но и из конца;
- при чтении данных можно указать величину таймаута, т.е. ждать данные не безусловно, а с ограничением по времени, что иногда оказывается очень полезным;
- т.к. элементами канала являются объекты, а не просто байты, то проблема, возникающая с **TChannel** при чередовании записи/чтения, с каналом на основе **OS::channel<>** не является фатальной, что позволяет, например, нескольким процессам одновременно записывать данные в канал – это применяется для организации очередей заданий, см. «**Приложение А Примеры использования**».

Новые возможности каналов дают эффективное средство для построения очередей сообщений. Причём в отличие от опасного, не наглядного и не гибкого способа организации очередей сообщений на основе указателя **void*,** очередь **OS::channel** предоставляет:

- безопасность на основе статического контроля типов как при создании очереди-канала, так и при записи в неё данных и чтении их оттуда;
- простоту использования – не нужно выполнять ручное преобразование типов, сопряжённое с необходимостью держать в голове лишнюю информацию о реальных типах данных, передаваемых через канал с целью правильного их использования;
- значительно большую гибкость использования - объектами очереди могут быть любые типы, а не только указатели.

По поводу последнего пункта следует сказать несколько слов: недостаток указателей **void*** в качестве основы для передачи сообщений состоит, в частности, в том, что пользователь должен где-то выделить память под сами сообщения. Это дополнительная работа, и целевой объект получается распределённым – очередь в одном месте, а собственно содержимое элементов очереди в другом. Главными достоинствами механизма сообщений на указателях является высокая

эффективность работы при больших размерах тел сообщений и возможность передачи разноформатных сообщений. Но если, например, сообщения небольшого размера – в пределах нескольких байт – и все имеют одинаковый формат, то нет никакой необходимости в указателях, гораздо проще создать очередь из требуемого количества таких сообщений, и все. При этом, как уже говорилось, не нужно выделять память под сами тела сообщений – поскольку сообщения целиком помещаются в очередь-канал, память под них в этом случае будет автоматически выделена непосредственно при создании канала компилятором.

Что касается сообщений на основе указателей, то и тут существует значительно более безопасное, удобное и гибкое решение на основе механизмов C++. Об этом будет сказано ниже – см. «**Приложение А Примеры использования**».

Определение шаблона канала – см. «**Листинг 5.11 Определение шаблона OS::channel**».

```
{1}   template<typename T, uint16_t Size, typename S = uint8_t>
{2}   class channel : public TService
{3}   {
{4}   public:
{5}       INLINE channel() : ProducersProcessMap(0)
{6}                        , ConsumersProcessMap(0)
{7}                        , pool()
{8}       {
{9}       }
{10}
{11}      //-------------------------------------------------------------------
{12}      //
{13}      //    Data transfer functions
{14}      //
{15}      void write(const T* data, const S cnt);
{16}      bool read (T* const data, const S cnt, timeout_t timeout = 0);
{17}
{18}      void push      (const T& item);
{19}      void push_front(const T& item);
{20}
{21}      bool pop     (T& item, timeout_t timeout = 0);
{22}      bool pop_back(T& item, timeout_t timeout = 0);
{23}
{24}      //-------------------------------------------------------------------
{25}      //
{26}      //    Service functions
{27}      //
{28}      INLINE S get_count()     const;
{29}      INLINE S get_free_size() const;
{30}      void flush();
{31}
{32}  private:
{33}      volatile TProcessMap ProducersProcessMap;
{34}      volatile TProcessMap ConsumersProcessMap;
{35}      usr::ring_buffer<T, Size, S> pool;
{36}  };
```

Листинг 5.11 Определение шаблона `OS::channel`

Использование **OS::channel** очень простое: сначала нужно определить тип объектов, которые будут передаваться через канал, затем определить тип канала либо создать объект-канал. Например, пусть данные, передаваемые через канал, представляют собой структуру:

```
struct TData
{
    int   A;
    char* p;
};
```

Теперь можно создать объект-канал путём инстанцирования шаблона **OS::channel**:

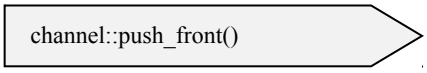
```
OS::channel<TData, 8> DataQueue;
```

Этот код объявляет объект-канал **DataQueue** для передачи объектов типа **TData**, ёмкость канала – 8 объектов. Теперь можно использовать канал для передачи.

Как уже говорилось, помимо возможности работать с объектами произвольного типа **OS::channel** предоставляет расширенную функциональность по сравнению с **OS::TChannel**, а именно – возможность записывать данные не только в конец очереди, но и в начало; читать данные не только из начала очереди, но и из конца. При чтении также имеется возможность указать величину таймаута. Обо всем об этом - чуть ниже.

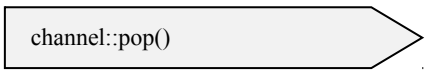
Для действий над объектом-каналом предоставляется следующий интерфейс:

1. записать элемент в конец очереди[1]. Функция **void push(const T& item)** записывает один элемент в канал, если там было для этого место. Если места не было, то процесс переходит в состояние ожидания до тех пор, пока в канале не появится место. Когда место появится, элемент будет записан. После этого делается проверка, не ждёт ли какой-нибудь процесс данных из канала;

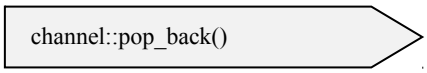


2. записать элемент в начало очереди. Функция **void push_front(const T& item)** записывает один элемент в канал, если там было для этого место. Если места не было, то процесс переходит в состояние ожидания до тех пор, пока в канале не появится место. Когда место появится, элемент будет записан. После этого делается проверка, не ждёт ли какой-нибудь процесс данных из канала;

[1] Имеется в виду очередь канала. Т.к. функционально канал представляет собой FIFO, то конец очереди соответствует входу FIFO, начало канала – выходу FIFO.

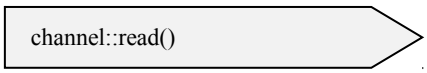


3. извлечь элемент из начала очереди. Функция **`bool pop(T& item, timeout_t timeout = 0)`** извлекает один элемент из канала, если канал не был пуст. Если канал был пуст, то процесс переходит в состояние ожидания до тех пор, пока в нем не появятся данные, либо до истечения таймаута, если таймаут был указан[1]. В случае вызова с таймаутом, если данные поступили до истечения таймаута, функция возвращает **`true`**, в противном случае **`false`**. Если вызов был без таймаута, функция всегда возвращает **`true`**, за исключением пробуждения по **`wake_up()`**, **`force_wake_up()`**. Когда данные появляются, элемент извлекается и делается проверка, не ждёт ли какой-нибудь процесс с целью записать в канал. Следует обратить внимание на тот факт, что при вызове этой функции данные, извлечённые из канала, передаются не путём копирования при возврате функции, а через передачу объекта по ссылке. Это обусловлено тем, что значение возврата занято для передачи результата таймаута;

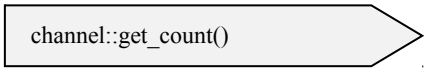


4. извлечь элемент из конца очереди. Функция **`bool pop_back (T& item, timeout_t timeout)`** извлекает один элемент из канала, если канал не был пуст. Вся функциональность ровно такая же, как и в случае с **`os::channel::pop()`** за исключением того, что элементы данных считываются из конца канала;

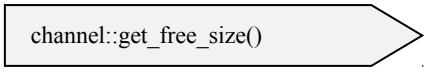
5. записать в конец очереди несколько элементов из памяти по адресу. Функция **`void write(const T* data, const S cnt)`** реализует эту операцию. Фактически это то же самое, что и записать один элемент в конец очереди (**`push`**), только ожидание продолжается до тех пор, пока в канале не появится достаточно места. Остальная логика работы такая же;



6. извлечь из канала несколько элементов и поместить их в память по адресу. Эту операцию выполняет функция **`bool read (T* const data, const S cnt, timeout_t timeout = 0)`**. То же самое, что и извлечь элемент из начала очереди, только ожидание продолжается до тех пор, пока в канале не окажется нужное количество байт или не сработает таймаут, если он был задействован.



7. получить величину количества элементов в канале. Функция **`S get_count() const`** является встраиваемой, поэтому эффективность её работы максимально высока;



8. получить величину количества элементов, которые можно записать в канал, т.е. под них есть свободное место. Функция

[1] Т.е. вызов был с передачей вторым аргументом целого числа в диапазоне $1..2^n-1$ (где n – разрядность типа **`timeout_t`**), которое и задает величину таймаута в тиках системного таймера.

S get_free_size() const является встраиваемой, поэтому эффективность её работы максимально высока;

9. произвести очистку канала. Функция **void flush()** очищает буфер путём вызова **usr::ring_buffer<>::flush()** и проверяет, не ждёт ли какой-либо процесс, пока в канале не появится свободное место. После вызова этой функции в канале нет ни одного элемента.

Простой пример использования – см. «**Листинг 5.12 Пример использования очереди на основе канала.**»

```
{1}  //---------------------------------------------------------------------
{2}  struct TCmd
{3}  {
{4}      enum TCmdName { cmdSetCoeff1, cmdSetCoeff2, cmdCheck } CmdName;
{5}      int Value;
{6}  };
{7}
{8}  OS::channel<TCmd, 10> CmdQueue; // Queue for Commands with 10 items depth
{9}  //---------------------------------------------------------------------
{10} template<> void TProc1::exec()
{11} {
{12}     ...
{13}     TCmd cmd = { cmdSetCoeff2, 12 };
{14}     CmdQueue.push(cmd);
{15}     ...
{16} }
{17} //---------------------------------------------------------------------
{18} template<> void TProc2::exec()
{19} {
{20}     ...
{21}     TCmd cmd;
{22}     if( CmdQueue.pop(cmd, 10) ) // wait for data, timeout 10 system ticks
{23}     {
{24}         ... // data incoming, do something
{25}     }
{26}     else
{27}     {
{28}         ... // timeout expires, do something else
{29}     }
{30}     ...
{31} }
{32} //---------------------------------------------------------------------
```

Листинг 5.12 Пример использования очереди на основе канала.

Как видно, использование достаточно простое и прозрачное. В одном процессе (**Proc1**) создаётся сообщение-команда **cmd** {13}, инициализируется требуемыми значениями {13} и записывается в очередь-канал {14}. В другом процессе (**Proc2**) происходит ожидание данных из очереди {22}, при приходе данных выполняется соответствующий код {23}-{25}, при истечении таймаута выполняется другой код {27}- {29}.

5.7. Заключительные замечания

Существует некий инвариант между различными средствами межпроцессного взаимодействия. Т.е. с помощью одних средств (или, что чаще, их совокупности) можно выполнить ту же задачу, что и с помощью других. Например, вместо использования канала можно создать статический массив и обмениваться данными через него, используя семафоры взаимоисключения для предотвращения совместного доступа и флаги события для уведомления ожидающего процесса, что данные для него готовы. В ряде случаев такая реализация может оказаться более эффективной, хотя и менее удобной.

Можно использовать сообщения для синхронизации по событиям вместо флагов событий – такой поход имеет смысл в случае, если вместе с флагом нужно ещё передать какую-то информацию. Собственно, **`OS::message`** именно для этого и предназначен.

Словом, разнообразие использования велико, и какой вариант подходит наилучшим образом в той или иной ситуации, определяется, в первую очередь, самой ситуацией.

СОВЕТ. Необходимо понимать и помнить, что любое средство межпроцессного взаимодействия при выполнении своих функций делает это в критической секции, т.е. при запрещённых прерываниях. Исходя из этого, не следует злоупотреблять средствами межпроцессного взаимодействия там, где можно обойтись без них. Например, при обращении к статической переменной встроенного типа использовать семафор взаимоисключения не является хорошей идеей по сравнению с простым использованием критической секции, т.к. семафор при захвате и освобождении тоже использует критические секции, пребывание в которых дольше, чем при простом обращении к переменной.

При использовании сервисов в прерываниях есть определённые особенности. Например, очевидно, что использовать **`TMutex::lock()`** внутри обработчика прерывания является достаточно плохой идеей, т.к., во-первых, семафоры взаимоисключения предназначены для разделения доступа к ресурсам на уровне процессов, а не на уровне прерываний, и, во-вторых, ожидать освобождения ресурса, если он был занят, внутри обработчика прерывания все равно не удастся и

это приведёт только к тому, что процесс, прерванный данным прерыванием, просто будет переведён в состояние ожидания в неподходящей и непредсказуемой точке. Фактически процесс будет переведён в неактивное состояние, из которого его вывести можно будет только с помощью функции **`TBaseProcess::force_wake_up()`**. В любом случае ничего хорошего из этого не получится.

Аналогичная в некотором роде ситуация может получиться при использовании объектов-каналов в обработчике прерываний. Ждать данных из канала внутри ISR не получится, и последствия будут аналогичны вышеописанным, а записывать данные в канал тоже не вполне безопасно. Если, к примеру, при записи в канал в нем не окажется достаточно места, то поведение программы окажется далеко не таким, как ожидает пользователь.

И только два типа из всех средств межпроцессного взаимодействия позволяют безопасное и полезное их использование. Это **`OS::TEventFlag`** и **`OS::message`**. Естественно, безопасность и полезность относятся не к любому их использованию, а только лишь к вызову функций **`TEventFlag::signal()`**, **`TEventFlag::signal_isr()`**, **`message<T>::send()`**, **`message<T>::send_isr()`**, и обуславливаются тем обстоятельством, что внутри этих функций ни при каких условиях не производится попыток встать на ожидание и никаких вопросов по поводу достаточного количества памяти (как в случае с каналами) нет. Кроме того, указанные функции сами по себе заметно компактнее и быстрее, чем, к примеру, любая из функций **`push()`** объектов-каналов.

Исходя из вышесказанного, внутри ISR рекомендуется по возможности использовать только **`TEventFlag::signal_isr()`** и **`message<T>::send_isr()`**. И не рекомендуется использовать все остальные средства межпроцессного взаимодействия – они предназначены для использования на уровне процессов. Если всё-таки возникает желание (необходимость) использовать внутри обработчика прерываний, например, объект канала, то следует проявить внимательность, осмотрительность и аккуратность, и, конечно, все действия выполнять на основе глубокого понимания, ни в коем случае не формально.

Ну и, конечно же, в случае, если имеющийся набор средств межпроцессного взаимодействия по каким-то причинам не удовлетворяет потребностей того или иного проекта, всегда есть возможность спроектировать сервисный класс под собственные нужды, опираясь на предоставленную базу в виде **`TService`**. При этом штатный набор сервисов можно использовать в качестве примеров для проектирования.

Глава 6 Отладка

6.1. Измерение потребления стека процессов

Существует вопрос, дать однозначный ответ на который в большинстве случаев оказывается довольно сложно: какой необходим объем оперативной памяти, выделенной под стек, чтобы её хватило для всех нужд программы и обеспечило правильную и безопасную её работу?

В случае программ, работающих без использования ОС, когда весь код выполняется с использованием одного-единственного стека, существуют средства оценки объёма памяти, выделенной под стек, необходимого для обеспечения правильной работы. Они основаны на построении дерева вызовов функций и известной информации о том, какой объём стека потребляет каждая функция. Эту работу может выполнить сам компилятор, поместив результаты в файл листинга после компиляции исходного файла.

Для получения окончательного результата остаётся к результату самой потребляющей функции прибавить потребности в стеке самого потребляющего обработчика прерываний.

К сожалению, описанный метод даёт только приблизительную оценку, т.к. компилятор не в состоянии точно построить дерево вызовов функций, возникающих на практике – в частности, косвенные вызовы функций, к которым относятся вызовы функций по указателю или вызовы виртуальных функций, не дают возможности учесть их вызов, т.к. на этапе компиляции ничего не известно о том, какая именно функция будет вызвана. В частных случаях, когда программист знает, какие функции могут быть вызваны косвенно, вычисление потребления стека может быть произведено вручную. Но этот способ неудобен – ведь это нужно делать при каждом сколько-нибудь значительном изменении программы, и он чреват ошибками.

В общем случае компилятор не обязан предоставлять такую информацию, а сторонние инструменты, выполняющие эту работу, также не способны обойти вышеописанные трудности, по какой причине не снискали себе популярности.

Всё это предъявляет разработчику программы выбор, какой указать размер стека. С одной стороны есть желание сэкономить ОЗУ, с другой - необходимо указать достаточный размер, чтобы не получить ошибки работы программы на этапе её выполнения, тем более, что ошибки, возникающие из-за неправильной работы с памятью, являются, как правило, весьма трудноуловимыми, т.к. их проявление всегда индивидуально и слабопредсказуемо. Поэтому на практике приходится указывать размер стека с некоторым запасом, что позволяет учесть ошибки в недооценке его размера.

В случае использования операционной системы ситуация усугубляется в силу того, что стек в программе не один, а их количество равно количеству процессов, указанному при конфигурации ОС, что порождает больший дефицит ОЗУ и вынуждает разработчика ещё больше экономить память и указывать размеры стеков с меньшим запасом.

Для решения вышеописанных проблем можно применить способ практического измерения объёмов потребления стека процессами. Эта возможность, как и другие возможности по отладке работы системы, включается в ***scmRTOS*** при конфигурации с помощью указания значения макроса **`scmRTOS_DEGUG_ENABLE`** равным **`1`**.

Суть метода состоит в том, чтобы на этапе подготовки стекового кадра заполнить пространство стека каким-либо заранее известным значением (паттерном), а при проверке результата просканировать область памяти, выделенную под стек процесса, начиная с конца, противоположного вершине стека (TOS), и найти место, где заканчивается заполнение паттерном. Количество ячеек, в которых паттерн не был перезаписан в процессе работы программы, показывает реальный запас по размеру стека процесса.

Заполнение стека паттерном производится в платформеннозависимой функции **`init_stack_frame()`** при разрешённом режиме отладки. Получить информацию о запасе по стеку[1] процесса можно в любой момент, вызвав для объекта процесса функцию **`stack_slack()`**, возвращающую целое число, указы-

[1] Т.е. сколько осталось в стеке ячеек, ни разу не использованных в программе.

вающее искомую величину. Исходя из этого, разработчик программы может откорректировать размеры стеков и тем самым исключить ошибки, возникающие из-за переполнения стеков.

6.2. Работа с зависшими процессами

В процессе разработки нередко возникает характерная ситуация, когда по каким-то не ясным причинам программа работает неверно, и по косвенным признакам легко определить, что не работает тот или иной процесс. Обычно это случается, если процесс находится в ожидании какого-то сервиса, и, чтобы найти причину зависания, нужно определить, какой именно сервис стал причиной ожидания.

Для определения сервиса, которого ждёт процесс, в режиме отладки ***scmRTOS*** включаются специальные средства – в частности, при переходе процесса в режим ожидания запоминается адрес сервиса, который вызывал переход к ожиданию. При необходимости пользователь может вызвать функцию процесса **`waiting_for()`**, которая возвращает указатель на сервис, а зная этот адрес всегда можно по файлу отчёта линкера определить имя объекта сервиса.

6.3. Профилировка работы процессов

Иногда бывает очень полезно узнать распределение нагрузки на процессы в программе. Эта информация позволяет оценить правильность работы алгоритмов программы и выявить ряд трудноуловимых логических ошибок. Для получения информации о загрузке процессов существует ряд методов определения относительного времени их активной работы, это называется профилировкой работы процессов.

В ***scmRTOS*** профилировка реализована в виде расширения и не входит в основной состав самой ОС. Профилировщик представляет собой класс-расширение, реализующий базовые функции по сбору информации об относительном времени работы и её обработку. Сбор этой информации может быть реализован двумя способами, имеющими свои достоинства и недостатки:

- статистический;
- измерительный.

6.3.1. Статистический метод

Статистический метод не требует для своей работы никаких дополнительных ресурсов, кроме тех, которые предоставляет операционная система. Принцип его работы основан на сэмплировании через равные интервалы времени переменной ядра **`CurProcPriority`**, которая указывает, какой процесс является активным в данный момент времени. Сэмплирование удобно организовать, например, в обработчике системного таймера – чем больше процессорного времени занимает процесс, тем чаще он будет активным при сэмплировании. Недостатком такого метода является низкая точность, позволяющая получить лишь качественную картину происходящего.

6.3.2. Измерительный метод

Этот метод лишён главного недостатка профилировки статистическим методом – низкой точности определения времени загрузки процессов. Принцип работы измерительного метода основан на измерении времени работы процессов (отсюда и название). Для этого пользователь должен предоставить средства для измерения времени работы – это может быть один из аппаратных таймеров МК или какие-либо другие средства – например, счётчик тактов процессора, если таковой имеется. Это цена за использование этого метода.

6.3.3. Использование

Для использования профилировщика в пользовательском проекте необходимо определить функцию измерения времени и подключить профилировщик к проекту. Подробнее об этом см. пример в приложении «**A.2 Разработка расширения: профилировщик работы процессов**».

6.4. Имена процессов

С целью повышения удобства отладки предусмотрена возможность задавать строковые имена процессам. Имя задаётся обычным для языка C++ способом — через аргумент конструктора:

```
TMainProc MainProc("Main Process");
```

Этот строковый аргумент может быть описан всегда, но задействование имени доступно только в отладочной конфигурации.

Для доступа из пользовательской программы в классе **TBaseProcess** определена функция:

```
const char *name();
```

Использование тривиально и ничем не отличается от работы с C-строками на языках C/C++. Пример вывода отладочной информации см. Листинг 6.1 Пример вывода отладочной информации.

```
{1}  //-----------------------------------------------------------------------
{2}  void TProcProfiler::get_results()
{3}  {
{4}      print("-----------------------------\n");
{5}      for(uint_fast8_t i = 0; i < OS::PROCESS_COUNT; ++i)
{6}      {
{7}      #if scmRTOS_DEBUG_ENABLE == 1
{8}          printf("#%d | CPU %5.2f | Slack %d | %s\n", i,
{9}                 Profiler.get_result(i)/100.0,
{10}                OS::get_proc(i)->stack_slack(),
{11}                OS::get_proc(i)->name() );
{12}     #endif
{13}     }
{14} }
{15} //-----------------------------------------------------------------------
```

Листинг 6.1 Пример вывода отладочной информации.

Приведённый код порождает следующий вывод:

```
-----------------------------
#0 | CPU 82.52 | Slack 164 | Idle
#1 | CPU  0.00 | Slack 178 | Background
#2 | CPU  0.07 | Slack 387 | GUI
#3 | CPU  0.23 | Slack 259 | Video
#4 | CPU  0.00 | Slack 148 | BiasReg
#5 | CPU 17.09 | Slack 165 | RefFrame
#6 | CPU  0.03 | Slack 204 | TempMon
#7 | CPU  0.00 | Slack 151 | Terminal
#8 | CPU  0.01 | Slack 129 | Test
#9 | CPU  0.01 | Slack 301 | IBoard
```

Глава 7 Порты

7.1. Общие замечания

Ввиду больших отличий как аппаратных архитектур, так и средств разработки под них, возникает необходимость в специальной адаптации кода ОС[1] под них. Результатом этой работы является платформеннозависимая часть, которая в совокупности с общей частью и составляет порт под ту или иную платформу. Процесс подготовки платформеннозависимой части называется портированием.

В настоящей главе будут рассмотрены, главным образом, платформеннозависимые части, их содержимое и особенности, также дана краткая инструкция по портированию ОС, т.е. что нужно сделать, чтобы создать порт.

Платформеннозависимая часть каждой целевой платформы содержится в отдельной директории и минимально содержит три файла:

- os_target.h – платформеннозависимые объявления и макросы.
- os_target_asm.ext[2] – низкоуровневый код, функции переключения контекста, старта ОС.
- os_target.cpp – определения функции инициализации стекового кадра процесса и функции обработчика прерывания от таймера, используемого в качестве системного.

Настройка кода ОС на целевую платформу осуществляется путём:

- определения специальных макросов препроцессора;
- директивами условной трансляции;
- определением типов, определяемых пользователем, реализация которых зависит от целевой платформы;
- заданием псевдонимов некоторых типов;
- определением функций, код которых вынесен на уровень порта.

Важной и «тонкой» частью кода порта является определение ассемблерных подпрограмм, осуществляющих старт системы, сохранение контекста преры-

[1] Это касается не только ОС, но и других многоплатформенных программ.
[2] Расширение ассемблерного файла для целевого процессора.

ваемого процесса, переключение указателей стека и восстановление контекста процесса, получившего управление, в том числе и обработчик программного прерывания, в теле которого производится переключение контекстов процессов. Чтобы реализовать этот код, от разработчика порта требуются глубокие знания целевой аппаратной архитектуры на низком уровне, а также умение использовать программный пакет (компилятор, ассемблер, линкер) для работы со «смешанными»[1] проектами.

Процесс портирования сводится, главным образом, к определению объектов портирования и написанию платформеннозависимого кода.

7.2. Объекты портирования

7.2.1. Макросы

Существует ряд платформеннозависимых макросов, которые должны быть определены. Если значение макроса в том или ином порте не требуется, то макрос должен быть определён пустым. Перечень макросов и их описания приведены ниже:

`INLINE`	Задаёт поведение функций при встраивании. Обычно состоит из платформеннозависимой директивы безусловного встраивания и ключевого слова **`inline.`**
`OS_PROCESS`	Квалифицирует исполняемую функцию процесса. Содержит платформеннозависимый атрибут, указывающий компилятору, что функция не имеет возврата, поэтому preserved[2] регистры процессора можно использовать без сохранения. Это экономит код и пространство в стеке.
`OS_INTERRUPT`	Содержит платформеннозависимое расширение, используемое для квалификации обработчиков прерываний на целевой платформе.
`DUMMY_INSTR()`	Макрос, определяющий пустую инструкцию целевого процессора (как правило, это инструкция «NOP»). Используется в цикле ожидания переключения контекстов в планировщике (в варианте с программным прерыванием переключения контекстов).

[1] Т.е. содержащими исходные файлы на разных языках программирования – в нашем случае C++ и ассемблер целевой аппаратной платформы.

[2] Те, значение которых перед использованием должно быть сохранено, а после использования восстановлено, чтобы вызывающая функция не получила искажения контекста при вызове другой функции.

INLINE_PROCESS_CTOR	Определяет поведение встраивания конструкторов процессов. Если нужно встраивание, то значение этого макроса должно **INLINE**, если встраивание не нужно, то значение макроса должно быть оставлено пустым.
SYS_TIMER_CRIT_SECT()	Используется в обработчике прерываний системного таймера и задаёт, будет ли использоваться в нём критическая секция, которая актуальна в случае, если целевой процессор имеет приоритетный многоуровневый контроллер прерываний, что может привести к тому, что обработчик прерываний от системного таймера может быть прерван в непредсказуемый момент другим, более высокоуровневым, обработчиком прерываний, который может производить доступ к тем же ресурсам ОС, что и обработчик прерываний от системного таймера.
CONTEXT_SWITCH_HOOK_CRIT_SECT	Определяет, будет хук переключателя контекстов выполняться в критической секции или нет. Очень важно, чтобы хук переключателя контекстов выполнялся целостно по отношению к манипуляциям с переменными ядра (**SchedProcPriority**, в частности), а это означает, что во время выполнения хука не должен вызываться планировщик. Вызов планировщика может произойти из обработчика прерываний в случае, если процессор имеет аппаратный приоритетный контроллер прерываний и программное прерывание переключения контекстов имеет более низкий приоритет по сравнению с другими прерываниями. В этом случае код хука переключателя контекстов должен выполняться в критической секции и значение макроса должно быть **TCritSect cs;**. Это очень важный момент, если его не соблюсти, то в процессе работы системы будут возникать трудноуловимые ошибки, поэтому при портировании тут нужно проявить внимательность и аккуратность.
SEPARATE_RETURN_STACK	Для платформ, имеющих отдельный стек возвратов, значение этого макроса должно быть равно 1. Для остальных платформ – 0.

7.2.2. Типы

stack_item_t	Псевдоним встроенного типа, задаёт тип элемента стека целевого процессора.
status_reg_t	Псевдоним встроенного типа, соответствующий разрядности статусного регистра целевого процессора.
TCritSect	Класс-«обёртка» для организации критической секции.
TPrioMaskTable	Класс, содержащий таблицу масок (тегов) приоритетов. Служит для повышения эффективности работы системы. Может отсутствовать на некоторых платформах, на таких, где есть аппаратные средства для вычисления тегов по значению приоритета, –

	например, аппаратный shifter.
TISRW	Класс-«обёртка» для обработчиков прерываний, в которых используются сервисы ОС.

7.2.3. Функции

get_prio_tag()	Преобразует номер приоритета в соответствующий тег. Функционально это сдвиг единицы в двоичном слове на количество позиций, равное номеру приоритета.
highest_priority()	Возвращает номер приоритета, соответствующего тегу наиболее приоритетного процесса в карте процессов, переданной функции в качестве аргумента.
disable_context_switch()	Запрещает переключение контекстов. В настоящее время реализуется путём запрещения прерываний.
enable_context_switch()	Разрешает переключение контекстов. В настоящее время реализуется через разрешение прерываний.
os_start()	Производит старт операционной системы. Сама функция реализована на ассемблере. Получает в качестве аргумента указатель стека самого приоритетного процесса и осуществляет передачу ему управления путём восстановления контекста из его стека.
os_context_switcher()	Функция, реализованная на ассемблере, производит переключение контекстов процессов в варианте с прямой передачей управления.
context_switcher_isr	Обработчик прерываний переключения контекстов. Реализуется на ассемблере. Производит сохранение контекста прерываемого процесса, переключение указателей стеков процессов путём вызова **context_switch_hook()**[1] и восстановление контекста активируемого процесса.
TBaseProcess:: **init_stack_frame()**	Функция подготовки стекового кадра, которая формирует значения ячеек памяти в стеке таким образом, чтобы состояние стека было таким, как будто процесс, которому принадлежит стек, прерван и контекст процесса сохранён в стеке. Функция используется конструктором процесса и при рестарте процесса.
system_timer_isr()	Обработчик прерываний системного таймера. Вызывает функцию **TKernel::system_timer()**.

[1] Через функцию-«обёртку» **os_context_switch_hook()**, имеющую спецификацию связывания "**extern C**".

7.3. Портирование

Для портирования, как правило, достаточно определить для целевой платформы все вышеперечисленные макросы, типы и функции.

Наиболее «тонкая» и ответственная работа при портировании выпадает на реализацию ассемблерного кода и на функцию подготовки стекового кадра. Ряд моментов, на которые следует обратить особое внимание:

- выяснить, какие используются соглашения о вызове функций у используемого компилятора, чтобы знать, какие регистры (или область стека) используются для передачи аргументов тех или иных типов;
- определить особенности работы процессора в части сохранения адресов возвратов и статусных регистров при возникновении прерывания – это необходимо для понимания, как формируется стековый кадр на целевой аппаратной платформе, что, в свою очередь, важно для реализации функции (и обработчика прерываний) переключения контекстов, и функции формирования стекового кадра;
- проверить схему кодирования экспортируемых/импортируемых имён ассемблера. В простейшем случае имена объектов и функций на C (и ”extern C”[1] на C++) на ассемблере видны без изменений, но на некоторых платформах[2] к самому имени могут добавляться префиксы и/или суффиксы, что потребует ассемблерные функции именовать в соответствии с этой схемой, иначе линкер не сможет правильно выполнить связи.

Весь ассемблерный код должен быть помещён в файл os_target_asm.ext, упомянутый выше. Определения макросов и типов, а также встраиваемых функций – в файл os_target.h. В файле os_target.cpp объявляются объекты типов, если необходимо, – например:

```
OS::TPrioMaskTable OS::PrioMaskTable;
```

а также определяются функция **`TBaseProcess::init_stack_frame()`** и обработчик прерывания системного таймера **`system_timer_isr()`**.

[1] Имена в C++ подвергаются специальному кодированию в целях поддержки перегрузки имён функций, а также для типобезопасного связывания, по какой причине получить к ним доступ на ассемблере задача трудновыполнимая. Поэтому имена функций, описанных в файлах, которые компилируются C++ компилятором и к которым необходим доступ из ассемблерного кода, должны быть объявлены в исходных файлах как ”extern C”.

[2] В частности, на Blackfin.

Вышеописанное является лишь общими сведениями, относящимися к порту ОС, при портировании возникает достаточно много нюансов, описание которых является весьма частным и выходит за рамки настоящего документа.

СОВЕТ. При создании нового порта имеет смысл взять за основу или в качестве примера один из существующих – это значительно облегчает процесс портирования. Какой именно выбрать из имеющихся портов, зависит от близости аппаратной и программной частей платформы, на которую осуществляется портирование.

Подробности, присущие портам под конкретные платформы, описаны в отдельных документах, посвящённых этим портам.

7.4. Запуск в составе рабочего проекта

Для повышения гибкости и эффективности использования часть платформеннозависимого кода, зависящая от частных особенностей того или иного проекта и конкретно используемого микроконтроллера, вынесена на уровень проекта. Сюда, как правило, относится выбор аппаратного таймера процессора, используемого в качестве системного, а также выбор прерывания переключения контекстов, если процессор не имеет специализированного программного прерывания.

Для конфигурирования порта проект должен содержать файлы:

- scmRTOS_config.h;
- scmRTOS_target_cfg.h;

scmRTOS_config.h содержит большинство конфигурационных макросов, задающих такие параметры, как количество процессов в программе, способ передачи управления, включение функции системного времени, разрешение использования пользовательских хуков, порядок нумерации значений приоритетов и т.д.

В scmRTOS_target_cfg.h размещён код управления ресурсами целевого процессора, выбранными для реализации системных функций – всё тот же системный таймер, прерывание переключения контекстов.

Содержимое обоих конфигурационных файлов подробно описано в документах, посвящённых конкретным портам.

Заключение

Использование операционных систем реального времени с вытесняющим планированием в небольших однокристальных микроконтроллерах на сегодняшний день не является чем-то сверхъестественным. При наличии достаточного минимума ресурсов использование ОСРВ становится предпочтительным, т.к. имеет ряд ключевых преимуществ перед вариантом, когда ОС не используется:

- во-первых, ОС предоставляет формализованный набор средств для распределения задач по процессам и по организации потока управления, что качественно меняет процесс разработки ПО в сторону упрощения, формализации логики работы программы как в пределах одного процесса, так и в пределах всей программы в целом;
- во-вторых, механизмы приоритетного планирования процессов (в т.ч. и при выходе из прерываний) ОСРВ дают возможность значительно улучшить поведение программы в смысле реакции на события;
- ну, и в-третьих, в силу более формализованного похода к разработке программ возникает тенденция к появлению типовых решений, что упрощает повторное использование кода в других проектах, а также упрощает переносимость между платформами хотя бы в рамках одной ОС.

Следует помнить, что применение ОС накладывает некоторые ограничения на применение процессора. Например, если нужно при возникновении прерывания максимально быстро на него среагировать, чтобы внутри обработчика прерывания, к примеру, «дёрнуть» ножкой МК, то ОС тут только помеха. Причина кроется в том, что переключение контекстов, а также работа средств межпроцессного взаимодействия выполняются в критических секциях, которые могут длиться десятки и сотни тактов процессора, и во время них прерывания заблокированы. Т.е. решение задач вроде формирования временных диаграмм при использовании ОСРВ становится крайне затруднительным (если не невозможным).

Процессор – как следует из самого слова – есть устройство для выполнения процессов в предположении, что процесс – длительный (по отношению к длительности выполнения команд процессора) промежуток времени. И требования ко времени реакции/формировании времён, сравнимых со временем выполнения команд, плохо совместимы с возможностями процессора, если только он не имеет «на борту» специальных периферийных устройств, которые делают работу аппаратно.

Можно, конечно, и при использовании ОС формировать жёсткие, выверенные по тактам, временные диаграммы, но при этом придётся заблокировать основные механизмы ОС, т.е. ОС будет в течение известного промежутка времени неработоспособной.

* * *

Развитие ***scmRTOS*** на текущий момент продолжается, в дальнейшем возможно добавление новых средств, расширение функциональности существующих портов, появление их под другие МК и прочие изменения.

Приложение А Примеры использования

А.1. Очередь заданий

А.1.1. Введение

Очередь заданий, которая будет рассмотрена в данном примере, представляет собой очередь сообщений на основе указателей на объекты-задания. Традиционно в ОС, написанных на языке программирования С, для реализации очередей сообщений используются указатели **`void*`** совместно с ручным преобразованием типов. Этот подход обусловлен имеющимися в наличие средствами языка С. Как уже было сказано, такой подход признан неудовлетворительным по соображениям удобства и безопасности. Поэтому вместо него будет применён другой способ, который доступен благодаря использованию языка С++ и предоставляет ряд преимуществ.

Во-первых, нет никакой необходимости в нетипизированных указателях – механизм шаблонов позволяет эффективно и безопасно использовать указатели на конкретные типы, что устраняет необходимость в ручном преобразовании типов.

Во-вторых, имеется возможность ещё более повысить гибкость сообщений на указателях, введя возможность передавать не только данные, но и в некотором смысле «экспортировать» действия – т.е. сообщение не только служит для передачи данных, но и позволяют производить определённые действия на приёмном конце очереди. Это достаточно легко реализуется на основе иерархии полиморф-

ных классов[1] сообщений. В данном примере и будет реализован упомянутый подход.

Поскольку в очередь передаются только указатели, сами тела сообщений размещаются где-то в памяти. Способ размещения может быть различным – от статического до динамического, в данном примере этот момент опущен, т.к. в контексте рассмотрения он не важен и на практике пользователь сам решает, как ему поступить, исходя из требований задачи, имеющихся ресурсов, личных предпочтений и т. п.

В данном примере будет продемонстрирован метод делегирования выполнения заданий, реализованный на основе очереди сообщений.

А.1.2. Постановка задачи

Разработка практически любой программы сводится к выполнению тех или иных действий, и эти действия по важности и приоритетности выполнения в общем случае различны, что и мотивирует использование операционных систем с приоритетными планировщиками. Нередко случается так, что в том или ином процессе при обработке событий возникает необходимость в выполнении некоего действия, требующего значительного процессорного времени[2] при отсутствии какой-то срочности в этом, т.е. это действие вполне может быть выполнено и в процессе с низким приоритетом. В этом случае разумно не тормозить текущий процесс выполнением этого действия, а перепоручить его выполнение другому процессу, имеющему низкий приоритет.

К тому же, в программе вышеописанные ситуации могут иметь место неоднократно, и для решения этой проблемы логично создать специальный низкоприоритетный процесс, которому и перепоручать (делегировать) выполнение зада-

[1] Для новичков в C++, но хорошо знакомых с языком C, можно привести аналогию по технической реализации. Суть полиморфизма состоит в выполнении разных действий при одном и том же описании. C++ поддерживает два вида полиморфизма – статический и динамический. Статический полиморфизм реализуется с помощью шаблонов (templates). Динамический – на основе виртуальных функций. Иерархия полиморфных классов строится с использованием динамического полиморфизма.
Технически механизм виртуальных функций реализуется на базе таблиц указателей на функции. Поэтому на языке C тоже можно было бы реализовать аналогичный механизм – например, на основе структур с указателями на массивы указателей на функции. Но в случае с C придется много делать руками, что чревато ошибками, не очень наглядно и, вследствие этого, трудоёмко и неудобно. C++ здесь просто перекладывает всю рутинную работу на компилятор, избавляя пользователя от необходимости писать низкоуровневый код с таблицами указателей на функции, их правильной инициализацией и использованием.

[2] Например, обширные вычисления или обновление контекста экрана в программе с графическим интерфейсом пользователя.

ний из других процессов, выполнение которых не хочется или нельзя по условиям задачи производить в самих высокоприоритетных процессах. Механизм передачи заданий для выполнения удобно выполнить на основе полиморфных классов-заданий и сервиса **`OS::channel`**, используемого в качестве транспорта для передачи объектов-заданий.

A.1.3. Реализация

Все задания, безотносительно к тому, какой процесс породил задание, что именно нужно сделать по заданию, имеют общее свойство – все они должны выполняться. Это позволяет использовать механизм, при котором запуск выполнения задания может быть произведён унифицированным способом, а реализация собственно задания сделана с помощью виртуальных функций. Для этого нужно определить абстрактный базовый класс, который задаёт интерфейс объектов-заданий:

```
class TJob
{
public:
    virtual void execute() = 0;
};
```

Т.е. есть объект-задание, у которого определено его главное общее свойство – он может выполняться.

Для краткости изложения будет рассмотрено два разных типа заданий[1], ресурсоёмких в смысле времени выполнения:

- вычислительное – например, вычисление полинома;
- пересылка значительно объёма данных – обновление экранного буфера.

Для этого нужно определить два класса:

```
class TPolyVal : public TJob
{
public:
    virtual void execute();
};
class TUpdateScreen : public TJob
{
public:
    virtual void execute();
};
```

Объекты этих классов и будут представлять собой задания, выполнение которых передаётся в низкоприоритетный процесс. Подробнее см. «**Листинг A.1 Типы и объекты примера делегирования заданий**».

[1] Очевидно, что при необходимости это количество можно можно легко увеличить.

```
{1}   //---------------------------------------------------------------------
{2}   class TJob             // абстрактный класс-задание
{3}   {
{4}   public:
{5}       virtual void execute() = 0;
{6}   };
{7}   //---------------------------------------------------------------------
{8}   class TPolyval : public TJob
{9}   {
{10}  public:
{11}      ...                       // конструкторы и остальной интерфейс
{12}      virtual void execute();
{13}
{14}  private:
{15}      ...                       // представление: коэффициенты
{16}      ...                       // полинома, аргументы,
{17}      ...                       // результат и т.д.
{18}  };
{19}
{20}  //---------------------------------------------------------------------
{21}  class TUpdateScreen : public TJob //
{22}  {
{23}  public:
{24}      ...                       // конструкторы и остальной интерфейс
{25}      virtual void execute();
{26}
{27}  private:
{28}      ...                       // представление
{29}  };
{30}  //---------------------------------------------------------------------
{31}  typedef OS::process<OS::pr1, 200> THighPriorityProc1;
{32}  ...
{33}  typedef OS::process<OS::pr3, 200> THighPriorityProc2;
{34}  ...
{35}  typedef OS::process<OS::pr7, 200> TBackgroundProc;
{36}
{37}  OS::channel<TJob*, 4> JobQueue; // очередь заданий на 4 элемента
{38}  TPolyval      Polyval;          // объект-задание
{39}  TUpdateScreen UpdateScreen;     // объект-задание
{40}  ...
{41}  THighPriorityProc1 HighPriorityProc1;
{42}  THighPriorityProc2 HighPriorityProc2;
{43}  ...
{44}  TBackgroundProc    BackgroundProc;
{45}  //---------------------------------------------------------------------
```

Листинг А.1 Типы и объекты примера делегирования заданий

Абстрактный базовый класс **`TJob`** задаёт интерфейс объектов-заданий, и объектов этого класса в программе быть не может. В данном случае интерфейс ограничен всего одной функцией **`execute()`**, что позволяет заданию выполняться[1]. Далее определены два конкретных класса-задания **`TPolyval`** и **`TUpdateScreen`**, которые уже нацелены на вполне чёткие цели: первый производит вычисление значения некоего полинома, второй обновляет экранный буфер.

[1] При необходимости можно расширить интерфейс с помощью других чистых виртуальных функций.

Дальнейший код не являет собой ничего необычного – это штатный способ определения типов и объектов, принятый в языке программирования C++ и рекомендованный для использования совместно с ***scmRTOS***. Следует заметить, что определения типов и объявления объектов могут быть размещены в разных файлах (заголовочных и исходных) так, как их удобнее использовать с точки зрения проекта. Конечно, для предотвращения возникновения ошибок при компиляции определения типов должны быть размещены так, чтобы быть доступными в точках объявления объектов, – это обыкновенное требование языков C/C++.

Ниже показан собственно код реализации делегирования заданий на основе очереди.

```
{1} //---------------------------------------------------------------------------
{2} template<> void THighPriorityProc1::exec()
{3} {
{4}     const timeout_t DATA_UPDATE_PERIOD = 10;
{5}     for(;;)
{6}     {
{7}         ...
{8}         sleep(DATA_UPDATE_PERIOD);
{9}         ...                         // загрузка данных в объект-задание[1]
{10}        JobQueue.push(&Polyval); // постановка задания в очередь
{11}    }
{12}}
{13}//---------------------------------------------------------------------------
{14}template<> void THighPriorityProc2::exec()
{15}{
{16}    for(;;)
{17}    {
{18}        ...
{19}
{20}        if(...)                             // элемент экрана изменён
{21}        {
{22}            JobQueue.push(&UpdateScreen); // постановка задания в очередь
{23}        }
{24}    }
{25}}
{26}//---------------------------------------------------------------------------
{27}template<> void TBackgroudProc::exec()
{28}{
{29}    for(;;)
{30}    {
{31}        TJob *Job;
{32}        JobQueue.pop(Job);          // извлечение задания из очереди
{33}        Job->execute();             // выполнение задания
{34}    }
{35}}
{36}//---------------------------------------------------------------------------
```

Листинг А.2 Исполняемые функции процессов

В этом примере два высокоприоритетных процесса часть работы, относящейся к их области ответственности, перепоручают (делегируют) другому, низко-

[1] Опционально. Нужна какая-либо загрузка окружения задания или нет, определяется целями приложения.

приоритетному процессу путём постановки заданий в очередь, которую он обрабатывает. Сам этот низкоприоритетный (фоновый) процесс ничего не «знает» о том, что нужно делать по заданиям, – в его компетенции только запустить указанное задание, которое само имеет достаточно информации о том, что и как необходимо сделать. Важно то, что выполняться делегированное задание будет с нужным (низким в данном случае) приоритетом, не тормозя высокоприоритетные процессы[1].

Очевидно, что в процессе-обработчике заданий можно легко организовать реализацию каких-либо действий, которые требуют периодического выполнения в фоне остальной программы. Для этого достаточно вызывать функцию **`pop()`** с таймаутом. По истечении таймаута процесс получит управление, и требуемые действия могут быть выполнены в этот момент. Как согласовать выполнение этих действий с выполнением заданий – это зависит от требований проекта и от решения, принимаемого пользователем.

Технические аспекты, на которые следует обратить внимание:

- несмотря на то, что тип элементов очереди – это указатель на базовый класс **`TJob`**, в очередь помещаются адреса объектов-заданий, которые являются производными от **`TJob`**. Это ключевой момент – на этом основан механизм работы виртуальных функций, являющийся центральным при реализации полиморфного поведения. При вызове **`Job->execute()`** реально будет вызвана функция, принадлежащая классу, адрес объекта которого помещён в очередь;
- сами объекты-задания в примере созданы статически. Это сделано для простоты – в данном случае способ создания этих объектов не важен, они могут быть размещены статически, они могут быть размещены в свободной памяти, важно, чтобы они имели нелокальное время жизни, т.е. могли существовать между вызовами функций. А факт существования активного задания состоит не в физическом существовании самого объекта-задания, а в помещении указателя с адресом объекта-задания в очередь.

В целом сам вышеописанный механизм достаточно прост, имеет низкие накладные расходы и позволяет гибко распределять программную нагрузку по приоритетам выполнения.

[1] Не только сами процессы, которые перепоручают выполнение задание низкоприоритетному процессу, но и другие процессы, выполнение которых может блокироваться длительным выполнением заданий в высокоприоритетных процессах.

ЗАМЕЧАНИЕ. Продемонстрированный выше механизм может быть применён не только для организации выполнения заданий с низким приоритетом, но и наоборот для выполнения их с высоким приоритетом – это актуально, если задание требует срочности выполнения, которую не обеспечивает приоритет того или иного процесса. Технически организация передачи заданий на выполнение точно такая же, как описано выше, с той лишь разницей, что процесс-обработчик заданий является не Background, а Foreground[1] процессом.

А.1.4. Семафоры взаимоисключения (mutex) и проблема блокировки высокоприоритетных процессов

При рассмотрении особенностей доступа к совместно используемым ресурсам из разных процессов через семафоры взаимоисключения была описана ситуация, решаемая методом инверсии приоритетов (стр 94). Суть её сводилась к тому, что при определённых обстоятельствах низкоприоритетный процесс может опосредованно блокировать выполнение высокоприоритетного процесса. Для решения этой проблемы часто используют приём под названием «инверсия приоритетов», суть которого сводится к тому, что высокоприоритетный процесс, пытаясь захватить семафор взаимоисключения, в случае, если семафор уже захвачен низкоприоритетным процессом, не просто переходит в состояние ожидания обычным образом, а меняется приоритетами с тем низкоприоритетным процессом, который захватил семафор, до момента освобождения семафора. Как уже было сказано ранее, этот метод не используется в ***scmRTOS*** ввиду наличия накладных расходов, сравнимых (или больших) с реализацией самого **`TMutex`**.

Для решения проблемы, упомянутой выше, можно предложить приём, описанный в данном примере. Только в качестве обработчика заданий использовать не низкоприоритетный процесс, а наоборот – высокоприоритетный. И программу организовать так, чтобы процессы, которые имеют доступ к совместно используемым ресурсам, эту работу не выполняли сами, а делегировали её в виде заданий высокоприоритетному процессу-обработчику. В этой ситуации никаких коллизий с приоритетностью выполнения не возникает, а накладные расходы на передачу заданий в виде указателей на объекты незначительны.

[1] По отношению к процессам, которые ставят задания в очередь.

А.2. Разработка расширения: профилировщик работы процессов

А.2.1. Постановка задачи

Профилировщик работы процессов – это объект, выполняющий действия по сбору информации об относительном времени активной работы процессов системы, её обработке и имеющий интерфейс, через который пользовательская программа может получить доступ к результатам профилировки.

Как было отмечено в «**6.3 Профилировка работы процессов**», сбор информации об относительном времени работы процессов можно выполнить разными способами – в частности, методом сэмплирования текущего активного процесса и путём измерения времени работы процессов. В сущности, сам класс профилировщика может быть одним и тем же, а выбор реализации обоих методов выполнен с помощью организации способов взаимодействия профилировщика с объектами ОС и использования аппаратных ресурсов процессора.

Реализация самого класса профилировщика требует доступа к внутренностям ОС, но все эти потребности могут быть удовлетворены штатными средствами операционной системы, которые предоставляются пользователю для подобных целей. Таким образом, профилировщик времени активной работы процессов может быть выполнен в виде расширения ОС.

Цель данного примера – показать, как можно создать полезное средство, расширяющее функциональные возможности операционной системы не изменяя исходный код ОС. Дополнительные требования:

- разрабатываемый класс не должен накладывать ограничений на способы использования профилировщика – т.е. период сбора информации и место использования должны полностью определяться пользователем;
- реализация должна быть как можно менее ресурсоёмкой, как по размеру исполняемого кода, так и по быстродействию, т.е., в частности, использование вычислений с плавающей точкой должно быть исключено.

А.2.2. Реализация

Профилировщик сам по себе выполняет две основные функции – это сбор информации об относительном времени активной работы процессов и обработка этой информации с целью получения результатов. Оценка времени работы процесса может быть реализована на основе счётчика, который накапливает информацию об этом. Соответственно, для всех процессов системы потребуется массив таких счётчиков. Также потребуется массив переменных, хранящих результаты профилировки. Итого, профилировщик должен содержать два массива переменных, функцию обновления счётчиков в соответствии с активностью процессов, функцию обработки значений счётчиков и сохранения результатов и функцию доступа к результатам профилировки. Для повышения гибкости использования основа профилировщика выполнена в виде шаблона – см. «**Листинг А.3 Профилировщик**».

```
{1}  template < uint_fast8_t sum_shift_bits = 0 >
{2}  class TProfiler : public OS::TKernelAgent
{3}  {
{4}      uint32_t time_interval();
{5}  public:
{6}      INLINE TProfiler();
{7}
{8}      INLINE void advance_counters()
{9}      {
{10}         uint32_t Elapsed = time_interval();
{11}         Counter[ cur_proc_priority() ] += Elapsed;
{12}     }
{13}
{14}     INLINE uint16_t get_result(uint_fast8_t index) { return Result[index]; }
{15}     INLINE void     process_data();
{16}
{17} protected:
{18}     volatile uint32_t  Counter[OS::PROCESS_COUNT];
{19}              uint16_t  Result [OS::PROCESS_COUNT];
{20} };
```

Листинг А.3 Профилировщик

Помимо перечисленного выше присутствует очень важная функция **`time_interval()`** {4}. Функция **`time_interval()`** определяется пользователем исходя из имеющихся у него ресурсов и выбранного способа сбора информации о времени работы процессов.

Вызов функции **`advance_counters()`** должен быть организован пользователем, и место вызова определяется выбранным методом профилировки – статистическим или измерительным.

Алгоритм обработки результатов сбора информации сводится к нормированию значений счётчиков, накопленных за период измерения, – см. «**Листинг А.4 Обработка результатов профилировки**».

```
{1}   template < uint_fast8_t sum_shift_bits >
{2}   void TProfiler<sum_shift_bits>::process_data()
{3}   {
{4}       // Use cache to make critical section as fast as possible
{5}       uint32_t CounterCache[OS::PROCESS_COUNT];
{6}       {
{7}           TCritSect cs;
{8}           for(uint_fast8_t i = 0; i < OS::PROCESS_COUNT; ++i)
{9}           {
{10}              CounterCache[i] = Counter[i];
{11}              Counter[i]       = 0;
{12}          }
{13}      }
{14}
{15}      uint32_t Sum = 0;
{16}      for(uint_fast8_t i = 0; i < OS::PROCESS_COUNT; ++i)
{17}      {
{18}          Sum += CounterCache[i];
{19}      }
{20}      Sum >>=  sum_shift_bits;
{21}
{22}      const uint32_t K = 10000;
{23}      for(uint_fast8_t i = 0; i < OS::PROCESS_COUNT; ++i)
{24}      {
{25}          Result[i]  = (CounterCache[i] >> sum_shift_bits) * K / Sum;
{26}      }
{27}  }
```

Листинг А.4 Обработка результатов профилировки

Как видно, все вычисления производятся в целых числах, а чтобы не блокировать работу прерываний на значительное время при обращении к массиву счётчиков[1], производится копирование этого массива во временный массив, который и используется при дальнейшей обработке данных.

Принятое разрешение результата профилировки составляет одну сотую долю процента, и конечные результаты хранятся в сотых долях процента. Реализуется это путём нормирования величины каждого счётчика, предварительно умноженного на коэффициент, задающий разрешение результата[2], к значению суммы величин всех счётчиков.

Из этих обстоятельств вытекает естественное ограничение на максимальную величину значения счётчика, которая используется при вычислениях. Тип переменных, выполняющих функции счётчиков профилировщика, является 32-раз-

[1] А это обращение необходимо сделать атомарным, дабы не нарушить целостность алгоритма из-за возможности асинхронного изменения значений счётчиков при вызове функции **`advance_counters()`**.

[2] В данном случае этот коэффициент равен 10000 (константа **`K`**), что и задаёт значение разрешения в 1/10000, которое соответствует 0.01%.

рядным беззнаковым целым, что позволяет представлять числа в диапазоне $0..2^{32}-1 = 0..4294967295$. При вычислениях производится умножение на коэффициент K, равный 10000, поэтому для предотвращения переполнения при вычислениях величина счётчика не должна превышать величину:

$$N = \frac{2^{32}-1}{10000} = \frac{4294967295}{10000} = 429496$$

Таким образом, величина каждого счётчика должна быть отмасштабирована (поделена) так, чтобы результат этой операции не превышал вычисленный выше предел. Самым эффективным способом достичь этого является деление значения счётчика на делитель, кратный 2, что при реализации сводится к обычному битовому сдвигу. Конкретная величина сдвига определяется требованиями проекта и задаётся пользователем в качестве аргумента шаблона.

Также пользователь должен позаботиться о том, чтобы за период профилировки не происходило переполнения счётчиков, т.е. накопленная любым счётчиком величина не превышала значения $2^{32}-1$. Удовлетворение этого требования достигается путём согласования периода профилировки и величины разрешения возвращаемого функцией **`time_interval()`** значения.

Подключение профилировщика к проекту осуществляется путём включения заголовочного файла profiler.h в конфигурационный файл проекта scmRTOS_extensions.h.

A.2.3. Использование

A.2.3.1. Статистический метод

В случае статистического метода вызов функции **`advance_counters()`** следует поместить в код, который периодически получает управление с равными интервалами времени, – например, в обработчик прерывания какого-либо таймера; в случае ***scmRTOS*** для этих целей хорошо подходит обработчик прерываний системного таймера, в этом случае вызов функции **`advance_counters()`** помещается в пользовательский хук системного таймера, вызов которого требуется разрешить при конфигурации. Функция **`time_interval()`** в этом случае всегда должна возвращать 1.

А.2.3.2. Измерительный метод

При выборе измерительного метода профилировки вызов функции **advance_counters()** должен производиться при переключении контекстов, что может быть достигнуто путём помещения её вызова в пользовательский хук прерывания переключения контекстов. Реализация функции **time_interval()** в этом случае получается несколько сложнее – функция должна возвращать значение, пропорциональное временному интервалу между предыдущим и текущим вызовами этой функции. Измерение этого временного интервала требует задействования тех или иных аппаратных ресурсов целевого процессора, и в большинстве случаев для этого подходит любой аппаратный таймер[1], позволяющий получать величину таймерного регистра[2].

Масштаб возвращаемого значения функции **time_interval()** должен быть согласован с периодом профилировки так, чтобы сумма всех возвращённых за период профилировки значений этой функции *для любого процесса* не превысила 2^{32}-1, – см. «**Листинг А.5 Пример функции измерения временных интервалов**».

```
{1}   template<uint_fast8_t sum_shift>
{2}   uint32_t TProfiler<sum_shift> TProfiler::time_interval()
{3}   {
{4}       static uint32_t Cycles;
{5}
{6}       uint32_t Cyc = sysreg_read(reg_CYCLES);
{7}       uint32_t Res = Cyc - Cycles;
{8}       Cycles       = Cyc;
{9}
{10}      return Res;
{11}  }
```

Листинг А.5 Пример функции измерения временных интервалов

На листинге А.5 представлен пример функции измерения временных интервалов. В данном примере для измерения временных интервалов используется аппаратный счётчик тактов процессора, работающего на частоте 200 МГц, что соответствует периоду следования тактов 5 нс. Период профилировки выбран равным 2 с. Отношение периодов таково, что счётчик успевает за период профилировки достичь величины в 2 с/5 нс = 400 000 000, что является величиной, меньше, чем 2^{32}-1, поэтому никаких дополнительных действий производить не надо.

[1] Некоторые процессоры, например, **Blackfin**, имеют в своём составе специальный аппаратный счётчик тактов процессора, который инкрементируется на каждом такте, что позволяет очень просто организовать процесс измерения временных интервалов.

[2] Например, **WatchDog Timer** МК **MSP430**, который вполне подходит для использования его в качестве системного таймера, не годится для целей измерения временных интервалов, т.к. не позволяет программе получить доступ к своему счётному регистру.

В противном случае возникла бы необходимость изменить код функции так, чтобы указанное условие соблюдалось.

Организация периода сбора информации об относительном времени активной работы процессов и способ отображения результатов профилировки находятся в ведении пользователя. Типовые схемы использования – определить класс-наследник от шаблона **`TProfiler`**[1], в котором добавить требуемую функциональность, или просто создать объект типа, напрямую инстанцированного из шаблона **`TProfiler`**, и использовать его возможности обычным образом.

А.2.4. Выводы

Приведённый пример показывает, что разработка подобного расширения не является очень сложной задачей, хотя требует хорошего понимания механизмов работы ОС.

При желании и/или необходимости пользователь может разработать свой собственный вариант профилировщика, который будет лучше удовлетворять потребностям пользовательского проекта и предпочтениям его разработчика.

Вышесказанное в полной мере относится и к любому другому расширению, которое может быть разработано для нужд прикладных проектов. Общий подход остаётся тем же самым – создаётся класс-наследник класса-интерфейса ядра **`TKernelAgent`** и добавляется всё необходимое представление и интерфейс.

* * *

Профилировщик, который показан в данном примере, является работоспособным и годным к использованию. Его исходный код можно найти в директории, предназначенной для расширений ОС.

[1] Указав аргумент шаблона, если необходимо масштабирование значений счётчиков.

Приложение В Вспомогательные средства

В.1. Утилита проверки целостности конфигурации системы

Как было сказано ранее, для корректной работы операционной системы, она должны быть правильно сконфигурирована – подробнее об этом см. **ЗАМЕЧАНИЕ** на стр 44, в параграфе 2.3.5.

Т.к. средствами программного пакета (компилятором, линкером) нет возможности проверить целостность (правильность и достаточность) конфигурации, для этой цели предоставляется инструментальное средство – специальная утилита **scmIC**[1].

Сама по себе утилита реализована в виде скрипта на языке Python и может использоваться как обычная утилита командной строки при наличии установленного интерпретатора языка Python. Формат запуска:

scmic.py src_folder1 [src_folder2...src_folderN] [options],

где *src_folder1..src_folderN* – директории с исходными файлами проекта, а options:

- -q - suppress output, подавляет вывод любых сообщений, кроме сообщений об ошибках и информации, выдаваемой по опции 's';
- -s - show summary, выводит информацию о процессах в виде таблицы, с указанием имён типов процессов, объектов процессов и их приоритетов;
- -r - recursive directory processing, включает рекурсивную обработку директорий, т.е. указанные в аргументах командной строки и все их вложенные.

При сканировании директорий ищутся файлы с расширениями 'h', 'c', 'cpp', которые и подвергаются анализу.

[1] IC означает Integrity Checker.

scmIC не производит проверку исходных файлов на соответствие их правилам языка C++, потому для минимизации ошибок рекомендуется запускать утилиту после успешной компиляции исходных файлов – например, перед запуском линкера или после него.

Пользователи, которые не имеют установленного интерпретатора языка Python, могут воспользоваться вариантом утилиты в виде исполняемого файла ОС **Windows** – scmic.exe. Этот файл является не отдельно написанным вариантом утилиты, а упаковкой скрипта, библиотек и интерпретатора в один самостоятельный исполняемый[1] файл.

Способ использования исполняемого файла точно такой же, как и в случае со скриптом.

Скрипт может также использоваться не только в виде отдельно запускаемой утилиты, но и из других скриптов языка Python – в частности, из скрипта сборки SConsturct системы управления сборкой проектов **SCons**. Формат вызова этой функции несколько отличается от использования в случае отдельного запуска и выглядит так:

```
{1} def checker(fld, Quiet = False, Summary = False, Recursive = False):
{2}     ...
```

Листинг B.1 Прототип функции проверки конфигурации

```
{1} import scmIC
{2} ...
{3} rcode = scmIC.checker(dir_list, ...)
```

Листинг B.2 Использование функции проверки конфигурации

где **`dir_list`** – список директорий, содержащих файлы проекта, **`rcode`** – код возврата, если проверка конфигурации завершилась успешно, этот код равен 0, иначе ненулевое целое значение.

[1] Что отражается в достаточно большом размере этого файла.

Предметный указатель