

sctRTOS

(Version 2)

**Операционная
система
реального времени**

**для однокристальных
микроконтроллеров**

**Новосибирск
2003-2006**

Оглавление

ОГЛАВЛЕНИЕ	3
ЛИСТИНГИ И ТАБЛИЦЫ	5
ПРЕДИСЛОВИЕ	7
ГЛАВА 1 ВВЕДЕНИЕ.....	11
ГЛАВА 2 ОБЗОР ОПЕРАЦИОННОЙ СИСТЕМЫ.....	19
2.1. ОБЩИЕ СВЕДЕНИЯ.....	19
2.2. СТРУКТУРА ОС	20
2.2.1. Ядро.	20
2.2.2. Процессы	20
2.2.3. Межпроцессное взаимодействие	21
2.3. ПРОГРАММНАЯ МОДЕЛЬ	22
2.3.1. Состав и организация.....	22
2.3.2. Внутренняя структура.....	23
2.3.3. Критические секции	25
2.3.4. Синонимы встроенных типов.....	26
2.3.5. Использование ОС	26
ГЛАВА 3 ЯДРО (KERNEL)	31
3.1. ОБЩИЕ СВЕДЕНИЯ И ФУНКЦИОНИРОВАНИЕ	31
3.1.1. Организация процессов.....	31
3.1.2. Передача управления.....	33
3.1.3. Планировщик.....	34
3.1.4. Прерывания.....	40
3.1.5. Достоинства и недостатки способов передачи управления	46
3.1.6. Поддержка межпроцессного взаимодействия	50
3.1.7. Системный таймер	50
3.2. СТРУКТУРА.....	53
ГЛАВА 4 ПРОЦЕССЫ.....	55
4.1. ОБЩИЕ СВЕДЕНИЯ И ВНУТРЕННЕЕ ПРЕДСТАВЛЕНИЕ	55
4.1.1. Процесс, как таковой.....	55
4.1.2. Стек.....	56
4.1.3. Таймауты.....	56
4.1.4. Приоритеты.....	56
4.1.5. Функция <i>Sleep()</i>	56
4.2. СОЗДАНИЕ И ИСПОЛЬЗОВАНИЕ ПРОЦЕССА	57
4.2.1. Определение типа процесса	57
4.2.2. Объявление объекта процесса и его использование.....	58
ГЛАВА 5 СРЕДСТВА МЕЖПРОЦЕССНОГО ВЗАИМОДЕЙСТВИЯ.....	59
5.1. OS::TMUTEX.....	59
5.2. OS::TEVENTFLAG	63
5.3. OS::TCHANNEL	66

5.4.	OS::MESSAGE.....	68
5.5.	OS::CHANNEL.....	70
5.6.	ЗАКЛЮЧИТЕЛЬНЫЕ ЗАМЕЧАНИЯ	76
ГЛАВА 6	ПОРТЫ.....	79
6.1.	ОБЩИЕ ЗАМЕЧАНИЯ	79
6.2.	MSP430	80
6.2.1.	Обзор.....	80
6.2.2.	Источник тактовой частоты.....	81
6.2.3.	Системный таймер.....	81
6.2.4.	Передача управления	81
6.2.5.	Прерывания	82
6.3.	AVR.....	83
6.3.1.	Обзор.....	83
6.3.2.	Системный таймер.....	86
6.3.3.	Передача управления	86
6.3.4.	Прерывания	87
6.4.	BLACKFIN.....	89
6.4.1.	Обзор.....	89
6.4.2.	Передача управления	90
6.4.3.	Прерывания	90
6.4.4.	Платформеннозависимые действия.....	91
6.4.5.	Системный таймер.....	91
ГЛАВА 7	ЗАКЛЮЧЕНИЕ.....	93
ПРИЛОЖЕНИЕ А	ИСТОРИЯ ИЗМЕНЕНИЙ.....	95
ПРИЛОЖЕНИЕ В	ПЕРЕНОС ПРОЕКТОВ С ВЕРСИЙ 1.XX.....	97
ПРИЛОЖЕНИЕ С	ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ.....	99
C.1.	Очередь сообщений на основе указателей.....	99
ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ.....		107

Листинги и таблицы

Листинг 2-1 Исполняемая функция процесса	21
Листинг 2-2 Определение типов процессов в заголовочном файле.....	29
Листинг 2-3 Объявление процессов в исходном файле и запуск ОС.....	29
Таблица 2-1 Конфигурационные макросы	30
Листинг 3-1 Функция регистрации процессов.....	31
Листинг 3-2 Функция запуска ОС	32
Листинг 3-3 Планировщик.....	34
Листинг 3-4 Цикл ожидания переключения контекстов.....	37
Листинг 3-5 Вариант планировщика, оптимизированный для использования в ISR.....	39
Листинг 3-6 Функция входа в прерывание.....	42
Листинг 3-7 Функция выхода из прерывания	42
Листинг 3-8 Класс-«обертка» обработчика прерываний.....	44
Листинг 3-9 Системный таймер (прямая передача управления).....	51
Листинг 3-10 Системный таймер (передача управления с помощью программного прерывания).....	52
Листинг 4-1 Определение шаблона типа процесса.....	57
Листинг 5-1 OS::TMutex	60
Листинг 5-2 Пример использования OS::TMutex.....	62
Листинг 5-3 OS::TEventFlag.....	64
Листинг 5-4 Использование TEventFlag.....	66
Листинг 5-5 OS::TChannel.....	67
Листинг 5-6 Шаблон OS : :message.....	69
Определение шаблона – см «Листинг 5-7 Определение шаблона OS::channel»	72
Листинг 5-7 Определение шаблона OS : : channel	72
Листинг 5-8 Пример использования очереди на основе канала.....	75
Листинг 6-1 Обработчик прерывания Timer_A Input Capture (прямая передача управления).....	82

Листинг 6-2 Обработчик прерывания Timer_В Overflow (передача управления на основе программного прерывания).....	83
Листинг 6-3 Определение типа процесса с отдельным стеком возвратов.....	85
Листинг 6-4 Обработчик прерывания системного таймера.....	88
Листинг 6-5 Пример определения прерывания (передача управления на основе программного прерывания).....	88
Листинг 6-6 Функция нахождения наивысшего приоритета процесса из готовых к выполнению.....	90
Листинг 6-7 Пример обработчика прерывания.....	91
Листинг 7-1 Пример использования очереди сообщений на указателях.....	105

Предисловие

Все приведенные ниже рассуждения, оценки, выводы основаны на личном опыте и знаниях автора и вследствие этого обладают известной долей субъективизма, что обуславливает наличие как неточностей, так и ошибок. Поэтому просьба рассматривать нижеизложенное с учетом 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 EC++ имеет ряд преимуществ (и ряд недостатков ☺) перед C. Основное преимущество – более защищенная модель программных объектов и, как следствие, более простое и безопасное использование. Основной недостаток – меньшая (и значительно) возможность переносимости на другие целевые платформы, т.к. количество соответствующих компиляторов количеству компиляторов языка C (которые есть почти подо все ☺). Мое личное мнение – это вопрос времени. Лед уже тронулся...

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 – фон Неймановской, в AVR при использовании пакетов IAR EWAVR реализованы

два стека: один для данных, другой для адресов возвратов), процесс переноса помог выявить ряд серьезных 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 реализована версия **scmRTOS** следующего поколения (версии 2.xx), где объекты процессов и ряд средств межпроцессного взаимодействия реализован на основе шаблонов. При этом изменения коснулись и других частей ОС – в частности, класс `os`, выполнявший функцию пространства имен для составных частей операционной системы заменен на настоящее пространство имен (*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 удален 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 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. ☺

В настоящем документе речь пойдет только о новой версии **scmRTOS** – 2.xx. Версии 1.xx будут упоминаться только в контексте сравнения.

* * *

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

Нужен, в первую очередь, соответствующий компилятор C++. В настоящее время такие компиляторы существуют под несколько платформ. **scmRTOS** пока имеет порты под MSP430 и AVR.

Второе, нужно, чтобы используемый микроконтроллер имел необходимый минимум ОЗУ. Эта величина определена как 512¹ байт. На МК с меньшим количеством ОЗУ ***scmRTOS*** **работать** не будет. То есть, можно, конечно, попытаться запустить ее и на 256 байтах – один или два крохотных процесса, возможно, влезут (я не проверял), но, по моему мнению, это баловство, т.к. 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 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 – очень маленькая ОС². В ней использованы очень простые³ (можно даже сказать, тривиальные) механизмы, потребление ОЗУ для служебных целей минимизировано. Благодаря простоте и малому размеру процесс освоения значительно упрощается. Главное – понять основные принципы, а затем, чтобы разобраться во внутренних механизмах, дело не станет! ☺

* * *

Эта ОС разрабатывалась для себя и для всех желающих ее использовать. Любой, изъявивший желание, может использовать и/или распространять ее совершенно бесплатно как в образовательных целях, так и в некоммерческих и коммерческих проектах. Единственное ограничение – копирайт, но это у нас никому не мешает. ☺

scmRTOS поставляется «как есть» (“as is”), никаких гарантий, естественно, не предоставляется. ☺

¹ На момент написания 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. ☺

² Исходные тексты, включая порты, занимают всего несколько десятков килобайт.

³ И, как следствие, быстрые.

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 четко понять контекст дальнейшего описания, рассуждений и выводов.

* * *

Что есть операционная система (ОС) вообще? Вопрос в достаточной мере абстрактен. Исходя из разных аспектов, 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 идет 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 проч.

Таким образом, исходя из того, что основная функция ОС – поддержка параллельного асинхронного исполнения разных процессов и взаимодействия между ними, встает вопрос о планировании (Scheduling) процессов, т.е. когда какой процесс должен получить управление, когда отдать управление другому процессу и проч. 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 такими планировщиками являются, например, широко известная и популярная коммерческая ОС реального времени uC/OS-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 передается 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 сишной 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 детерминированы, 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 в любом

случае можно провести анализ и сделать более-менее точную оценку. ОС, которая не гарантирует никаких времен, соответственно, не является ОСРВ.

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. Остальные типы планировщиков уступают им по скорости реакции.

По детерминированности времени отклика из оставшихся на первом месте — ОС с вытесняющими round-robin планировщиками, в этом случае, событие будет обработано тогда, когда дойдет 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), где каждый процесс организован как автомат состояний, 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, вся система встанет «колом»! ☺

Вышесказанное дает основания сомневаться, что операционные системы с кооперативными планировщиками можно в полной мере отнести к системам реального времени. О степени соответствия можно более-менее судить по совокупности ОС + прикладной код: если прикладной код организован так, чтобы обеспечить максимальную скорость реакции, а планировщик ОС является приоритетным (т.е. при отдаче управления текущим процессом, его (управление) получает процесс с наибольшим приоритетом, для которого есть «работа»), то такая система (не ОС, а именно ОС + прикладная программа) вполне может «претендовать на звание» СРВ. ☺

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 серьезный 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, его (процесса) «окружение» (содержимое регистров процессора, состояние стека – то, что называется контекстом процесса) 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 статических¹ переменных + размер стека, который является общим для всех, то при использовании ОС с вытесняющей 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 MSP430² размер контекста равен порядка 30 байт (12 регистров + SR + SP = 28 байт), при вложенности вызовов функций до 10 (и это немного) – это еще 20 байт, уже получаем порядка 50 байт только для обеспечения работы ОС, 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 объемом ОЗУ многих однокристаллок. Поэтому использование ОС с вытесняющим планированием натывается на принципиальные трудности при использовании МК с объемом ОЗУ менее полкилобайта. Такие МК – ниша для кооперативных ОС.

Еще одно принципиальное ограничение – аппаратный стек в некоторых МК. По этой причине вытесняющая ОС, видимо, никогда не сможет работать на PIC16, поэтому МК этого семейства также кандидаты на использование их с кооперативными ОС.

Кооперативные ОС тоже имеют контекст, который переключается при передаче управления от одного процесса к другому, но размер этого контекста очень мал (по сравнению контекстами процессов в вытесняющих ОС). Сохранять все рабочие регистры процессора нет необходимости, т.к. управление передается не в произвольный момент времени, а во вполне определенный, поэтому сохранение регистров происходит таким же образом, как при вызове любой функции.

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.

Что касается ОС с чистым вытесняющим планировщиком round-robin (т.е. не приоритетным), то причина существования таких ОС (лично мне) не очень понятна: вытеснение (т.е. асинхронное отбирание управления и передача его другому процессу) реализовано, соответственно, ресурсы для отдельного стека каждому процессу имеются, т.е. главное ограничение преодолено, почему было не сделать приоритетный планировщик, не понятно: он не настолько сложнее карусельного (round-robin), чтобы от него отказываться. Справедливости ради надо отметить, что реально ОС с такими планировщиками (по крайней мере, в сегменте малых процессоров) не видно ни одной.

Еще 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, а преимущества как-то не особенно заметны.

* * *

Операционная система реального времени для однокристальных микроконтроллеров **scmRTOS** (Single-Chip Microcontroller Real-Time Operating System) 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 очень небольшое количество доступной оперативной памяти (ОЗУ): у подавляющего большинства микроконтроллеров эта величина на текущий момент не превышает 2-4 килобайт¹, и это уже приличные по остальным ресурсам кристаллы – с объемом ПЗУ 16 и более (исчисляется десятками) килобайт, большим количеством периферийных устройств (таймеров, АЦП, портов ввода-вывода, последовательных портов, как синхронных (SPI, I²C), так и асинхронных (UART)). Таким образом, внутреннее ОЗУ является одним из наиболее дефицитных ресурсов МК для реализации ОС с вытесняющим планированием.

¹ Конечно, есть вполне «толстые» МК, например M16C (Mitsubishi), с объемом внутреннего ОЗУ до 20 килобайт, или MB90xxx (Fujitsu), где объем ОЗУ начинается от 2 килобайт (есть кристаллы с 8-ю килобайтами), но это пока отдельные примеры, которые уступают более мелким МК по другим характеристикам, таким как цена (M16C), энергопотребление (оба упомянутых).

Как показано выше, существуют принципиальные ограничения на использование вытесняющего механизма – главным образом из-за требований к ОЗУ. При разработке описываемой ОС ставилась цель получить минимально ресурсоемкое решение, чтобы его можно было реализовать на однокристальных МК с объемом ОЗУ от 512 байт. Основные характеристики ОС закладывались на основе этой главной цели. Именно этим обусловлено небольшое количество процессов (до 15¹ пользовательских – большее количество на 1-2 килобайтах ОЗУ и не получится), отказ от механизмов динамического создания/удаления процессов, изменения приоритетов процессов на этапе выполнения программы и проч., словом, всего того, что может повлечь за собой дополнительные накладные расходы как по размеру памяти, так и по времени выполнения.

Благодаря исходной ориентации на мелкие МК, что позволило применить упрощенные и облегченные решения, удалось добиться сравнительно неплохого результата. Выигрыш по быстродействию достигнут, главным образом, благодаря очень простому механизму планирования (и вычисления приоритетов), возможность использования которого обеспечена малым количеством процессов в *scmRTOS*.

Упрощенная функциональность также дает выигрыш по ресурсоемкости: ядро в *scmRTOS* занимает порядка $8-12 + 2 \cdot \text{Кол-во процессов байт}$, 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, поддерживающих оба этих механизма.

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

- ARM;
- AVR (Atmel);

¹ 15 пользовательских процессов – это не так уж и мало! Реально, как показывает практика, в системах на таких МК вполне хватает 4-8 процессов, т.ч. 15 - это даже с запасом.

- MSP430 (Texas Instruments);
- PIC18 (Microchip);
- 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 дорогих) компиляторов фирмы IAR Systems существует семейство бесплатных, но достаточно гибких и мощных компиляторов GCC (GNU Compiler Collection).

Существуют компиляторы и для других платформ. Процесс этот развивается, и тенденция такова, что в дальнейшем C++ займет нишу C, т.к. он покрывает все возможности C и добавляет принципиально новые возможности, 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 еще одна причина, обуславливающая выбор C++ в качестве языка разработки ОС.

scmRTOS v2 разработана с использованием компилятора C++ для MSP430 от IAR Systems и на текущий момент имеет порты под EW430 v3.xx, EWAVR v4.xx и VisualDSP++ v4.0/Blackfin¹.

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'a 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.

Глава 2

Обзор операционной системы

— Гого, ты знаешь, что такое «ОС»?

— Канэшина знаю: «ОС» - это бал-
шой, паласатый мух!

— Нэт! Балшой паласатый мух –
эта имел, а «ОС» - эта то, на чем
вэртится наша Земла!

анекдот.

2.1. Общие сведения

scmRTOS является операционной системой реального времени с приоритетной вытесняющей многозадачностью. ОС поддерживает до 16 процессов (включая системный процесс `idleProcess`, т.е. до 15 пользовательских процессов), 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 очень ограничены. Динамическое удаление процессов также не реализовано, т.к. в этом немного смысла – ПЗУ, используемое процессом, при этом не освобождается, а ОЗУ для последующего использования должно иметь возможность быть выделяемым/освобождаемым с помощью диспетчера памяти, который сам по себе весьма «толстая» вещь и, как правило, не используется в проектах на однокристальных МК.

В текущей версии приоритеты процессов также статические, т.е. каждый процесс получает приоритет на этапе сборки, и приоритет не может быть изменен во время выполнения программы. Такой подход также обусловлен стремлением сделать систему как можно более легкой в части требований к ресурсам и мобильной в смысле быстродействия, т.к. изменение приоритетов совсем нетривиальный механизм, который для корректной работы требует анализа состояния всей системы (ядра, сервисов) с последующей модификацией составляющих ядра и, возможно, остальных частей ОС (семафоров, флагов событий и проч.)

2.2. Структура ОС

Система состоит из ядра (Kernel), процессов, 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).

Подробнее о структуре, составе, функциях и механизмах ядра см «Глава 3 Ядро (Kernel)»

2.2.2. Процессы

Процессы реализуют возможность создавать отдельный (асинхронный по отношению к остальным) поток управления. Каждый процесс для этого предоставляет функцию, которая должна содержать бесконечный цикл, являющийся главным циклом процесса – см «Листинг 2-1 Исполняемая функция процесса».

```
{1} OS_PROCESS 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 передается 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 «Глава 4 Процессы». В коде процесса не должно также быть операторов возврата из функции – **return**.

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 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 прерывания запрещены, или использовать специальные средства для межпроцессного взаимодействия. К таким средствам в **scmRTOS** относятся:

- флаги событий (OS::TEventFlag);
- семафоры взаимного исключения (OS::TMutex);

¹ При этом никакой другой процесс не должен «будить» этот спящий перед выходом процесс, иначе возникнет неопределенное поведение и система, скорее всего, «упадет».

- 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 ☺.

2.3. Программная модель

2.3.1. Состав и организация

Исходный код *scmRTOS* разбит на две основные части: общая (Common) и платформеннозависимая (Target).

Общая часть содержит объявления и определения функций ядра, процессов (кроме конструктора процесса), системных сервисов.

Платформеннозависимая часть – объявления и определения, отвечающие за реализацию функций, присущих данной целевой платформе, расширения языка для текущего компилятора, а также небольшую библиотеку поддержки, содержащую некоторый полезный код, часть которого непосредственно используется ОС. К платформеннозависимой части относятся код переключения контекста, определение процесса `IdleProcess` на данной платформе, конструктор класса `TBaseProcess`¹, который, помимо прочего, формирует структуру стека (stack frame), определение класса-«обертки» (wrapper) критической секции, а также макросы, реализующие поддержку аппаратного таймера данной платформы, используемого в качестве системного, и другое платформеннозависимое поведение.

Конфигурирование производится путем определения макросов и указания их значений в специальном конфигурационном заголовочном файле `scmRTOS_config.h`, который должен быть подключен к проекту.

¹ На основе которого построен шаблон `process`.

Исходные тексты общей части содержатся в пяти файлах:

- OS_Kernel.cpp – определения функций ядра;
- OS_Services.cpp – определения функций сервисов;
- scmRTOS.h – главный заголовочный файл, содержит основные объявления и определения;
- scmRTOS_defs.h – вспомогательные объявления и макросы.
- services.h – определения шаблонных функций сервисов.

Исходный код платформеннозависимой части находится в трех файлах:

- OS_Target.h – платформеннозависимые объявления и макросы;
- OS_Target_asm.ext¹ - низкоуровневый код, функции переключения контекста;
- OS_Target_cpp.cpp – определения функции-конструктора процесса и системного процесса `IdleProcess`.

В состав **scmRTOS** входит еще небольшая библиотека поддержки, где находится код, используемый средствами ОС². Состав и реализация библиотеки для разных платформ может различаться. Поскольку сама по себе эта библиотека не входит в состав ОС, то внимания к ее (библиотеки) рассмотрению в текущем документе уделено не будет.

2.3.2. Внутренняя структура

Все, что относится к **scmRTOS**, за исключением нескольких ассемблерных функций (которые имеют спецификацию связывания **extern "C"**), помещено внутрь пространства имен **OS** – таким способом реализовано отдельное пространство имен для составных частей ОС.

¹ Расширение ассемблерного файла для целевого процессора.

² В частности, класс/шаблон кольцевого буфера.

Внутри этого пространства имен объявлены следующие классы¹:

- **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 стр. 31;
- **TBaseProcess**. Реализует тип объекта, являющегося основой для построения шаблона **process**, на основе которого реализуется любой (пользовательский или системный) процесс. Подробнее см стр. 55;
- **process**. Шаблон, на основе которого создается тип любого процесса ОС.
- **TISR_Wrapper**. Это класс-«обертка» для облегчения и автоматизации процедуры создания кода обработчиков прерываний. Его конструктор выполняет действия при входе в обработчик прерывания, а деструктор – соответствующие действия при выходе.
- **TEventFlag**. Предназначен для межпроцессных взаимодействий путем передачи бинарного семафора (флага события). Подробнее см стр. 63;
- **TMutex**. Бинарный семафор, предназначенный для организации взаимного исключения доступа к совместно используемым ресурсам. Подробнее см стр. 59;
- **TChannel**. Реализует тип объекта, являющегося базовым для объектов-каналов, выполняющих функции канала передачи данных. Ширина канала – один байт. Глубина задается при определении типа. Подходит также для организации очередей данных, состоящих из байт. Подробнее см стр. 66;
- **message**. Шаблон для создания объектов-сообщений Сообщение «похоже» на **EventFlag**, но вдобавок может еще содержать объект произвольного типа (обычно это структура), представляющий собой тело сообщения. Подробнее см стр. 68;
- **channel**. Шаблон для создания канала передачи данных произвольного типа. Служит основой для построения очередей сообщений. Более подробно см стр. 70.

Как видно из приведенного выше списка, отсутствуют счетные семафоры. Причина этого в том, что при всем желании не удалось увидеть острой необходимости в них. Ресурсы, которые нуждаются в контроле с помощью счетных 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 типа **TChannel** или созданных на основе шаблона **channel**, внутри которых в том или ином виде уже реализован соответствующий механизм.

scmRTOS предоставляет пользователю несколько функций для контроля:

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.

- **Run ()** ; Предназначена для запуска ОС. При вызове этой функции начинается собственно функционирование операционной системы – управление передается процессам, работа которых и взаимное взаимодействие определяется пользовательской программой;
- **LockSystemTimer ()** ; Блокирует прерывания от системного таймера;
- **UnlockSystemTimer ()** ; Разблокирует прерывания от системного таймера;
- **WakeUpProcess (TBaseProcess& p)** ; 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;
- **ForceWakeUpProcess (TBaseProcess& p)** ; 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;
- **IsProcessSleeping (const TBaseProcess& p)** ; Проверяет, находится ли процесс в состоянии «спячки», т.е. в состоянии ожидания с таймаутом события;
- **IsProcessSuspended (const TBaseProcess& p)** ; Проверяет, находится ли процесс в неактивном состоянии.
- **GetTickCount ()** ; Возвращает количество тиков системного таймера. Счетчик тиков системного таймера должен быть разрешен при конфигурировании системы.

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, когда необходимо исключить возможность прервать процесс во время выполнения определенного фрагмента кода. Это достигается запрещением прерываний 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-«обертка» **tcritsect**. В конструкторе этого класса запоминается состояние процессорного ресурса, управляющего разрешением/запрещением глобальных прерываний, и прерывания запрещаются. В деструкторе этот процессорный ресурс приводится к тому состоянию, в котором он пребывал перед запрещением прерываний. Т.е. если прерывания были запрещены, то они и останутся запрещены. Если были разрешены, то будут разрешены. Реализация этого класса

платформеннозависима, поэтому ее определение содержится в соответствующем файле `OS_Target.h`.

Использование `tcritsect` тривиально: достаточно просто завести объект этого типа, и от места объявления до конца блока прерывания будут запрещены¹.

2.3.4. Синонимы встроенных типов

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

- `byte` – `unsigned char`;
- `word` – `unsigned short`;
- `dword` – `unsigned long`;
- `TProcessMap` – тип для определения переменной, выполняющий функцию карты процессов. Ее размер зависит от количества процессов в системе. Каждому процессу соответствует один бит, расположенный в соответствии с приоритетом этого процесса. Процессу с наибольшим приоритетом соответствует младший бит (позиция 0). При количестве пользовательских процессов менее 8 размер карты процессов – 8 бит, а тип – `byte`. При 8 и более пользовательских процессов размер – 16 бит, а тип – `word`;
- `TStackItem` – тип элемента стека. Зависит от целевой архитектуры. Например, на 8-разрядном AVR этот тип определен как `byte`, на 16-разрядном MSP430 – `word`, а на 16/32-разрядном Blackfin – `dword`.

2.3.5. Использование ОС

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

В первую очередь это касается процессов. Перед использованием каждого процесса должен быть определен его тип², где указывается имя типа процесса, его приоритет и размер области ОЗУ, отведенной под стек процесса³. 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 Каждый процесс – это отдельный тип (класс), производный от общего базового класса `TBaseProcess`.

³ Более подробно см «Глава 4 Процессы».

Здесь определен процесс с приоритетом 2 и размером стека в 200 байт. Такое объявление может показаться несколько неудобным из-за некоторой многословности, т.к. при необходимости ссылаться на тип процесса придется писать полное объявление – например, при определении исполняемой функции процесса:

```
OS_PROCESS 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 нужно пользоваться синонимами типов, вводимыми через `typedef`. Это рекомендуемый стиль кодирования: сначала определить типы 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. При этом приведенный выше пример выглядит так:

```
// В заголовочном файле
typedef OS::process<OS::pr2, 200> TMainProc;
...
// В исходном файле
TMainProc MainProc;
...
OS_PROCESS void TMainProc::Exec()
...
```

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



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 введен специальный перечислимый тип `TPriority`, который описывает допустимые значения приоритетов¹.

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` связаны с количеством процессов, указанным в значении 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.

Кроме того, приоритеты всех процессов должны идти подряд, «дырок» 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. Старшинство (порядок следования) приоритетов может быть задано пользователем на этапе сборки – старшинство может быть задано по возрастанию или по убыванию. Например, если в системе 4 пользовательских процесса (т.е. всего 5 процессов – один системный процесс `IdleProcess`), то если выбран порядок по возрастанию, то значения приоритетов должны быть `pr0`, `pr1`, `pr2`, `pr3` (`pr4` – для `IdleProcess`), где `pr0` – самый высокоприоритетный процесс, а `pr3` – самый низкоприоритетный (не считая `IdleProcess`); если выбран порядок по убыванию, то значения приоритетов должны быть `pr4`, `pr3`, `pr2`, `pr1`, где `pr4` – самый высокоприоритетный процесс, а `pr1` – самый низкоприоритетный (опять же не считая `IdleProcess`). Самый низкоприоритетный процесс – это всегда `IdleProcess`. Этот процесс существует в системе всегда, его описывать не нужно.

За пропусками в нумерации приоритетов процессов, а также за уникальностью значений приоритетов процессов компилятор не следит, т.к., придерживаясь принципа отдельной 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 `scmlC` (`IC` – Integrity Checker), и позволяет «выловить» подавляющее большинство типовых ошибок конфигурирования ОС. К сожалению, утилита для версий 1.xx не подходит для версий 2.xx, поэтому для текущей версии подобный инструмент пока отсутствует. Но он появится в будущем.

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} //-----
{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}     OS::Run();
{16} }
{17} //-----

```

Листинг 2-3 Объявление процессов в исходном файле и запуск ОС

Каждый процесс, как уже упоминалось выше, имеет исполняемую функцию. При использовании приведенной выше схемы исполняемая функция процесса называется **Ехес** и выглядит, как показано на «Листинг 2-1 Исполняемая функция процесса».

Конфигурационная информация задается в специальном заголовочном файле `scmRTOS_config.h`. Состав и значения¹ конфигурационных макросов — см Таблица 2-1 Конфигурационные макросы.

Название	Знач.	Описание
<code>scmRTOS_PROCESS_COUNT</code>	5	Количество процессов в системе
<code>scmRTOS_SYSTIMER_NEST_INTS_ENABLE</code>	—	Разрешает вложенные прерывания в обработчике прерывания от системного таймера.
<code>scmRTOS_SYSTEM_TICKS_ENABLE</code>	—	Включает использование счетчика тиков системного таймера.

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

<code>scmRTOS_SYSTIMER_HOOK_ENABLE</code>	–	Включает в обработчике прерывания системного таймера вызов функции <code>SystemTimerUserHook()</code> . В этом случае указанная функция должна быть определена в пользовательском коде.
<code>scmRTOS_IDLE_HOOK_ENABLE</code>	–	Включает в системном процессе <code>IdleProcess</code> вызов функции <code>IdleProcessUserHook()</code> . В этом случае указанная функция должна быть определена в пользовательском коде.
<code>scmRTOS_START_HOOK_ENABLE</code>	–	Включает в функции запуска ОС вызов функции <code>SystemStartUserHook()</code> . В этом случае функция должна быть определена в пользовательском коде. Функция вызывается непосредственно перед передачей управления процессам, т.е. вся оставшаяся работа перед этим уже сделана, т.ч. имеется возможность что-то изменить.
<code>scmRTOS_CONTEXT_SWITCH_SCHEME</code>	0/1	Задаёт способ переключения контекстов. Подробнее см ...
<code>scmRTOS_PRIORITY_ORDER</code>	0/1	Задаёт порядок старшинства приоритетов. Значение 0 соответствует варианту <code>pr0</code> – наиболее приоритетный процесс, значение 1 соответствует варианту <code>pr0</code> – наименее приоритетный процесс.
<code>scmRTOS_USER_PLATFORM_SPECIFIC_STUFF</code>	–	Исключает использование макроса <code>EXECUTE_PLATFORM_SPECIFIC_STUFF</code> из кода запуска RTOS.
<code>scmRTOS_USER_START_SYSTEM_TIMER</code>	–	Исключает использование макроса <code>START_SYSTEM_TIMER</code> из кода запуска RTOS.

Таблица 2-1 Конфигурационные макросы

Глава 3

Ядро (Kernel)

3.1. Общие сведения и функционирование

Как уже было сказано выше, ядро выполняет:

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

3.1.1. Организация процессов

Функция по организации процессов сводится к регистрации созданных процессов. При этом в конструкторе каждого процесса вызывается функция ядра `RegisterProcess(TBaseProcess*)`, которая помещает значение указателя на процесс, переданного в качестве аргумента, в таблицу `ProcessTable` (см ниже) 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::RegisterProcess(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} void OS::Run()
{2} {
{3} #ifndef scmRTOS_USER_PLATFORM_SPECIFIC_STUFF
{4}     EXECUTE_PLATFORM_SPECIFIC_STUFF();
{5} #endif
{6}
{7} #ifndef scmRTOS_USER_START_SYSTEM_TIMER
{8}     START_SYSTEM_TIMER();
{9} #endif
{10}
{11} #ifdef scmRTOS_START_HOOK_ENABLE
{12}     SystemStartUserHook();
{13} #endif
{14}
{15}     TStackItem* sp =
{16}     Kernel.ProcessTable[scmRTOS_MOST_READY_PROCESS]->
{17}     StackPointer;
{18}     OS_Start(sp);
{19} }
```

Листинг 3-2 Функция запуска ОС

Здесь производятся платформеннозависимые действия и старт системного таймера. Это достигается с помощью использования макросов `EXECUTE_PLATFORM_SPECIFIC_STUFF()` и `START_SYSTEM_TIMER()`. Первый выполняет какие-то действия, необходимые для правильной работы целевого процессора в рамках данной ОС. Второй – запускает системный таймер. Оба макроса определены в файле `OS_Target.h`. Если пользователя по каким-то причинам не устраивает функциональность этих макросов, то он может отключить их использование путем определения в файле конфигурации соответствующих макросов `scmRTOS_USER_PLATFORM_SPECIFIC_STUFF` и `scmRTOS_USER_START_SYSTEM_TIMER`. Естественно, что в этом случае пользователь должен самостоятельно позаботиться о том, чтобы целевой процессор функционировал в требуемом режиме, а также запустить системный таймер.

Затем, если разрешено при конфигурировании, вызывается пользовательская функция `SystemStartUserHook()` {12}, с помощью которой имеется возможность что-то добавить или изменить. Это является одним из способов изменить настройки, заданные по умолчанию, не модифицируя исходный код ОС.

После этого из таблицы процессов извлекается указатель на стек самого приоритетного процесса {15} и производится собственно старт системы {18} пу-

тем запуска низкоуровневой функции `os_start()` с передачей ей в качестве аргумента извлеченного 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.

3.1.2. Передача управления

Передача управления может происходить двумя способами:

- процесс сам отдает управление, когда ему (пока) нечего больше делать, или в результате своей работы процесс должен войти в межпроцессное взаимодействие с другими процессами (захватить семафор взаимного исключения (`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 свое задание и не отдаст управление¹.

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 путем вызова из планировщика² низкоуровневой³ функции переключателя контекста. Другой способ – передача управления путем активации специального программного прерывания, где и происходит переключение контекста. *scmRTOS v2.xx* поддерживает оба способа. И тот, и другой способы имеют свои достоинства и недостатки, которые будут подробно рассмотрены ниже.

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, который готов к выполнению.

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

³ Обычно реализуемой на ассемблере.

3.1.3. Планировщик

Исходный код планировщика представлен в функции `Scheduler()` – 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.

```

{20} #if scmRTOS_CONTEXT_SWITCH_SCHEME == 0
{21} void OS::TKernel::Scheduler()
{22} {
{23}     TCritSect cs;
{24}
{25}     if(ISR_NestCount) return;
{26}
{27}     byte NextPrty = GetHighPriority(ReadyProcessMap);
{28}     if(NextPrty != CurProcPriority)
{29}     {
{30}         TStackItem* Next_SP = ProcessTable[NextPrty]->StackPointer;
{31}         TStackItem** Curr_SP_addr =
{32}             &(ProcessTable[CurProcPriority]->StackPointer);
{33}         CurProcPriority = NextPrty;
{34}         OS_ContextSwitcher(Curr_SP_addr, Next_SP);
{35}     }
{36} }
{37} #else
{38} void TKernel::Scheduler()
{39} {
{40}     byte NextPrty = GetHighPriority(ReadyProcessMap);
{41}     if(NextPrty != CurProcPriority)
{42}     {
{43}         SchedProcPriority = NextPrty;
{44}         RaiseContextSwitch();
{45}         if(ISR_NestCount) return;
{46}         TStatusReg sr = GetInterruptState(); EnableInterrupts();
{47}         WaitForContextSwitch();
{48}         SetInterruptState(sr);
{49}     }
{50} }
{51} #endif // scmRTOS_CONTEXT_SWITCH_SCHEME

```

Листинг 3-3 Планировщик

3.1.3.1. Планировщик с прямой передачей управления

Все действия, выполняемые внутри планировщика, не должны быть прерываемы, поэтому код этой функции выполняется в критической секции. Это достигается объявлением объекта `cs` типа `TCritSect` {4}: от этого места до конца функции прерывания будут блокированы.

Следующим шагом производится проверка того, что вызов планировщика произведен из обработчика прерывания {6}: в обработчике прерываний значение

переменной `ISR_NestCount` отлично от нуля, и если вызов произведен из обработчика прерываний, то дальнейшая работа планировщика прекращается.

Далее, если вызов производился не из обработчика прерываний, то производятся действия по перепланированию процессов. Первым делом вычисляется приоритет самого высокоприоритетного процесса, готового к выполнению (путем 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 остается 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 путем 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 (обычно реализованной на ассемблере) `os_ContextSwitcher()`, которая вызывается из планировщика {15}. Этой функции передаются в качестве аргументов два параметра:

- адрес указателя стека текущего процесса, куда будет помещен сам указатель по окончании сохранения контекста текущего процесса. {12};
- указатель стека следующего процесса {11}.

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

3.1.3.2. 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 путем непосредственного вызова функции-переключателя контекстов, а путем активирования 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. Рассмотрим ситуацию 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 нарушит 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 в необходимости перепланировки. Если такая необходимость имеется, то происходит фиксация значения приоритета в промежуточной переменной `SchedProcPriority {24}` следующего процесса и активация программного прерывания переключения контекстов.

Для того, чтобы прерывание реально произошло, необходимо, чтобы прерывания глобально были разрешены¹. Эту операцию выполняет код {27}, где записывается во временной переменной состояние регистра процессора, управляющего глобальным разрешением прерываний, после чего производится глобальное разрешение прерываний.

Далее программа входит в цикл ожидания переключения контекстов {28}. Здесь кроется довольно тонкий момент. Ведь почему бы, например, было бы просто не сделать зону разрешенных прерываний в виде пары пустых (dummy) команд (чтобы аппаратура процессора успела осуществить собственно само прерывание)? Такая реализация таит в себе следующую трудноуловимую ошибку: представим себе, что на момент разрешения прерываний {27} кроме программного прерывания были активированы одно или несколько других прерываний, причем приоритет некоторых из них выше, чем приоритет программного прерывания. При этом, естественно, поток управления будет передан в обработчик соответствующего прерывания, по окончании которого будет произведен возврат в прерванную программу. Теперь, в основной программе – т.е. внутри функции-

¹ Как правило, в этой точке программы они запрещены, т.к. работа функции-планировщика всегда выполняется в критической секции – т.е. планировщик запускается из функций-сервисов ОС, а они все работают в критической секции.

планировщика, процессор может выполнить одну или несколько инструкций¹, прежде чем может быть выполнено следующее прерывание. При этом программа сможет дойти до кода, восстанавливающего состояние регистра процессора, управляющего глобальным разрешением прерываний, что приведет 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 разрешенных прерываний используется цикл ожидания переключения контекстов. Т.е. сколько бы прерываний не стояло в очереди, пока реального переключения контекстов не произойдет, поток управления программы дальше этого цикла не пойдет.

Для обеспечения работоспособности описанного необходим критерий того, что перепланирование реально произошло. Таким критерием может выступать равенство переменных ядра **CurProcPriority** и **SchedProcPriority**. Эти переменные становятся равными друг другу (т.е. значение текущего приоритета становится равным запланированному значению) только после выполнения переключения контекстов. Реализация этого цикла – см «Листинг 3-4 Цикл ожидания переключения контекстов».

```
{1} void OS::TKernel::WaitForContextSwitch() const volatile
{2} {
{3}     byte cur;
{4}     byte sched;
{5}     do
{6}     {
{7}         cur = CurProcPriority;
{8}         sched = SchedProcPriority;
{9}     }
{10}     while (cur != sched);
{11} }
```

Листинг 3-4 Цикл ожидания переключения контекстов

Чтобы предотвратить оптимизации компилятора при реализации цикла, функция объявлена как **volatile**. Альтернативный вариант – объявить сами переменные **CurProcPriority** и **SchedProcPriority** как **volatile**. Выбор сделан в

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.

пользу функции по той причине, что не хочется подавлять оптимизации компилятора в части этих переменных (главным образом, `CurProcPriority`) в других местах программы, где в этом нет необходимости.

Естественно, прерывания разрешаются только в случае, если планировщик запущен не из прерывания. Эту проверку производит условие {26}.¹

Как видно, никаких обновлений переменных, содержащих указатели стеков и значения текущего приоритета, тут нет. Вся эта работа производится позднее при непосредственном переключении контекстов путем вызова специальной функции ядра `OS_ContextSwitchHook`. Может возникнуть вопрос: зачем такие сложности? Чтобы ответить на этот вопрос, представим себе 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_ContextSwitcher(Curr_SP_addr, Next_SP);
```

присутствует:

```
RaiseContextSwitch();  
if(ISR_NestCount) return;  
TStatusReg sr = GetInterruptState(); EnableInterrupts();  
WaitForContextSwitch();  
SetInterruptState(sr);
```

Теперь представим, что в момент, когда разрешаются прерывания, на очереди стоит еще одно или несколько прерываний, причем хотя бы одно из них более приоритетное, чем наше программное, и в этом прерывании вызывается какая-либо из функций сервисов (средств межпроцессного взаимодействия). Что при этом получится? При этом будет тоже вызван планировщик и произойдет еще 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, произойдет «наложение» планировок и нарушение целостности работы системы. Поэтому крайне важно, чтобы фактическое обновление значения

¹ Нелишне напомнить, что все обработчики прерывания в программе должны содержать объявление объекта `TISR_Wrapper` до любого вызова функции-сервиса (т.е. где имеет место вызов планировщика). Рекомендуется делать это в первом же выражении кода обработчика прерывания.

`CurProcPriority` и переключение контекстов процессов были «атомарны» – 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 же планировка и переключение контекстов несколько «разнесены» во времени. Поэтому само переключение и изменение текущего приоритета происходят непосредственно во время выполнения обработчика программного прерывания¹. В нем сразу после сохранения контекста текущего процесса вызывается функция `OS_ContextSwitchHook`, где и производится непосредственно обновление значения `CurProcPriority`, а также вычисляются адреса объектов, содержащих значения указателей стеков текущего и следующего процессов. Эти значения передаются в обработчик программного прерывания, где они используются для сохранения указателя стека текущего процесса и извлечения указателя 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 облегченная встраиваемая версия планировщика, используемая некоторыми функциями-членами объектов-сервисов, оптимизированными для применения их в ISR. Код этой версии планировщика см «Листинг 3-5 Вариант планировщика, оптимизированный для использования в ISR».

```
{1} void OS::TKernel::SchedISR()
{2} {
{3}     byte NextPrty = GetHighPriority(ReadyProcessMap);
{4}     if(NextPrty != CurProcPriority)
{5}     {
{6}         SchedProcPriority = NextPrty;
{7}         RaiseContextSwitch();
{8}     }
{9} }
```

Листинг 3-5 Вариант планировщика, оптимизированный для использования в ISR

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

3.1.4. Прерывания

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. Поскольку в текущей версии *scmRTOS* существует два варианта передачи управления, то рассмотрим их по очереди.

3.1.4.1. Прямая передача управления

Код любого обработчика прерывания, работающего по этой схеме, должен на входе выполнять следующие действия:

- сохранить контекст текущего процесса в стек этого процесса;
- проинкрементировать переменную `ISR_NestCount`;

а на выходе:

- вызвать функцию `ISR_Exit()`, которая декрементирует `ISR_NestCount` и по ее 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;
- поместить код восстановления контекста после `ISR_Exit()` на тот случай, если прерывание оказалось вложенным или прерванный процесс был самым приоритетным из готовых к выполнению, т.е. поток управления программой вернулся из `ISR_Exit()`.

Когда возникает прерывание, управление передается обработчику прерывания, который работает со стеком прерванного процесса. Это означает, что стек процесса должен иметь размер, достаточный для работы как кода самого процесса, так и кода обработчиков прерываний. Причем дополнительные требования к размеру стека определяются самым потребляющим обработчиком прерывания. Более того, эти требования распространяются на все процессы, т.е. любой процесс должен иметь размер стека с учетом возможности работы в этом стеке самого «толстого» обработчика прерывания. В случае вложенных прерываний ситуация еще более усугубляется.

Из описанного следуют два неприятных вывода:

- требования к потреблению ОЗУ для стеков процессов могут значительно возрасти даже в случае простого, «легкого» процесса. Например, если процесс использует в своем стеке всего 10 байт, то в принципе ему было бы достаточно 10 + размер контекста + доп. размер на адреса возвратов,

который определяется из глубины вложенности вызова функций. А так ему еще нужно учесть вызов обработчика прерываний с его потреблением и глубиной вложенности. Эти требования к дополнительному размеру стека могут значительно превосходить то, что потребляет сам процесс;

- оценка требуемого размера стека для каждого процесса значительно усложняется, т.к. любой процесс может быть прерван любым обработчиком прерывания. В случае вложенных прерываний ситуация может стать катастрофической – могут возникать различные комбинации из вложенных прерываний с разными требованиями к размеру стека. При этом для гарантии правильной работы нужно закладывать размер стека для каждого процесса по наихудшему варианту (оценка которого тоже может быть непростой задачей).

Для обхода этой ситуации можно использовать отдельный стек для обработчиков прерываний. Т.е. при входе в обработчик прерывания после сохранения контекста прерванного процесса происходит переключение стеков: указатель стека прерванного процесса сохраняется в стеке процесса, а текущим стеком становится специальный стек прерываний. Это происходит только при прерывании основной программы, т.е. при входе в первый обработчик прерывания – при вложенных прерываниях этого делать не нужно, т.е. все вложенные обработчики прерываний работают в том же стеке. При выходе из обработчика прерывания также производится анализ вложенности, и если обработчик прерывания последний, то происходит передача управления наиболее приоритетному из готовых к выполнению процессов путем переключения стеков (со стека прерываний на стек процесса), восстановления контекста процесса и перехода к выполнению кода процесса.

Таким образом, имеется возможность изолировать стеки процессов от стека прерываний, что позволяет как сэкономить ОЗУ, которое, как уже говорилось, является самым дефицитным ресурсом в однокристальных МК, так и значительно упростить оценку размера стека для каждого процесса.

В **scmRTOS** применяется отдельный стек для прерываний¹.

Для упрощения использования и переносимости код, выполняемый на входе и выходе обработчиков прерываний, «обёрнут» в макросы **OS_ISR_ENTER()** и **OS_ISR_EXIT()**. Поскольку действия, выполняемые этими макросами платформеннозависимые, оба макроса определены в **OS_Target.h**.

Макрос **OS_ISR_ENTER()** выполняет следующие действия:

- сохраняет контекст;

¹ В версиях 1.xx существует, также, вариант без отдельного стека, но он, в силу описанных недостатков, не является основным и вниманию не предлагается.

- вызывает функцию ядра `ISR_Enter()` (см Листинг 3-6), которая сохраняет указатель стека текущего процесса в его стеке путем вызова низкоуровневой функции `OS_ISR_ProcessStackSave()`, возвращающей указатель стека прерываний;
- помещает значение возвращенного функцией `OS_ISR_ProcessStackSave()` указателя стека прерываний в аппаратный указатель стека процессора, завершая тем самым процедуру переключения стека процесса на стек прерываний.

```

{1} word ISR_Enter()
{2} {
{3}     if(ISR_NestCount++ == 0)
{4}     {
{5}         TStackItem** Curr_SP_addr =
{6}             &(ProcessTable[CurProcPriority]->StackPointer);
{7}         return OS_ISR_ProcessStackSave(Curr_SP_addr);
{8}     }
{9}     else
{10}         return 0;
{11} }

```

Листинг 3-6 Функция входа в прерывание

Макрос `OS_ISR_EXIT()` в свою очередь выполняет следующие действия:

- вызывает функцию ядра `ISR_Exit()` (см Листинг 3-7), которая в случае выхода из обработчика прерывания в основную программу производит извлечение указателя стека наиболее приоритетного (из готовых к выполнению) процесса, и путем вызова низкоуровневой функции `OS_ISR_Exit_ContextRestorer()` восстанавливает контекст этого процесса и передает ему управление;
- помещает вслед за вызовом `ISR_Exit()` код восстановления контекста, необходимый для случаев выхода из вложенных прерываний.

```

{1} void OS::Kernel::ISR_Exit()
{2} {
{3}     TCritSect cs;
{4}
{5}     if(--ISR_NestCount) return;
{6}
{7}     byte NextProcPriority = GetHighPriority(ReadyProcessMap);
{8}     TStackItem* Next_SP =
{9}         ProcessTable[NextProcPriority]->StackPointer;
{10}     CurProcPriority = NextProcPriority;
{11}     OS_ISR_Exit_ContextRestorer(Next_SP);
{12} }

```

Листинг 3-7 Функция выхода из прерывания

Для реализации механизма перепланирования процессов в прерывании пользователь должен позаботиться о том, чтобы первой строкой в коде обработчика прерывания был макрос `OS_ISR_ENTER()`, а последней строкой – макрос

`OS_ISR_EXIT()`. Это очень важный момент, невыполнение этого условия может повлечь (и несомненно повлечет) за собой «падение» системы.



ВАЖНОЕ ЗАМЕЧАНИЕ. При использовании этого механизма организации прерываний может возникнуть неприятность, заключающаяся в том, что компилятор в ряде случаев «самостоятельно» вставляет код, модифицирующий указатель стека¹. Это происходит обычно, когда в теле самого обработчика написан какой-то (пользовательский, естественно) код, требующий создания автоматических объектов в стеке. Если эта вставка компилятором кода, модифицирующего указатель стека, происходит ДО сохранения контекста процесса и переключения стеков, то последствия самые катастрофические – система «упадет»!

Чтобы избежать такой ситуации, нужно не «провоцировать» компилятор использованием автоматических объектов прямо в теле обработчика прерывания. Лучше вынести всю функциональность в отдельную функцию, которую и вызывать из обработчика прерывания. Накладные расходы на вызов функции очень небольшие по сравнению с затратами на сохранение контекста и переключение стеков, т.ч. лучше на этом не экономить. Зато у компилятора не будет причин для «самовольных» манипуляций с указателем стека. В любом случае неплохо бы убедиться по сгенерированному коду (в файле листинга), что код обработчика прерывания начинается с сохранения контекста, что перед ним нет никаких «неожиданных» инструкций.

Здесь может возникнуть возражение: «А чё такие ограничения!?!». Ответ прост: «Языки C/C++ ничего не знают о таких вещах, как прерывания, стеки и операционные системы. Поэтому компилятор действует в рамках Стандарта, который регламентирует, как и положено, только поведение языковых конструкций. Для потребностей, которые не входят в компетенцию языка, но все же необходимы для работы и достижения эффективности, вводят специальные расширения языка². Исходя из этого, для полной гарантии получения требуемого кода есть один путь – писать обработчики прерываний полностью на ассемблере, хотя это, мягко говоря, не слишком удобно. Так что, есть выбор – либо писать все на ассемблере, либо выполнять пару простых правил и контролировать указанный момент (хотя при выполнении упомянутых правил никаких эксцессов возникать не должно – это скорее рекомендация, куда смотреть, если вдруг что-то не работает).»

¹ Имеет на это полное право.

² Являющиеся нестандартными, и, поэтому, очень отличающимися от компилятора к компилятору.

Для упрощения использования можно применить уже описанный механизм «обёрток», т.е. классов, которые в конструкторе выполняют первую, подготовительную часть работы, а в деструкторе – завершающую. В состав *scmRTOS* входит специальный класс `TISR_Wrapper`, предназначенный для упрощения и повышения безопасности использования. Достаточно в коде обработчика прерывания первой строкой объявить объект этого класса, и всю остальную работу сделает компилятор. К сожалению, не со всеми компиляторами можно использовать этот способ, т.к. очень важно, чтобы и конструктор и деструктор были встроенными, а некоторые компиляторы подавляют встраивание. Более подробно см «Глава 6 Порты».

3.1.4.2. Передача управления на основе программного прерывания

При использовании этого варианта подход к определению и использованию прерываний совсем другой. По сути он мало отличается от обычного использования прерываний на целевом МК, что является несомненным преимуществом данного варианта.

В рассматриваемом варианте требования к оформлению обработчика прерываний значительно упрощаются – фактически они сводятся тоже к созданию на входе объекта `TISR_Wrapper`, но никаких дополнительных платформенно-зависимых требований нет. Т.е. нет требования, чтобы конструктор `TISR_Wrapper` был встраиваемым, нет требований, чтобы перед сохранением контекста компилятор не вставлял инструкции, модифицирующие указатель стека – словом, все значительно проще. Не нужно объявлять сам обработчик прерываний с какими-то дополнительными расширениями (типа `__raw`) и т.д. – иными словами, дополнительных требований почти никаких нет. Единственное требование – создание на входе в ISR объекта `TISR_Wrapper`. Код этого класса для рассматриваемого варианта передачи управления – см «Листинг 3-8 Класс-«обертка» обработчика прерываний»

```
{1}     class TISR_Wrapper
{2}     {
{3}     public:
{4}         INLINE  TISR_Wrapper()    { Kernel.ISR_NestCount++; }
{5}         INLINE  ~TISR_Wrapper()  { Kernel.ISR_NestCount--; }
{6}     };
```

Листинг 3-8 Класс-«обертка» обработчика прерываний

Таким образом, как видно, все действия сводятся только к модификации переменной ядра `ISR_NestCount`, значение которой анализируется в планировщике для определения, произошел ли вызов планировщика из основной программы или из обработчика прерываний. Очевидно, что никаких особых требований к этому коду нет. Главное, чтобы объект этого класса был создан до вызова любой функции-члена объектов сервисов.

Благодаря тому, что при входе в обработчик прерываний не происходит полного сохранения контекста, потребление ОЗУ стека процесса также значительно скромнее. В наихудшем случае, если имеет место вызов невстраиваемой функции, компилятор сохранит `scratch`¹ регистры². Кроме того, желательно избегать вызовов невстраиваемых функций из обработчиков прерываний, т.к. даже частичное сохранение контекста ухудшает характеристики и по скорости, и по коду. В связи с этим в текущей версии *scmRTOS* у некоторых объектов межпроцессного взаимодействия появились специальные дополнительные облегченные функции для использования их в обработчиках прерываний. Функции эти являются встраиваемыми и используют облегченную версию планировщика, которая также является встраиваемой. Подробнее об этом см «Глава 5 Средства межпроцессного взаимодействия».

Исходя из этого, не возникает острой необходимости в отдельном стеке прерываний и переключении на него. Эта память может быть использована, например, под стек системного процесса `IdleProcess`.



ЗАМЕЧАНИЕ. В текущей версии *scmRTOS* при использовании передачи управления на основе программного прерывания вложенные прерывания не поддерживаются. Точнее, не поддерживаются вложенные прерывания, где задействуются механизмы ОС – главным образом, касающиеся планировки процессов. Остальные прерывания могут быть использованы в обычном порядке.

Отказ от вложенных прерываний обусловлен желанием упростить и ускорить код ОС, вызываемый в ISR, а вложенность прерываний, сигнализирующих события, никаких значимых преимуществ не дает – такие прерывания могут спокойно выполняться по очереди, а собы-

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. Иногда группу `preserved` еще называют `local`.

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.

тия все равно будут обработаны по окончании всех прерываний в коде основной программы.

3.1.5. Достоинства и недостатки способов передачи управления

Оба способа имеют свои достоинства и недостатки. Достоинства одного способа передачи управления являются недостатками другого и наоборот. Рассмотрим их подробнее.

3.1.5.1. 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 быстроедействие по сравнению с вариантом программного прерывания, т.к. в последнем случае имеются дополнительные накладные расходы на вызов `OS_ContextSwitchHook`.

У варианта с прямой передачей управления существует два основных недостатка. Первое – это необходимость во всяких специальных платформеннозависимых требованиях по оформлению прерываний – специальные нестандартные ключевые слова вроде `__raw`¹. Фактически при отсутствии таких расширений обработчик прерываний приходится делать в виде обычной функции с размещением вектора прерывания в отдельном ассемблерном файле вручную.

Здесь же существует очень важное требование, чтобы обработчик прерывания всегда начинался с инструкций сохранения контекста текущего процесса, чтобы перед ними не было инструкций, модифицирующих стек. А именно это и случается, когда в коде обработчика прерывания пользователь заводит объекты, размещаемые компилятором в стеке, или если уровень оптимизации при компиляции недостаточно высок. Практика показала, что почти все обращения пользователей к автору связаны именно с этим моментом. И хотя при соблюдении пра-

¹ Конечно, при определении обработчика прерываний в любом случае используются расширения языка – ключевые слова `__interrupt`, прагмы `#pragma interrupt` и т.д., но эти расширения в том или ином виде всегда присутствуют в любом пакете, и никаких дополнительных требований и сложностей с их использованием нет, чего нельзя сказать о том же `__raw` (и подобном), которое есть разве что в очень ограниченном количестве пакетов Embedded Workbench фирмы IAR Systems и в пакетах GCC в виде `__attribute__((naked))`.

вил и рекомендаций все, как правило, работает нормально, все равно полной гарантии нет и компилятор в любом случае имеет право нарушить функционирование, т.е. все равно имеется жесткая зависимость от реализации, что является серьезным недостатком данного подхода. Полную гарантию в этом случае может дать только реализация ISR полностью на ассемблере, но это весьма ухудшает удобство использования (особенно, когда весь проект полностью реализуется на C/C++), поэтому этого очень желательно избежать.

Вторым не менее серьезным недостатком является тот факт, что на входе в обработчик прерываний всегда сохраняется полный контекст. Что в этом плохого? Дело в том, что сохранение контекста нужно только тогда, когда в прерывании реально происходит перепланировка и при выходе из ISR управление передается в другой процесс. Но далеко не всегда ситуация именно такая. Рассмотрим пример.

Представим, что МК участвует в обмене данными через UART. Обмен производится пакетами, состоящими из заголовка, «тела» пакета, где заключена собственно передаваемая информация, и трейлера, содержащего контрольную сумму. Логика работы такова, что сначала принимается весь пакет, а затем управление получает процесс, ожидающий пакет и обрабатывающий заключенную в нем информацию. Совершенно очевидно, что при приеме заголовка и «тела» пакета нет никакой необходимости в передаче управления процессу, т.е. никакого перепланирования реально происходить не будет – перепланирование с последующей передачей управления процессу, ожидающему пакет, будет иметь место только после приема трейлера, проверки контрольной суммы и в случае отсутствия ошибок¹. Таким образом, реально полное сохранение контекста требуется только один раз на весь пакет, но при рассматриваемом подходе будет происходить каждый раз при приеме очередного символа приемником UART. Т.е. налицо очень значительный оверхед, который весьма снижает производительность системы и очень ограничивает скорость передачи по UART'у.

Еще одним недостатком является то, что каждый ISR включает в себя код по полному сохранению и восстановлению контекста, что влечет за собой накладные расходы по размеру кода программы.

¹ Как ошибок контрольной суммы, так и других – например, ошибок о типе пакета, переданном в заголовке.

3.1.5.2. Передача управления на основе программного прерывания

Этот вариант лишен обоих вышеописанных недостатков. Для организации ISR здесь не требуется абсолютно никаких специальных мер по сравнению с обычным способом организации обработчиков прерываний на данной платформе. Единственным требованием, как уже было сказано, является включение в код ISR объекта `TISR_Wrapper`, который модифицирует внутреннюю переменную ядра `ISR_NestCount`, значение которой анализируется в планировщике с целью определить, не из обработчика ли прерываний вызван планировщик. Сам по себе этот код никаких требований не предъявляет и влечет никакого платформеннозависимого поведения.

Благодаря тому, что сам по себе ISR выполняется обычным образом и никакой перепланировки из него не делается, полное сохранение контекста также не производится, что значительно сокращает накладные расходы и повышает производительность системы. В рассмотренном выше примере с приемом пакетов через UART накладные расходы по частичному сохранению контекста полностью будут определяться реализацией пользовательского кода – насколько много в нем будет использовано временных объектов, требующих сохранения регистров процессора в стеке, или размещенных в стеке, будут ли использоваться вызовы невстраиваемых функций и т.д. Фактически, при грамотном написании кода, степень оверхеда можно свести к минимуму, а чтобы не испортить картину вызовом невстраиваемой функции-члена сервисного объекта межпроцессного взаимодействия рекомендуется пользоваться специальными, облеченными, встраиваемыми версиями таких функций – об этом подробнее см «Глава 5 Средства межпроцессного взаимодействия».

Также благодаря тому, что нет необходимости в полном сохранении контекста процесса на входе в прерывание, требования к потреблению стека значительно снижены, а учитывая, что размер стека любого процесса рассчитан на полное сохранение в нем контекста, размера стека текущего процесса должно хватать и для нужд прерываний, что позволяет не использовать отдельный стек прерываний и не тратить время на переключение указателя стека на стек прерываний. В этом случае, также, память, используемую для стека прерываний, можно использовать для служебных целей – например, для стека системного процесса `IdleProcess`, что и реализовано в ряде случаев.

При приеме пакета в случае отсутствия ошибок в обработчике прерывания инициируется перепланировка, которая будет произведена сразу после выхо-

да из обработчика прерывания путем вызова обработчика программного прерывания, где и произойдет полное переключение контекстов – т.е. ровно один раз на пакет.

Благодаря тому, что переключение контекстов производится всегда в одном и том же месте и не на уровне пользовательского кода, эта операция – ISR программного прерывания, – реализована полностью на ассемблере, что, во-первых, исключает побочные эффекты поведения компилятора, которые могут иметь место в случае организации ISR при прямом способе передачи управления, что уже было описано выше, во-вторых, дает возможность реализовать этот критичный по времени выполнения код с максимальной эффективностью, т.к. реализация здесь полностью делается вручную.

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 тут может возникнуть необходимость в корректировке кода ОС в части target. Более подробно о текущей реализации программного прерывания, использующего аппаратное прерывание процессора, на конкретных платформах см «Глава 6 Порты».

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

3.1.5.3. Выводы

Учитывая вышеприведенный анализ достоинств и недостатков обоих способов передачи управления, рекомендуется по возможности использовать передачу управления, реализованную на основе программного прерывания. В этом случае:

- нет жестких требований к оформлению кода ISR;
- отсутствует возможность неправильной работы из-за платформеннозависимого поведения компилятора при реализации кода ISR;
- имеется возможность получить более стройный дизайн и значительно лучшую производительность в ISR;
- отсутствует дублирующийся код сохранения/восстановления контекстов в каждом ISR;

- отсутствует настоятельная необходимость в отдельном стеке прерываний, что позволяет использовать память стека прерываний для нужд ОС и сокращает общее потребление памяти операционной системой.

Использование прямой передачи управления оправдано только 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.

3.1.6. 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».

3.1.7. Системный таймер

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

В качестве системного таймера используется обычно один из аппаратных таймеров процессора¹.

Функциональность системного таймера реализуется в функции ядра `systemTimer()`. Существует две реализации этой функции – одна относится к способу прямой передачи управления, другая – к передаче управления с помощью программного прерывания. Основные отличия касаются способов вызова и обусловлены соображениями эффективности. В случае прямой передачи управления вызов функции системного таймера производится обычным способом – тут нет противопоказаний т.к. контекст все равно полностью сохранен, стек переключен на стек прерываний, т.е. все накладные расходы имеются в полной мере и вызов

¹ Для этого подходит самый простой (без «наворотов») таймер. Единственное принципиальное требование к нему – он должен быть способен генерировать прерывание по переполнению, - это «умеют» почти все таймеры. Желательно, также, чтобы имелась возможность управлять величиной периода переполнения, чтобы подобрать подходящую частоту системных тиков.

функции почти ничего не добавляет. Этот вариант - см «Листинг 3-9 Системный таймер (прямая передача управления)».

```
{1} void OS::TKernel::SystemTimer()
{2} {
{3}     #ifdef scmRTOS_SYSTEM_TICKS_ENABLE
{4}     SysTickCount++;
{5}     #endif
{6}
{7}     for(byte i = 0; i < scmRTOS_PROCESS_COUNT; i++)
{8}     {
{9}         TBaseProcess* p = ProcessTable[i];
{10}
{11}         if(p->Timeout > 0)
{12}             if(--p->Timeout == 0) SetProcessReady(p->Priority);
{13}     }
{14} }
```

Листинг 3-9 Системный таймер (прямая передача управления)

Как видно из исходного кода, действия очень простые:

1. если разрешен счетчик тиков, то переменная счетчика инкрементируется {4};
2. далее в цикле проверяются значения таймаутов всех зарегистрированных процессов, и если значение проверяемой переменной не равно 0¹, тогда значение декрементируется и проверяется на 0. При равенстве (после декремента) 0 (т.е. таймаут данного процесса истек) 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 истек, то сразу при выходе из прерывания он получит управление.

В случае же передачи управления с помощью программного прерывания, само по себе прерывание не нуждается в полном сохранении контекста, поэтому имеется возможность получить более легкий и быстрый обработчик прерывания. В этом случае вызов невстраиваемой функции нецелесообразен, т.к. повлечет за собой частичное сохранение контекста (scratch регистры). По этой причине определение функции системного таймера вынесено в заголовочный файл и использует специальную облегченную встраиваемую версию планировщика. Код функции системного таймера с передачей управления путем программного прерывания

¹ Это означает, что процесс находится в ожидании с таймаутом.

СМ «Листинг 3-10 Системный таймер (передача управления с помощью программного прерывания)».

```

{1} void OS::TKernel::SystemTimer()
{2} {
{3} #ifdef scmRTOS_SYSTEM_TICKS_ENABLE
{4}     SysTickCount++;
{5} #endif
{6}
{7}     bool Reschedule = false;
{8}     for(byte i = 0; i < scmRTOS_PROCESS_COUNT; i++)
{9}     {
{10}         TBaseProcess* p = ProcessTable[i];
{11}
{12}         if(p->Timeout > 0)
{13}         {
{14}             if(--p->Timeout == 0)
{15}             {
{16}                 SetProcessReady(p->Priority);
{17}                 Reschedule = true;
{18}             }
{19}         }
{20}     }
{21}     if(Reschedule) SchedISR();
{22} }

```

Листинг 3-10 Системный таймер (передача управления с помощью программного прерывания)

Здесь отличия лишь в том, что при определении истечения таймаута процесса, если процесс должен быть переведен в готовые к выполнению, устанавливается значение переменной-флага **Reschedule** {17}, что означает, что возможно требуется перепланировка, и при выходе из функции системного таймера при значении **Reschedule** равном **true** вызывается планировщик (та самая облегченная встраиваемая версия) {21}.



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 наклад-

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?

ные расходы (по времени выполнения)¹ 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 похожая тенденция, поэтому разумно ожидать, что для малых процессоров период системного тика меньше, чем для больших – в больших системах и накладные расходы БОльшие в виду, как правило, БОльшей загрузки более мощного процессора и, как следствие, меньшей его «поворотливости».

3.2. Структура

Ядро содержит минимально необходимый набор данных и функций. Оно представляет собой класс, содержащий следующие члены-данные²:

- **CurProcPriority** – переменная, содержащая номер приоритета текущего 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)³;
- **ReadyProcessMap** – карта процессов, готовых в выполнении. Каждый бит этой переменной соответствует тому или иному процессу. Лог. 1 указывает на то, что процесс готов к выполнению, лог. 0 – на то, что процесс не готов;
- **ProcessTable** – массив указателей на процессы, зарегистрированные в системе;
- **PrioMaskTable** – массив битовых паттернов-масок, используемый для манипуляций с объектами **TProcessMap**;
- **ISR_NestCount** – переменная-счетчик входов в прерывания. При каждом входе она инкрементируется, при каждом выходе декрементируется;
- **SysTickCount** – переменная-счетчик 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.

¹ В виду малого количества процессов, а также простого и быстрого планировщика.

² Объекты, помеченные ‘*’, присутствуют только в варианте с использованием передачи управления на основе программного прерывания.

³ 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 больше, чем размер переменной для хранения приоритета.

- `CS_StackData*` – структура, содержащая адреса указателей стеков текущего и следующего процессов. Является объектом, указатель на который возвращает функция `OS_ContextSwitchHook`. Его данные используются внутри обработчика программного прерывания.

При использовании варианта с прямой передачей управления существует еще одна (или две – в зависимости от того, сколько у процесса стеков) переменная, имеющая отношение к функционированию ядра, хотя и в силу специфики не входящая в структуру ядра – это `OS_ISR_SP`. В этой переменной хранится значение адреса стека, используемого в прерываниях.

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, все-таки отличаются – у них разные размеры стеков и разные значения приоритетов (которые, как мы помним, задаются статически). В версиях 1.xx для определения процесса использовался макрос. В текущей версии в виду наличия поддержки компилятором шаблонов C++ сделан отказ от использования макросов препроцессора, как менее безопасного средства. Основное преимущество перед макросами, получаемое от использования шаблонов, состоит в том, что при шаблонной реализации компилятор производит анализ этого кода и делает соответствующий контроль типов со значительно более внятной диагностикой, нежели это может быть в случае препроцессорного макроса. Главная проблема макросов – это то, что пользователь видит не тот код, который подается на вход компилятору, из-за чего иногда совсем неочевидно, в чем состоит ошибка.

4.1.2. 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 отдельных стека – один для данных, другой для адресов возвратов. *scmRTOS* поддерживает такую возможность, позволяя ассоциировать с каждым процессом 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`.

4.1.3. Таймауты

Каждый процесс имеет специальную переменную `Timeout` для контроля за временным поведением процесса: при ожиданиях событий с таймаутами или при «спячке».

4.1.4. Приоритеты

Каждый процесс имеет также еще поле данных, содержащее приоритет процесса. Это поле является как бы 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 – см стр. 31.

4.1.5. 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 перейдет в «спячку» до тех пор, пока его не разбудит, например, какой-либо другой процесс с помощью функции `OS::ForceWakeUpProcess()`. Если функция вызывается с аргументом (целое число в диапазоне от 1 до 65535), то процесс будет «спать» указанное количество тиков

системного таймера, после чего будет «разбужен», т.е. приведен в состояние готового к выполнению. В этом случае «спячка» также может быть прервана другим процессом или обработчиком прерывания с помощью функций `OS::WakeUpProcess()`, `OS::ForceWakeUpProcess()`.

4.2. Создание и использование процесса

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

4.2.1. Определение типа процесса

Тип конкретного процесс определен с помощью специального шаблона `process`: см «Листинг 4-1 Определение шаблона типа процесса»

```
{1} template<TPriority pr, word stack_size>
{2} class process : public TBaseProcess
{3} {
{4} public:
{5}     process() : TBaseProcess(&Stack[stack_size/sizeof(TStackItem)]
{6}                               , pr
{7}                               , (void (*)())Exec)
{8}     {
{9}     }
{10}
{11}     OS_PROCESS static void Exec();
{12}
{13} private:
{14}     TStackItem Stack[stack_size/sizeof(TStackItem)];
{15} };
```

Листинг 4-1 Определение шаблона типа процесса

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

1. стек процесса `stack` с размером `stacksz`. Размер задается в байтах;
2. 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 программой не покидал процессной функции, в противном случае, в силу того, что эта функция не была вызвана обычным образом, при выходе из нее поток управления попадет, грубо говоря, в неопределенные адреса, что повлечет неопределенное поведение программы (хотя на практике поведение вполне определенное – программа не работает! ☺);
- использовать функцию `os::WakeUpProcess()` нужно с осторожностью и внимательно, а `os::ForceWakeUpProcess()` с особой осторожностью, т.к. неаккуратное использование может привести к несвоевременной «побудке» спящего (отложенного) процесса, что может привести к коллизиям в межпроцессном взаимодействии.

Глава 5

Средства межпроцессного взаимодействия

— Мальчик, тебя как зовут?

— Чего? ...

— Ты что – тормоз?

— Вася.

— Что «Вася»?

— Я не тормоз.

анекдот.

К средствам межпроцессного взаимодействия **scmRTOS** относятся:

- `OS::TMutex;`
- `OS::TEventFlag;`
- `OS::TChannel;`
- `OS::channel;`
- `OS::message;`

5.1. OS::TMutex

Семафор Mutex (от Mutual Exclusion – взаимное исключение), как видно из названия, служит для организации взаимного исключения доступа к нему. Т.е. 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, так чтобы к нему имелся доступ различных частей программы, 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 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 взаимного исключения: 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, т.е. производить обращение через семафор.

Для реализации бинарных семафоров этого типа в *scmRTOS* определен класс `OS::TMutex`, см «Листинг 5-1 OS::TMutex».

```
{1} class TMutex
{2} {
{3} public:
{4}     TMutex() : ProcessMap(0), ValueTag(0) { }
{5}     void Lock();
{6}     void Unlock();
{7}
{8}     INLINE void LockSoftly()
{9}     {
{10}         TCritSect cs;
{11}         if(ValueTag) return; else Lock();
{12}     }
{13}
{14}     INLINE bool IsLocked() const
{15}     {
{16}         TCritSect cs;
{17}         if(ValueTag) return true; else return false;
{18}     }
{19}
{20} private:
{21}     TProcessMap ProcessMap;
{22}     TProcessMap ValueTag;
{23}
{24} };
```

Листинг 5-1 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.

Как видно из интерфейса класса, с объектами этого типа можно делать четыре вещи:

`TMutex::Lock()`

1. захватывать. Функция `Lock()` выполняет эту задачу. Если до этого семафор не был захвачен, то внутреннее значение будет переведено в состояние, соответствующее захваченному, и поток управления вернется обратно вызываемую функцию. Если семафор был захвачен, то процесс будет переведен в ожидание, пока семафор не будет освобожден, а управление отдано ядру;

`TMutex::Unlock()`

2. освобождать. Это выполняет функция `Unlock()`. Она переводит внутреннее значение в состояние, соответствующее освобожденному семафору, и проверяет, не ждет ли какой-либо другой процесс этого семафора. Если ждет, то управление будет отдано ядру, которое произведет 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 произведет, объект останется в том же состоянии;

`TMutex::LockSoftly()`

3. «мягко» захватывать. Функция `LockSoftly()`. Разница с простым захватом состоит в том, что захват будет иметь место только в случае, если семафор свободен. Например, нам нужно поработать с ресурсом, но кроме этого у нас еще есть куча другой работы. Пытаясь захватить «жестко», можно встать в 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 отдает высокоприоритетный;

TMutex::IsLocked()

4. проверять. Для этого предназначена функция `IsLocked`. Функция просто проверяет значение и возвращает `true`, если семафор захвачен, и `false` в противном случае. Иногда бывает удобно использовать семафор с 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 5-2 Пример использования OS::TMutex»

```

{1} OS::TMutex Mutex;
{2} byte buf[16];
{3} ...
{4} OS_PROCESS 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 5-2 Пример использования OS::TMutex



ЗАМЕЧАНИЕ. В текущей версии *scmRTOS* существует некоторое отличие функциональности `OS::TMutex::Unlock()` от аналогичной функции из прежних версий – ранее любой процесс, вызвавший функцию снятия блокировки реально производил разблокировку. Такое поведение, как оказалось, в ряде случаев не отвечает потребностям дизайнера – иногда бывает удобно построить функцию процесса, например, в виде конечного автомата, так, что блокировка мутекса происходит в нескольких местах (состояниях автомата), а общая разблокировка в одном. При этом может возникнуть ситуация, когда произойдет вызов функции разблокировки мутекса, заблокированного другим процессом, что, естественно, является ошибкой. В текущей версии *scmRTOS* этот нюанс учтен – разблоки-

ровать семафор может только тот процесс, который его (семафор) заблокировал.

5.2. OS::TEventFlag

При работе программы часто возникает необходимость в синхронизации между процессами. Т.е. например, один из процессов для выполнения своей работы должен дожидаться 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 получают 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, как только событие произойдет, передать управление процессу.

Эту функциональность в *scmRTOS* реализуется с помощью объектов `OS::TEventFlag` (флаг события). Определение класса: см «Листинг 5-3 OS::TEventFlag».

```

{1} class TEventFlag
{2} {
{3}     enum TValue
{4}     {
{5}         efOn = 1,    // prefix 'ef' means: "Event Flag"
{6}         efOff= 0
{7}     };
{8}
{9} public:
{10}    TEventFlag (TValue init_val = efOff)
{11}        : ProcessMap(0)
{12}        , Value(init_val)
{13}    {
{14}    }
{15}
{16}    bool Wait (word timeout = 0);
{17}    void Signal();
{18}    void Clear () { TCritSect cs; Value = efOff; }
{19}
{20} #if scmRTOS_CONTEXT_SWITCH_SCHEME == 1
{21}     INLINE inline void SignalISR();
{22} #endif // scmRTOS_CONTEXT_SWITCH_SCHEME
{23}
{24}    bool IsSignaled()
{25}    {
{26}        TCritSect cs;
{27}        if(Value == efOn)
{28}            return true;
{29}        else
{30}            return false;
{31}    }
{32}
{33} private:
{34}    TProcessMap ProcessMap;
{35}    TValue Value;
{36}
{37} };

```

Листинг 5-3 OS::TEventFlag

С объектами этого типа можно делать четыре вещи:

TEventFlag::Wait()

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 еще не произошло), то процесс переводится в состояние ожидания этого флага (события) и управление отдается ядру, которое, перепланировав процессы, запустит следующий. Если вызов функции был произведен 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 `OS::ForceWakeUpProcess()`. (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 65535), 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()` вернет `false`. Таким образом реализуются ожидание безусловное и ожидание с таймаутом;

`TEventFlag::Signal()`

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 очередности приоритетов);

`TEventFlag::SignalISR()`

3. вариант вышеописанной функции, оптимизированный для использования в прерываниях (при использовании способа передачи управления на основе программного прерывания). Функция является встраиваемой и использует специальную облегченную встраиваемую версию планировщика. Этот вариант нельзя использовать вне кода обработчика прерываний.

`TEventFlag::Clear()`

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`;

`TEventFlag::IsSignaled()`

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

Пример использования флага события – см «Листинг 5-4 Использование `TEventFlag`».

В этом примере один процесс (`Proc1`) ждет 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 ВСЕ процессы, ожидавшие этот флаг, будут переведены в состояние готовых к выполнению. Другими словами, все, кто ждал, дождался. Управление они, конечно, получают в порядке очередности их приоритетов, но событие не будет «пропущено» ни одним процессом, успевшим встать на ожидание, независимо от приоритета процесса.

```

{1} OS::TEventFlag EFlag;
{2} ...
{3} //-----
{4} OS_PROCESS 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} OS_PROCESS void Proc2::Exec()
{23} {
{24}     for(;;)
{25}     {
{26}         ...
{27}         if( ... ) EFlag.Signal();
{28}         ...
{29}     }
{30} }
{31} //-----

```

Листинг 5-4 Использование TEventFlag

5.3. OS::TChannel

Объект `OS::TChannel` представляет собой кольцевой буфер, позволяющий безопасно с точки зрения межпроцессного взаимодействия записывать в него и считывать из него байты. Следует отметить, что в текущей версии *scmRTOS* появилось другое средство с аналогичным назначением – шаблон `OS::channel`, на

основе которого можно создать канал для объектов произвольного типа и, к тому же, обладающий большей функциональностью. `OS::TChannel` оставлен в составе ОС по соображениям совместимости¹. Вторая причина состоит в том, что ряд пользователей испытывают недоверие к шаблонам. Чтобы не лишать их возможности передачи данных через каналы, `OS::TChannel` оставлен в составе ОС ☺.

Класс `TChannel` не является законченным типом канала: в его определении не выделена память под собственно массив буфера. Таким образом, при создании объекта-канала, нужно выделить память под буфер и передать адрес этой памяти, а также ее размер в качестве аргументов конструктору канала. Чтобы эта память не «болталась» снаружи, удобно поступить так: создать производный от `TChannel` класс, в котором и выделить память под буфер.

Для уменьшения писанины и предотвращения ошибок удобно использовать макрос `DefineChannel(Name, Capacity)`, где нужно указать имя типа канала и его емкость в байтах. После этого нужно объявить объект этого типа.

Определение класса `OS::TChannel` – см «Листинг 5-5 `OS::TChannel`».

```

{1} class TChannel
{2} {
{3} public:
{4}     TChannel (byte* buf, byte size) : Cbuf(buf, size) { }
{5}     void Push (byte x);
{6}     byte Pop ( );
{7}     void Write(const byte* data, const byte count);
{8}     void Read (byte* const data, const byte count);
{9}     byte GetCount() const { TCritSect cs; return Cbuf.get_count(); }
{10}
{11} private:
{12}     TProcessMap PushersProcessMap;
{13}     TProcessMap PopersProcessMap;
{14}     TCbuf Cbuf;
{15}
{16} private:
{17}     void CheckWaiters (TprocessMap& pm);
{18} };
    
```


Листинг 5-5 `OS::TChannel`

Использование объектов класса `OS::TChannel` сводится к следующим действиям:

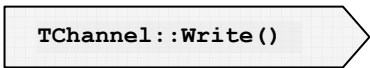
`TChannel::Push()`

1. записать байт. Функция `Push()` записывает один байт 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


¹ Хотя полной совместимости между версиями 1.xx и 2.xx нет, все-таки имеется цель свести несовместимые моменты к минимуму, дабы облегчить процедуру переноса рабочих проектов на новую версию.



`TChannel::Pop()`



`TChannel::Write()`



`TChannel::Read()`

- появится место. Когда место появится, байт будет записан. После этого делается проверка, не ждет ли какой-нибудь процесс данных в канале;
2. извлечь байт. Функция `Pop()` извлекает один байт 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. Когда данные появляются, байт извлекается, и делается проверка, не ждет ли какой-нибудь процесс с целью записать в канал;
 3. записать несколько байт из памяти по адресу. То же самое, что и записать байт, только ожидание продолжается до тех пор, пока в канале не появится достаточно места. Остальная логика работы такая же;
 4. извлечь из канала несколько байт и поместить их в память по адресу. То же самое, что и извлечь байт, только ожидание продолжается до тех пор, пока в канале не окажется нужное количество байт.



ЗАМЕЧАНИЕ. Если обмен через канал производится не парой процессов, а большим количеством одновременно, то нужно проявлять осторожность, т.к. запись в канал и чтение из него разными процессами одновременно приведет к чередованию байтов, и это может вызвать ошибки при чтении – читающий процесс может получить «неожиданные» данные. То же самое и при чтении – например, если один читающий процесс вытеснен более приоритетным, который тоже читает из этого же канала, то вытесняющий процесс просто «украдет» байты, предназначенные для первого процесса, из канала. Т.е. одновременно писать (и/или читать) в один канал двум и более процессам можно, но это не очень хорошая идея. Лучше либо разносить это по времени, либо использовать канал только для работы пары процессов. Это, в общем, и есть основное применение каналов.

5.4. OS::message

`OS::message` представляет собой C++ шаблон для создания объектов, реализующих обмен между процессами путем передачи структурированных данных. `OS::message` очень похож на `OS::TEventFlag` и отличается главным обра-

зом тем, что кроме самого флага содержит еще и объект произвольного типа, составляющий собственно тело сообщения.

Определение шаблона – см «Листинг 5-6 Шаблон OS::message».

```

{1} //-----
{2} template<class T>
{3} class message
{4} {
{5} public:
{6}     message()                : ProcessMap(0), NonEmpty(false)      { }
{7}     message(const T& msg)    : ProcessMap(0), NonEmpty(false), Msg(msg) { }
{8}
{9}     void send();
{10}    bool wait (word timeout = 0);
{11}
{12}    INLINE bool is_non_empty() const { TCritSect cs; return NonEmpty; }
{13}    INLINE void reset          ()    { TCritSect cs; NonEmpty = false; }
{14}
{15}    void operator=(const T& msg) { TCritSect cs; Msg = msg; }
{16}    operator      T() const     { TCritSect cs; return Msg; }
{17}
{18} #if scmRTOS_CONTEXT_SWITCH_SCHEME == 1
{19}     INLINE inline void sendISR();
{20} #endif // scmRTOS_CONTEXT_SWITCH_SCHEME
{21}
{22} private:
{23}     TProcessMap ProcessMap;
{24}     bool NonEmpty;
{25}     T Msg;
{26} };
{27} //-----
    
```

Листинг 5-6 Шаблон OS::message

Как видно из листинга, реализация шаблона достаточно проста. Над объектами, созданными по шаблону, можно производить следующие действия:

message::send()

1. посылать сообщение¹. Функция `void send()` выполняет эту операцию, которая сводится к переводу процессов, ожидающих сообщение, в состояние готовых к выполнению и вызову планировщика;

message::sendISR()

2. вариант вышеописанной функции, оптимизированный для использования в прерываниях (при использовании способа передачи управления на основе программного прерывания). Функция является встраиваемой и использует специальную облегченную встраиваемую версию планировщика. Этот вариант нельзя использовать вне кода обработчика прерываний.

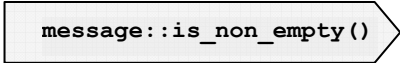
message::wait()

3. ждать сообщение². Для этого предназначена функция `void wait (word timeout = 0)`, которая проверяет, не пустое ли сообщение и если


¹ Аналог функции `OS::TEventFlag::Signal()`.

² Аналог функции `OS::TEventFlag::Wait()`.

не пустое, то возвращает `true`, если пустое, 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 пошлет сообщение или текущий процесс не будет «разбужен» с помощью функции `os::ForceWakeUpProcess()`¹. Если в качестве аргумента было указано целое число в диапазоне от 1 до 65535, 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 произойдет до истечения таймаута, что означает, приход сообщения до того, как таймаут истек, то функция вернет `true`. В противном случае, т.е. если таймаут истечет до того, как сообщение будет послано, функция вернет `false`;



```
message::is_non_empty()
```



```
message::reset()
```

4. проверить сообщение. Функция `bool is_non_empty()` возвращает `true`, если сообщение было послано, и в `false` противном случае;
5. сбросить сообщение. Функция `void reset()` сбрасывает сообщение, т.е. переводит сообщение в состояние `empty`;

5.5. OS::channel

`os::channel` представляет собой C++ шаблон для создания объектов, реализующих кольцевые буфера² 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 задается на этапе инстанцирования шаблона в пользовательском коде. Шаблон канала `os::channel` основан на шаблоне кольцевого буфера

¹ В последнем случае нужно проявлять крайнюю осторожность.

² Функционально это FIFO.

`usr::ring_buffer<class T, word size, class S = byte>`, определенного в библиотеке¹, входящей в состав поставки *scmRTOS*.

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.

Новые возможности каналов дают эффективное средство для построения очередей сообщений. Причем 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 нее данных и чтении их оттуда;
- простоту использования – не нужно выполнять ручное преобразование типов, сопряженное с необходимостью держать в голове кучу лишней информации;
- 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, часто связанная с динамическим выделением памяти². 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 це-

¹ Таким образом, библиотека также является неотъемлемой частью ОС – без нее часть средств работать не будет.

² Хотя это и не обязательно – можно выделить память под объекты-сообщения статически, если условия задачи и ресурсы это позволяют.

ликом помещаются в очередь-канал, память под них в этом случае будет выделена непосредственно при создании канала.

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 С Примеры использования.

Определение шаблона – см «Листинг 5-7 Определение шаблона OS::channel»

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

Листинг 5-7 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 путем инстанцирования шаблона `OS::channel`:

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

Этот код объявляет объект-канал `DataQueue` для передачи объектов типа `TData`, емкость 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 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:

- | | |
|--|--|
| <div style="border: 1px solid black; padding: 5px; display: inline-block; margin-bottom: 20px;"> <code>channel::push()</code> </div> | <p>1. записать элемент в конец очереди¹. Функция <code>void push(const T& item)</code> 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 ждет ли какой-нибудь процесс данных из канала;</p> |
| <div style="border: 1px solid black; padding: 5px; display: inline-block; margin-bottom: 20px;"> <code>channel::push_front()</code> </div> | <p>2. записать элемент в начало очереди. Функция <code>void push_front(const T& item)</code> 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 ждет ли какой-нибудь процесс данных из канала;</p> |
| <div style="border: 1px solid black; padding: 5px; display: inline-block;"> <code>channel::pop()</code> </div> | <p>3. извлечь элемент из начала очереди. Функция <code>bool pop(T& item, word timeout = 0)</code> 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,</p> |

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

если таймаут был указан¹. 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`. Когда данные появляются, элемент извлекается и делается проверка, не ждет ли какой-нибудь процесс с целью записать в канал. Следует обратить внимание на тот факт, что при вызове этой функции данные, извлеченные из канала, передаются не путем копирования при возврате функции, а через передачу объекта по ссылке. Это обусловлено тем, что значение возврата занято для передачи результата таймаута;

`channel::pop_back()`

4. извлечь элемент из конца очереди. Функция `bool pop_back (T& item, word timeout = 0)` 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;

`channel::write()`

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;

`channel::read()`

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, word 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.

`channel::get_count()`

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

`channel::get_free_size()`

8. получить величину количества элементов, которые можно записать в канал, т.е. под них есть свободное место. Функция `S get_free_size() const` является встраиваемой, поэтому эффективность ее работы максимально высока.

`channel::flush()`

9. произвести очистку канала. Функция `void flush()` очищает буфер путем вызова `usr::ring_buffer<class T, word size, class S = byte>::flush()` и проверяет, не ждет ли ка-

¹ Т.е. вызов был с передачей вторым аргументом целого числа в диапазоне 1..65535, которое и задает величину таймаута в тиках системного таймера.

кой-либо процесс пока в канале не появится свободное место. После вызова этой функции в канале нет ни одного элемента.

Простой пример использования – см «Листинг 5-8 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} OS_PROCESS void TProc1::Exec()
{11} {
{12}     ...
{13}     TCmd cmd = { cmdSetCoeff2, 12 };
{14}     CmdQueue.push(cmd);
{15}     ...
{16} }
{17} //-----
{18} OS_PROCESS 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 5-8 Пример использования очереди на основе канала.

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

5.6. 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 взаимного исключения 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.

С другой стороны, глобальные массивы в обрамлении мутексов и флагов событий можно использовать и для передачи сообщений. В текущей версии **scmRTOS** этот подход уже не столь актуален как это было в версиях 1.xx в виду появления шаблонных сервисов, которые позволяют сочетать удобство и безопасность использования с эффективностью. Правда, надо отметить, что эффективность реализации типов на шаблонах в значительной степени зависит от качества используемого компилятора. Но сегодня это уже не является острой проблемой – современные компиляторы обладают высоким качеством кодогенерации в т.ч. и при инстанцировании объектов по шаблону.

Каналы **OS::TChannel** также можно использовать для передачи сообщений – достаточно придумать формат сообщений. Недостатком такого подхода является отсутствие статического контроля типов. Как уже было сказано, это средство межпроцессного взаимодействия в текущей версии является устаревшим и оставлено, главным образом, только для уменьшения количества проблем при переносе рабочих проектов на новую версию. Вместо них рекомендуется использовать каналы произвольных типов данных **OS::channel**.

Можно использовать сообщения для синхронизации по событиям вместо флагов событий – такой подход имеет смысл в случае, если вместе с флагом нужно еще 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 запрещенных 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 взаимного исключения 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 определенные особенности. Например, очевидно, что использовать `TMutex::Lock()` внутри обработчика прерывания является достаточно плохой идеей, т.к., во-первых, мутексы 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 приведет только к тому, что процесс, прерванный данным прерыванием, просто будет переведен состояние ожидания в неподходящей и непредсказуемой точке. Фактически процесс будет переведен в неактивное состояние, из которого его вывести можно будет только с помощью функции `ForceWakeUpProcess`. 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::SignalISR`, `message<T>::send`, `message<T>::sendISR`, 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 использовать только `TEventFlag::Signal` и `message<T>::send` (`TEventFlag::SignalISR` и `message<T>::sendISR` в случае варианта передачи управления с помощью программного прерывания). И очень не рекомендуется использовать все остальные средства межпроцессного взаимодействия – они предназначены для использования на уровне процессов.

Глава 6

Порты

6.1. Общие замечания

Ввиду больших отличий как целевых архитектур, так и средств разработки под них, возникает необходимость в специальной адаптации кода ОС¹. Результатом этой работы является платформеннозависимая часть, которая в совокупности с общей частью и составляет порт под ту или иную платформу. В настоящей главе будут рассмотрены, главным образом, платформеннозависимые части.

На сегодняшний день **scmRTOS** v2.xx имеет поддержку трех целевых архитектур: MSP430 (Texas Instruments) – IAR EW430, AVR (Atmel) – IAR EWAVR и Blackfin (Analog Devices) – VisualDSP++. Итого, существует 3 порта ОС – по одному на каждую платформу.

В заголовочном файле `OS_Target.h` для каждой платформы определен ряд макросов и синонимов встроенных типов.

В файле `OS_Target_cpp.cpp` определен конструктор процесса, системный процесс `IdleProcess`, а также обработчик прерывания от системного таймера `SystemTimer_ISR()`.

В файле `OS_Target_asm.ext` определены низкоуровневые функции запуска первого процесса, переключения контекстов, восстановления контекста при выходе из прерывания, переключения стеков процесса и прерывания, а также обработчик программного прерывания, используемого как переключатель контекстов при соответствующей схеме передачи управления.

Прерывания, как ключевой элемент организации быстрой реакции на события, заслуживают пристального внимания.

¹ Это касается не только ОС, но и других многоплатформенных программ.

В случае прямой передачи управления прерывания можно разделить на две категории: с поддержкой ОС и без поддержки ОС.

Прерывания с поддержкой ОС – это прерывания, при выполнении кода обработчиков которых, происходит сохранение контекста текущего процесса и переключение указателя стека на стек прерываний. А при выходе происходит перепланирование и запуск наиболее приоритетного процесса.

Прерывания без поддержки ОС – это обычные прерывания. Их использование нежелательно и должно производиться с осторожностью. Причины для этого две:

1. при таком прерывании не происходит перепланирования процессов, следовательно время реакции на событие тут ничем не лучше, чем при использовании кооперативных ОС или вообще при работе без ОС;
2. при таком прерывании обработчик прерывания работает в стеке прерванного процесса, поэтому если обработчик прерывания потребляет значительные ресурсы для своей работы, это потребует выделения этих ресурсов в стеке каждого процесса, в противном случае, может произойти ошибка работы с памятью и система, скорее всего, «упадет».

В случае передачи управления на основе программного прерывания такого разделения нет, тут все намного проще и для достижения эффективности используются обычные подходы. Со стороны *scmRTOS* имеется ряд средств для поддержки написания эффективного кода, заключающихся, главным образом, в ряде дополнительных функций, оптимизированных для использования их в ISR, о чем выше уже неоднократно говорилось.

6.2. MSP430

6.2.1. Обзор

Этот МК имеет простую, стройную архитектуру, что сделало подготовку платформеннозависимой части для него несложной задачей.

IAR EW430 использует один стек для данных и адресов возвратов.

Нумерация приоритетов в данном порте традиционно идет по возрастанию: `pr0` – наивысший приоритет, по мере возрастания номеров, уровень приоритета уменьшается.

6.2.2. Источник тактовой частоты

В качестве источника MCLK используется DCO, настроенный на максимальную частоту, которая составляет приблизительно 5 МГц. Настройка выполняется с помощью макроса `EXECUTE_PLATFORM_SPECIFIC_STUFF()`. Как уже говорилось, пользователь может отключить использование кода, подставляемого этим макросом, и написать вместо этого более подходящий для него (пользователя) код.

6.2.3. Системный таймер

В качестве системного таймера используется Watchdog Timer в режиме интервального таймера. Источник тактовой частоты таймера – MCLK. Настройка периода переполнения таймера (т.е. системного тика) такова, что при выбранном источнике тактовой и его частоте, этот период равен приблизительно 1.6 мс. Запуск таймера реализован с помощью макроса `START_SYSTEM_TIMER()`. Использование этого макроса, также, может быть отключено пользователем. В этом случае пользователь должен сам позаботиться о запуске системного таймера, в противном случае все функции, связанные с системным временем, работать не будут.

6.2.4. Передача управления

Передача управления в данном порте реализована обоими способами – способ можно выбрать на этапе конфигурирования.

В случае с прямой передачей управления никаких дополнительных особенностей нет.

В случае с передачей управления на основе программного прерывания ситуация следующая. MSP430, к сожалению, не имеет отдельного программного прерывания, поэтому для реализации функций перепланирования и переключения контекстов используется одно из аппаратных прерываний. Из всех

имеющихся в наличие аппаратных прерываний наиболее удобным на текущий момент представляется прерывание от Analog Comparator. Никаких проблем с использованием не возникает – нет необходимости даже включать сам компаратор, достаточно установить флаг прерываний в соответствующем регистре специальных функций и аппаратно процессор генерирует соответствующее прерывание. Если это прерывание разрешено и прерывания глобально разрешены, то поток управления оказывается в обработчике этого прерывания, а этот обработчик есть не что иное, как обработчик программного прерывания, где и происходит переключение контекстов процессов.

Для инициирования переключения контекстов (т.е. генерации программного прерывания) используется функция:

```
// set flag and enable interrupt
inline void RaiseContextSwitch() { CACTL1 |= 0x03; }
```

6.2.5. Прерывания

6.2.5.1. Прямая передача управления

При использовании прерываний с поддержкой ОС возникает требование, чтобы компилятор не пытался сохранять регистры, т.к. это делает ОС. Для того, чтобы подавить у компилятора «желание» сохранять/восстанавливать регистры, вместе со словом `__interrupt` используется слово `__raw`.

В случае прерываний с поддержкой ОС можно (и нужно) использовать класс-«обертку» `TISR_Wrapper`, который упрощает использование и уменьшает вероятность ошибок. Пример обработчика прерываний – см «Листинг 6-1 Обработчик прерывания Timer_A Input Capture (прямая передача управления)».

```
{1} OS_INTERRUPT void TTimer_A::Capture_ISR()
{2} {
{3}     OS::TISR_Wrapper ISR;
{4}     __enable_interrupt();
{5}
{6}     ... // some code
{7}
{8}     Timer_A_Complete.Signal();
{9} }
```

Листинг 6-1 Обработчик прерывания Timer_A Input Capture
(прямая передача управления)

6.2.5.2. Передача управления на основе программного прерывания

Здесь, как раз, никаких особенностей нет, все достаточно просто.

Определение обработчика любого прерывания должно быть похоже на нижеприведенное – см «Листинг 6-2 Обработчик прерывания Timer_B Overflow (передача управления на основе программного прерывания)».

Ввиду того, что полное сохранение контекста в ISR не используется, переключения на отдельный стек прерываний не происходит, а эта область ОЗУ используется в качестве стека системного процесса `IdleProcess`.

```
{1}      #pragma vector = TIMERB0_VECTOR
{2}      __interrupt void Timer_B_ISR()
{3}      {
{4}          OS::TISR_Wrapper ISR;
{5}
{6}          Timer_B_Ovf.Signal();
{7}      }
```

Листинг 6-2 Обработчик прерывания Timer_B Overflow (передача управления на основе программного прерывания)

Здесь главное и единственное требование состоит в том, чтобы объявление объекта `ISR {4}` было сделано до вызова функции `Signal () {6}`.

6.3. AVR

6.3.1. Обзор

С AVR'ом все оказалось гораздо «интереснее». AVR имеет два весьма «кривых» момента с точки зрения ОС.

Во-первых, у него слишком много регистров. Используется из них едва ли половина, но при переключении контекстов таскать нужно все 32. Для улучшения ситуации в предыдущих версиях было принято решение заблокировать часть регистров от использования их компилятором. Компилятор из пакета IAR EWAVR дает возможность это сделать с помощью ключа `--lock_regs`, AVR-GCC – с помощью ключа `-ffixed-<register>`, и в тех портах были заблокированы регистры `r4-r15`.

К сожалению, и этот подход не лишен недостатков. Во-первых, возникает необходимость в перекомпиляции run-time библиотек. Это, как показала практика, не всем нужно и не всем удобно – для пересборки библиотек нужны их исходные тексты, которые также не всегда и не всем доступны, да и сам процесс сборки библиотек не слишком простой в виду большого количества исходных файлов и опций компилятора. Во-вторых, собранные с ограниченным набором регистров библиотеки не подходят для всех проектов. В частности, при использовании с пакетом EWAVR арифметики с 64-битными double, используется часть регистров, которые попадают под блокирование, потому что использование арифметики с такими типами невозможно совместно с использованием библиотеки с заблокированными регистрами.

Исходя из этих предпосылок, текущий вариант порта выполнен без блокировки регистров. Это, разумеется, привело к тому, что размер контекста увеличился на 12 байт и время переключения контекста также возросло на $12 \cdot 4^1 = 48$ тактов. Зато отсутствуют вышеперечисленные недостатки.

Второй «кривой» момент AVR – это его убогий указатель стека. Как известно, указатель стека в AVR – это пара регистров специальных функций, расположенная в области IO регистров. Из-за этого функциональность указателя стека просто никакая – он может только содержать адрес «верхушки» стека (top of stack). Никаких продвинутых возможностей по косвенной адресации, адресной арифметике у него нет. Поэтому разработчики IAR Systems приняли правильное решение использовать в качестве стека данных отдельную область ОЗУ, адресуемую регистровой парой `r28:r29` – `x-pointer`². При этом прежний стек никуда не делся – аппаратура МК все равно его использует для сохранения адресов возвратов из подпрограмм и прерываний.

Такая ситуация характерна для любого процессора, у которого указатель стека такой же убогий, как у AVR.

Таким образом, при использовании компиляторов от IAR Systems имеется два стека: один стек для данных, второй – для адресов возвратов.

¹ 2 такта на сохранение каждого регистра и два такта на восстановление.

² Интересно, куда они смотрели, когда два норвега разрабатывали сам МК, ведь по информации от производителя AVR разрабатывался как процессор, ориентированный на применение его с языками высокого уровня и в тесном сотрудничестве с фирмой-разработчиком компиляторов ЯВУ для МК???

scmRTOS учитывает этот момент и поддерживает по два стека для каждого процесса, причем размер каждого стека можно задать индивидуально, исходя из требований ресурсоемкости прикладной задачи.

При создании процесса нужно кроме описанных в «4.2.1 Определение типа процесса» параметров указать еще размер стека возвратов. Определение шаблона типа процесса с отдельным стеком возвратов представлен на «Листинг 6-3 Определение типа процесса с отдельным стеком возвратов.». Пример определения типа процесса с отдельным стеком возвратов:

```
typedef OS::process<OS::pr0, 80, 32> TParams;
```

определили тип процесса с именем **TParams** и двумя стеками. Размер стека данных равен 80 байт, размер стека возвратов – 32 байта (что эквивалентно 16 уровням вложенности вызова подпрограмм). Общее потребление ОЗУ под стеки для этого процесса составляет $80 + 32 = 112$ байт.

```
{1} template<TPriority pr, word stack_size, word rstack_size>
{2} class process : public TBaseProcess
{3} {
{4} public:
{5}     process() : TBaseProcess( &Stack[stack_size/sizeof(TStackItem)]
{6}                             , &RStack[rstack_size/sizeof(TStackItem)]
{7}                             , pr
{8}                             , (void (*)())Exec)
{9}     {
{10}    }
{11}
{12}     OS_PROCESS static void Exec();
{13}
{14} private:
{15}     TStackItem Stack [stack_size/sizeof(TStackItem)];
{16}     TStackItem RStack[rstack_size/sizeof(TStackItem)];
{17} };
```

Листинг 6-3 Определение типа процесса с отдельным стеком возвратов.

Еще один важный момент. При использовании МК, у которых объем флешки превышает объем 64 кбайт (ATmega103, ATmega128 и др.), в них используется регистр-указатель страницы RAMPZ. Для корректной работы ОС этот регистр также нуждается в сохранении/восстановлении при переключении процессов, т.е. входит в контекст процесса. В **scmRTOS** этот момент учитывается и реализована соответствующая поддержка. Со стороны пользователя необходимо указать конкретный тип МК (например, --cpu=m128) для компилятора и определить макрос HAS_RAMPZ для ассемблерных файлов – например, передавать его через командную строку -DHAS_RAMPZ.

Нумерация приоритетов в данном порте традиционно идет по возрастанию: `pr0` – наивысший приоритет, по мере возрастания номеров, уровень приоритета уменьшается.

6.3.2. Системный таймер

В качестве системного таймера используется `Timer/Counter0` со значением предделителя, равным 64, что дает при 8 МГц тактовой частоты период переполнения 2.048 мс. Настойку таймера можно изменить путем определения пользовательской функции `SystemStartUserHook()`, вызываемой при старте системы.

Использование этого макроса может быть отключено пользователем. В этом случае пользователь должен сам позаботиться о запуске системного таймера, в противном случае все функции, связанные с системным временем, работать не будут.

6.3.3. Передача управления

Передача управления в данном порте реализована обоими способами – способ можно выбрать на этапе конфигурирования.

В случае с прямой передачей управления никаких дополнительных особенностей, как и в случае с `MSP430`, нет.

В случае с передачей управления на основе программного прерывания ситуация похожа на ту же, что и с `MSP430`. `AVR`, к сожалению, не имеет отдельного программного прерывания, поэтому для реализации функций перепланирования и переключения контекстов используется одно из аппаратных прерываний. Из всех имеющихся в наличие аппаратных прерываний наиболее удобным на текущий момент, также как и в случае с `MSP430`, представляется прерывание от `Analog Comparator`.

К сожалению, и тут ситуация несколько хуже, нежели с `MSP430` – простой записью в регистр специальных функций здесь дело не обходится. Чтобы сгенерировать прерывание от аналогового компаратора в `AVR`, необходимо включить сам компаратор, и сгенерировать на его входе сигналы – только в этом случае взводится флаг прерываний от аналогового компаратора. Для того, чтобы добиться этого с наименьшими затратами, использован следующий под-

ход: на один из входов подключается внутренний источник опорного напряжения, вывод МК, соединенный со вторым входом компаратора, настраивается на вывод. При необходимости сгенерировать прерывание достаточно переключить уровень выходного сигнала на этом выводе. Описанный способ не годится для всех МК семейства AVR, а только лишь для ATmega, т.к. встроенный источник опорного напряжения у аналогового компаратора имеется не во всех МК, но использовать *scmRTOS* с другими членами семейства AVR вряд ли удастся. Еще одним недостатком описанного подхода является то, что под нужны ОС уходит один за выводов МК, что иногда может оказаться неприемлемым. Возможно, в ряде случаев использование прерывания от аналогового компаратора в качестве программного прерывания окажется невозможным или нецелесообразным. В этой ситуации необходимо немного переработать код, изменив функцию инициирования программного прерывания и откорректировав вектор прерываний. Например, представляется несложной задачей, скажем, переделать порт на использование в качестве программного прерывания любого из внешних прерываний.

После инициирования прерывания путем изменения логического уровня на выводе МК аппарататура процессора генерирует соответствующее прерывание. Если это прерывание разрешено и прерывания глобально разрешены, то поток управления оказывается в обработчике этого прерывания, а этот обработчик есть не что иное, как обработчик программного прерывания, где и происходит переключение контекстов процессов.

Для инициирования переключения контекстов по вышеописанной схеме (т.е. генерации программного прерывания) используется функция:

```
inline void RaiseContextSwitch()
{
    PORTB |= (1 << 3); PORTB &= ~(1 << 3); // set flag
}
```

6.3.4. Прерывания

Поскольку используются два способа передачи управления, то и подхода к организации прерываний тоже два.

6.3.4.1. Прямая передача управления

При использовании прерываний с поддержкой ОС возникает требование, чтобы компилятор не пытался сохранять регистры, т.к. это делает ОС. Начиная с EWAVR v4.12, наконец, ввели новое ключевое `__raw`, поэтому появи-

лась возможность полностью определять ISR прямо в коде C/C++, как это делается для MSP430, поэтому подход, применявшийся ранее, где требовалось описывать ISR как обычную функцию с `extern "C"` связыванием и заданием вектора прерывания вручную на ассемблере, теперь устарел и не используется.

В случае прерываний с поддержкой ОС желательно использовать класс-«обертку» `TISR_Wrapper`, который упрощает использование и уменьшает вероятность ошибок.

В качестве примера обработчика прерываний можно рассмотреть обработчик прерываний системного таймера – см «Листинг 6-4 Обработчик прерывания системного таймера».

```
{1} OS_INTERRUPT void OS_SystemTimer_ISR()  
{2}     {  
{3}         TISR_Wrapper ISR;  
{4}         Kernel.SystemTimer();  
{5}  
{6}         #ifdef scmRTOS_SYSTIMER_NEST_INTS_ENABLE  
{7}         MCU_ENABLE_INTERRUPT();  
{8}         #endif  
{9}  
{10}        #ifdef scmRTOS_SYSTIMER_HOOK_ENABLE  
{11}        SystemTimerUserHook();  
{12}        #endif  
{13}     }
```

Листинг 6-4 Обработчик прерывания системного таймера

6.3.4.2. Передача управления на основе программного прерывания

Определение прерывания по этой схеме очень просто и не налагает никаких особых требований. См «Листинг 6-5 Пример определения прерывания (передача управления на основе программного прерывания)».

```
{1}     #pragma vector=TIMER1_OVF_vect  
{2}     __interrupt void Timer1_overflow_ISR()  
{3}     {  
{4}         OS::TISR_Wrapper ISRW;  
{5}  
{6}         ... // some code  
{7}  
{8}         Timer1_Ovf.Signal();  
{9}     }
```

Листинг 6-5 Пример определения прерывания (передача управления на основе программного прерывания)

Как и в случае с MSP430 существует единственное требование, чтобы объявление объекта `ISRW {4}` было сделано до вызова функции `Signal () {8}`.

6.4. Blackfin

6.4.1. Обзор

Blackfin – процессор, изначально рассчитанный на использование его под управлением операционных систем, поэтому в его составе имеется ряд средств для поддержки ОС. Сюда относятся наличие, например, режимов User и Supervisor и поддержка программного прерывания. Для нас самым важным из них является поддержка программного прерывания.

В данном порте режим User по понятным причинам не используется, а вся работа происходит в режиме Supervisor, переключение на который происходит на этапе выполнения кода Startup.

Нумерация приоритетов в данном порте идет по убыванию: максимальное значение номера приоритета соответствует процессу с наивысшим приоритетом, по мере убывания номеров, уровень приоритета уменьшается и `pr0` – низший приоритет, соответствует системному процессу `IdleProcess`. Такая схема выбрана из соображений эффективности при вычислении приоритетов процессов – порядок битов в объектах `TProcessMap` получается такой, что наивысший приоритет соответствуют старшим битам, а это, в свою очередь, позволяет эффективно находить номер самого приоритетного процесса из готовых к выполнению. Для этого используется инструкция процессора `signbits`. Реализацию функции нахождения наивысшего приоритета процесса из готовых к выполнению – см «Листинг 6-6 Функция нахождения наивысшего приоритета процесса из готовых к выполнению».

```
{1}     byte OS::GetHighPriority(TProcessMap pm)
{2}     {
{3}         byte pr;
{4}         asm
{5}         (
{6}             " %0.1 = signbits %1; " :
{7}             "=d" (pr) :
{8}             "d" (pm)
{9}         );
{10}        return 30 - pr;
{11}    }
```

Листинг 6-6 Функция нахождения наивысшего приоритета процесса из готовых к выполнению

Вообще, такой порядок задания приоритетов целесообразен для любой архитектуры, имеющей инструкции для поддержки операций с плавающей точкой – вроде упомянутой `signbits`.

6.4.2. Передача управления

Поскольку данный процессор имеет поддержку программного прерывания, прямая передача управления в данном порте не реализована за ненадобностью. В качестве программного прерывания для переключения контекстов используется Software Interrupt 1 (IVG14). Инициирование переключения контекстов производится с помощью вызова функции:

```
// raise software interrupt
inline void RaiseContextSwitch() { asm(" raise 14;"); }
```

6.4.3. Прерывания

Определение прерываний мало отличается от принятого в среде VisualDSP++ и содержит то же самое требование, что и порты для MSP430 и AVR, а именно – чтобы внутри ISR был определен объект `TISR_Wrapper`, и сделано это было до любого вызова функций-членов сервисов *scmRTOS* (объектов межпроцессного взаимодействия) – см «Листинг 6-7 Пример обработчика прерывания».

```
{1}     EX_INTERRUPT_HANDLER(Timer0_ISR)
{2}     {
{3}         OS::TISR_Wrapper ISR;
{4}
{5}         MMR16(TIMER_STATUS) = TIMIL0; // clear flag
{6}
{7}         ef_timer0.SignalISR();
{8}     }
```

Листинг 6-7 Пример обработчика прерывания

Ввиду того, что полное сохранение контекста в ISR не используется, переключения на отдельный стек прерываний не происходит, а эта область ОЗУ используется в качестве стека системного процесса `IdleProcess`.

6.4.4. Платформеннозависимые действия

Платформеннозависимые действия выполняются путем вызова макроса `EXECUTE_PLATFORM_SPECIFIC_STUFF` и сводятся к регистрации двух обработчиков прерываний: от системного таймера и программного прерывания переключателя контекстов.

6.4.5. Системный таймер

В качестве системного таймера используется таймер ядра процессора (Core Timer). Поскольку существует большое разнообразие вариантов как задания периода таймера, так и задания тактовых частот процессора, настройка и запуск таймера вынесены из состава ОС и полностью находятся в ведении пользователя. Логично всю инициализацию – настройку тактовых частот, установку напряжения питания ядра, настройку и запуск таймера ядра и др., – сделать в функции `main` проекта до запуска `OS::Run`.

Глава 7

Заключение

Использование операционных систем реального времени с вытесняющим планированием в мелких 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 подхода 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 применения процессора. Например, если нужно при возникновении прерывания максимально быстро на него среагировать, чтобы внутри обработчика прерывания, к примеру, «дернуть» 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 – он для осуществления процессов в предположении, что процесс – длительный (по отношению к длительности выполнения команд процессора) промежуток времени. И требования ко времени реакции/формированию времён, сравнимых со временем выполнения команд, плохо совместимы с возможно-

стями процессора, если только он не имеет «на борту» специальных периферийных устройств, которые делают работу аппаратно.

Можно, конечно, и при использовании ОС формировать жесткие, выверенные по тактам, временные диаграммы, но при этом придется заблокировать основные механизмы ОС, т.е. ОС будет в течение известного промежутка времени неработоспособной. В любом случае, процессы и «временка» - «две сути не совместимы» ©.

* * *

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

Приложение А

История изменений

Версия	Дата	Статус	Описание
V2.03	xx.02.2006	Бета	<ol style="list-style-type: none"> 1. Класс <code>os::kernel</code> со статическими членами заменен на <code>os::tkernel</code> с обычными (нестатическими) членами. 2. Введено задание порядка следования приоритетов. 3. Введен альтернативный способ передачи управления – через программное прерывание. Альтернативная реализация планировщика, прерываний и некоторых функций сервисов. 4. Введена возможность отключать конфигурационные макросы <code>EXECUTE_PLATFORM_SPECIFIC_STUFF</code> и <code>START_SYSTEM_TIMER</code>. 5. Тип, задающий значения приоритетов процессов, перенесен внутрь пространства имен <code>os</code>. 6. Функция регистрации процессов <code>os::tkernel::RegisterProcess</code> упрощена – убраны проверки времени выполнения за бесполезностью. 7. Из функции запуска ОС <code>os::run</code> убраны проверки времени выполнения за бесполезностью. 8. Функциональность <code>os::tmutex</code> изменена: теперь снять блокировку может только процесс, заблокировавший семафор. 9. Добавлен порт для Blackfin/VisualDSP++ 4.0.
V2.02	30.01.2006	Бета	Исправлены две ошибки в реализации <code>os::tchannel</code> и <code>os::channel</code> .

V2.01	25.04.2005	Бета	Первый выпуск. Порты под EW430 v3.xx, EWAVR v4.10/4.11.
-------	------------	------	---

Приложение В

Перенос проектов с версий 1.xx

Перенос рабочих проектов с версий 1.xx на текущую требует следующий действий:

1. Переопределения объявлений процессов.
2. Переименования типа `OS::TCritSec` в `OS::TCritSect`, если он использовался в пользовательской программе.
3. Модификации кода, использующего `MailBox`'ы и `MemoryManager`, т.к. эти части *scmRTOS* версий 1.xx в версиях 2.xx не присутствуют.

Первый пункт реализуется достаточно просто – для этого нужно заменить определения типов процессов и объявления объектов процессов. Например, пусть был объявлен тип и определен объект процесса:

```
DefineProcess(TSlonProc, 100);  
...  
TSlonProc SlonProc(pr1);
```

Тогда в новом варианте это будет выглядеть:

```
typedef OS::process<OS::pr1, 100> TSlonProc;  
...  
TSlonProc SlonProc;
```

Выполнение второго пункта тривиально – операция поиска и замены имени легко реализуется средствами любого приличного программистского редактора ☺.

Третий пункт самый сложный. Здесь путь решения не поддается формальному описанию и требует творческого подхода. Самое простое и правильное – это использовать новые сервисы на шаблонах – `OS::message` и `OS::channel`. Единственное, чего не предоставляют эти два средства – это возможности размещать объекты в свободной памяти. Если все же существует настоятельная необходимость в диспетчере памяти, а под рукой нет подходящего (и нет време-

ни/желания реализовывать это собственными силами), а использованный в версии 1.xx устраивает, то не составляет труда просто перенести MemoryManager из версий 1.xx в новый проект – все исходные тексты открыты, примеры использования описаны и опыт применения уже имеется.

Приложение С

Примеры использования

С.1. Очередь сообщений на основе указателей.

Рассмотрим пример¹ использования передачи сообщений на основе очереди указателей. Традиционно для реализации очередей сообщений используются указатели `void*` совместно с ручным преобразованием типов. Этот подход обусловлен имеющимися в наличии средствами языка С. Как уже было сказано, он (подход) признан неудовлетворительным по соображениям удобства и безопасности. Поэтому, отказываясь от этого варианта использования, реализуем другой способ, который доступен благодаря использованию языка С++, где у нас есть возможность обойти недостатки очередей указателей `void*`.

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 еще 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 определенные действия на приемном конце очереди. Это достаточно легко реализуется на основе иерархии полиморфных классов² сообщений. В данном примере и будет реализован упомянутый подход.

¹ Сразу хочется принести извинения за некоторую многословность – дело в том, что реализация очереди сообщений, использованная в примере, обладает некоторой абстрактностью и людям, не знакомым с С++ и объектно-ориентированным программированием (ООП), может показаться малопонятной и запутанной. Поэтому для облегчения ситуации опишем пример более подробно и конкретно, чтобы лучше была видна основная идея и «точка приложения» подхода к практике. ☺

² Возможно, для тех, кто не знаком с этими вещами, это звучит устрашающе ☺, но на деле ничего страшного там нет, все достаточно просто – при аналогичной с реализацией на С эта цель достигается путем использования массивов указателей на функции, только в случае с С придется много делать руками, что чревато ошибками, не очень наглядно и, вследствие этого, не слишком удобно. С++ здесь просто перекладывает всю рутинную работу на компилятор, избавляя пользователя от необходимости писать низкоуровневый код с таблицами указателей на функции, их правильной

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 мы опустим этот момент, 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 проч.

* * *

Итак, представим, что наш МК является частью системы, где он выполняет сбор данных, управление исполнительными элементами, измерение сигналов от датчиков, обработку сигналов от органов управления, вывод информации на жидкокристаллический (ЖК) дисплей (LCD) – словом, все то, что часто входит в круг задач МК.

Одной из задач, возложенных на прибор, является отображение различной информации на ЖК дисплее, а также управление режимами его работы – регулировка яркости подсветки, автоматическое отключение через определенный интервал времени бездействия (т.е. когда в течение означенного промежутка времени не было никакой активности на органах управления) – например, 3 секунды, и т.д.

Далее пусть среди органов управления имеется кнопка «Режим», которая служит для переключения режимов работы прибора. При нажатии на нее происходит переход из одного режима работы в другой, что, среди прочего, сопровождается соответствующей индикацией на ЖК дисплее – скажем, в левом верхнем углу.

Пусть, также, прибор производит периодическое измерение некоего параметра – например, температуры окружающей среды, и по команде (по нажатии на соответствующую кнопку или от другого источника) отображает эту информацию также на ЖК дисплее – в верхнем правом углу.

Ну, и третья функция, попавшая в рассмотрение – принудительное¹ включение/выключение ЖК дисплея.

инициализацией и использованием. Думается, после рассмотрения данного примера, все станет понятно. ☺

¹ При отсутствии активности дисплей сам отключается, например, через 3 секунды.

Понятно, что список этих функций можно продолжить, но для примера хватит и этих, а расширить функциональность при необходимости труда не составит.

* * *

Конечно, существует много путей реализации – например, непосредственно в момент поступления внешнего сигнала выполнить все манипуляции, связанные с ЖК дисплеем. Но здесь придется позаботиться об обеспечении атомарности доступа к ресурсу (ЖКД) – использовать мутексы, а это может ухудшить реакцию на события, например, в случае, когда низкоприоритетный процесс на длительное время заблокирует доступ к ресурсу и высокоприоритетный будет вынужден простаивать.

Или посылать сообщения через очередь просто в виде данных-сигнатур и производить анализ на приемном конце. Но и тут придется городить код анализа на принимающем конце. Это, в свою очередь, повлечет необходимость обеспечить все классы-инициаторы сообщений соответствующим интерфейсом. В любом случае код разрастется, реализация «размажется».

Реализация через классы-сообщения с виртуальными функциями ничем не уступает вышеприведенным способам ни по эффективности, ни по замыслу. Здесь работу с ЖК дисплеем можно вынести в относительно низкоприоритетный процесс, чтобы он не мешал более высокоприоритетным, в которых просто при необходимости ставить сообщения в очередь и заниматься дальше своей более приоритетной, чем загрузка и управление ЖКД, работой. Т.е. по сути здесь имеет место постановка в очередь действий, которые не требуют немедленного выполнения. Одновременно решается проблема с атомарностью доступа к ЖК дисплею.

* * *

Итак, имеем три типа разных сообщений:

- Отобразить текущий режим.
- Отобразить измеренную температуру.
- Включить/выключить дисплей.

Для реализации требований поступим следующим образом: сначала создадим абстрактный класс сообщения:

```
class TMsg
{
public:
    virtual void apply() = 0;
};
```

Здесь определен только интерфейс, т.е. то, что можно делать с сообщением – а именно: сообщение можно применить¹.

Далее на основе этого абстрактного класса-сообщения создадим типы конкретных сообщений:

```
class TMsgMode : public TMsg
{
public:
    virtual void apply();
};
class TMsgTemperature : public TMsg {...};
class TMsgOnOff       : public TMsg {...};
```

И так далее. Естественно, в каждом конкретном классе-сообщении мы должны определить функцию `void apply()`, которая и будет выполнять необходимые действия. В итоге можем написать такой конечный код², см «Листинг 7-1 **Пример использования очереди сообщений**».

¹ Возможно, не самое удачное имя, но оно отражает смысл использования сообщений в данном контексте – здесь сообщения не читаются, не посылаются, они применяются по назначению в пункте назначения.

² Код, естественно, не полный – здесь просто иллюстрируется основной принцип.

```

{1} //-----
{2} // Статическая функция-член класса TLCD, служит для загрузки
{3} // символов в ЖКД
{4} //
{5} void TLCD::LoadData
{6}     (word const row,           // номер строки
{7}     word const col,         // номер столбца
{8}     const char* const data1) // адрес строки символов
{9} {
{10}     ... // реализация загрузки строки символов в ЖК дисплей
{11} }
{12} //-----
{13} // Статическая функция-член класса TLCD, служит для
{14} // включения и выключения отображения информации на ЖКД
{15} //
{16} void TLCD::OnOff(bool On)
{17} {
{18}     if(On) ... // код, включающий отображение на ЖКД
{19}     else ... // код, выключающий отображение на ЖКД
{20}     ...      // опциональный вспомогательный код
{21} }
{22} //-----
{23} //-----
{24} class TMsg           // абстрактный класс-сообщение
{25} {
{26} public:
{27}     virtual void apply() = 0;
{28} };
{29} //-----
{30} class TMsgMode : public TMsg // отобразить текущий режим
{31} {
{32} public:
{33}     TMsgMode();
{34}     void apply();
{35}     ...           // остальной интерфейс
{36} private:
{37}     char CurrName[8]; // строка, хранящая имя текущего режима
{38} };
{39} void TMsgMode::apply()
{40} {
{41}     TLCD::LoadData(1, 1, CurrName); // row = 1, col = 1
{42} }
{43} //-----
{44} //-----
{45} class TMsgTemperature : public TMsg // отобразить температуру окр. среды
{46} {
{47} public:
{48}     TMsgTemperature();
{49}     void apply();
{50}     ...           // остальной интерфейс
{51} private:
{52}     char Value[6]; // строковое значение температуры, напр.: -14°C
{53} };
{54} void TMsgTemperature::apply()
{55} {
{56}     TLCD::LoadData(1, 150, Value); // row = 1, col = 150
{57} }
{58} //-----
{59} //-----
{60} class TMsgOnOff : public TMsg // включить/выключить ЖКД
{61} {
{62} public:
{63}     TMsgOnOff() : On(false) { }

```

¹ Для простоты используется C-строка с терминальным нулем (\0).

```

{64}     void apply();
{65}     ...           // остальной интерфейс
{66} private:
{67}     bool On; // значение, определяющее включить или выключить ЖКД
{68} };
{69} void TMsgOnOff::apply()
{70} {
{71}     TLCD::OnOff(On);
{72} }
{73} //-----
{74} //-----
{75} OS::channel<TMsg*, 12> MsgQueue; // Очередь сообщений
{76} ...
{77} TMsgMode MsgMode(...);
{78} ...
{79} TMsgTemperature MsgTemperature(...);
{80} ...
{81} TMsgOnOff MsgOnOff;
{82} ...
{83} //-----
{84} //-----
{85} OS_PROCESS void TExtControlsProc::Exec()
{86} {
{87}     for(;;)
{88}     {
{89}         ...
{90}         TMsg* msg
{91}         if(...)
{92}         {
{93}             ...
{94}             msg = &MsgMode;
{95}         }
{96}         else if(...)
{97}         {
{98}             ...
{99}             msg = &MsgOnOff;
{100}        }
{101}        MsgQueue.push(msg); // поместить сообщение в очередь
{102}        ...
{103}    }
{104} }
{105} //-----
{106} //-----
{107} OS_PROCESS void TMonitorProc::Exec()
{108} {
{109}     for(;;)
{110}     {
{111}         ...
{112}         TMsg* msg = &MsgTemperature;
{113}         MsgQueue.push(msg); // поместить сообщение в очередь
{114}         ...
{115}     }
{116} }
{117} //-----
{118} //-----
{119} OS_PROCESS void TLCDProc::Exec()
{120} {
{121}     for(;;)
{122}     {
{123}         TMsg* msg;
{124}         if( MsgQueue.pop(msg, 3000/21) ) // ждать до 3-х секунд
{125}             msg->apply();           // обработать сообщение
{126}         else

```

¹ В предположении, что один тик системного таймера равен 2 мс.


```
{127}          TLCD::OnOff(false);          // выключить ЖК дисплей
{128}          }
{129} //-----
```

Листинг 7-1 Пример использования очереди сообщений на указателях

Несколько замечаний. Следует обратить внимание, что адреса объектов-сообщений присваиваются указателю на базовый класс. Это ключевой момент – на этом основан механизм работы виртуальных функций, являющийся центральным при реализации полиморфного поведения.

Следующий момент. Можно заметить, что сами объекты-сообщения 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 нелокальный класс памяти, т.е. могли существовать между вызовами функций. А сам факт существования активного сообщения состоит не в существовании самого объекта-сообщения, а в помещении указателя с адресом объекта-сообщения в очередь.

Предметный указатель

AVR	7, 16, 26, 79, 83, 84	TChannel	24, 59, 66, 67, 76
AVR-GCC	79	TEventFlag	21, 24, 33, 59, 63, 65, 68
Blackfin	17, 26, 79, 89, 95	TISR_Wrapper	24, 82
channel		TMutex	21, 24, 33, 59, 60
clear()	74	UnlockSystemTimer()	25
get_count()	74	WakeUpProcess()	25, 57, 58
get_free_size()	74	OS_ContextSwitcher	35
pop()	73	OS_ISR_ENTER	41, 42
pop_back()	74	OS_ISR_EXIT	41, 42, 43
push()	73	OS_ISR_Exit_ContextRestorer	42
push_front()	73	OS_ISR_ProcessStackSave	42
read()	74	OS_ISR_SP	54
write()	74	OS_Kernel.cpp	23
Cooperative	12	OS_Services.cpp	23
CurProcPriority	53	OS_Start()	33
DefineChannel	67	OS_Target.h	23, 26, 32, 41, 56, 79
Exec	29	OS_Target_asm.ext	23, 79
IAR Systems	7, 16, 17, 79, 84	OS_Target_cpp.cpp	23, 79
IdleProcess	19, 22, 28, 30, 79	preemptive	12
ISR_Enter	42	ProcessTable	31, 53
ISR_Exit	40, 42	ReadyProcessMap	35, 53
ISR_NestCount	35, 40, 53	round-robin	12, 13, 15
Kernel	20, 31	RTOS	12
message	68	Runtime Config	
is_empty()	70	EXECUTE_PLATFORM_SPECIFIC	
reset()	70	_STUFF	32, 81
send()	69	START_SYSTEM_TIMER	32, 81
wait()	69	Scheduler	12, 34
MSP430	7, 14, 17, 26, 79, 80	Scheduling	12, 20, 31
Mutex	59, 60	scmRTOS	15
OS		scmRTOS.h	23
channel	22, 24, 59, 70, 76, 97	scmRTOS_DEFS.h	23
ForceWakeUpProcess()	25, 56, 57, 58, 64, 70	scmRTOS_PROCESS_COUNT	27
GetTickCount()	25	SEPARATE_RETURN_STACK	56
IsProcessSleeping()	25	SystemStartUserHook	30, 86
IsProcessSuspended()	25	SystemTimer	50
Kernel	24	SystemTimerUserHook	30
LockSystemTimer()	25	SysTickCount	53
message	22, 24, 59, 97	TBaseProcess	
Run()	25	Sleep()	21, 56
TBaseProcess	22, 24, 25, 31, 55	TChannel	
		Pop()	68

Push()	67	scmRTOS_SYSTIMER_NEST_INT	
Read()	68	S_ENABLE	29
Write()	68	Критические секции	25
TCritSect	25, 26	Межпроцессное взаимодействие ...	21
TEventFlag		операционная система	11
Clear()	65	Операционные системы	
IsSignaled()	65	proc	12
Signal()	65	Salvo	12
Wait()	64	uC/OC-II	12
TISR_Wrapper	44, 88	ОСРВ	12, 13, 93
TMutex		Передача управления с помощью	
IsLocked()	62	программного прерывания ...	35, 44,
Lock()	61	48, 83, 88	
LockSoftly()	61	планировщик	12
Unlock()	61	Планировщик	34
TProcess	26	Поддержка межпроцессного	
TProcessMap	26	взаимодействия	50
TStackItem	26	Прерывания	40, 79
Конфигурационная информация ...	29	Приоритет	56
Конфигурационные макросы		Процессы	20, 55
scmRTOS_IDLE_HOOK_ENABLE		Прямая передача управления ...	33, 40,
.....	30	46, 82, 87	
scmRTOS_PROCESS_COUNT	29	Семафоры взаимoisключения	21
scmRTOS_START_HOOK_ENABLE		Системный таймер	50, 81, 86
.....	30	Стек прерываний	41
scmRTOS_SYSTEM_TICKS_ENAB		Стек процесса	56, 57
LE	29	Таймауты	56
scmRTOS_SYSTIMER_HOOK_ENA		Флаги событий	21
BLE	30	Ядро	20, 31, 53

