SALVO RTOS
РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

ВЕРСИЯ 4.1.2
(предварительное)

ДЛЯ ВСЕХ ДИСТРИБУЦИЙ

SALVO RTOS USER'S MANUAL v.4.1.2

Перевод: Андрей Шлеенков
http://andromega.narod.ru
mailto:andromega@narod.ru

Salvo™

The RTOS that runs in tiny places.™

Перевод: Андрей Шлеенков • http://andromega.narod.ru • mailto:andromega@narod.ru

Быстрый старт

Благодарим вас за приобретение Salvo – "The RTOS that runs in tiny places™". Компания Pumpkin специализируется на обеспечении пользователей мощными, эффективными и недорогими программными решениями для встроенных систем. Мы надеемся, что вам понравится то, что мы сделали для вас.

Если вы еще не использовали Salvo, пожалуйста, прочтите *Главу 1 • Введение*, чтобы получить понятие о том, что такое Salvo, что она может, и какие другие инструменты вам понадобятся для ее использования. См. *Главу 2 • Основы RTOS*, если вы ранее не использовали RTOS (Real Time Operating System – операционная система реального времени). Затем попробуйте выполнить перечисленные ниже шаги в указанном порядке.

Замечание: Вы не обязаны приобретать Salvo для выполнения демонстрационных и обучающих программ или использования свободно распространяемых библиотек для построения вашего многозадачного приложения Salvo – все они являются частью Salvo Lite – свободно распространяемой версии Salvo.

Выполнение демонстрационной программы

Если вы имеете совместимую целевую среду, вы можете выполнить на ваших собственных аппаратных средствах одно из демонстрационных приложений Salvo, содержащихся в `\Pumpkin\Salvo\Examples\...` Откройте проект демонстрационной версии, скомпилируйте его, загрузите или запрограммируйте его в ваше устройство и дайте ему выполниться. Большинство из демонстрационных программ обеспечивает обратную связь в реальном времени. Если используется демо-версия Salvo Lite и общедоступные аппаратные средства, вы можете создать ваше собственное приложение, модифицируя и компилируя исходный код примера.

См. *Приложение C • Описание файлов и программ* для получения дополнительной информации о демонстрационных программах.

Учебные программы

В *Главе 4 • Учебник* создается многозадачное управляемое событиями приложение Salvo за шесть простых шагов. Учебник ознакомит вас с терминологией Salvo, сервисами пользователя и процессом построения рабочего приложения. Набор учебных проектов включен в каждый дистрибутив Salvo для встроенных систем, позволяя вам строить каждое учебное приложение, просто загружая и компилируя проект в соответствующей среде разработки.

Salvo Lite

Компилятор, сертифицированный для использования вместе с Salvo – это все, что нужно для использования Salvo Lite – свободно распространяемой версии Salvo. Вы можете написать ваше собственное небольшое многозадачное приложение с обращениями к сервисам Salvo и скомпоновать его со свободно распространяемыми библиотеками. См. *Главу 4 • Учебник* и документы *Salvo Application Note* для вашего компилятора и/или процессора для получения дополнительной информации.

Salvo LE

Salvo LE добавляет стандартные библиотеки Salvo к версии Salvo Lite. Это означает, что число задач, событий, и т.д. в вашем приложении будут ограничены только объемом доступной памяти RAM (Random Access Memory – памяти с произвольным доступом или оперативной памяти).

Salvo Pro

С Salvo Pro вы получите полный доступ ко всему исходному коду, стандартным библиотекам, тестовым программам и поддержке приоритетов задач. Если вы этого еще не сделали, поработайте с *Главой 4 • Учебник* в качестве первого шага к созданию вашего собственного приложения. Затем используйте опции конфигурации из *Главы 5 • Конфигурация* и сервисы вместе с их примерами, описанные в *Главе 7 • Справочник* для настройки Salvo на требования вашего приложения. Если у вас возникнут проблемы или вопросы, вы найдете большое количество полезной информации в *Главе 6 • Часто задаваемые вопросы (FAQ)* и в *Главе 11 • Советы и приемы*.

Получение помощи

Некоторые из полезных ресурсов для новичков и опытных пользователей Salvo находятся на форумах пользователей Salvo, размещенных на веб-сайте Pumpkin http://www.pumpkininc.com/. Ищите там обновленную информацию о последних выпусках Salvo.

Контакты и техническая поддержка

Контакты с Pumpkin

Почтовый адрес, номера телефона и факса Pumpkin:


```
Pumpkin, Inc.
750 Naples Street
San Francisco, CA 94112 USA
tel: 415-584-6360
fax: 415-585-7948

info@pumpkininc.com
sales@pumpkininc.com
support@pumpkininc.com
```


Временная зона: GMT–0800 (стандартное тихоокеанское время)

Доступ к веб-сайту Pumpkin

Для доступа к Pumpkin используйте обозреватель интернета:

http://www.pumpkininc.com

Информация, доступная на веб-сайте, включает:

- Последние новости
- Загрузку и обновление программ
- *Руководства пользователя*
- *Справочные руководства компиляторов*
- *Заметки о приложениях*
- *Руководства по сборке*
- *Заметки о выпусках продукта*
- Пользовательские форумы

Пользовательские форумы Salvo

Pumpkin поддерживает на своем веб-сайте форумы пользователей Salvo. Форумы содержат массу практической информации об использовании Salvo и посещаются пользователями Salvo также для получения технической поддержки Pumpkin.

Как связаться с Pumpkin для поддержки

Pumpkin обеспечивает интерактивную поддержку Salvo при помощи форумов пользователей Salvo на сайте Pumpkin во всемирной сети (WWW). Файлы и информация на данном сайте доступны всем пользователям Salvo. Для доступа к сайту вам необходим доступ к сети и обозреватель (например, Netscape, Opera, Internet Explorer).

Интернет (WWW)

Форумы пользователей Salvo находятся по адресу

http://www.pumpkininc.com

и являются *предпочтительным* методом для отправки по почте ваших вопросов о приобретении, общих вопросов или вопросов технической поддержки.

Электронная почта

Обычно мы просим, чтобы вы отправляли ваши вопросы технической поддержки по почте на форумы пользователей Salvo, находящиеся на нашем веб-сайте. Мы следим за форумами и отвечаем на вопросы технической поддержки интерактивно.

В экстренном случае вы можете получить техническую поддержку через электронную почту:

support@pumpkininc.com

Мы приложим все усилия, чтобы ответить на ваши почтовые запросы о технической поддержке в течение 1 рабочего дня. Пожалуйста, убедитесь, что сообщили о вашей проблеме столько информации, сколько возможно.

Почта, телефон и факс

Если вы не смогли найти ответ на ваш вопрос в этом руководстве, проверьте веб-сайт Salvo и форумы пользователей Salvo (см. ниже) на наличие дополнительной информации, которая, возможно, была размещена недавно. Если и в этом случае вы не смогли решить ваши вопросы, пожалуйста, свяжитесь с нами непосредственно по номерам, указанным выше.

Что требуется для запроса поддержки

Зарегистрированные пользователи, запрашивающие техническую поддержку Salvo, должны сообщить:

- Номер версии Salvo;
- Название и номер версии компилятора;
- Фрагменты исходного кода пользователя;
- Пользовательский файл `salvocfg.h`;
- Все остальные относящиеся к делу файлы, детали, и т.д.

Небольшие секции кода могут быть присланы непосредственно на форумы пользователей Salvo. См. интерактивно зарегистрированные FAQ (Frequently Asked Questions – часто задаваемые вопросы) о том, как использовать теги кода UBB (`[code]` and `[/code]`) для предохранения форматирования кода и как сделать код более читаемым.

Если возникнет потребность послать большие секции кода или даже полный проект, пожалуйста, заархивируйте файлы и вышлите их по электронной почте непосредственно службе технической поддержки Salvo (см. выше). Пожалуйста, убедитесь, что выслали все необходимые файлы, чтобы дать возможность группе технической поддержки создать ваше приложение Salvo локально, чтобы попытаться решить вашу проблему. Имейте в виду, что без соответствующих целевых аппаратных средств, поддержка в этих случаях вообще ограничена решением проблем, не относящихся ко времени выполнения. Группа технической поддержки будет использовать весь пользовательский код строго конфиденциально.

О руководстве пользователя Salvo

Претензии

Компания Pumpkin Incorporated ("Pumpkin") прилагает все усилия, чтобы данный документ содержал полную и точную информацию. Однако из-за постоянного совершенствования и изменения продукции, Pumpkin и ее лицензиары не берут на себя ответственность за технические или редакционные ошибки в данном документе, а также за любой прямой или косвенный ущерб из-за расхождений между документом и описываемым продуктом.

Информация предоставляется по принципу "как есть" и может быть изменена без уведомления, а также не представляет собой обязательств со стороны Pumpkin Incorporated и ее лицензиаров.

Торговые марки

Названия и логотипы Pumpkin и Salvo, а также "The RTOS that runs in tiny places" являются торговыми марками Pumpkin Incorporated.

Отсутствие указанных названий или логотипов продукта или службы не означает отказ от торговой марки или прав на интеллектуальную собственность Pumpkin, относящуюся к этим именам или логотипам.

Все другие упоминаемые названия продуктов и компаний могут быть торговыми марками их владельцев. Все упомянутые слова и термины, известные как торговые или сервисные марки, набраны прописными буквами. Pumpkin Incorporated не может гарантировать точности этой информации. Использование термина не должно быть расценено как подтверждение правильности любой торговой или сервисной марки.

Этот список может быть неполным.

Патентная информация

Описываемый программный продукт может иметь действующие или заявленные патенты США.

Системы жизнеобеспечения

Продукция Pumpkin не авторизована для использования в качестве критических компонентов в устройствах или системах обеспечения жизни без письменного подтверждения президента компании. Здесь:

1) Устройства или системы жизнеобеспечения – это средства, предназначенные для имплантации или поддержки жизни, при отказе которых при использовании в соответствии с правилами может ожидаться значительный ущерб для пользователя.
2) Критический компонент – любой компонент устройства или системы жизнеобеспечения, при отказе которого может ожидаться отказ устройства или системы жизнеобеспечения или ухудшение безопасности или эффективности.

Ограниченная гарантия на носители

Компания Pumpkin хочет, что вы были довольны приобретением Salvo и поэтому предлагает вам перед покупкой оценить систему. Вы можете загрузить и оценить полнофункциональную свободно распространяемую версию Salvo Lite с веб-сайта Pumpkin. Если вы имеете вопросы по использованию Salvo Lite, пожалуйста проконсультируйтесь на форумах пользователей Salvo, свяжитесь с нашим персоналом поддержки по адресу support@pumpkininc.com или свяжитесь с Pumpkin непосредственно.

На основании наличия возможности свободного изучения и того, что приобретенная версия может содержать полный исходный текст Salvo, Pumpkin не предусматривает возврата суммы, уплаченной за приобретенное программное обеспечение.

Pumpkin бесплатно заменит дефектные носители с дистрибутивами или руководство, если вы возвратите их компании вместе со свидетельством покупки в течение 90-дневного периода после приобретения. Дополнительные подробности могут быть найдены в разделе *11 – Ограниченная гарантия на носитель* в главе *Лицензионное соглашение* (в англоязычном оригинале – прим. пер.).

Создание документации

Данный документ (речь идет об оригинале – прим. перев.) создан при помощи программных инструментов Microsoft Word, Creative Softworx Capture Professional, CorelDRAW!, Adobe Photoshop, Adobe Illustrator и Adobe Acrobat.

Документ:	SalvoUserManual.doc (главный документ)
Шаблон:	User's Manual - Template (TT).dot
Дата:	22:02, Четверг, 6 февраля 2008 г.
Страниц:	530
Слов:	96788 (в англоязычном оригинале – прим. пер.)

Авторы

Автор:	Andrew E. Kalman
Оформление:	Laura Macey, Elizabeth Peartree, Andrew E. Kalman
Советы по языку Си:	Russell K. Kadota, Clyde Smith-Stubbs, Dan Henry
Советы по компиляторам:	Matthew Luckman, Jeffrey O'Keefe, Paul Curtis, Richard Man

Содержание

Рисунки

Листинги

Таблицы

Введение

Что нового

Пожалуйста, обратитесь к файлу `salvo-whatsnew.txt` из комплекта дистрибутива за подробной информацией о том, что нового появилось в выпуске версии v4.1.2.

Замечания о выпуске

Пожалуйста, обратитесь к общим (`salvo-release.txt`) и специфическим для дистрибутивов замечаниям (`salvo-release-`*`targetname`*`.txt`) за подробной информацией относительно изменений и обновлений в выпуске версии v4.1.2.

Версии инструментов третьих сторон

Пожалуйста, обратитесь к специфическим для дистрибутивов замечаниям (`salvo-release-`*`targetname`*`.txt`) для получения номеров версий инструментов третьих сторон (компиляторы, компоновщики, библиотекари и т.д.) для выпуска версии v4.1.2.

Процессоры и компиляторы

С момента выпуска версии v4.1.2, Salvo поддерживает разнообразные 8, 16 и 32-битные процессоры и компиляторы:

Пожалуйста, обратитесь к специфическим для дистрибутивов замечаниям (`salvo-release-`*`targetname`*`.txt`) за номерами версий инструментов третьих сторон (компиляторы, компоновщики, библиотекари и т.д.) для выпуска v4.1.2. Если вы имеете названный компилятор более старой версии, чем указано в списке, вам может понадобиться обновить его для работы с Salvo. Свяжитесь с поставщиком компилятора для получения информации по обновлению.

Предисловие

История создания

Salvo версии 1 была внутренним выпуском, написанным на ассемблере и предназначенным специально для Microchip PIC17C756 PICmicro в частной внутренней системе сбора данных. Эта версия 1998 года поддерживала большое число базовых функций, позднее использовавшихся в последующих версиях.

После анализа рынка Pumpkin Inc приняла решение о расширении функциональности версии 1, переписав Salvo на языке Си. При этом возникли возможности многочисленных опций конфигурации и оптимизации, что сделало Salvo по сравнению с предшественницей 1 не только более мощной и гибкой, но также полностью переносимой.

В 2000 году Salvo версии 2 стала первой коммерческой версией кооперативной, основанной на приоритетах многозадачной RTOS от Pumpkin, Inc. Она предназначалась для всего ряда микроконтроллеров Microchip PICmicro®.

В 2002 году вышла Salvo версии 3. Это означило расширение Salvo RTOS для поддержки новых микропроцессоров, таких, как 8-битные 8051 и 16-битные MSP430.

Salvo версии 4 вышла в 2005 году. Этот выпуск означал как первую поддержку Salvo 32-битных встроенных процессоров, так и учет многих уроков разработки, полученных за предыдущие 6 лет, позволивших достичь максимума конфигурируемости и эффективности. В Salvo версии 4 были удалены аппаратные зависимости кода, не поддерживаемые набором команд. Это дает пользователю полную гибкость и конфигурируемость Salvo для получения максимальной эффективности в реальном времени.

Типографские соглашения

В данном руководстве используются различные стили текста, чтобы повысить удобочитаемость. Примеры и фрагменты кода, имена путей и файлов выполнены `моноширинным шрифтом`. Новые, особо полезные и требующие акцентирования термины набраны *курсивом*. Ввод пользователя (например, в командной строке DOS) показан в таком стиле. Некоторые термины и последовательности чисел набраны **полужирным** шрифтом. Важные замечания, советы, предостережения и предупреждения заключены в рамки:

> **Замечание:** Исходный код Salvo использует шаг табуляции равный 4, т.е. символ табуляции эквивалентен 4-м символам пробела. (в переводе используется шаг, равный 2 – прим. пер.)

Буквосочетание xyz используется для обозначения одного из нескольких возможных вариантов имен, например `OSSygnalXyz()` относится к `OSSygnalBinSem()`, `OSSygnalMsg()`, `OSSygnalMsgQ()`, `OSSygnalSem()` и т.п. Сочетание `Xyz` является нечувствительным к регистру символов.

Символ | используется стенографически для обозначения множества схожих имен, например `sysa|e|f` означает `sysa`, `syse` или `sysf`.

Имена путей DOS и Windows используют символ '\'. Имена путей Linux/Unix используют символ '/'. В данном документе они используются как взаимозаменяемые.

Стандартизированная схема нумерации

Salvo использует *стандартизированную схему нумерации* для всех выпусков программного обеспечения. Схема нумерации версии/редакции использует несколько полей[1], показанных ниже:

`salvo-distribution-target-MAJOR.MINOR.SUBMINOR[-PATCH]`

где:

- `distribution` относится к версиям Lite, tiny, SE, LE или Pro, отличающимися ограничениями для целевого процессора, поддерживаемого дистрибутивом.
- `MAJOR` меняется при добавлении важных свойств (например режим массива).
- `MINOR` меняется при добавлении или изменении менее важных свойств (например, новый сервис пользователя).
- `SUBMINOR` меняется при альфа и бета тестировании и при добавлении файлов поддержки (например новых *Salvo Application Notes*).
- `PATCH` присутствует и изменяется каждый раз при исправлении ошибок и/или добавлении новой документации. `PATCH` также используется для кандидатов на новый выпуск, например `rc4`.

Все `MAJOR.MINOR.SUBMINOR` версии выпускаются со своими собственными полными программами инсталляторами. `-PATCH` может быть использован в полной или минимальной программе инсталляции или архивах, модифицирующих или добавляющих новые файлы к существующим кодам или документации Salvo.

Примеры:

`salvo-lite-pic-2.2.0`	v2.2 Salvo Lite для PICmicro MCU, инсталлятор, выпуск 2
`salvo-le-8051-3.1.0-rc3`	v3.1.0 Salvo LE для 8051 семейства, инсталлятор, кандидат на выпуск 3
`salvo-pro-avr-4.1.2`	v4.1.2 Salvo Pro для AVR и MegaAVR, инсталлятор, выпуск 4

Выпуски Salvo обобщенно упоминаются по их номерам `MAJOR.MINOR`, например “выпуск 4.0".

[1] Последнее поле присутствует только в пакете исправлений.

Стиль кодирования Salvo

Господство конфигурируемости

Salvo является чрезвычайно гибко конфигурируемой системой для удовлетворения требований самого широкого круга пользователей встраиваемых микроконтроллеров. Она также обеспечивает пользователя всеми необходимыми заголовочными файлами, инструментами пользователя, предопределенными константами, типами данных, функциями, и т.д., что позволяет вам создавать свое собственное приложение Salvo настолько быстро и свободно от ошибок, насколько возможно.

Берегите драгоценные ресурсы

Исходный код Salvo написан на оптимальном уровне для использования в целевом приложении минимально возможного объема ресурсов. Ресурсы включают RAM (ОЗУ), ROM (ПЗУ), стек адресов возврата, количество циклов команд. *Большинство из ресурсоемких свойств функциональности Salvo запрещены по умолчанию.* Если вы хотите использовать особые свойства (например, флаги событий), то их необходимо разрешить *опцией конфигурации* (например, `0SENABLE_EVENT_FLAGS`) и перекомпилировать ваше приложение. Это позволяет управлять кодом Salvo в вашем приложении из одного места – файла конфигурации Salvo `salvocfg.h`.

Учитесь любить препроцессор

Salvo делает непростым использование препроцессора Си и символов, предопределенных компилятором, Salvo или пользователем для конфигурирования исходного кода при компиляции. Хотя поначалу это может показаться мучением, вы найдете, что это делает управление проектами Salvo намного более простым.

Документировать, но не дублировать

Везде, где только возможно, ни исходный текст, ни документация в Salvo не повторяются. Это упрощает нам поддержку и проверку кода и обеспечивает точную и своевременную информацию.

Мы не совершенны

Несмотря на то, что было сделано все, чтобы обеспечить работу Salvo, как обещано и без ошибок, вполне возможно, что мы могли упустить какую-либо проблему или нам не удалось отловить какую-нибудь ошибку. Если, по вашему мнению, продукт содержит ошибку или двусмысленность, пожалуйста, свяжитесь с нами, чтобы мы могли решить проблему насколько возможно быстро и позволить вам спокойно продолжить создавать ваши приложения Salvo[2].

[2] См. детальную информацию в Соглашениях о лицензировании программного продукта Pumpkin Salvo.

Замечание: Мы полагаем, что нет необходимости модифицировать исходный код Salvo для достижения функциональности, слабо отличающейся от той, какую Salvo уже обеспечивает. Мы настоятельно рекомендуем вам сначала обратиться к нам с вашими вопросами перед изменением исходного кода, так как мы не поддерживаем модифицированные версии Salvo. Во многих случаях, мы можем предложить решение вашей проблемы, и возможно также включить решение проблемы в следующий выпуск Salvo.

Глава 1 • Введение

Приглашение

В гонке внедрения инноваций срок появления на рынке является решающим для успешного запуска нового продукта. Если вы не используете преимущества фирменной или доступной коммерческой программной базы и инструментальной поддержки, то за вас это сделает ваш конкурент. Но стоимость – тоже важная статья, а в области полупроводников (как и в реальной жизни) цены растут, поскольку продукты становятся больше. Если ваш проект может позволить себе большой объем памяти и возможно также большой микропроцессор, то идите и получите это. Так делают все вокруг...

Но что если этого сделать нельзя?

Что, если вас попросили сделать невозможное – встроить сложные функции реального времени в дешевый микроконтроллер и сделать все это в рамках жестких планов? Что, если ваш процессор имеет всего несколько килобайт ROM и еще меньше RAM? Что, если единственные инструменты, которые вы имеете – компилятор, некоторые отладчики, пара книг и ваше воображение? Собираетесь ли вы опять увязнуть с конечными автоматами, таблицами переходов, сложными схемами прерываний и кодом, который вы не сможете объяснить кому-либо другому? Довольно скоро станет не до веселья. Почему для вас должно быть закрыто использование эффективных программных решений, используемых 'крутыми' парнями?

Они говорят, что многозадачность требует много памяти и поэтому не годится для вашего проекта. Но так ли это в реальности?

Ни в коем случае. Только не с Salvo. Salvo – полностью разместимая в ROM многозадачная система. Она занимает поразительно малый объем памяти, сравнимый по величине с объемом кода, занимаемым функцией `printf()`[3]! Многозадачность, приоритеты, события, системный таймер – все это имеется. Нет прерываний? Это, скорее всего, тоже не проблема. Вы получите большую функциональность от вашего процессора быстрее, чем вы вероятно когда-либо думали. И вы можете сразу же установить Salvo для вашей работы.

Что такое Salvo?

Salvo – мощная, высокоэффективная и свободная от отчислений RTOS (Real Time Operating System), требующая очень мало памяти и не требующая стеков задач. Это легкий в использовании программный инструмент, способный помочь вам быстро создавать мощные, надежные и сложные приложения (программы) для встроенных систем.

Salvo была разработана с самого начала для использования в микропроцессорах и микроконтроллерах со строго ограниченными ресурсами, и потребует обычно в 5…100 раз меньшего объема памяти, чем другие RTOS. Фактически, требования памяти Salvo настолько минимальны, что она будет выполняться там, где никакая другая RTOS не сможет работать.

[3] Сравнение основано на реализации полной функциональности функции `printf()`.

Salvo пригодна для размещения в ROM, легко масштабируется и чрезвычайно легко переносима. Она выполняется почти на любом процессоре от PIC до Pentium.

Почему я должен использовать Salvo?

Если вы разрабатываете очередную актуальную встроенную программу, вы знаете, что срок поступления на рынок является определяющим для обеспечения успеха. Salvo предоставляет мощный и гибкий каркас, на который вы быстро можете нарастить ваше приложение.

Если перед вами стоит сложный проект и ограниченные ресурсы процессора, Salvo поможет вам максимально использовать то, что доступно в вашей системе.

Если вы пытаетесь увеличить функциональность или уменьшить стоимость существующего проекта, Salvo может оказаться тем, чего вам не доставало, потому что она послужит вам рычагом увеличения уже имеющейся мощности обработки информации.

До Salvo, программисты встроенных систем могли только мечтать о наличии RTOS в простых процессорах. Они были лишены преимуществ, которые RTOS может принести в проект, включая уменьшение сроков выхода на рынок, управление сложностью, повышение устойчивости к ошибкам и улучшение совместного и многократного использования кода. Они не имели возможности воспользоваться преимуществом многих известных возможностей RTOS, разработанных для решения общих и повторяющихся проблем в программировании встроенных систем.

Теперь эта мечта стала действительностью. Вы можете перестать волноваться относительно глубинной структуры и надежности вашей программы и сосредоточиться непосредственно на приложении.

К какому типу относится RTOS Salvo?

Salvo – многозадачная кооперативная RTOS с полной поддержкой сервисов событий и таймера. Многозадачность основана на приоритетности с поддержкой пятнадцати отдельных уровней приоритета. Задачи, имеющие один и тот же приоритет, будут выполняться циклически (round-robin). Salvo обеспечивает сервис для использования семафоров, сообщений и очередей сообщений для межзадачных связей и управления ресурсами. Поддерживается полный набор функций RTOS (например, переключатель контекста останавливает задачу, ожидает сигнал семафора, и т.д.). Также поддерживаются функции таймера, включая задержки и таймауты.

Salvo написан на языке ANSI C, с очень немногими специфическими для процессора расширениями, некоторые из которых написаны на ассемблере процессора. Высокая степень конфигурируемости поддерживает уникальные запросы вашего частного приложения.

Несмотря на то, что Salvo предназначена для встроенных приложений, она обладает универсальностью и может также использоваться для создания приложений для других типов систем (например, 16-разрядные приложения DOS).

Как выглядит программа Salvo?

Программа Salvo выглядит похожей на любую другую, выполняемую в многозадачной среде RTOS. Листинг 1 содержит исходный текст для удаленного автомобильного кресла, нагреваемого до установленной пользователем температуры. Микроконтроллер интегрирован в кресло и требует только четырех проводов для связи с остальной электроникой автомобиля – питание, общий, Rx (получение желатемой температуры от системы управления, установленной в другом месте) и Tx (для сообщения о состоянии). Желаемая температура поддерживается через `TaskControl()`. `TaskStatus()` посылает каждую секунду или одиночный 50ms импульс, чтобы указать, что кресло еще не нагрелось, или два последовательных 50ms импульса, для указания, что желаемая температура достигнута.

```
#include <salvo.h>

typedef unsigned char t_boolean;
typedef unsigned char t_temp;

/* local flag */
t_boolean warm = FALSE;

/* seat temperature functions */
extern t_temp UserTemp( void );
extern t_temp SeatTemp( void );
extern t_boolean CtrlTemp( t_temp user, seat );

/* moderate-priority (i.e. 8) task (i.e. #1) */
/* to maintain seat temperature. CtrlTemp() */
/* returns TRUE only if the seat is at the */
/* the desired (user) temperature. */
void TaskControl( void )
{
  while (1) {
    warm = CtrlTemp(UserTemp(), SeatTemp());
    OS_Yield();
  }
}

/* high-priority (i.e. 3) task (i.e. #2) to */
/* generate pulses. System ticks are 10 ms. */
void TaskStatus( void )
{
  /* initialize pulse output (low). */
  TX_PORT &= ~0x01;

  while (1) {
    OS_Delay(100);
    TX_PORT |= 0x01;
    OS_Delay(5);
    TX_PORT &= ~0x01;
    if (warm) {
      OS_Delay(5);
      TX_PORT |= 0x01;
      OS_Delay(5);
      TX_PORT &= ~0x01;
    }
  }
}
```

```
/* initialize Salvo, create and assign */
/* priorities to the tasks, and begin */
/* multitasking. */
int main( void )
{
  OSInit();

  OSCreateTask(TaskControl, OSTCBP(1), 8);
  OSCreateTask(TaskStatus,  OSTCBP(2), 3);

  while (1) {
    OSSched();
  }
}
```

Листинг 1: Простая программа Salvo

Важно заметить, что, когда программа выполняется, управление температурой продолжается постоянно, тогда как `TaskStatus()` периодически задерживается. Вызов `OS_Delay()` не вызывает программного цикла на какое-либо время и позволяет продолжать выполнение программы. Цикл был бы пустой тратой ресурсов процессора, то есть циклов команд. Взамен этого простой вызов Salvo приостанавливает генератор импульсов и гарантирует продолжение работы после определенного периода времени. `TaskControl()` выполняется всегда, тогда как `TaskStatus()` приостанавливается.

Исключая создание простого файла конфигурации Salvo и привязки Salvo к 10ms периодическим прерываниям в вашей системе, код Си, описанный выше – это все, что необходимо, чтобы выполнять эти две задачи одновременно. Представьте насколько просто добавить большее количество задач к этому приложению, чтобы расширить функциональные возможности.

См. *Главу 4 • Учебник* для получения большей информации о программировании в среде Salvo.

Какие ресурсы требуются Salvo?

Объем ROM, требуемый Salvo, будет зависеть от того, сколько возможностей Salvo вы используете. Минимальное многозадачное приложение для RISC-процессора могло бы использовать несколько сотен команд. Полное приложение Salvo на том же самом процессоре использует в ROM около тысячи команд.

В традиционных RTOS значительный объем оперативной памяти выделяется для индивидуальных стеков задач. Так как Salvo не требует наличия стеков задач, ее требования к объему оперативной памяти предельно минимальны, и поэтому компиляторы Си могут использовать стек для локальных (авто) переменных, параметров функций и т.п. Принципы построения Salvo позволяют применять ее в процессорах с ограниченным аппаратным стеком возврата[4] вместо стека общего назначения.

[4] Аппаратный стек возврата используется только для хранения адреса возврата из функции и ограничен некоторой глубиной (например 16 уровней для процессоров PIC17). Аппаратный стек возврата не может быть использован для локальных (авто) переменных. Также процессоры с аппаратным стеком возврата не имеют инструкций PUSH и POP.

Объем RAM, требуемый Salvo, также зависит от специфики вашей конфигурации. В RISC-приложении[5], каждая задача требует 4...12 (обычно 7) байт, каждое событие – 3...4 байта[6], и еще 4...6 байт требуются для управления задачами, событиями и задержками.

В любом случае, объем требуемой RAM, прежде всего, зависит от размера указателей на ROM и RAM (8, 16 или 32 бит), используемых в вашем приложении, то есть является зависимым от приложения. В некоторых приложениях (например, с CISC процессорами) может потребоваться дополнительная RAM для хранения регистров общего назначения.

Если вы планируете использовать задержки и таймауты без использования прерываний, Salvo потребует, чтобы вы регулярно вызывали сервис таймера. Удобнее вызывать сервис таймера периодическим прерыванием. Но это прерывание не требуется самой Salvo – оно может использоваться для ваших собственных целей.

Количество задач и событий ограничено только объемом доступной памяти.

Подробности см. в *Главе 6 • Часто Задаваемые Вопросы (FAQ)*.

Чем отличается Salvo?

Salvo – это кооперативная RTOS, не использующая стеки задач[7]. Фактически все другие RTOS используют стек, и многие являются приоритетными и кооперативными. Salvo отличается, прежде всего, следующим:

- Salvo – кооперативная RTOS, поэтому вы должны явно управлять переключением задач[8].
- Переключение задач может производиться только на уровне задачи, то есть непосредственно внутри ваших задач, а не из функций, вызываемых вашей задачей или где-либо в другом месте. Это – следствие отсутствия универсального стека для совместного сохранения данных и состояния задач, и может несколько повлиять на структуру вашей программы.
- По сравнению с другими кооперативными или приоритетными RTOS, нуждающимися в большом объеме RAM (обычно в виде универсального стека), потребности Salvo весьма невелики. Не имея большого объема RAM, ваш выбор – только Salvo RTOS.

Salvo способна обеспечить большинство эффективных возможностей RTOS, размещенной в ROM, используя очень небольшую часть памяти. С Salvo вы можете быстро создавать мощные, быстрые, сложные и устойчивые многозадачные приложения.

Что мне нужно для использования Salvo?

Рекомендуются практические навыки в языке Си. Но даже если вы – новичок в Си, вы не должны испытать большие трудности, изучая использование Salvo.

[5] Серия PIC16 (например, PIC16C64). Указатели на ROM требуют 2 байта, указатели на RAM требуют 1 байт.
[6] Очереди сообщений требуют дополнительной RAM.
[7] Под "не требующая стек" мы подразумеваем, что Salvo не использует RAM для хранения данных, которые традиционные RTOS хранят в стеке общего назначения, включая адрес возврата, временные данные, локальные переменные, сохраняемые регистры и т.п.
[8] Мы объясним этот термин позже, но сейчас это означает находиться в одной задаче и освобождать управление процессором так, чтобы могла выполняться другая задача.

Полезны некоторые знания основных принципов RTOS, но это не обязательно. Если вы плохо знакомы с применением RTOS, изучите *Главу 2 • Основы RTOS*.

Вам потребуется хороший ANSI-C совместимый компилятор для процессора, который вы используете. Он должен быть способен создавать код Salvo из исходного текста на языке Си, используя многие свойства языка, включая (но не ограничиваясь):

- массивы,
- объединения,
- битовые поля,
- структуры,
- статические переменные,
- множественные исходные файлы,
- косвенные вызовы функций,
- многоуровневая косвенность,
- передача всех типов параметров,
- передача многобайтных параметров,
- широкое использование препроцессора Си,
- указатели на функции, массивы, структуры, объединения и т.д.
- поддержка списков переменных аргументов[9] (`va_arg()`, и т.д.)

Листинг 2: Требования к свойствам компилятора Си

Если ваш Си-компилятор может использовать встроенный ассемблер, то это только плюс. Чем мощнее встроенный ассемблер, тем лучше.

Наконец, ваш компилятор должен быть способен компилировать исходные файлы в объектные модули (`*.o`) и библиотеки (`*.lib`) и компоновать объектные модули и библиотеки вместе для создания конечной выполнимой программы (обычно `*.hex`).

Мы советуем вам использовать компилятор, рекомендованный для использования с Salvo. Если ваш компилятор или процессор еще не поддерживается, вы можете, вероятно, за несколько часов перенести под них исходный код. *Глава 10 • Перенос кода* поясняет этот процесс. Всегда сверяйтесь с производителем для получения последних новостей о поддерживаемых компиляторах и процессорах.

Какие процессоры и компиляторы поддерживает Salvo?

Посетите веб-сайт Pumpkins для получения необходимой информации.

Как распространяется Salvo?

Система RTOS Salvo доступна для загрузки через Интернет, как программа инсталляции в среде операционной системы Windows 95/98/ME/NT/2000/XP. После установки системы Salvo на ваш компьютер, вы получите группу папок и подпапок с исходными кодами Salvo, примерами, демонстрационными программами, руководствами по применению и другими файлами.

[9] Это не является абсолютно необходимым, но желательно. `va_arg()` является частью стандарта ANSI C.

Что содержится в этом руководстве?

Глава 1 • Введение представлено в этой главе.

Глава 2 • Основы RTOS является введением в программирование RTOS. Если вы знакомы только с традиционными типами программ типа "суперцикл" или "передний/задний план (фон)", вы должны изучить эту главу.

Глава 3 • Инсталляция объясняет установку Salvo на компьютер.

Глава 4 • Учебник является руководством по использованию Salvo. Он содержит примеры, представляющие функциональные возможности Salvo и их использование в вашем приложении. Даже программисты знакомые с другими RTOS должны изучить эту главу.

Глава 5 • Конфигурация объясняет параметры конфигурации Salvo. Начинающим и опытным пользователям эта информация нужна для оптимизации размера и эффективности Salvo в приложении.

Глава 6 • Часто задаваемые вопросы (FAQ) содержит ответы на многие часто задаваемые вопросы пользователей.

Глава 7 • Справочник является руководством по всем сервисам Salvo.

Глава 8 • Библиотеки перечисляет доступные свободные и стандартные библиотеки и объясняет их использование.

Глава 9 • Эффективность освещает вопросы реального времени и как максимизировать производительность вашего приложения Salvo.

Глава 10 • Перенос кода освещает проблемы, которые возникнут при переносе Salvo в среду компилятора или процессора, которые формально еще не рекомендовались или не поддерживаются Salvo.

Глава 11 • Советы, приемы, неполадки содержит информацию о проблемах, с которыми вы можете встретиться, и их решения.

Приложение A • Рекомендуемая литература содержит ссылки на литературу о многозадачности и связанные с ней документы.

Приложение B • Другие ресурсы содержит информацию о других ресурсах, которые могут быть полезны совместно с Salvo.

Приложение C • Описание файлов и программ содержит описания всех файлов и типов файлов, являющихся частью инсталляции Salvo.

Глава 2 • Основы RTOS

Замечание: Если вы уже знакомы с основами RTOS, вы можете пропустить данную главу и перейти к *Главе 3 • Инсталляция*.

Введение

"Когда-то я создавал системы опроса. Худшие из приложений – те, которые имеют дело с несколькими различными более или менее одновременными процессами без использования многозадачности. Программа в обоих случаях неизменно представляет замысловатую путаницу. Двадцать лет назад, я по наивности создавал без RTOS датчик толщины стали для производства обувных рожков. Получалось так много асинхронных процессов, что встроенный код разрастался до диковинных размеров и сложности".
Джек Г. Гэнсл[10].

Большинство программистов знакомо с традиционными системами, которые используют конструкцию цикла для выполнения основной части приложения, и прерывания для обработки критических по времени событий. Это комбинация процедур *переднего плана* и *заднего плана* (*фона)* или *суперцикл* – системы, где прерывания выполняются на *переднем плане* (так как имеют приоритет над остальными процедурами) а основной цикл выполняется на *заднем плане – фоне*, когда ни одно из прерываний не активно. Поскольку приложения растут в размере и сложности, этот подход теряет привлекательность, потому что становится все более трудно учитывать взаимодействие между передним планом и фоном.

Альтернативный метод структурирования приложений состоит в том, чтобы использовать программный каркас, который управляет всем выполнением программы согласно набору ясно определенных правил. Установив эти правила, эффективность приложения может быть повышена относительно простым способом независимо от размера и сложности.

Многие встроенные системы могут извлечь пользу от подхода, включающего использование нескольких параллельных задач, сообщающихся между собой, управляемых единым ядром, и с ясно определенным поведением во время выполнения. Этот подход к программированию – RTOS. Эти и другие принципы описаны ниже.

Замечание: Данная глава является кратким введением в устройство и применение RTOS. *Приложение А • Рекомендуемая Литература* содержит ссылки на дополнительные более углубленные источники информации.

[10] "Interrupt Latency", *Embedded Systems Programming*, Vol. 14 No. 11, October 2001, p. 73.

Основные термины

Задача (Task) – это последовательность команд, иногда повторно исполняемых, для выполнения некоторых действий (например, чтения клавиатуры, отображения сообщений на ЖКИ (LCD), зажигания светодиодов (LED) или генерации формы сигнала). Другими словами, это – обычно маленькая программа внутри большой программы. При выполнении на относительно простом процессоре (например, Z80, 68HC11, PIC), задача может владеть всеми ресурсами системы независимо от того, сколько задач используются в приложении.

Прерывание (Interrupt) – это внутреннее или внешнее аппаратное событие, которое вызывает приостановку выполнения программы. Для возникновения прерывания оно должно быть разрешено. Когда оно происходит, процессор переходит по вектору определяемой пользователем *Процедуры Обработки Прерывания (Interrupt Service Routine – ISR)*, которая выполняется до своего завершения. Затем выполнение программы возобновляется в том месте, где было приостановлено. Из-за способности приостанавливать выполнение программы, прерывания, как говорят, выполняются на переднем плане, а остальная часть программы выполняется в фоновом режиме.

Приоритет (Priority) задачи означает степень важности задачи относительно других задач. Он может быть фиксированным или переменным, уникальным или совпадать с другими задачами.

Переключение задачи (Task Switch) происходит, когда одна задача приостанавливает выполнение, а другая начинает или возобновляет выполнение. Это также называется *переключением контекста (context switch)*, потому что контекст задачи (в общем случае это содержание стека и значения регистров) обычно сохраняется для продолжения использования, когда задача возобновляет свое выполнение.

Вытеснение (Preemption) задачи происходит, когда выполнение одной задачи прерывается, а управление получает другая задача, готовая к выполнению. Альтернатива вытесняющей системе – *кооперативная (cooperative)* система, в которой задача должна добровольно освободить процессор для выполнения другой задачи. Задача программиста – структурировать задачи так, чтобы это всегда выполнялось. Если текущая задача не будет кооперироваться с другими, то другие задачи не смогут выполняться, и приложение не сможет работать правильно.

Ядро (Kernel) выполняет переключение контекста. Ядро управляет переключением задач (также называемое *диспетчирование* или *планирование – scheduling*) и межзадачной связью. Ядро в общем случае гарантирует, что самая высокоприоритетная готовая к выполнению задача – это выполняемая задача (приоритетное планирование) или задача, которая будет выполняться следующей (кооперативное планирование). Программное обеспечение ядра пишется как можно более компактным и быстрым, чтобы гарантировать высокую эффективность прикладной программы[11].

[11] Некоторые ядра обеспечивают функции ввода-вывода и другой сервис, например управление памятью. Здесь эти вопросы не обсуждаются.

Задержка (Delay) – это количество времени (часто определяемое в миллисекундах) на которое может быть приостановлено выполнение задачи. Приостановленная задача должна занимать минимум ресурсов процессора, чтобы максимизировать эффективность полного приложения, которое, вероятно, включает другие одновременно не приостановленные задачи. Как только задержка *истекает (elapsed, expired)*, задача возобновляет выполнение. Программист определяет, как долго длится задержка и как часто она происходит.

Событие (Event) – это какое-либо происшествие (например, нажатие клавиши, возникновение ошибки или отсутствие ожидаемого ответа), которого может ожидать задача. Также, почти любая часть программы может сигнализировать о происхождении события, позволяя, таким образом, знать другим, что событие произошло.

Межзадачная связь (Intertask Communication) – это упорядоченная передача информации от одной задачи другой задаче, следующая некоторым известным концепциям программирования. *Семафоры*, *сообщения*, *очереди сообщений* и *флаги событий (Semaphores, messages, message queues and event flags)* могут использоваться для передачи в той или другой форме информации между задачами и, в некоторых случаях, прерываниями.

Таймаут (Timeout) – это количество времени (часто определяемое в миллисекундах) в течение которого задача может ожидать событие. Таймауты являются опциональными – задача также может ждать событие неопределенно долго. Если задача определяет таймаут при ожидании события, и событие не происходит, считается, что произошел таймаут, и вызывается его специальная обработка.

Состояние (State) задачи описывает, что задача в настоящее время делает. Задачи изменяют одно состояние на другое по ясно определенным правилам. Общие состояния задачи могут быть следующими: *готова / готова продолжать, выполняется, задержана, ожидает, остановлена и разрушена / неинициализирована* (*ready / eligible*, *running*, *delayed*, *waiting*, *stopped and destroyed / uninitialized)*.

Таймер (Timer) – другой фрагмент программного обеспечения, который следит за прошедшим временем и/или реальным временем для задержек, таймаутов и других, связанных со временем сервисов. Таймер имеет такую точность, какую обеспечивают системные часы.

Система находится в *холостом (idling)* состоянии, когда не имеется ни одной выполняющейся задачи.

Операционная система (Operating System – OS) содержит ядро, таймер и остальное программное обеспечение (называемое сервисом) для обработки задач и событий (например, создание задачи, сигнализация события). *Операционную Систему Реального Времени (RTOS)* выбирают, когда некоторые операции являются критическими и должны быть завершены корректно в течение некоторого количества времени. *Приложение* или *программа* с RTOS – конечный продукт объединения ваших задач, ISR, структур данных, и т.д. с RTOS, для формирования цельной программы.

Далее все эти термины и некоторых другие исследуются более подробно.

Системы переднего и заднего плана

Простейшая структура программы состоит из *основного цикла* (иногда называемого *суперциклом*) вызывающего функции в упорядоченной последовательности. Так как выполнение программы может переключаться с основного цикла на ISR и обратно, основной цикл, как говорят, выполняется в фоне – на заднем плане, а ISR выполняются на переднем плане. С этим стилем программирования сталкиваются многие начинающие при изучении программирования простых систем.

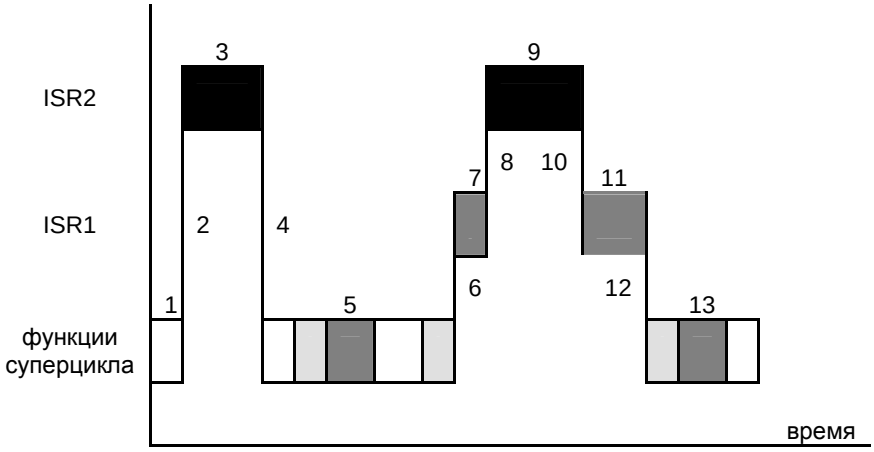

Рисунок 1: Процессы переднего и заднего планов

На Рисунке 1 мы видим группу функций, повторяющихся в основном цикле много раз [1, 5, 13]. Прерывания могут происходить в любое время и даже на нескольких уровнях. Когда происходит прерывание (высокоприоритетное прерывание в [2] и [8], низкоприоритетное прерывание в [6]), процессы в функции приостановлены, пока прерывание не закончится, после чего программа возвращается к основному циклу или к предыдущему прерванному ISR. Функции основного цикла выполняются в строго последовательном порядке, все одного приоритета, без каких-либо средств его изменения, когда или даже если функция обязана выполниться. ISR должны использоваться, чтобы быстро ответить на внешние события и могут быть распределены по приоритетам, если поддерживаются несколько уровней прерываний.

Системы переднего / заднего плана относительно просты с точки зрения программирования, пока имеется слабое взаимодействие функций в основном цикле и между ними и ISR. Но они имеют несколько недостатков: на синхронизацию цикла воздействуют любые изменения в коде цикла и/или ISR. Также, *ответ* системы на ввод недостаточен по скорости, потому что информация, сделанная ISR доступной для функции в цикле не может быть обработана функцией, пока она не сможет выполняться. Этот жестко последовательный характер выполнения программы в суперцикле предоставляет очень немного гибкости программисту и усложняет критические по времени операции. Чтобы частично решить эту проблему могут использоваться конечные автоматы. Поскольку приложение растет, синхронизация цикла становится непредсказуемой, и возникает ряд других факторов усложнения.

Реентерабельность

Один такой фактор – *реентерабельность* или повторная входимость. Реентерабельная функция может использоваться одновременно в одном или нескольких частях приложения без разрушения данных. Если функция написана не реентерабельной, одновременные вызовы могут разрушить внутренние данные функции, с непредсказуемыми результатами в приложении. Например, если приложение имеет нереентерабельную функцию `printf()`, и она вызывается и из кода основного цикла (в фоновом режиме), а также изнутри ISR (то есть на переднем плане), каждый раз имеется превосходный шанс что время от времени при вызове из основного цикла:

```
printf("Here we are in the main loop.\n");
```

и при вызове из процедуры ISR в то же самое время:

```
printf("Now we are servicing an interrupt.\n);
```

результат вывода может быть подобен следующему:

```
Here we aNow we are servicing an interrupt.
```

Листинг 3: Ошибка реентерабельности с printf()

Это явная ошибка. Случилось то, что первый экземпляр `printf()` (вызванный из основного цикла) произвел печать первых 9 символов (`"Here we a"`) строкового параметра перед самым прерыванием. ISR, также включающая вызов `printf()`, заново инициализировала локальные переменные и вызвала печать всей 36-символьной строки (`"Now we ... interrupt.\n"`). После завершения ISR, `printf()` основного цикла продолжилась там, где была прервана. Но внутренние переменные уже отражали вывод строкового параметра до конца, и никакого дальнейшего вывода больше не требовалось, так что функция просто осуществила возврат в основной цикл.

Замечание: Вызов нереентерабельных функций как будто они были реентерабельны, редко приводит к благоприятному результату.

Чтобы избежать подобной проблемы нереентерабельной функции `printf()`, могут быть использованы различные методы. Один метод заключается в запрете прерываний на время нереентерабельной функции. Другой метод заключается в том, чтобы переписать `printf()` таким образом, чтобы она использовала только локальные переменные (то есть переменные, которые хранятся в стеке). Стек играет очень важную роль в реентерабельных функциях.

Ресурсы

Ресурс – это какой-либо объект вашей программы, который может использоваться другими частями программы. Ресурсом может быть регистр, переменная, структура данных, или физическое устройство, такое как LCD или звонок. Общий (разделяемый) ресурс (*Shared resource*) – это ресурс, который может использоваться больше чем одной частью программы. Если две отдельных части программы борются за тот же самый ресурс, вы будете должны управлять этим при помощи взаимного исключения (*mutual exclusion*). Когда часть вашей программы хочет использовать ресурс, она должна получить к нему монопольный доступ, чтобы избежать его разрушения.

Многозадачность и переключение контекста

Разбивая приложение переднего / заднего плана на несколько независимых задач, может быть получено много преимуществ. В *многозадачности*, когда все задачи пытаются выполниться одновременно, должен существовать некоторый механизм передачи управления процессором и его ресурсов от одной задачи другой. Это работа *планировщика* или *диспетчера* – части ядра, которое (среди других обязанностей) приостанавливает одну задачу и продолжает другую при выполнении некоторых условий. Он делает это, сохраняя счетчик команд одной задачи и восстанавливая счетчик команд другой. Чем быстрее диспетчер способен переключать задачи, тем выше эффективность всего приложения, так как время переключения задач – время, потраченное без выполнения какой-либо задачи.

Переключатель контекста, который приостанавливает задачу и переключатель контекста, который продолжает ее, должен быть одним и тем же и должен быть прозрачен для самой задачи, чтобы перед приостановкой и после возобновления выполнения задача имела одинаковый контекст. Этим способом задача A может быть прервана в любое время, чтобы позволить диспетчеру выполнить более приоритетную задачу B. Как только задача B завершена, задача A продолжается с того места, где была приостановлена. Эффект воздействия переключателя контекста на задачу A только в том, что она была приостановлена на потенциально длительное время. Следовательно, задачи, имеющие критические по времени операции, должны предотвращать переключение контекста в течение критических периодов.

Переключение контекста может быть вынужденным по причинам внешним по отношению к задаче или намеренным из-за желания программиста приостановить задачу, чтобы сделать другие дела.

Большинство процессоров поддерживает универсальные стеки и имеет много регистров. Восстановление только соответствующего счетчика команд будет недостаточным, чтобы гарантировать продолжение выполнения задачи. Дело в том, что стек и значения регистров будут уникальны в задаче в момент переключения контекста. Переключатель контекста сохраняет весь контекст задачи (например, счетчик команд, регистры, содержание стека). Большинство архитектур процессоров требует, чтобы каждой задаче была выделена память для поддержки переключения контекста.

Задачи и прерывания

Как и в случае систем переднего / заднего плана, многозадачные системы часто и широко используют прерывания. Задачи должны быть защищены от эффектов прерываний, ISR должна быть максимально быстрой и большинство времени прерывания должны быть разрешены. Прерывания и задачи сосуществуют одновременно – прерывание может происходить прямо среди задачи. Запрет прерываний в течение задачи должен быть минимизирован, однако прерывания должны управляться, чтобы избежать конфликтов между задачами и прерываниями когда оба обращаются к общим ресурсам.

Время ожидания прерывания (*Interrupt latency*) определяется как максимальное время, в течение которого прерывания запрещены, плюс время до выполнения первой команды ISR. Другими словами, это задержка в самом неудачном случае между тем, когда прерывание происходит и когда ISR начинает выполняться.

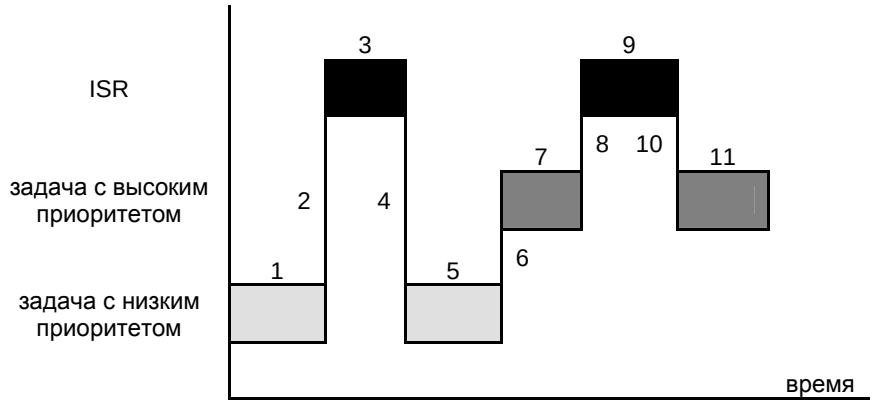

Рисунок 2: Прерывания могут происходить во время выполнения задачи

На Рисунке 2 низкоприоритетная задача выполняется [1], когда происходит прерывание [2]. В этом примере, прерывания всегда разрешены. Прерывание [3] выполняется до завершения [4], после чего низкоприоритетная задача [5] продолжает выполнение. Происходит переключение контекста [6], и начинает выполняться высокоприоритетная задача [7]. Переключатель контекста управляется диспетчером (не показано). Высокоприоритетная задача также прерывается [8-10] перед продолжением [11].

Приоритетное и кооперативное планирование

Имеются два типа планировщиков: приоритетный и кооперативный. Приоритетный планировщик может вызывать вытеснение текущей задачи другой задачей. Вытеснение происходит, когда задача с более высоким приоритетом, чем у текущей задачи становится готовой продолжать выполнение. Так как это может происходить в любое время, вытеснение требует использования прерываний и управления стеком, чтобы гарантировать правильность переключения контекста. Временно отключая вытеснение, программист может предотвращать нежелательные сбои в своих программах в критических секциях кода.

Приоритетное планирование

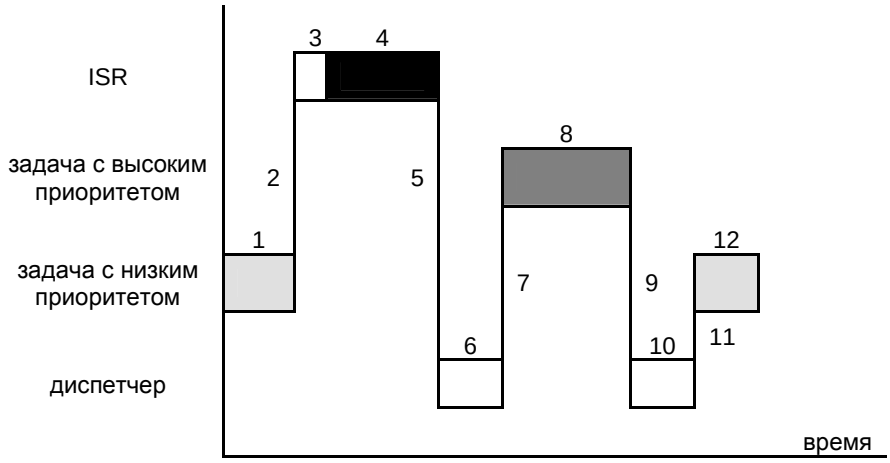

Рисунок 3: Приоритетное планирование

Рисунок 3 иллюстрирует работу приоритетного диспетчера. Низкоприоритетная задача [1] выполняется, когда происходит внешнее событие [2], вызывающее прерывание. Контекст задачи и некоторая другая информация диспетчера сначала сохраняются [3] в ISR, и прерывание обслуживается [4]. В этом примере высокоприоритетная задача ждет это специфическое событие и должна выполниться как можно скорее после того, как происходит событие. Когда ISR завершается [5], продолжает диспетчер [6], который начинает [7] высокоприоритетную задачу [8]. Когда она завершается, управление возвращается диспетчеру [9, 10], который восстанавливает контекст низкоприоритетной задачи и позволяет ей продолжить выполнение в прерванном месте [11, 12].

Приоритетное планирование интенсивно использует стек. Диспетчер содержит отдельный стек для каждой задачи, чтобы, когда задача продолжила выполнение, все значения в стеке, являющиеся уникальными в каждой задаче, оказались бы на своем месте. Это обычно адреса возврата из подпрограмм, параметры и локальные переменные (для языка подобного Си). Диспетчер может также сохранять в стеке контекст приостановленной задачи.

Приоритетные диспетчеры вообще очень сложны из-за огромного числа проблем, относящихся к поддержке правильного контекста, переключающегося в произвольный момент. Это особенно актуально в отношении обработки прерываний. Также, как может быть замечено на Рисунке 3, существует запаздывание между тем, когда прерывание случается и когда соответствующая ISR сможет выполниться. Это, плюс время ожидания прерывания, является временем *ответа прерывания* (t4 - t2). Время между концом ISR и возобновлением выполнения задачи – время *восстановления из прерывания* (t7 - t5). *Время ответа на событие* системы составляет (t7 - t2).

Кооперативное планирование

Кооперативный диспетчер более простой, чем его приоритетный коллега. Так как, для переключений контекста, все задачи должны сотрудничать, диспетчер меньше зависит от прерываний и может быть меньше и потенциально быстрее. Также программист точно знает момент переключения контекста и может защитить критический код, уствновив вызов переключателя контекста вне этого кода. Из-за его относительной простоты данный диспетчер имеет ряд преимуществ.

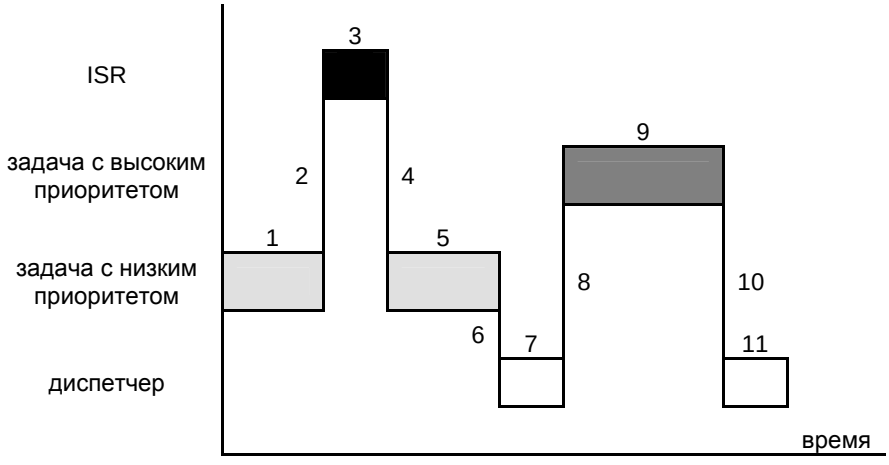

Рисунок 4: Кооперативное планирование

Рисунок 4 иллюстрирует работу кооперативного диспетчера. Как и в предыдущем примере, высокоприоритетная задача выполнится после события, управляемого прерыванием. Событие происходит при выполнении низкоприоритетной задачи [1, 5]. ISR обслуживается [2-4], и диспетчер узнает о событии, но переключения контекста не происходит, пока низкоприоритетная задача явно это не позволит [6]. Как только диспетчер сможет выполниться [7], он запускает высокоприоритетную задачу [8-10]. После нее диспетчер [11] запустит любую готовую продолжаться задачу с наивысшим приоритетом.

По сравнению с приоритетным, кооперативное планирование проще и имеет более короткие ответ на прерывание и восстановление. Но реальный отклик хуже, так как высокоприоритетная готовая продолжать задача не может выполняться, пока низкоприоритетная не освободит процессор явно через переключатель контекста.

Дополнительно о многозадачности

Вы можете думать о задачах как о маленьких программах, которые выполняются внутри большой программы (вашего приложения). Фактически, используя многозадачную RTOS, ваше приложение может выглядеть как каркас для задач и контроля над тем, как и когда они выполняются. Когда приложение выполняется, это означает, что группа небольших программ (задач) ведет себя так, как будто они выполняются одновременно. Конечно, в каждый отдельный момент фактически может выполняться только одна задача. Чтобы получить полное преимущество многозадачных RTOS, вы захотите определить задачи так, чтобы в любой момент времени процессор наилучшим образом использовал ресурсы, выполняя наиболее важную задачу. Если приоритеты определены правильно, об остальном позаботится диспетчер.

Структура задачи

Как фактически выглядит задача в многозадачном приложении? Задача в общем случае – это операция, обязанная выполняться в вашем приложении много раз. Структура очень проста и состоит из инициализации и бесконечного основного цикла. С приоритетным диспетчером, задача может выглядеть следующим образом:

```
Initialize();
while (1)
{
  ...
}
```

Листинг 4: Структура задачи в приоритетной многозадачности

Это корректно потому что приоритетный диспетчер может прервать задачу в любой момент. С кооперативным диспетчером задача должна выглядеть так:

```
Initialize();
while (1)
{
  ...
  TaskSwitch();
  ...
}
```

Листинг 5: Структура задачи в кооперативной многозадачности

Единственное различие между двумя версиями – необходимость явно вызывать переключатель контекста в кооперативной версии. В кооперативной многозадачности каждой задаче необходимо объявить, когда она потенциально пожелает передать управление процессором другой задаче. Такие переключатели контекста обычно безусловны – вызов диспетчера может потребоваться даже если текущая задача – единственная задача, готовая продолжать выполнение. В приоритетной многозадачности это никогда бы не произошло, поскольку диспетчер активирует переключатель контекста только тогда, когда высокоприоритетная задача станет готовой продолжать выполняться.

Замечание: Переключение контекста в любой задаче может происходить много раз, как в приоритетной, так и в кооперативной многозадачной системе.

Простая многозадачность

Простейшая форма многозадачности включает "разделение" процессора поровну между двумя или более задачами. Каждая задача выполняется по очереди некоторый период времени. Это называют *карусель* (*round-robin*) или циклическое выполнение задач.

Это ограниченная утилита, и она страдает из-за проблем архитектуры суперцикла. Дело в том, что все задачи имеют равный, не взвешенный доступ к процессору, и последовательность их выполнения вероятно фиксирована.

Приоритетная многозадачность

Добавление задачам приоритетов кардинально изменяет ситуацию. Дело в том, что, назначая задачам приоритеты, вы можете гарантировать, что в любой момент, ваш процессор выполняет наиболее важную в вашей системе задачу.

Приоритеты могут быть статические или динамические. *Статические приоритеты* – приоритеты, назначенные задачам во времени компиляции и не изменяющиеся во время выполнения приложения. С *динамическими приоритетами* задача может изменять свой приоритет во время выполнения.

Очевидно что, если самой высокоприоритетной задаче позволить выполняться непрерывно, то система больше не будет многозадачной. Как могут сосуществовать в многозадачной системе много задач с различными приоритетами? Ответ в том, как задачи ведут себя фактически – они выполняются не всегда! Вместо этого, то что делает некоторая задача в любое время зависит от ее *состояния* и от других факторов, таких как *события*.

Состояния задачи

RTOS поддерживает каждую задачу в одном из нескольких состояний. Рисунок 5 иллюстрирует различные состояния задачи и разрешенные переходы между ними. *Выполнение* – только одно из нескольких исключительных состояний задачи. Задача также может быть *готова* к выполнению, *задержана*, *остановлена* и даже *разрушена* / *неинициализирована* и может *ждать* событие. Состояния задачи объясняются ниже.

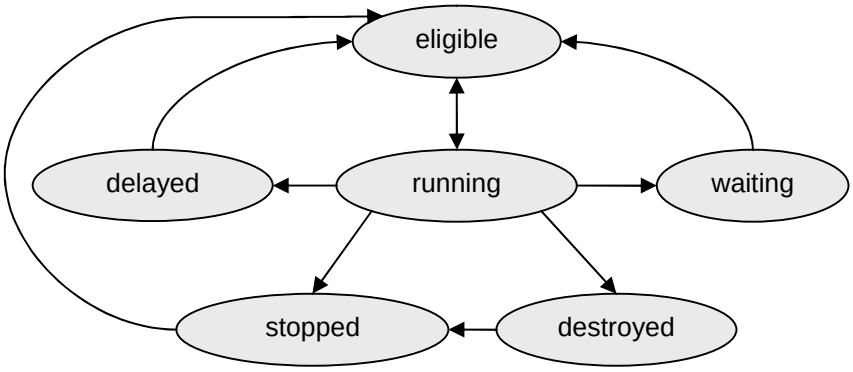

Рисунок 5: Состояния задачи

Прежде, чем задача будет создана, она находится в неинициализированном состоянии. Она возвращается к этому состоянию, когда и если она разрушена (*destroyed*). Немного вы можете сделать с разрушенной задачей, кроме как создать другую на ее месте, или воссоздать ту же самую задачу снова. Задача переходит из разрушенного состояния в остановленное (*stopped*) через обращение к сервису RTOS, которое создает задачу.

Готовая (*eligible*) задача – та, которая готова выполняться, но не может, потому что она – не задача с высшим приоритетом. Она останется в этом состоянии, пока диспетчер не определит, что это – самая приоритетная готовая задача и заставит ее выполняться. Остановленные (*stopped*), задержанные (*delayed*) или ждущие (*waiting*) задачи могут стать готовыми через вызовы соответствующих сервисов RTOS.

Выполняющаяся задача (*running*) возвратится в состояние готовой после простого переключения контекста. Однако она может перейти в разные состояния двумя способами: или задача вызовет сервис RTOS, который уничтожает, останавливает, задерживает или заставляет задачу ждать, или задача будет переведена в одно из этих состояний вызовом сервиса RTOS из другого места приложения.

Задержанная задача – та, которая прежде выполнялась, но теперь приостановлена и ждет окончания работы таймера задержки. Как только таймер отработал, RTOS снова делает задачу готовой.

Остановленная задача прежде выполнялась, а затем была приостановлена на неопределенный срок. Она не будет выполняться снова, пока не произойдет вызов RTOS, который запустит задачу.

Ждущая задача приостановлена и останется в этом состоянии пока не произойдет событие, которого она ожидает (См. "Управляемая событиями многозадачность" ниже).

Для многозадачного приложения типично чтобы различные задачи находились в различных состояниях в любой данный момент времени. Периодические задачи, вероятно, будут задерживаться в некоторые моменты времени. Низкоприоритетные задачи могут быть готовы, но неспособны выполняться, потому что уже выполняется высокоприоритетная задача. Некоторые задачи, вероятно, будут ждать событие.

Задачи могут быть даже разрушены или остановлены. Диспетчер управляет всеми задачами и гарантирует, что каждая задача выполняется тогда, когда нужно. Диспетчер и другие части RTOS гарантируют правильный переход задач из состояния в состояние.

Замечание: Основа приоритетного многозадачного приложения – диспетчер, имеет отношение только к одному – выполнению самой высокоприоритетной готовой задачи. Вообще говоря, диспетчер взаимодействует только с выполняющейся задачей и задачами, которые являются готовыми.

RTOS, вероятно, обрабатывает все задачи в одном состоянии одним способом, и таким образом повышает эффективность вашего приложения. Например, он не должен расходовать время процессора на задачи, которые остановлены или разрушены. Они останутся такими неопределенно долго, пока ваша программа не сделает их готовыми.

Задержки и таймер

Большинство программистов встроенных систем знакомы с простой конструкцией цикла задержки, например:

```
...
for ( i=0; i<100; i++ )
  asm("nop"); /* do nothing for 100 somethings */
...
```

Листинг 6: Цикл задержки

Недостаток выполнения задержек подобных показанной в Листинге 6 в том, что ваше приложение не может делать никакой полезной фоновой работы, пока выполняется цикл. Несомненно, прерывания могут происходить на переднем плане, но разве не было бы полезно быть способным делать кое-что еще в течение задержки?

Другая проблема с кодом Листинга 6 состоит в том, что он является зависимым от компилятора, процессора и быстродействия. Компилятор может уметь или не уметь оптимизировать команды ассемблера, составляющие цикл, приводя к вариациям фактической задержки. Замена процессора может также изменить время задержки. Если увеличить частоту процессора, задержка соответственно уменьшится. Чтобы обойти эти проблемы, циклы задержки часто пишут на ассемблере, но он жестко ограничивает мобильность кода.

RTOS обеспечивает механизм для отслеживания прошедшего времени через системный таймер. Этот таймер часто вызывается в вашем приложении через периодическое прерывание. Каждый раз, когда он вызывается, таймер увеличивает счетчик, содержащий число импульсов системного времени. Текущее число импульсов обычно читаемо и возможно даже записываемо для переустановки счетчика.

Частота вызова таймера выбирается для получения достаточного разрешения, чтобы быть полезной для сервиса времени, например, задерживать задачу или отслеживать прошедшее время. Монитор уровня жидкости может, вероятно, обойтись системными импульсами 1 Hz, но считыватель клавиатуры нуждается в системном сигнале 100 Hz, чтобы определить задержки для алгоритма подавления дребезга контактов. Чрезмерная частота импульсов приведет к реальной перегрузке и меньшему времени, остающемуся для приложения.

Должно также иметься достаточно памяти, выделенной счетчику импульсов системы, чтобы гарантировать, что он не переполнится в течение самого длинного периода времени его использования. Например, однобайтный таймер с 10ms периодом счета обеспечит максимальную задержку задачи 2.55 с. В этом примере будут получены ошибочные результаты, если таймер читается реже, чем через 2.55 с. Задержки задач с подобными ограничениями не работают. Например, система с 10ms системными импульсами и 32-разрядным таймером может задерживать задачу до 497 дней!

Так как задержки используются везде, RTOS может обеспечить встроенный сервис задержек, оптимизированный для минимизации затрат и увеличения производительности. Помещая желаемую задержку внутри задачи, мы можем приостановить задачу на время счета задержки, и затем продолжить задачу, как только задержка истекла. Определение задержки как количество реального времени также значительно улучшит мобильность кода. Код для задержки задачи через RTOS выглядит отличным от Листинга 6:

```
...
OS_Delay(100); /* delay for 100 ticks @ 50 Hz */
...
```

Листинг 7: Задержка через RTOS

В Листинге 7, обращение к сервису RTOS `OS_Delay()` меняет состояние задачи с выполняющейся на задержанную. Так как задача больше не выполняется и даже не готова продолжаться (помните, она задержана), происходит переключение контекста, и (если имеется) самая высокоприоритетная готовая задача начинает выполняться.

`OS_Delay()` также определяет задержку в 100 системных импульсов. Если система имеет импульсы частотой 50 Гц, то задача будет задержана на две полных секунды (100 импульсов по 20 мс.) перед продолжением выполнения, став самой высокоприоритетной готовой задачей. Представьте, как много работы могут сделать другие готовые задачи за две полных секунды!

RTOS может поддерживать несколько, одновременно задержанных задач. Задача разработчика RTOS – максимизировать эффективность – то есть минимизировать издержки, связанные с работой таймера, независимо от того, сколько задач задержано в любой момент времени. Издержки работы таймера не могут быть удалены, они могут быть только минимизированы.

Разрешающая способность и *точность* таймера системы может быть важна для вашего приложения. В простой RTOS, разрешающая способность и точность таймера обе равняются периоду импульса системного времени. Например, задержка задачи n импульсами системного времени приведет к задержке в пределах от n-1 до n импульсов системного времени в реальном времени (например, миллисекунды). Это происходит из-за асинхронной природы системного таймера – если вы задерживаете задачу немедленно после вызова (прерывания) таймера, первый импульс сигнала задержки будет длиться почти полный период импульса системного времени. Если, с другой стороны, вы задерживаете задачу непосредственно перед импульсом системного времени, первый импульс задержки будет в действительности очень короток.

Управляемая событиями многозадачность

Вы могли заметить, что задержанная задача фактически чего-то ждет – она ждет исчерпания таймера задержки. Завершение работы таймера задержки – это пример *события*, а события могут вызвать изменение состояния задачи. Следовательно, события используются для управления выполнением задачи. Примеры событий включают:

- прерывание,
- возникновение ошибки,
- завершение работы таймера,
- периодическое прерывание,
- освобождение ресурса,
- изменение состояния линии ввода-вывода,
- нажатие клавиши на клавиатуре,
- прием или посылка символа по RS-232,
- передача информации от одной части приложения к другой.

Листинг 8: Примеры событий

Кратко говоря, событием может быть любое действие, происходящее внутри или вне процессора. Вы связываете событие с остальной частью вашего приложения (прежде всего с задачами, но также и с ISR и с фоновым кодом) через службы событий RTOS. Взаимодействие между событиями и задачами следует некоторым простым правилам:

- *Создание* события делает его доступным остальной части вашей системы. Вы не можете сообщать о событии, и любая задача не может ожидать события, пока оно не было создано. События могут быть созданы с различными начальными значениями.
- Как только событие создано, оно может *сигнализировать*. Когда событие сигнализирует, мы говорим, что событие произошло. События могут сигнализировать из задачи или из другого фонового кода или из ISR. Что происходит затем, зависит от того, имеется ли одна или большее количество задач, ждущих события.
- Как только событие было создано, одна или большее количество задач могут его *ожидать*. Одна задача может одновременно ждать только одно событие, но то же самое событие может ожидать любое число задач. Если одна или более задач ждут событие, и событие сигнализирует, самая высокоприоритетная задача или первая задача, ожидающая событие, становится готовой к выполнению, в зависимости от того, как RTOS реализует эту возможность. Если множество ждущих задач имеют один и тот же приоритет, RTOS будет иметь четкую схему[12] управления тем, какая задача становится готовой.

[12] В общем случае LIFO или FIFO, т.е. самая последняя по времени создания задача или первая задача соответственно, для ожидания события, чтобы стать готовой, когда событие пришлет сигнал.

Выполняющимся задачам есть резон в прямом ответе на событие, чтобы гарантировать, что в любое время система может ответить на событие так быстро, как только возможно. Это так потому, что ждущие задачи не[13] потребляют временных ресурсов – они остаются в ожидании неопределенно долго, до того как событие, которого они ждут, наконец, произойдет. Кроме того, вы можете выбирать, когда системе реагировать на событие (то есть выполнить связанную задачу) на основании относительной важности, то есть на основании приоритета задачи, связанной с событием.

Ключом к пониманию многозадачных утилит является то, чтобы знать, как структурировать задачи в вашем приложении. Если вы привыкли к программированию суперцикла, вначале может быть трудно. Дело в том, что общий тип мышления состоит в следующем: "Сначала я должен делать это, затем то, затем другое, и т.д. И я должен делать это снова и снова, наблюдая, произошли ли некоторые события и когда". Другими словами, система суперцикла контролирует события последовательным способом и действует соответственно.

Для программирования управляемой событиями многозадачности, вы можете хотеть думать по этим правилам: "Какие события случаются в моей системе, внутри и снаружи, и какие действия я предприму при каждом событии?" Отличие здесь в том, что система является полностью *управляемой событиями*. События могут происходить повторно или непредсказуемо. Задачи выполняются в ответ на события, и доступ задачи к процессору – функция ее приоритета[14]. Задача может реагировать на событие, как только больше не имеется более высокоприоритетной выполняющейся задачи.

Замечание: Приоритеты связаны с задачами, но не событиями.

Чтобы использовать события в вашем многозадачном приложении, вы должны сначала спросить себя:

- что делает моя система?
- как я разделяю действия на отдельные задачи?
- чем занимается каждая задача?
- когда каждая задача выполняется?
- что представляют события?
- какое событие заставляет задачу выполняться?

Замечание: События не должны быть связаны с задачами взаимно однозначно. Задачи могут взаимодействовать со многими событиями и наоборот. Также, задачи, которые не взаимодействуют ни с какими событиями, легко интегрируются в систему, но им назначаются обычно низкие приоритеты, чтобы они выполнились только тогда, когда системе нечего больше делать.

События и межзадачные коммуникации

RTOS поддерживает разные способы связи с задачами. В многозадачности, основанной на событиях, для того, чтобы задача реагировала на событие, событие должно иметь некоторый вид связи с задачей. Задачи могут также желать связываться друг с другом. Для межзадачной связи используются *семафоры*, *сообщения* и *очереди сообщений*, объясняемые ниже.

[13] Если они не ожидают с таймаутом, который требует таймера.
[14] Приоритеты задач легко вводятся в основанную на событиях многозадачность.

Общими для всех этих межзадачных коммуникаций являются два действия: передача *сигнала* (также называемая *регистрация* или *посылка*) и *ожидание* (также называемое *отложенность* или *получение*). Каждая связь также требует инициализации (*создания*).

> **Замечание:** Все операции, включающие семафоры, сообщения и очереди сообщений обрабатываются через обращения к операционной системе.

Семафоры

Имеются два типа семафоров: *двоичные* семафоры и *счетные* семафоры. Двоичный семафор может иметь только два значения, 0 или 1. Счетный семафор может иметь диапазон значений, основанный на его размере – например, значение 8-битного счетного семафора может быть от 0 до 255. Счетные семафоры могут также быть 16-битные или 32-битные. Рисунок 6 иллюстрирует, как мы будем представлять семафоры и их значения:

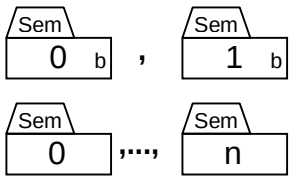

Рисунок 6: Двоичные и счетные семафоры

Прежде, чем использоваться, семафор должен быть создан вместе с начальным значением. Соответствующее значение будет зависеть от того, как семафор используется.

Флаги событий

Флаги событий – одно из использований двоичных семафоров. Они указывают факт происхождения события. Если семафор инициализирован значением 0, это означает, что событие еще не произошло. Когда событие происходит, семафор устанавливается в 1 передачей *сигнала семафору*.

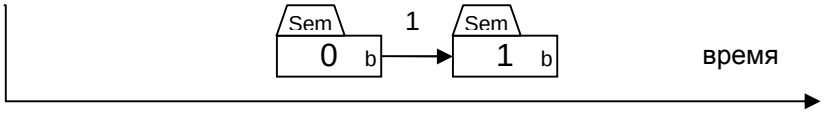

Рисунок 7: Передача сигнала двоичному семафору

Рисунок 7 показывает ISR, задачу или другой фоновый код, передающий сигнал [1] двоичному семафору. Как только семафор (двоичный или счетный) достигает своего максимального значения, дальнейшая передача сигналов является ошибкой.

В дополнение к передаче сигнала семафору, задача может также *ожидать семафор*. Только задачи могут ждать семафор – ISR и другой фоновый код не может. Рисунок 8 иллюстрирует случай уже происшедшего события, когда задача ждет семафор.

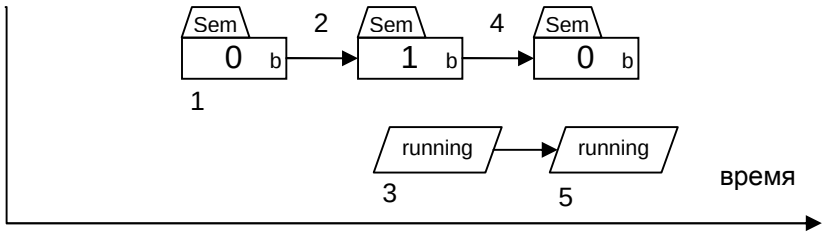

Рисунок 8: Ожидание двоичного семафора когда событие уже произошло

На Рисунке 8 двоичный семафор инициализирован значением 0 [1]. Некоторое время позже, происходит событие, передавая сигнал семафору [2]. Когда задача в заключение выполняется [3] и ждет семафор, семафор будет сброшен [4] так, чтобы он мог сигналить снова, а задача продолжит выполняться [5].

Замечание: Семафоры всегда инициализируются без каких-либо ждущих задач.

Если событие еще не произошло, когда задача ждет семафора, то задача будет *блокирована*. Она останется такой (то есть в ждущем состоянии), пока событие не произойдет. Это показано на Рисунке 9.

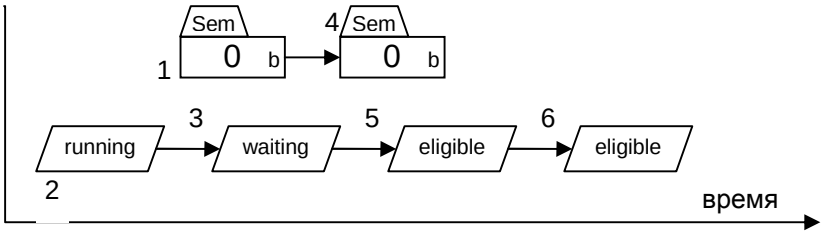

Рисунок 9: Сигнализация двоичного семафора когда задача ждет соответствующее событие

На Рисунке 9, событие еще не сигналило [1], когда выполняющаяся задача [2] ждет двоичный семафор. Так как семафор не установлен, задача блокирована и должна ждать [3] неопределенное время. Операционная система знает, что эта задача блокирована, потому что она ждет специальное событие. Когда семафор, в конечном счете, просигналит извне задачи [4], операционная система делает задачу готовой опять [5], и она выполнится, когда станет наиболее готовой продолжаться [6]. Семафор остается очищенным, потому что задача ждала его, когда он просигналил. Сравните это с Рисунком 7, где семафор сигналит без задач, ждущих его.

Также возможно комбинировать флаги событий, используя конъюнкцию (логическое И) или дизъюнкцию (логическое ИЛИ) флагов событий. Событие сигнализирует, когда установлены все (И) или, по крайней мере, один (ИЛИ) из флагов событий.

Замечание: Одновременно ждать события могут одна или более задач. Какая из задач становится готовой, зависит от операционной системы. Например, некоторые операционные системы могут делать готовой первую из задач ожидающих события (FIFO), а другие могут делать готовой самую высокоприоритетную задачу. Некоторые операционные системы конфигурируются для выбора одной из схем.

Синхронизация задач

Так как задачи могут быть сделаны ожидающими события перед продолжением выполнения, двоичные семафоры могут использоваться как средства *синхронизации* выполнения программы. Многозадачная синхронизация также возможна – Рисунок 10 показывает две задачи, синхронизирующие свое выполнение с помощью двух отдельных двоичных семафоров.

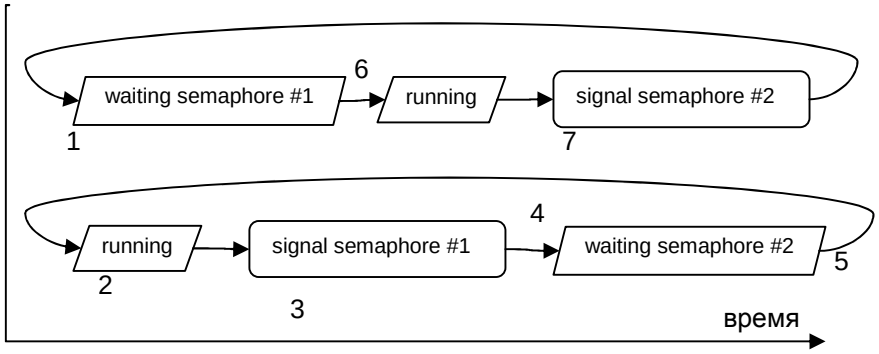

Рисунок 10: Синхронизация двух задач с флагами событий

На Рисунке 10, двоичные семафоры #1 и #2 инициализированы значениями 0 и 1, соответственно. Верхняя задача начинается, ожидая семафора #1, и блокирована [1]. Нижняя задача начинает выполняться [2], и когда она становится готовой ждать верхнюю задачу, она сигнализирует семафором #1 [3], затем ждет семафора #2 [4], и блокируется [5], так как он был инициализирован значением 0. Затем начинает выполняться верхняя задача [6], так как семафор #1 просигнализировал, и когда становится готовой ждать нижнюю задачу, она сигнализирует семафором #2 [7], затем ждет семафора #1, и блокируется [1]. Это продолжается неопределенно долго. Листинг 9 показывает псевдокод для этого примера.

```
initialize binary semaphore #1 to 0;
initialize binary semaphore #2 to 1;

UpperTask()
{
  while (1) {
    /* wait for LowerTask() */
    wait binary semaphore #1;
    do stuff;
    signal binary semaphore #2;
  }
}

LowerTask()
{
  while (1) {
    do stuff;
    signal binary semaphore #1;
    /* wait for UpperTask() */
    wait binary semaphore #2;
  }
}
```

Листинг 9: Синхронизация задач при помощи двоичных семафоров

Ресурсы

Семафоры могут также использоваться для управления ресурсами при помощи *взаимного исключения*. Ресурс доступен, если двоичный семафор установлен в 1, и недоступен, если он равен 0. Задача, которая хочет использовать ресурс, должна *завладеть* им, дождавшись соответствующего значения двоичного семафора. Как только задача завладела ресурсом, двоичный семафор устанавливается в 0 и, следовательно, любые другие задачи, желающие использовать ресурс должны ждать, пока он не будет *освобожден* (сигналом двоичного семафора) задачей, владеющей ресурсом.

```
initialize binary semaphore to 1;

TaskUpdateTimeDate()
{
  while (1) {
    ...
    prepare time & date string;
    wait binary semaphore;
    write time & date string to display;
    signal binary semaphore;
    ...
  }
}

TaskShowAlert()
{
  while (1) {
    wait binary semaphore;
    write alert string to display;
    signal binary semaphore;
  }
}
```

Листинг 10: Использование двоичного семафора для управления доступом к ресурсу

В Листинге 10 двоичный семафор используется для управления доступом к разделяемому ресурсу – дисплею (например, LCD). Для получения к нему доступа, семафор должен быть инициализирован значением 1. Задача, желающая осуществить вывод на дисплей, должна завладеть ресурсом, дождавшись соответствующего сигнала семафора. Если ресурс недоступен, задача будет заблокирована до тех пор, пока ресурс не будет освобожден. После завладения ресурсом и вывода на дисплей, задача должна освободить семафор передачей ему сигнала.

Ресурсы могут также управляться счетными семафорами. В этом случае, значение счетного семафора представляет, *как много ресурсов* доступно для использования. Общий пример такого случая - кольцевой буфер. Кольцевой буфер имеет место для **m** элементов. Элементы добавляются к нему и удаляются из него различными частями приложения. Рисунок 11 показывает схему передачи символьных строк через RS-232, используя счетный семафор для управления доступом к кольцевому буферу.

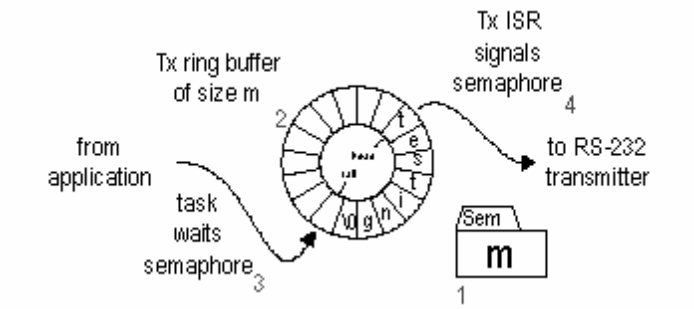

Рисунок 11: Использование счетного семафора для реализации кольцевого буфера

На Рисунке 11 счетный семафор инициализирован значением **m** [1] для представления числа мест, доступных в пустом кольцевом буфере [2]. Кольцевой буфер заполняется с хвоста[15] задачей [3] и освобождается из головы процедурой обработки прерывания ISR [4]. Перед добавлением символа в буфер задача должна дождаться семафора. Если она блокирована, это означает, что буфер полон и не может больше принимать символы. Если буфер не полон, семафор декрементирован, задача помещает символ в хвост буфера и увеличивает указатель хвоста списка. Если в буфере имеются символы[16], каждый символ будет извлечен из буфера прерыванием Tx ISR, передан и семафор увеличит свое значение передачей сигнала. Соответствующий псевдокод показан[17] в Листинге 11.

```
initialize counting semaphore to m;

TaskFillTxBuffer()
{
  while (1) {
    wait semaphore;
    place char at TxBuff[tail pointer];
    increment tail pointer;
  }
}

ISRTxChar()
{
  send char at TxBuff[head pointer] out RS-232;
  increment head pointer;
  signal semaphore;
}
```

Листинг 11: Использование счетного семафора для управления доступом к ресурсу

[15] Указатель хвоста указывает на следующее доступное свободное место для вставки очередного элемента в кольцевой буфер. Указатель головы указывает на первый доступный элемент для извлечения и удаления из кольцевого буфера.

[16] Обычно это означает разрешение прерываний по передаче.

[17] Управление прерываниями Tx, которые отличаются в зависимости от конфигурации передатчика, не показано.

Используя задачу для заполнения кольцевого буфера, приложению не требуется опрашивать состояние буфера равномерно для определения момента, когда вставлять новые символы. Приложение также не обязано ожидать в цикле освобождения места в буфере для вставки символов. Если вставлена только часть строки, прежде чем задача будет блокирована (то есть строка больше чем доступное место в буфере), задача автоматически продолжит вставлять дополнительные символы каждый раз, когда прерывание ISR передаст сигнал в счетный семафор. Если приложение посылает строки нечасто, вероятно приложение удовлетворит низкий приоритет задачи. Иначе может потребоваться высокий приоритет.

Замечание: Оперативная память, требуемая для семафора, используемого для управления ресурсом, отделена от оперативной памяти, выделенной самому ресурсу. RTOS выделяет память для семафора – *пользователь должен выделить память для ресурса.* В данном примере, 8-разрядные счетные семафоры ограничивают размер кольцевого буфера 256 символами. Семафор требует одного байта оперативной памяти независимо от фактического (объявленного пользователем) размера кольцевого буфера.

Сообщения

Сообщения обеспечивают средства посылки задаче произвольной информации. Информация может быть числом, строкой, массивом, функцией, указателем или чем-нибудь еще. Любое сообщение в системе может быть произвольным пока и отправитель, и получатель сообщения понимают его содержание. Даже тип сообщения может изменяться от одного сообщения к другому, пока и отправитель и получатель знают об этом! Как с семафорами, операционная система обеспечивает средства создания, передачи и ожидания сообщений.

Для обеспечения универсальности содержания, когда сообщение посылается и получается, фактическое содержание сообщения несет не саму информацию, а, скорее всего, *указатель на информацию*. Указатель или *адрес информации* сообщает, где найти саму информацию. Получатель сообщения использует указатель, чтобы получить информацию, содержащуюся в сообщении. Это называется *разыменование* указателя[18].

Если сообщение инициализировано пустым содержанием, оно содержит *пустой указатель*. Пустой указатель имеет значение 0. По соглашениям, пустой указатель не указывает ни на что – следовательно, он не несет никакой другой информации. Пустой указатель не может быть разыменован.

Передача сигнала (то есть посылка) сообщения более сложна, чем передача сигнала семафора. Дело в том, что функция для передачи сигнала сообщения операционной системы требует указателя сообщения как параметр. Указатель, переданный функции, должен правильно указывать на информацию, которую вы желаете передать как сообщение. Этот указатель обычно отличен от нуля и иллюстрируется Рисунком 12.

[18] В языке Си символ `&` представляет адрес оператора, а символ `*` является унарным оператором косвенного обращения. Поэтому, если `var` представляет переменную, а `p` указывает на нее, то `p = &var` ,а `*p` является эквивалентом `var`.

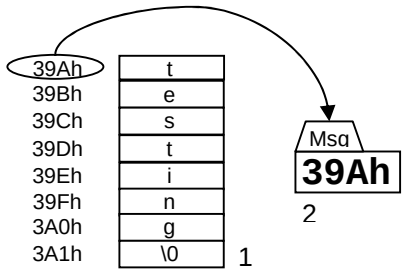

Рисунок 12: Сигнализация сообщения с указателем на содержимое сообщения

На Рисунке 12 строка символов языка Си[19] [1] посылается в сообщении [2] передачей сигнала сообщения с указателем. Строка постоянно находится по некоторому физическому адресу. Сообщение не содержит первый символ строки – оно содержит адрес первого символа строки (то есть указатель на строку), а значение указателя равно 39Ah. Псевдокод для посылки этого сообщения показано в Листинге 12.

```
string[] = "testing";
p = address(string);
signal message with p;
```

Листинг 12: Передача сигнала сообщения с указателем

Для получения содержания сообщения, задача должна ждать сообщение. Задача будет блокирована до тех пор, пока сообщение не прибудет. Затем задача извлекает содержимое сообщения (то есть указатель) и использует его любым выбранным способом. В Листинге 13, принимающая задача преобразует строку, на которую указывает сообщение, в строку с прописными буквами.

```
TaskCaps()
{
  while (1) {
    wait message containing string pointer p;
    while ((p) is not null) {
      if ('a' <= (p) <= 'z')
      (p) = (p) – 32;
      increment p;
    }
  }
}
```

Листинг 13: Прием сообщения и операция над его содержимым

Сообщение может содержать максимум одну единицу информации (то есть указатель) одновременно. Если сообщение пусто, оно может сигнализировать. Если оно заполнено, сообщение не может быть послано.

Сообщения могут использоваться подобно двоичным семафорам. Сообщение, содержащее пустой указатель эквивалентно двоичному семафору со значением 0, а сообщение, содержащее непустой указатель эквивалентно двоичному семафору со значением 1. Это полезно когда двоичные семафоры не поддерживаются RTOS явно.

[19] В языке Си символьная строка завершается символом NUL (`'\0'`).
[20] "(pointer)" представляет псевдокод для "what is pointed to by the pointer".

Очереди сообщений

Очереди сообщений являются расширением сообщений. Очередь сообщений может содержать множественные сообщения (до определенного количества) в любое время. Посылка сообщений может продолжаться до тех пор, пока почтовый ящик сообщений не заполнен. Задача, ожидающая очередь сообщений, будет получать их, пока очередь сообщений не пуста.

RTOS будет должна выделить некоторую дополнительную оперативную память, чтобы управлять каждой очередью сообщений. Эта память будет использоваться, чтобы следить за числом сообщений и порядком, в котором сообщения существуют в очереди.

Сводка о взаимодействии задач и событий

Представим свод правил управления взаимодействием задач и событий (т.е. семафорами, сообщениями и очередями сообщений).

- События должны быть инициализированы. Инициализация не требует наличия каких-либо ожидающих его задач.
- Задача не может ожидать событие до его инициализации.
- Несколько задач могут ожидать одно событие.
- Задача может ожидать одновременно только одно событие.
- Значение семафора может иметь величину от 0 до его максимального значения, определяемого размером семафора.
- Сообщение содержит указатель на некоторую информацию.
- Очередь сообщений может содержать несколько сообщений одновременно.
- Прерывание (ISR), задача или другой фоновый код могут сигнализировать о событии.
- Только задача может ожидать событие.
- Задача будет блокирована (например, она изменит статус на ожидающую) если ожидаемое событие недоступно.
- Какая из ждущих задач станет готовой, когда событие сигнализирует, зависит от того, как операционная система реализует обслуживание событий.
- Если событие уже просигналило, и отсутствуют задачи, ожидающие его, но событие снова сигналит, то либо произошла ошибка, либо сигналящая задача может быть заблокирована. Это зависит от того, как операционная система осуществляет обслуживание события.

Конфликты

В многозадачной среде могут происходить разнообразные конфликты. Их разновидности описаны ниже.

Взаимоблокировка

Взаимоблокировка происходит с двумя или большим числом задач, когда каждая задача ждет ресурс, контролируемый другой задачей. Так как все, участвующие в конфликте задачи находятся в ожидании, ни один из ресурсов не имеет никакой возможности стать доступным. Следовательно, все задачи будут блокированы, то есть они будут неопределенно долго ожидать.

Решение для всех задач, желающих завладеть ресурсами, состоит в следующем:

- всегда завладевайте ресурсами в предопределенном порядке,
- завладевайте всеми ресурсами перед продолжением, и
- освобождайте ресурсы в обратном порядке.

Используя *таймаут* можно прерывать взаимоблокировку. При попытке приобрести ресурс, может быть определен опциональный период времени. Если ресурс не приобретен в заданный период времени, задача продолжается, но с кодом ошибки, указывающим, что истекло время ожидания ресурса. Затем может вызываться специальная обработка ошибок.

Инверсия приоритетов

Инверсия приоритетов происходит тогда, когда высокоприоритетная задача ожидает ресурс, контролируемый низкоприоритетной задачей. Высокоприоритетная задача должна ждать, пока низкоприоритетная задача не освободит ресурс, после чего она может продолжиться. В результате, приоритет высокоприоритетной задачи реально понижается до приоритета низкоприоритетной задачи.

Имеется ряд способов избежать этой проблемы (например, *наследование приоритета*), большинство из которых включают динамическое изменение приоритета задачи, которая контролирует ресурс, основанное на приоритете задач, желающих, завладеть ресурсом.

Эффективность RTOS

Код, реализующий многозадачный режим RTOS может быть большим, чем тот, который требуется для реализации суперцикла. Дело в том, что каждая задача требует ряда дополнительных инструкций, совместимых с диспетчером задач. Но даже в этом случае, многозадачное приложение, вероятно, будет иметь много большую эффективность и лучшую реакцию, чем приложение с суперциклом. Дело в том, что хорошо написанная RTOS может пользоваться преимуществом того, что задачи, которые не требуют частого выполнения, почти не потребляют ресурсы процессора. Это означает, что вместо того, чтобы тратить циклы команд, опрашивая флаги, проверяя счетчики и отслеживая события, многозадачное приложение максимально использует ресурсы процессора, затрачивая их там, где вы в этом наиболее нуждаетесь – в самой высокоприоритетной задаче, готовой продолжать выполнение.

Реальный пример

Рассмотрим интересный пример приложения – контроллер удаленного торгового автомата продажи содовой в банках. Он должен указывать (светодиодами в кнопках) отсутствие выбора, обрабатывать вставку монет и суммарный счет, правильно интерпретировать выбор заказчика, отпускать ему правильный товар, и правильно производить все изменения. Современный, управляемый микропроцессором торговый автомат мог бы также регулировать внутреннюю температуру (например, для банок с содовой), быть соединенным с сетью, чтобы передавать информацию удаленному компьютеру, и быть привязанным к системе защиты от вандализма. И конечно все это должно быть выполнено без ошибок, независимо от того, сколько непредсказуемых действий совершает заказчик в попытке удовлетворить свой голод или жажду.

Стандартный подход с суперциклом

Охлаждающий, вандалоустойчивый торговый автомат в нашем примере имеет интерфейс пользователя, состоящий из массива кнопок выбора товара и щелей для счетов и монет. Основной цикл для псевдокода версии традиционной реализации суперцикла мог бы выглядеть следующим образом:

```
Initialize();
do forever
{
  ControlTemps();
  ShowEmpties();

  AcceptCurrency();

  flagSelectionGood = FALSE;
  ReadButtons();

  If ( flagSelectionGood ) {
    ReleaseItem();
    MakeChange();
  }

  if ( Tilt() ) {
    CallPolice();
  }
}
```

Листинг 14: Суперцикл торгового автомата

Здесь использованы некоторые обработчики прерываний – ISR (не показаны), чтобы выполнять процедуры, подобные подавлению дребезга кнопок. Листинг 14 также не показывает ни индивидуальных функций (например, `ReleaseItem()`) ни глобальных переменных, требуемых, чтобы передавать информацию между функциями, например между `ReadButtons()` и `ReleaseItem()`.

Исследуем Листинг 14 более подробно. В суперцикле мы вызываем `ControlTemps()` один раз каждый цикл. На 8-разрядном 8 MHz процессоре, вероятно используемом в таком приложении, мы могли бы получить вызов `ControlTemps()` один раз каждые 200 микросекунд при отсутствии действий пользователя. Это – огромная трата временных ресурсов процессора, поскольку мы знаем, что в реальности необходимо вызывать это только один раз в минуту. Мы вызываем `ControlTemps()` в 5,000 раз более часто, чем это необходимо! Если это еще может быть приемлемо в торговом автомате, то вряд ли так будет в более требовательном приложении.

Один из подходов исправления этого мог бы выделить периодическое прерывание для установки каждую секунду глобально видимого бита. Затем мы могли бы проверять этот бит и вызывать `ControlTemps()` когда бит установлен в высокий уровень. Этот подход не слишком разумен, потому что мы все еще делаем операцию (проверку бита) каждые 200 микросекунд. Другой подход мог бы переместить `ControlTemps()` полностью в обработчик прерывания – ISR, который вызывался бы каждую секунду, но это было бы опрометчиво, особенно, если `ControlTemps()` – большая и сложная функция.

В нашем примере, `ReleaseItem()` выполнится только тогда, когда деньги находятся в машине, и кнопка была нажата. Другими словами, функция ждет события – события, характеризуемого присутствием соответствующего количества денег И нажатия допустимой кнопки выбора.

Как иллюстрирует Листинг 14, программные проекты с суперциклом переднего/заднего плана помещают большинство требуемой обработки в одиночном основном цикле, который процессор выполняет снова и снова. Внешние события и критические по времени процедуры обрабатываются на переднем плане обработчиками прерываний ISR. Обратите внимание, что ни одна операция в суперцикле не имеет приоритета над любой другой. Выполнение функций продолжается жестко последовательным способом с использованием многих иерархических циклов. При добавлении большей функциональности системе подобной этой, основной цикл, вероятно, станет большим и медленным, возможно будет необходимо большее количество ISR, и сложность системы увеличится в попытке сохранить все в целом работающим.

Например, в вышеприведенном варианте для покупателя не имеется никакого способа отменить приобретение. Как бы вы изменили код, чтобы обработать это дополнительное требование? Вы могли бы написать расширенный конечный автомат, чтобы обработать различные сценарии или использовать большое количество прерываний от таймера для контроля того, как часто могут выполняться различные функции. Но как вы думаете, кто-нибудь другой понял бы то, что вы написали? Или даже вы, два года спустя?

Управляемый событиями подход RTOS

Если мы начали говорить о понимании, поддержке и модификации кода переднего/заднего плана от умеренной до значительной сложности, он теряет привлекательность. Дело в том, что в суперцикле не имеется никаких чистых связей ни между различными функциями, ни между функциями и переменными флагов, ни между ISR и суперциклом. Попробуем подход, основанный на задачах и событиях.

Это список задач, которые мы можем идентифицировать в предыдущем примере:

- Контроль и управление температурой – `ControlTemps()`
- Индикация пустых ячеек светодиодами – `ShowEmpties()`
- Прием или возврат денег и их суммирование – `AcceptCurrency()`
- Подавление дребезга и чтение кнопок – `ReadButtons()`
- Проведение изменений – `MakeChange()`
- Отпуск выбранного товара клиенту – `ReleaseItem()`
- Попытка защиты автомата от вандализма – `CallPolice()`

Исследуем каждую из этих задач несколько более детально. Мы рассмотрим степень важности каждой из них от 1 (наиболее важная) до 10 (наименее важная) и когда каждая задача должна выполняться.

`ControlTemps()` очевидно важна, поскольку мы хотим хранить содовую прохладной. Но она, вероятно, не должна выполниться чаще чем, скажем, один раз в минуту, чтобы точно контролировать и регулировать температуру. Мы дадим этой задаче приоритет 4.

`ShowEmpties()` не слишком важна. Кроме того, состояние пустых ячеек каждый раз изменяется только при отпуске товара заказчику. Так что мы дадим этой задаче приоритет 8, и мы хотели бы, чтобы она выполнялось в начале и по одному разу при каждом отпуске товара.

`ReadButtons()` должна иметь достаточно высокий приоритет, чтобы не имелось значительного запаздывания, когда заказчик нажимает кнопки машины. Так как кнопки нажимаются асинхронно, мы хотим проверять массив кнопок на активность регулярно. Дадим этой задаче приоритет 3, и будем выполнять ее каждые 40 миллисекунд.

`AcceptCurrency()` также является частью интерфейса пользователя. Мы дадим ей тот же приоритет, что и `ReadButtons()` и будем выполнять каждые 20 миллисекунд.

Изготовитель машины не считает `MakeChange()` важной, так что мы дадим ей приоритет 10. Мы свяжем ее с `ReleaseItem()` так как изменение должно быть сделано только после того, как выбранный товар отпущен заказчику.

`ReleaseItem()` интересна тем, что мы нуждаемся в ней только тогда, когда было принято соответствующее количество денег и нажата кнопка товара. Для быстроты ответа дадим ей приоритет 2, и мы хотели бы, чтобы она выполнялась, когда случается вышеупомянутая комбинация денег и нажатия кнопки.

Изготовитель машины также придает большое значение ее вандалоустойчивости. Она даже способна обнаруживать нападение (встроенными датчиками наклона) и вызывать локальную службу безопасности. Мы дадим `CallPolice()` самый высокий приоритет – 1 и будем каждые 2 секунды проверять датчики наклона на нападение.

Шаг за шагом

Наш пример торгового автомата требует семи задач с шестью различными приоритетами и таймера с разрешающей способностью 20 ms. Чтобы создать многозадачное приложение из этих функций, нам необходимо:

- инициализировать операционную систему,
- модифицировать структуру задач для совместимости с операционной системой и событиями,
- создать задачи с приоритетами из функций задач,
- связать реальные события с событиями, которые понимает операционная система,
- создать системный таймер, для того, чтобы следить за прошедшим временем,
- запустить различные задачи и
- начать выполнение многозадачности.

Инициализация операционной системы

Инициализация операционной системы обычно проста, например:

```
InitializeMultitasking();
```

Это создает необходимые (пустые) структуры, которые операционная система использует для управления выполнением задач и событиями. В этом месте все задачи системы находятся в *неинициализированном*/*разрушенном* состоянии.

Структурирование задач

Задачи для многозадачного приложения, выглядят подобно задачам, для приложения с суперциклом. Наибольшее отличие состоит в структуре полной программы. Задачи для многозадачности не входят в какие-либо циклы или большие функции – они являются независимыми функциями. `ReleaseItem()`отпускает товар один раз, если условия выполнены, и на псевдокоде может выглядеть так:

```
ReleaseItem()
{
  do forever {
    WaitForMessage(messageSelection, item);
    Release(item);
  }
}
```

Листинг 15: Версия задачи ReleaseItem()

В Листинге 15 `ReleaseItem()` бесконечно ожидает специфическое сообщение и не делает ничего, пока сообщение не поступает. В то Пока ожидается сообщение, `ReleaseItem()` находится в ждущем состоянии. Когда сообщение послано, `ReleaseItem()` становится готовой продолжать выполнение, и когда это случается, задача извлекает содержимое сообщения (в данном случае, код для желаемого товара, например "B3") и отпускает товар заказчику. `ReleaseItem()` не находится внутри какого-либо большого цикла и не вызывается какими-либо кроме косвенного вызова диспетчером, см. ниже). `CallPolice()` имеет подобную самостоятельность:

```
CallPolice()
{
  do forever {
    Delay(1000);
    If ( Tilt() ) {
      SendMsgToPoliceHQ();
    }
  }
}
```

Листинг 16: Версия задачи CallPolice()

`CallPolice()` вводит бесконечный цикл, где задерживает себя на 1000 x 20 ms, или 2 секунды, и затем посылает сообщение полицейской штаб-квартире, если датчики наклона торгового автомата обнаруживают нападение. Она повторяет эту последовательность неопределенно долго. Пока задача задержана, `CallPolice()` находится в *задержанном* состоянии.

Приоритизация задач

Вызов операционной системы назначает приоритет задаче и подготавливает задачу к многозадачности. Например,

```
CreateTask(ShowEmpties(), 8)
```

Листинг 17: Приоритизация задачи

сообщает операционной системе, что она должна дать `ShowEmpties()` приоритет 8 и добавить ее к задачам, чье выполнение она будет контролировать. `ShowEmpties()` теперь находится в *остановленном* состоянии.

Интерфейс с событиями

В Листинге 15 `ReleaseItem()` использует сообщение, чтобы обработать событие – а именно отпуск товара. Это сообщение должно быть инициализировано:

```
CreateEvent(messageSelection, empty);
```

Листинг 18: Создание события сообщения

Инициализируя `messageSelection` как пустое (то есть допустимый выбор не был сделан), `ReleaseItem()` отпустит товар только один раз когда произойдут требуемые события (достаточно вставленных денег и нажата подходящая кнопка).

Добавление системного таймера

RTOS нуждается в некотором способе слежения за реальным временем. Это обычно обеспечивается некоторым видом функции таймера, которую приложение должно вызывать регулярно с предопределенной частотой. В данном случае эта частота равна 50 Hz или вызов каждые 20 ms. Вызов системного таймера часто выполняется через прерывание, например:

```
InterruptEvery20ms()
{
  SystemTimer();
}
```

Листинг 19: Вызов системного таймера

Запуск задач

Приложения должны создавать все свои задачи и события прежде, чем любые из них будут фактически использоваться. Обеспечивая явные средства старта задач, RTOS позволяет вам управлять запуском системы предсказуемым способом:

```
StartTask(ControlTemps());
StartTask(ShowEmpties());
StartTask(AcceptCurrency());
StartTask(ReadButtons());
StartTask(MakeChange());
```

```
StartTask(ReleaseItem());
StartTask(CallPolice());
```

Листинг 20: Запуск всех задач

Так как многозадачный режим еще не начался, порядок, в котором задачи стартуют несущественен и в любом случае не зависит от их приоритетов. В этом месте все задачи находятся в *готовом* состоянии.

Разрешение многозадачности

Как только все находится на своем месте, события были инициализированы и задачи были запущены (то есть все они готовы к выполнению), многозадачный режим может начинаться:

```
StartMultitasking();
```

Листинг 21: Начало мгогозадачности

Диспетчер будет брать готовую задачу с самым высоким приоритетом и выполнять ее – то есть эта задача будет в состоянии *выполнения*. С этого момента, диспетчер гарантирует, что в любое время единственной выполняющейся будет самая высокоприоритетная задача.

Объединение всего вместе

Листинг 22 представляет полный текст управляемого задачами и событиями приложения торгового автомата на псевдокоде:

```
#include "operatingsystem.h"

extern AlertPoliceHQ()
extern ButtonPressed()
extern DisplayItemCounts()
extern InterpretSelection()
extern NewCoinsOrBills()
extern PriceOf()
extern ReadDesiredTemp()
extern Refund()
extern ReleaseToCustomer()
extern SetActualTemp()
extern Tilt()

ControlTemps()
{
  do forever {
    Delay(500);
    ReadActualTemp();
    SetDesiredTemp();
  }
}

ShowEmpties()
{
  DisplayItemCounts();
  do forever {
    WaitForSemaphore(semaphoreItemReleased);
    DisplayItemCounts();
  }
}
```

```
AcceptCurrency()
{
  do forever {
    Delay(1);
    money += NewCoinsOrBills();
  }
}

ReadButtons()
{
  do forever {
    Delay(2);
    button = ButtonPressed();
    if ( button ) {
      item = InterpretSelection(button);
      SignalMessage(messageSelection, item);
    }
  }
}

MakeChange()
{
  do forever {
    WaitForMessage(messageCentsLeftOver, change);
    Refund(change);
  }
}

ReleaseItem()
{
  CreateEvent(semaphoreItemReleased, 0);
  CreateEvent(messageCentsLeftOver, empty);

  do forever {
    WaitForMessage(messageSelection, item);
    if ( money >= PriceOf(item) ) {
      ReleaseToCustomer(item);
      SignalSemaphore(semaphoreItemReleased);
      SignalMessage(messageCentsLeftOver,
        money - PriceOf(item));
        money = 0;
    }
  }
}

CallPolice()
{
  do forever {
    Delay(1000);
    if ( Tilt() ) {
      AlertPoliceHQ();
    }
  }
}

InterruptEvery20ms()
{
  SystemTimer();
}
```

```
main()
{
  money = 0;

  InitializeMultitasking();

  CreateTask(ControlTemps(),   4)
  CreateTask(ShowEmpties(),    8)
  CreateTask(AcceptCurrency(), 3)
  CreateTask(ReadButtons(),    3)
  CreateTask(MakeChange(),    10)
  CreateTask(ReleaseItem(),    2)
  CreateTask(CallPolice(),     1)

  CreateEvent(messageSelection, empty);

  StartTask(ControlTemps());
  StartTask(ShowEmpties());
  StartTask(AcceptCurrency());
  StartTask(ReadButtons());
  StartTask(MakeChange());
  StartTask(ReleaseItem());
  StartTask(CallPolice());

  StartMultitasking();
}
```

Листинг 22: Торговый автомат на основе RTOS

Различия в RTOS

Программа в Листинге 22 имеет полностью отличную структуру, чем суперцикл в Листинге 14. Несколько различий совершенно очевидны:

- Она несколько больше – в основном из-за расходов на создание обращений к операционной системе.
- Имеются четко определенные приоритеты времени выполнения, связанные с каждой задачей.
- Сами задачи имеют простые структуры и легки для понимания. Задачи, которые связываются с другими задачами или ISR, используют очевидные механизмы (например, семафоры и сообщения). Инициализация задач может иметь специфику.
- Использование глобальных переменных минимизировано.
- Отсутствуют циклы задержки.
- Очень просто изменять, добавлять или удалять одну задачу без воздействия на другие.
- Полное поведение приложения в значительной степени зависит от приоритетов задач и межзадачной связи.

Возможно, наиболее важным является то, что RTOS обрабатывает сложность приложения автоматически – задачи выполняются на основании приоритетов, переключение задач и изменения состояний обрабатываются автоматически, задержки требуют минимума ресурсов процессора, и механизмы межзадачных связей скрыты от просмотра.

Имеются и другие различия, которые становятся более очевидными во время выполнения. Если проследить во времени за активностью задачи, то можно заметить:

- Каждые 2 секунды `CallPolice()` пробуждается для проверки вмешательства и затем возвращается в состояние задержки,
- Каждую 1 секунду `ControlTemps()` пробуждается для коррекции внутренней температуры и затем возвращается в состояние задержки,
- Каждые 40 ms `ReadButtons()` пробуждается для обработки дребезга любой нажатой кнопки и затем возвращается в состояние задержки,
- Каждые 20 ms `AcceptCurrency()` пробуждается для мониторинга вставки монет и счетов и затем возвращается в состояние задержки, и
- `ShowEmpties()`, `MakeChange()` и `ReleaseItem()`ничего не делают, пока не будет сделан допустимый выбор, после чего они ненадолго "приходят в себя", предоставляют выбранный товар, обрабатывают любые изменения и показывают состояния полных и пустых ячеек перед возвратом в состояние ожидания.

Другими словами, подавляющее большинство времени выполнения микроконтроллер торгового автомата имеет очень немного работы, потому что диспетчер видит только задержанные и ждущие задачи. Изготовитель торгового автомата может захотеть поддержать "Связь по Интернет для расширенного управления, удаленного запроса и повышения прибыли" как дополнительную возможность. Потребовалось бы добавление дополнительной задачи для передачи данных о продаже (например, какая содовая приобретена, в какое время и день, и какова наружная температура) и выполнение простого сервера сети. Это также просто, как добавление новой задачи к описанным выше и назначение ей соответствующего приоритета.

Глава 3 • Инсталляция

Введение

Salvo поставляется в виде самораспаковывающегося архива. Каждый инсталлятор устанавливает файлы, необходимые для создания приложения Salvo для целевого процессора и компилятора, а также дополнительные файлы, такие как *Salvo Compiler Reference Manuals*. Все файлы Salvo содержатся в сжатой и зашифрованной форме внутри инсталлятора.

Замечание: В этом разделе предполагается, что вы инсталлируете Salvo на PC совместимый компьютер с Microsoft Windows XP. Инсталляция в другие версии Windows аналогична.

Запуск инсталлятора

1. Запустите инсталлятор, соответствующий дистрибутиву `salvo-lite|tiny|LE|SE|Pro-target-version.exe`, на вашем Wintel PC. Появится экран приглашения:

Рисунок 13: Экран приглашения

Замечание: Большинство экранов инсталлятора содержит кнопки **Next**, **Back** и **Cancel**. Нажмите кнопку **Back** для возврата к предыдущему экрану. Нажмите кнопку **Cancel**, чтобы прервать установку. Нажмите кнопку **Next** для продолжения процесса инсталляции.

2. При нажатии **Next** появится соглашение о лицензировании Salvo:

Рисунок 14: Экран соглашения о лицензировании

Этот экран содержит *Лицензионное соглашение Pumpkin Salvo*. Прочтите его внимательно. Этот документ будет включен в папку Salvo по окончании инсталляции. Чтобы продолжить инсталляцию Salvo, вы должны принять условия Лицензии, нажатием **I Agree**. Если вы не принимаете Лицензию, нажмите **Cansel** и возвратите продукт[21].

3. Экран выбора компонентов:

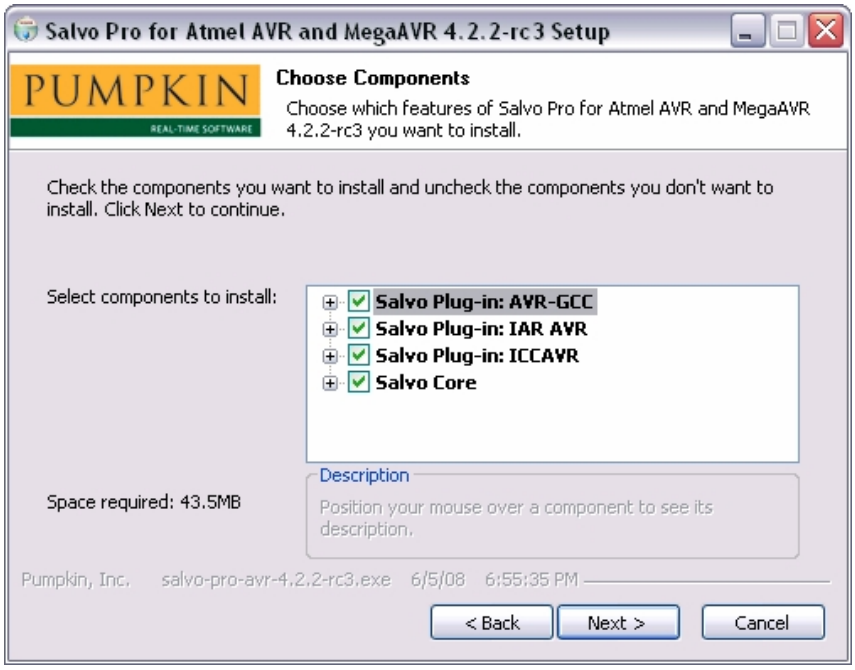

Рисунок 15: Экран выбора компонентов

[21] Инструкции по возврату программного продукта содержатся в Лицензии и Руководстве Пользователя.

Обычно стоит этот выбор оставить в состоянии по умолчанию. Компоненты типично содержат ядро Salvo (все независимые от целевого процессора и компилятора компоненты Salvo), а также специфические для целевого процессора и компилятора файлы.

Совет: Описание состава и функции каждого индивидуально выбираемого компонента доступно при раскрытии узлов дерева и пометке мышью интересующего компонета.

Этот экран позволяет удобно восстанавливать компоненты Salvo без полной переинсталляции пакета. Например, если вы случайно удалили библиотеку Salvo для определенного компилятора, вы можете через этот экран переинсталлировать только указанные вами библиотеки Salvo.

Совет: Инсталлированные компоненты для неинсталлированных инструментальных средств обычно не создают проблем. Поэтому рекомендуется оставить все компоненты выбранными по умолчанию.

4. Экран выбора места инсталляции.

Рисунок 16: Экран выбора места инсталляции

Этот экран позволяет вам определить каталог, в который будет установлена Salvo. Инсталлятор поместит в выбранном месте несколько[22] каталогов, некоторые из которых содержат вложенные подкаталоги. Вы можете оставить каталог установки по умолчанию (`C:\Pumpkin\Salvo`) или изменить его, нажав на кнопку **Browse...** и выбрав другой каталог.

Совет: Во избежание потенциальных проблем компилятора с длинными именами путей и пробелами, рекомендуется выбор по умолчанию. Выбор чрезмерно глубоко вложенного каталога (как `C:\My Projects\Programming\Tools\RTOS\Salvo`) может вызвать проблемы с некоторыми инструментальными средствами из-за очень длинных имен путей файлов Salvo. Также некоторые компиляторы не поддерживают пробелов.

[22] См. Рисунок 20 "Содержание типичной директории Salvo".

5. Далее, после нажатия кнопки **Next**, появится экран выбора папки стартового меню:

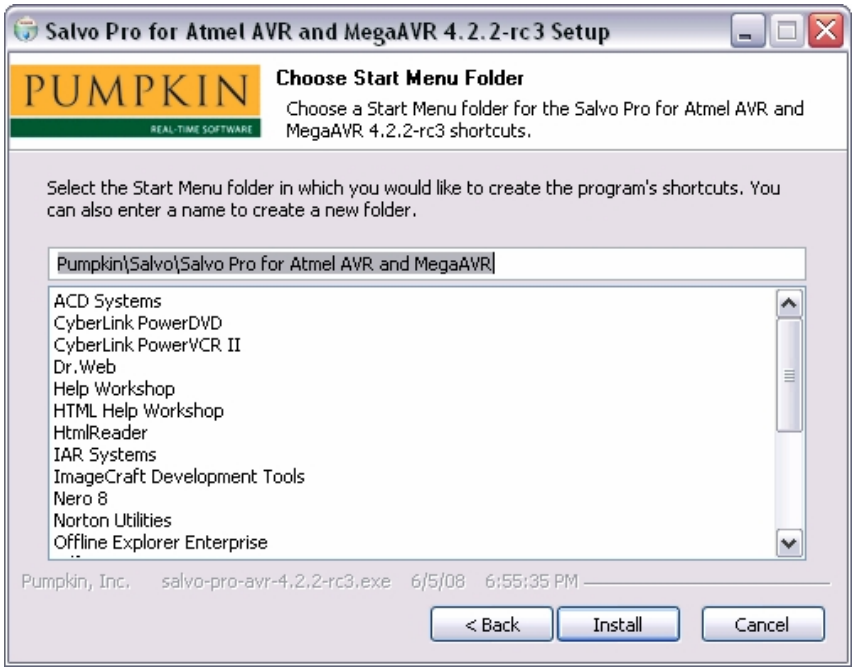

Рисунок 17: Экран выбора папки стартового меню

6. Выберите **Install** для продолжения. Инсталляция начнется и закончится на экране завершения установки:

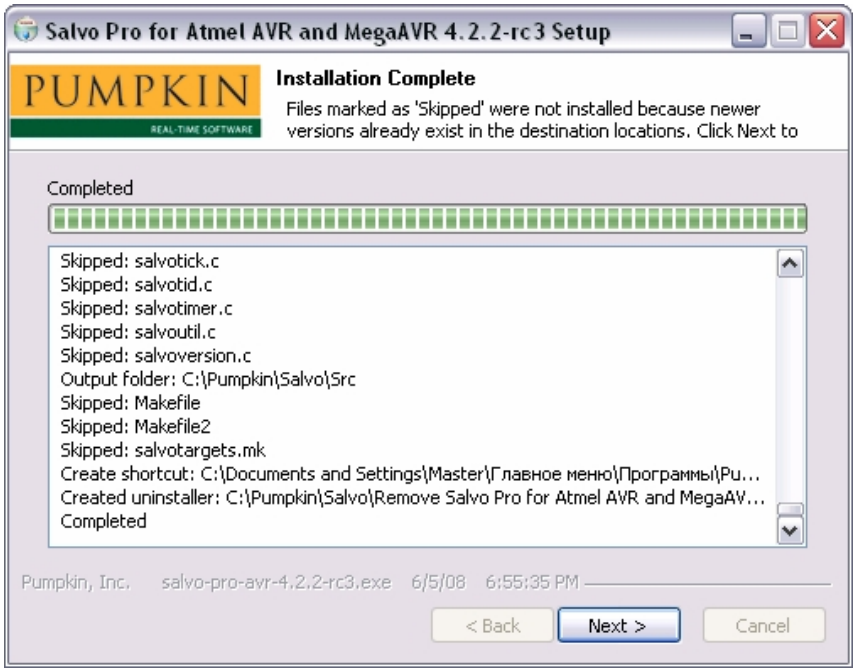

Рисунок 18: Экран завершения установки

Этот экран содержит список всех файлов дистрибутива Salvo, инсталлированных на ваш компьютер. Отдельные файлы маркированы, как **Extract** (файл инсталлирован) или **Skipped** (файл был пропущен, потому что был обнаружен уже установленный более новый файл с тем же именем).

Совет: Вывод на этот экран может быть просмотрен при помощи линейки прокрутки справа.

7. Когда инсталляция файлов завершится, щелкните кнопку **Next**. Вы получите приветствие на экране окончания.

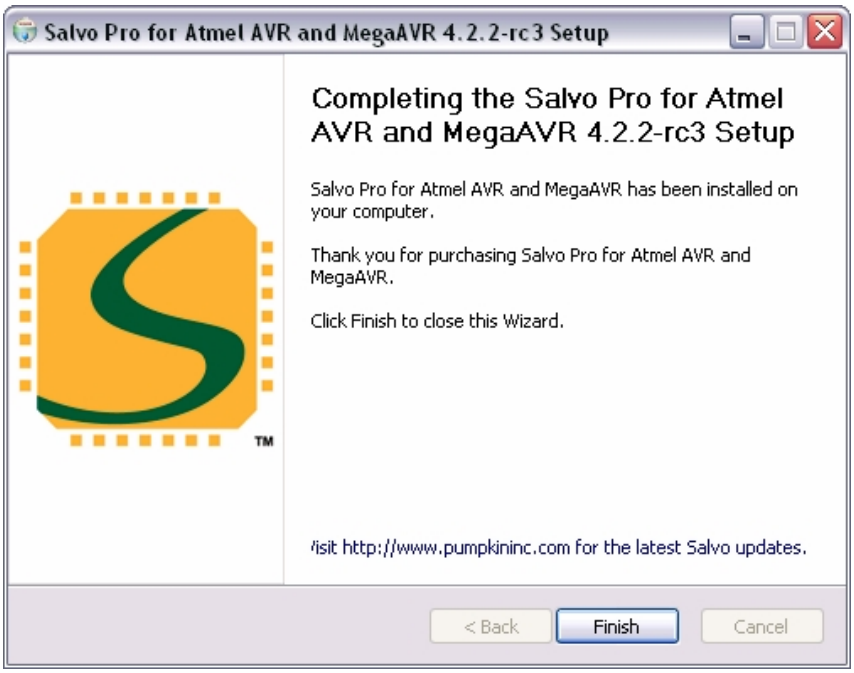

Рисунок 19: Экран окончания установки

Сетевая инсталляция

Если вы работаете над разделяемым кодом в сетевой среде (например, для управления версиями) и должны установить Salvo на разделяемом сетевом диске, запустите инсталлятор на Wintel PC и выберите каталог на сетевом диске как целевой каталог. Вы можете найти удобным создать ярлыки в папке Salvo Start Menu на каждой машине, которая обращается к Salvo по сети.

Замечание: Сетевые инсталляции должны выполнять условия *Лицензионного соглашения Salvo*. См. Лицензию для получения подробной информации.

Инсталляция Salvo на не-Wintel платформах

Если вы разрабатываете приложения Salvo на не-Wintel платформе, вы все таки будете нуждаться в доступе к Wintel машине для выполнения инсталлятора. Инсталлятор поместит все файлы Salvo с несколькими подкаталогами в выбранный целевой каталог (по умолчанию – `C:\Pumpkin\Salvo`). Вы можете затем скопировать весь подкаталог на другую машину через сеть или съемное запоминающее устройство (например, Zip, Jaz, лента, и т.д.).

Замечание: Лицензионное соглашение Salvo позволяет делать только одну копию каталогов Salvo на инсталляцию. Вы должны удалить весь каталог Salvo с Wintel машины после того, как вы переместили его в вашу не-Wintel среду разработки. См. Лицензию для подробной информации.

В качестве альтернативы, если вы работаете в сетевой среде с кросс-платформенным разделением файлов, вы можете выполнить инсталлятор на Wintel PC и выбрать каталог (удаленный) на вашей не-Wintel платформе как целевой каталог установки. Все файлы Salvo будут установлены в удаленный каталог. После выполнения установки, вы можете захотеть удалить элементы Start Menu с Wintel PC, если вы не будете их использовать.

Завершенная инсталляция

Директория Salvo после типовой установки должна выглядеть подобно следующей:

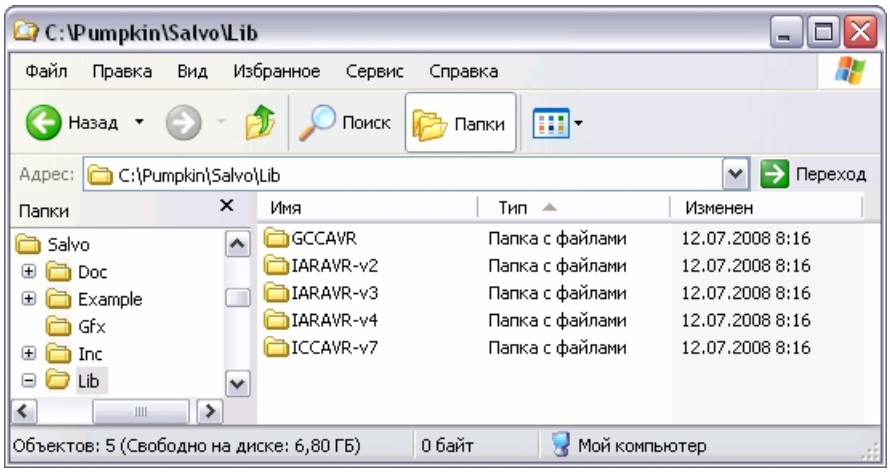

Рисунок 20: Содержание типичной директории инсталляции Salvo. (Выбрана подпапка Lib)

Деинсталляция Salvo

Программа установки автоматически создает деинсталлятор. Для его использования выберите соответствующий элемент Remove Salvo как показано ниже:

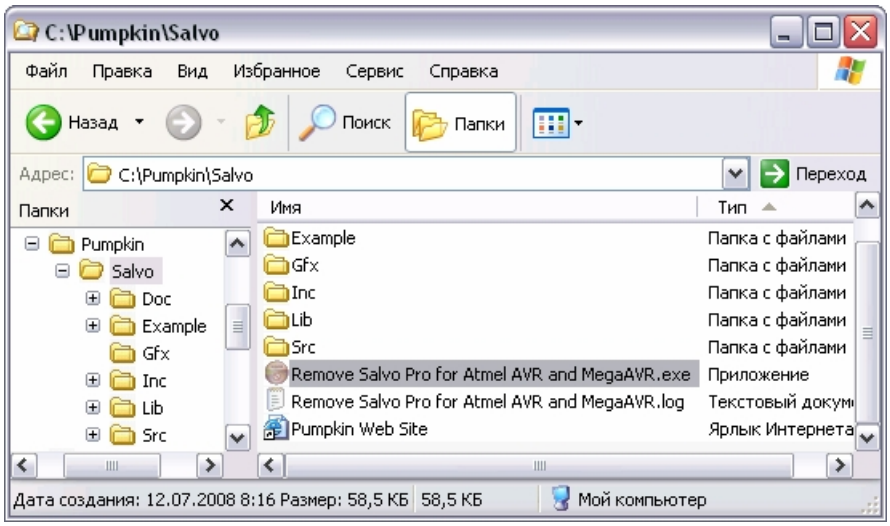

Рисунок 21: Расположение деинсталлятора

Когда появится приглашение деинсталлятора, нажмите кнопку **Да** для подтверждения удаления файлов:

Рисунок 22: Экран подтверждения удаления файлов

Щелкните **Да** для начала деинсталляции Salvo:

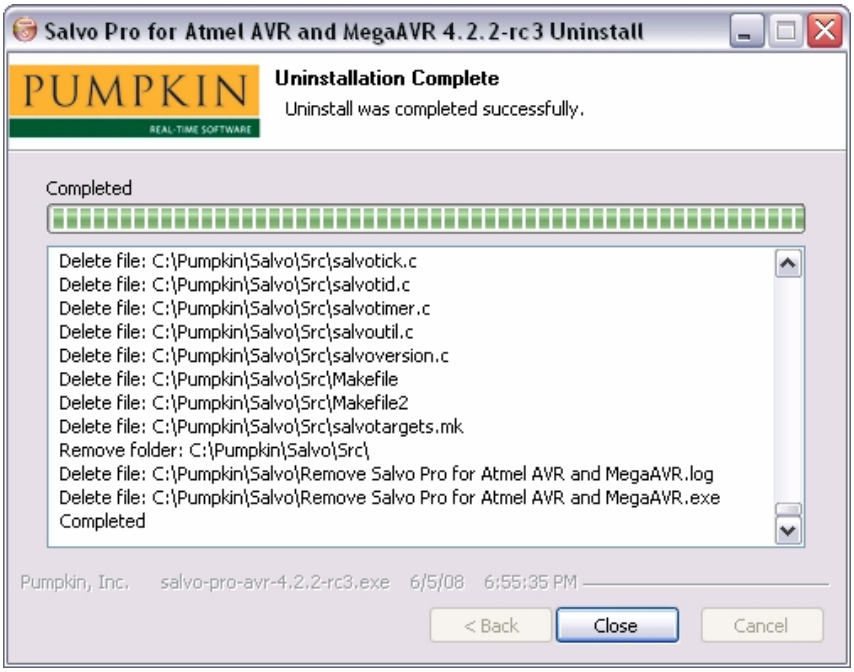

Рисунок 23: Экран завершения деинсталляции

После успешного удаления Salvo из среды разработки деинсталлятор выведет следующий экран:

Рисунок 24: Экран окончания деинсталляции

Нажмите кнопку **OK** для окончания деинсталляции Salvo.

Деинсталляция Salvo на не-Wintel машинах

Если вы используете Salvo на другой платформе (например, Linux), просто удалите целевую директорию Salvo и все ее поддиректории.

Инсталляция с несколькими дистрибутивами Salvo

Инсталлятор Salvo поддерживает несколько дистрибутивов различных типов, установленных в один каталог (обычно `C:\Pumpkin\Salvo`)[23]. Например, можно иметь Salvo Lite для TI's MSP430 а также Salvo Pro для семейства 8051, установленные вместе в `C:\Pumpkin\Salvo`.

Поведение инсталлятора

Инсталляторы Salvo заменяют разделяемые файлы всех дистрибутивов, только если устанавливаемые файлы более новые, чем существующие. При установке разделяемый файл помечается только для чтения. Разделяемые файлы включают независимый от целевого процессора файл заголовка Salvo и исходные файлы. Файлы, уникальные для дистрибутива (например, файлы проекта), всегда устанавливаются, то есть перезаписываются инсталлятором.

Инсталляция нескольких дистрибутивов Salvo

Обычно не требуется никаких дополнительных предосторожностей при установке дополнительных Salvo инсталляций на PC, содержащий один или большее количество существующих Salvo инсталляций. На основании поведения инсталлятора, только последние разделяемые файлы должны остаться на PC после завершения каждой инсталляции.

Деинсталляция с несколькими дистрибутивами Salvo

Так как деинсталлятор удалит разделяемые файлы, необходимо деинсталлировать на PC все инсталляции Salvo и затем переустановить желаемые.

Копирование файлов Salvo

Пользователям Salvo *категорически не рекомендуется* копировать любые из разделяемых файлов Salvo в места вне нормальных каталогов установки файлов. Наличие дубликатов файлов Salvo может привести к непредвиденному поведению и значительному усложнению отладки.

Пользователи с системами управления версиями, желающие добавить Salvo в их репозитории файлов, могут делать это, используя для добавления и извлечения единый источник (например, файловый сервер).

[23] Как в Salvo v3.2.2.

Модифицирование файлов Salvo

Модифицирование разделяемых файлов Salvo может также вести к непредсказуемому поведению и, следовательно, *категорически не рекомендуется*. Вообще говоря, только пользователи Salvo Pro должны изменять разделяемые файлы Salvo, и только тогда, когда проблемы с файлами и их решения были официально объявлены. Если доступны обновленные дистрибутивы Salvo, они должны автоматически заменить измененные файлы обновленными.

Глава 4 • Учебник

Введение

Эта глава представляет собой пошаговый учебник в двух частях, который поможет вам создавать приложения Salvo с самого начала. Первая часть служит введением в использование Salvo для написания многозадачной программы на языке Си. Во второй части мы скомпилируем ее в рабочее приложение.

Часть 1: Написание приложения Salvo

Создадим многозадачное приложение Salvo шаг за шагом, постепенно вводя различные понятия и свойства Salvo. Начнем с построения минимального приложения на Си. Мы объясним назначение и использование каждого нового свойства Salvo и подробно опишем, что происходит в приложении.

Совет: Каждый из нижеприведенных листингов Си является законченным приложением и сопровождается файлами проекта, исходного и исполнимого кода в каталоге `\Pumpkin\Salvo\Example\...\Tut\Tut5`. Вы можете найти их полезными для лучшего понимания их работы.

Пример 1: Инициализация Salvo и запуск многозадачности

Каждое рабочее приложение Salvo является комбинацией вызовов пользовательских сервисов Salvo и специфичного для приложения кода. Начнем использовать Salvo, создавая многозадачное приложение.

Минимальное приложение Salvo показано в Листинге 23.

```
#include "main.h"
#include <salvo.h>

int main( void )
{
  Init();

  OSInit();

  while (1) {
    OSSched();
  }
}
```

Листинг 23: Минимальное приложение Salvo

Эта элементарная программа вызывает два сервиса пользователя Salvo, прототипы функций которых объявлены в `salvo.h`. Один раз вызывается `OSInit()`, и из бесконечного цикла многократно вызывается `OSSched()`.

Совет: Приводимые учебные программы используют конструкцию языка Си `while (1) { }` для создания бесконечного цикла. Конструкции `for (;;) { }` и `do { } while (1)` являются функциональными эквивалентами и при создании бесконечного цикла взаимозаменяемы.

Совет: Все вызываемые пользователем функции Salvo имеют префикс "`OS`" или "`OS_`".

Замечание: Функция `Init()` в `main()` предназначена для инициализации[24] устройства. Она и файл заголовка `main.h` ничего не делают с кодом Salvo, но присутствуют для полноты структуры программы.

OSInit()

`OSInit()` инициализирует структуры данных, указатели и счетчики Salvo, и должна вызываться перед любыми другими вызовами функций Salvo. Отсутствие вызова `OSInit()` перед любыми другими вызовами Salvo приведет к непредсказуемому поведению программы.

OSSched()

`OSSched()` – это многозадачный диспетчер Salvo. Выполняться могут только задачи, находящиеся в состоянии готовности, и каждый вызов `OSSched()` приводит к выполнению наиболее готовой задачи до встречи следующего переключателя контекста в этой же задаче. Для продолжения многозадачности `OSSched()`вызывается повторно.

Совет: Для лучшего использования стека возврата процессора вы должны вызывать `OSSched()` непосредственно из `main()`.

Подробности

Так как никаких готовых задач в программе не имеется, диспетчер в Листинге 23 выполняет минимальную работу.

Пример 2: Создание, запуск и переключение задач

Многозадачность требует наличия готовых к выполнению задач, которые диспетчер может исполнить. Многозадачное приложение Salvo с двумя задачами показано в Листинге 24.

```
#include "main.h"
#include <salvo.h>

void TaskA( void )
{
  while (1) {
    OS_Yield();
  }
}

void TaskB( void )
{
  while (1) {
    OS_Yield();
  }
}
```

[24] Например, выбор осциллятора и настройка портов ввода-вывода микроконтроллера.

```
int main( void )
{
  Init();

  OSInit();

  OSCreateTask(TaskA, OSTCBP(1), 10);
  OSCreateTask(TaskB, OSTCBP(2), 10);

  while (1) {
    OSSched();
  }
}
```

Листинг 24: Многозадачное приложение Salvo с двумя задачами

Задачи `TaskA()` и `TaskB()` ничего не делают, но работают и переключают контекст снова и снова. Поскольку они обе имеют одинаковый приоритет (10), то они выполняются одна за другой непрерывно, отделяясь только переходом через диспетчер задач.

Для функционирования многозадачного режима, работающая задача должна возвращать управление диспетчеру. Это происходит через переключатель контекста (переключатель задач) внутри задачи. Поскольку Salvo разработана для работы без использования стека, она поддерживает переключение контекста только на уровне задачи.

Предупреждение: Переключение контекста Salvo на уровне ниже, чем из задачи (например, в пределах подпрограммы, вызванной из задачи), вызовет непредсказуемое поведение программы.

Для многозадачности Salvo, вы должны создать и запустить задачи. Задачи являются функциями, которые включают опциональную инициализацию, сопровождаемую бесконечным циклом, содержащим, по крайней мере, один переключатель контекста. Задачи Salvo не могут иметь никаких параметров. Когда задача создается через `OSCreateTask()`, вы назначаете для нее неиспользованный *блок управления задачи (`tcb`)*, и она переходит в остановленное состояние. Задача может быть создана в разных местах вашей программы. Задачи часто создаются до начала многозадачного режима, но они могут быть также созданы и впоследствии.

Чтобы задача могла работать, она должна быть в готовом состоянии. `OSStartTask()` может сделать остановленную задачу готовой. Однако, с целью уменьшения размера кода Salvo, `OSCreateTask()` *автоматически* запускает задачу, которую создает[25]. Поэтому вызов `OSStartTask()` не является необходимым. Как только задача сделана готовой, она будет запущена диспетчером, как только станет самой готовой, то есть готовой задачей с наивысшим приоритетом.

Замечание: Когда в группе готовых задач все имеют тот же самый приоритет, они выполняются циклически одна за другой.

Остановленная задача может быть запущена в разных местах вашей программы. Задачи могут быть запущены только после того, как они созданы. Задача может быть запущена после того, как начинается многозадачный режим.

[25] Опционально, задача может быть оставлена в остановленном состоянии с помощью `OSDONT_START_TASK`.

OS_Yield()

Каждая задача должна переключать контекст, по крайней мере, однажды. `OS_Yield()` – безусловный переключатель контекста Salvo. Обычное место нахождения `OS_Yield()` в конце, но в пределах бесконечного цикла задачи.

> **Замечание:** Все пользовательские сервисы Salvo с условным или безусловным переключением контекста имеют префикс `"OS_"`.

OSCreateTask()

Для создания задачи, вызовите `OSCreateTask()` со *стартовым адресом задачи*[26], *указателем tcb* и *приоритетом* как с параметрами. Стартовый адрес – это обычно начало задачи, определяемое именем задачи. Каждая задача нуждается в ее собственном, уникальном tcb. tcb содержит всю информацию, необходимую Salvo для управления задачей – такие как ее адрес начала/продолжения, состояние, приоритет, и т.д. Для использования доступны `OSTASKS` tcb, пронумерованные от 1 до `OSTASKS`. Макро `OSTCBP()` – краткий[27] способ определить указатель на специфический Salvo tcb, например `OSTCBP(2)` – указатель на второй tcb. Приоритет задачи может быть от 0 (самый высокий) и до 15 (самый низкий), и не обязан быть уникальным для задачи. Как только создана, задача находится в остановленном состоянии.

Поведение по умолчанию для `OSCreateTask()` состоит в том, чтобы также запустить задачу Salvo с указанного в tcb указателя, делая ее готовой к выполнению. Это может быть сделано прежде, чем задача фактически заработает, в зависимости от приоритета задачи, состояния задач с более высоким приоритетом, и момента когда диспетчер снова получит управление.

> **Совет:** Многие сервисы Salvo возвращают коды ошибок, которые вы можете использовать для обнаружения проблем в вашем приложении. См. *Главу 7 • Справочник* для получения дополнительной информации.

Подробности

Листинг 24 иллюстрирует некоторые из фундаментальных понятий RTOS: задачи, планирование задач, приоритеты задач и переключение контекста. Задачи – функции со специфической структурой, обычно использующие бесконечный цикл. Задача будет работать всякий раз, когда она – наиболее готовая задача, и диспетчер решает, какая задача наиболее готова на основании приоритета задачи. Так как Salvo – кооперативная RTOS, каждая задача должна возвращать управление обратно диспетчеру, иначе ни одна другая задача не будет иметь шанса работать. В этом примере, это достигается через `OS_Yield()`. В следующих примерах, мы будем использовать другие переключатели контекста вместо `OS_Yield()`.

[26] В Си это эквивалентно имени задачи (функции).
[27] `&OStcbArea[n – 1]` является более длинным путем.

Возможно, это пока не очевидно, но Листинг 24 также иллюстрирует другую основную концепцию RTOS – это состояния задачи. В Salvo все задачи начинаются как разрушенные. Создание задачи изменяет ее на остановленную, а старт задачи делает ее готовой к работе. Когда задача выполняется фактически, говорят, что она работает. В этом примере, после создания и старта, каждая задача меняет состояние между готовой и работающей много раз. И есть короткий период времени в течение повторения основного цикла `while()`, когда ни одна задача не работает, то есть они обе имеют состояние готовой. Это время работы диспетчера.

Планирование задач в Salvo следует двум очень простым правилам: Первое – любая задача с высшим приоритетом будет работать при следующем вызове диспетчера. Второе – все задачи с одним и тем же приоритетом будут работать циклически (round-robin) пока они – наиболее готовые задачи. Это означает, что они отработают одна за другой, пока не отработают все, и затем цикл повторится снова.

Пример 3: Добавление функциональности задачам

Листинг 25 показывает многозадачное приложение с двумя задачами, которые делают больше чем только переключение контекста. В этот раз мы будем использовать более описательные имена задач.

```
#include "main.h"
#include <salvo.h>

unsigned int counter;

void TaskCount( void )
{
  while (1) {
    counter++;

    OS_Yield();
  }
}

void TaskShow( void )
{
  InitPORT();

  while (1) {
    PORT = (PORT & ~0xFE) | ((counter >> 8) & 0xFE);

    OS_Yield(TaskShow1);
  }
}

int main( void )
{
  Init();

  OSInit();

  OSCreateTask(TaskCount, OSTCBP(1), 10);
  OSCreateTask(TaskShow,  OSTCBP(2), 10);

  counter = 0;
```

```
    while (1) {
      OSSched();
    }
  }
```

Листинг 25: Многозадачность с двумя нетривиальными задачами

Две задачи в Листинге 25 работают независимо и обе имеют доступ к разделяемой глобальной переменной – 16-битному счетчику. Счетчик инициализируется[28] прежде, чем стартует многозадачность. Первая задача инкрементирует счетчик каждый раз, как только получает возможность работать. Другая задача выбирает значение счетчика и выводит старшие 7 бит в 8-битный порт (PORT) с 8 подключенными к нему светодиодами. Это продолжается неопределенно долго.

Замечание: Так как Salvo – кооперативная RTOS, только одна задача может иметь доступ к глобальной переменной `counter` в каждый момент времени.

Подробности

В Листинге 25, фактически ни одна задача не заработает, пока не начнется многозадачный режим с вызовом диспетчера Salvo. Каждый раз, когда вызывается `OSSched()`, определяется, какая задача наиболее готова к выполнению, и выполнение программы передается этой задаче. Так как обе задачи имеют один и тот же приоритет, и одновременно готовы к выполнению, Salvo решает, какая задача заработает первой.

В этом примере `TaskCount()` запускается первой[29]. Вначале инкрементируется счетчик, затем контекст переключится через `OS_Yield()`. Эта макрокоманда отмечает, где находится выполнение программы в `TaskShow()` (в конце `while()` цикла), и затем возвратит выполнение программы диспетчеру. Тогда диспетчер проверит `TaskCount()`, чтобы увидеть готова ли она все еще продолжать выполнение. В данном случае, так как мы не делали никаких изменений состояния, она заработает снова, когда станет наиболее готовой задачей.

Диспетчер завершает свою работу и вызывается снова, потому что он находится в бесконечном `while()` цикле. В данном случае, так как задачи Salvo имеют равный приоритет, диспетчер решает, что `TaskShow()` – самая готовая задача, и делает ее рабочей. Сначала, `PORT` конфигурируется как выходной порт и инициализируется[30]. Затем `TaskShow()` входит в бесконечный цикл в первый раз, `PORT` инициализирован значением 0x00 (счетчик теперь равен 0x0001), и опять `OS_Yield()` возвращает выполнение программы диспетчеру после отметки, "куда возвращаться" в `TaskShow()`. `TaskShow()` также остается готовой продолжать работу.

[28] Строго говоря, эта инициализация необязательна, т.к. ANSI компилятор установит счетчик в 0 перед `main()`.
[29] Потому-что была запущена первой и обе задачи имеют равный приоритет.
[30] В этом примере, каждый вывод порта `PORT` может конфигурироваться как вход или как выход. При включении питания, все выводы конфигурированы как входы, следовательно необходимо сконфгурировать их как выходы через `InitPORT()`. `InitPORT()` также устанавливает 8-битовый порт ввода/вывода в 0x00.

После окончания своей работы, диспетчер теперь вызывается в третий раз. Опять `TaskCount()` – самая готовая задача, и таким образом она выполняется снова. Но в этот раз – это возобновление выполнения с того места, где мы оставили ее, то есть в конце `while()` цикла. Так как это – бесконечный цикл, выполнение продолжается с начала цикла. `TaskCount()` инкрементирует счетчик и передает контроль обратно диспетчеру.

В следующий раз, когда вызывается диспетчер, `TaskShow()` продолжается там, где задача была оставлена, переходит к началу `while()` цикла, записывает значение в `PORT` и выходит обратно в диспетчер. Это весь процесс возобновления задачи, где она была оставлена, выполняясь и возвращая контроль обратно диспетчеру, повторяясь бесконечно, с каждой задачей, выполняющейся поочередно с каждым вызовом диспетчера.

Когда выполняется программа из Листинга 25, создается видимость двух процессов, происходящих одновременно. Обе задачи самостоятельны, то есть чем быстрее микропроцессор, тем быстрее они выполняются. Счетчик оказывается инкрементированным и выведен в порт одновременно. Однако мы знаем, что задействованы две отдельные задачи, поэтому мы называем эту программу многозадачное приложение. Это пока не очень мощное приложение, и его функции могли бы быть продублированы многими другими способами. Но поскольку мы развиваем это приложение, мы увидим, что использование Salvo позволит нам управлять все более и более сложной системой с минимальными усилиями кодирования, и мы сможем также максимизировать ее эффективность.

Пример 4: Применение событий

Предыдущий пример не использовал одно из самых мощных инструментов RTOS – межзадачные связи. Это также бесполезно тратит вычислительную мощность, так как `TaskShow()` работает непрерывно, но `PORT` изменяется только один раз за каждые 512 вызовов `TaskCount()`. Используем межзадачную коммуникацию для повышения эффективности использования вычислительной мощности.

В листинге 26, показанном ниже, мы использовали несколько директив препроцессора `#define` для улучшения читабельности.

```
#include "main.h"
#include <salvo.h>

#define TASK_COUNT_P         OSTCBP(1) /* task #1 */
#define TASK_SHOW_P          OSTCBP(2) /* task #2 */
#define PRIO_COUNT           10 /* task priorities*/
#define PRIO_SHOW            10 /* "" */
#define BINSEM_UPDATE_PORT_P OSECBP(1) /* binsem 1
*/

unsigned int counter;

void TaskCount( void )
{
  while (1) {
    counter++;

    if ( !(counter & 0x01FF) ) {
```

```
      OSSignalBinSem(BINSEM_UPDATE_PORT_P);
    }

    OS_Yield();
  }
}

void TaskShow( void )
{
  InitPORT();

  while (1) {
    OS_WaitBinSem(BINSEM_UPDATE_PORT_P,
                  OSNO_TIMEOUT);

    PORT = (PORT & ~0xFE)|((counter >> 8) & 0xFE);
  }
}

int main( void )
{
  Init();

  OSInit();

  OSCreateTask(TaskCount, TASK_COUNT_P, PRIO_COUNT);
  OSCreateTask(TaskShow, TASK_SHOW_P, PRIO_SHOW);

  OSCreateBinSem(BINSEM_UPDATE_PORT_P, 0);

  counter = 0;

  while (1) {
    OSSched();
  }
}
```

Листинг 26: Многозадачность с использованием событий

В Листинге 26 мы общаемся между двумя задачами, чтобы обновлять порт только тогда, когда это требуется. Используем двоичный семафор для представления этого события. Мы инициализируем его значением 0, подразумевая, что событие еще не произошло. `TaskCount()` сигнализирует двоичным семафором всякий раз, когда старшие 7 бит счетчика изменяются. `TaskShow()` ждет события и затем копирует старшие 7 бит счетчика в `PORT`.

OSCreateBinSem()

`OSCreateBinSem()` создает двоичный семафор с указателем ecb и начальным значением. Двоичный семафор создан без задач, ждущих его. Двоичный семафор должен быть создан прежде, чем он может сигналить или ожидаться.

OSSignalBinSem()

Двоичный семафор передает сигнал через `OSSignalBinSem()`. Если ни одна из задач не ожидает двоичный семафор, то он просто инкрементируется. Если одна или более задач ждут двоичный семафор, то самая высокоприоритетная ждущая задача становится готовой после сигнала двоичного семафора.

OS_WaitBinSem()

Задача будет ждать двоичный семафор, пока он не передаст сигнал. Если двоичный семафор равен 0, когда его ждут задачи, то задачи переключаются в состояние ожидания и возвращаются через диспетчер. Это продлит ожидание двоичного семафора до тех пор, пока он не передаст сигнал, и задача с наивысшим приоритетом ожидает двоичный семафор. Поэтому специфическое событие может ожидать больше чем одна задача.

Если, с другой стороны, двоичный семафор равен 1, когда задача его ждет, то двоичный семафор сбрасывается в 0, и задача продолжает выполнение без переключения контекста.

Совет: Префикс "OS_" в `OS_WaitBinSem()` должен напоминать вам, что переключение контекста произойдет *безусловно* при каждом вызове `OS_WaitBinSem()` независимо от значения двоичного семафора. Если `binSem` установлен (равен 1) и задача является наиболее приоритетной готовой задачей, то выполнение продолжится в задаче. Если нет, выполение задачи продолжится позже при выполнении *обоих* условий.

Совет: Вы должны всегда определять таймаут[31] при ожидании двоичного семафора через `OS_WaitBinSem()`. Если вы хотите иметь задачу, неограниченно долго ожидающую двоичный семафор, который передаст сигнал, используйте предопределенную величину `OSNO_TIMEOUT`.

Замечание: В этом примере, `OS_WaitBinSem()` используется вместо `OS_Yield()`. Фактически, макро `OS_WaitBinSem()` включает вызов `OS_Yield()`. Вы не должны вызывать `OS_Yield()`, используя условный переключатель контекста такой, как `OS_WaitBinSem()` – он сделает это за вас.

Подробности

Для повышения эффективности нашего приложения, мы хотели бы обновлять `PORT` только тогда, когда старшие 7 бит счетчика изменяются. Чтобы добиться этого, мы используем механизм сигнализации между двумя задачами, называемый двоичным семафором. Здесь, двоичный семафор – это флаг, который инициализируется нулем, чтобы означать, что обновлять порт нет необходимости. Когда двоичный семафор сигналит, то есть устанавливается в 1, это означает, что требуется обновление `PORT`.

Коммуникация между задачами достигнута использованием двоичного семафора, чтобы передать ожидающей задаче (в данном случае `TaskShow()`) сигнал о том, что требуется обновление `PORT`. Это сделано в `TaskCount()` вызовом `OSSignalBinSem()` с параметром, являющимся указателем на двоичный семафор, при наличии ожидания `TaskShow()` двоичного семафора.

Замечание: `TaskCount()` не знает, какие задачи ожидают сигнал двоичного семафора, а `TaskShow()` не знает, как двоичный семафор передает сигнал.

[31] Параметр таймаута требуется независимо от того, действительно ли ваше приложение построено с кодом Salvo (исходные файлы или библиотеки), который поддерживает таймауты. Это позволяет перестроить приложение для таймаутов без каких-либо изменений исходного пользовательского кода.

В первый раз, когда `TaskShow()` входит в диспетчер, вызывается `OS_WaitBinSem()`. Так как двоичный семафор инициализирован нулем, `TaskShow()` отдает управление обратно диспетчеру и изменяет свое состояние с готовой на ожидающую. Теперь есть только одна готовая задача – `TaskCount()`, и диспетчер перезапускает ее повторно.

Когда TaskCount() наконец сигнализирует двоичным семафором, `TaskShow()` становится готовой к выполнению и будет выполнена однократно когда `TaskCount()` возвратится через диспетчера. Наконец, так как старшие 7 бит счетчика изменяются только каждые 512 вызовов `TaskCount()`, нет никакого смысла в управлении этим более часто. При использовании двоичного семафора, `TaskShow()` работает только тогда, когда необходимо обновить `PORT`. Остаток времени программа ждет и не потребляет *никакой* вычислительной мощности (циклов инструкций).

Эффективность этого приложения примерно вдвое выше (то есть счетчик инкрементируется с двойной скоростью) чем в приложении в Листинге 25. Так как ожидающая задача не отнимает времени процессора вообще, диспетчер управляет только готовыми задачами. Так как `TaskShow()` ждет двоичного семафора более чем 97% времени[32], она работает только в редких случаях, когда изменяется старший байт счетчика. Остальную часть времени диспетчер управляет `TaskCount()`.

Должно быть, очевидно, что здесь вызовы `OS_WaitBinSem()` и `OSSignalBinSem()`применяют довольно высокую функциональность. В этом примере, сервисы событий Salvo контролируют, когда `TaskShow()` будет работать используя двоичный семафор для коммуникаций между задачами. Здесь двоичный семафор – простой флаг (1 бит информации). Salvo поддерживает использование двоичных и счетных семафоров, так же как и других механизмов, передающих больше информации (например количество, или указатель) от одной задачи другой.

Листинг 26 содержит полную программу Salvo – ничто не пропущено. Нет ничего "работающего на заднем плане" (в фоновом режиме), не надо проверять, должна ли ожидающая задача быть сделана готовой и т.д. Другими словами, нет никакого продолжительного опроса – все действия Salvo управляются событиями, которые вносят вклад в его высокую эффективность. `TaskShow()` переходит от ожидания к готовности в вызове `OSSignalBinSem()`, и от работы к ожиданию через `OS_WaitBinSem()`. С Salvo вы имеете полный контроль над тем, что процессор делает в любой момент, и таким образом вы можете оптимизировать эффективность вашей программы без нежелательных помех от RTOS.

Пример 5: Задержка задач

Один элемент, отсутствующий в предыдущем примере – понятие работы в реальном времени, работает только "открытая петля". Если мы добавим в приложение дополнительные задачи равного или более высокого приоритета, скорость инкрементирования счетчика снизится. Давайте посмотрим на то, как RTOS может обеспечить работу в реальном времени, добавляя задачу, которая работает с частотой 2Hz, независимо от того, что делает остальная часть системы. Мы сделаем это, периодически задерживая задачу.

[32] Измерено на Test System A.

Возможность задерживать задачу на указанный период времени может быть очень полезным свойством. Задача останется в задержанном состоянии не готовой продолжать, пока не истечет определенное время задержки. Ядро контролирует задержки и возвращает задержанную задачу в готовое состояние.

Приложение в Листинге 27 мигает светодиодом на младшем бите PORT с частотой 1Hz, создав и запустив задачу, которая задерживает себя на 500ms после переключения бита порта, делая это повторно. Программа находится в `\Pumpkin\Salvo\Example\...\Tut\Tut5` каждой дистрибуции Salvo.

```
#include "main.h"
#include <salvo.h>

#define TASK_COUNT_P         OSTCBP(1) /* task #1 */
#define TASK_SHOW_P          OSTCBP(2) /* '''' #2 */
#define TASK_BLINK_P         OSTCBP(3) /* '''' #3 */
#define PRIO_COUNT           10 /* task priorities*/
#define PRIO_SHOW            10 /* '''' */
#define PRIO_BLINK           2  /* '''' */
#define BINSEM_UPDATE_PORT_P OSECBP(1) /* binSem #1
*/

unsigned int counter;

void TaskCount( void )
{
  while (1) {
    counter++;

    if ( !(counter & 0x01FF) ) {
      OSSignalBinSem(BINSEM_UPDATE_PORT_P);
    }

    OS_Yield();
  }
}

void TaskShow( void )
{
  while (1) {
    OS_WaitBinSem(BINSEM_UPDATE_PORT_P,
                  OSNO_TIMEOUT);

    PORT = (PORT & ~0xFE)|((counter >> 8) & 0xFE);
  }
}

void TaskBlink( void )
{
  InitPORT();

  while (1) {
    PORT ^= 0x01;

    OS_Delay(50);
  }
}
```

```
void main( void )
{
  Init();

  OSInit();

  OSCreateTask(TaskCount, TASK_COUNT_P, PRIO_COUNT);
  OSCreateTask(TaskShow,  TASK_SHOW_P,  PRIO_SHOW);
  OSCreateTask(TaskBlink, TASK_BLINK_P, PRIO_BLINK);

  OSCreateBinSem(BINSEM_UPDATE_PORT_P, 0);

  counter = 0;

  enable_interrupts();

  while (1) {
    OSSched();
  }
}
```

Листинг 27: Многозадачность с задержкой

Для этого дополнительно потребуются прерывания для вызова `OSTimer()` с желаемой частотой системных часов 100Hz. The code to do this is located in the source file `tut5_sr.c` that accompanies the project. An example for the PIC16 is shown below[33].

```
#include <salvo.h>

#define TMR0_RELOAD 156 /* for 100Hz ints @ 4MHz */

void interrupt IntVector( void )
{
  if ( T0IE && T0IF ) {
    T0IF = 0;
    TMR0 -= TMR0_RELOAD;

    OSTimer();
  }
}
```

Листинг 28: Вызов OSTimer() с частотой системных часов

Чтобы использовать задержки в приложении Salvo, вы должны добавить в него системный таймер. В данном примере мы добавили 10ms системный таймер, периодически вызывая `OSTimer()` с частотой приблизительно 100Hz. Частота получается периодическим переполнением таймера, вызывая прерывание. Прерывания нужно разрешить для возможности вызова `OSTimer()` – следовательно, вызов `enable_interrupts();` должен предшествовать старту многозадачности. Так как задержки определены в единицах системных часов, задача мигания задерживается на 50*10ms, или на 500ms.

[33] `IntVector()` также используется в примере 6 далее. `IntVector()` (и следовательно содержание `tut5_isr.c`) специфично для контроллера и компилятора.

OSTimer()

Чтобы использовать сервис задержек Salvo, вы должны вызывать `OSTimer()` с постоянной частотой. Это обычно делается периодическим прерыванием. Частота с которой ваше приложение вызывает `OSTimer()` определяет разрешение задержек. Если периодические прерывания происходят каждые 10ms, вызывая `OSTimer()` из ISR (Процедуры обработки прерывания), вы будете иметь период системных часов 10ms, или частоту 100Hz. С данной частотой, вы можете определять задержки с разрешением в один период системных часов. Например, возможны задержки 10ms, 20ms...1s, 2s...

Замечание: Свойства таймера Salvo гибко конфигурируемы с задержками до 32-битных системных часов и с опциональным предделителем. Подробнее см. *Главу 5 • Конфигурация* и *Главу 6 • Часто задаваемые вопросы (FAQ)*.

OS_Delay()

Имея `OSTimer()` и периодически вызывая его с системной частотой, вы можете теперь задерживать задачу, заменяя `OSYield()` вызовом `OSDelay()`, который вызовет переключатель контекста и задержит задачу на определенное число периодов системных часов. Задача автоматически станет готовой, как только истечет указанная задержка.

Подробности

В Листинге 27, каждый раз `TaskBlink()` отрабатывает, задерживает себя на 500ms и входит в задержанное состояние, возвращаясь в диспетчер. Когда через 500ms истекает задержка `TaskBlink()`, она автоматически снова становится готовой, и заработает после переключения контекста текущей (работающей) задачи. Поэтому `TaskBlink()` имеет более высокий приоритет чем любая из задач `TaskCount()` или `TaskShow()`. Делая `TaskBlink()` задачей самого высокого приоритета в нашем приложении, мы гарантируем минимум задержки (времени ожидания) между истечением таймера задержки и моментом, когда `TaskBlink()` переключит бит 0 в `PORT`. Поэтому `TaskBlink()` будет работать каждые 500ms с минимальным временем ожидания, независимо от того, что делают другие задачи.

Замечание: Если `TaskBlink()` имела бы тот же самый приоритет, что `TaskCount()` и `TaskShow()`, то она иногда оставалась бы готовой (но не работающей), так как обе `TaskCount()` и `TaskShow()` отрабатывали бы перед ней. Это увеличило бы ее максимальное время ожидания. Если бы `TaskBlink()` имела более низкий приоритет, она не работала бы вообще.

Инициализация `PORT` перемещена в `TaskBlink()` из-за приоритета. Это будет первая работающая задача, и поэтому `PORT` будет инициализирован как выходной перед первым запуском `TaskShow()`.

Salvo обрабатывает задержанные задачи один раз за вызов `OSTimer()`, и накладные расходы обработки не зависят от числа задержанных задач [34].

[34] За исключением случая, когда одна или более задержек задач истекают одновременно

Это иллюстрирует то, что системный таймер полезен по многим причинам. Один ресурс процессора (например, периодическое прерывание) может использоваться с `OSTimer()`, чтобы задерживать неограниченное число задач. Более важно, что задержанные задачи потребляют только очень маленькое количество вычислительной мощности процессора – много меньше чем работающие задачи.

Передача сигналов из нескольких задач

Многозадачность в программировании приносит реальные выгоды, когда приоритеты оптимально определены и функции программ ясно очерчены.

Рассмотрим код в Листинге 29, чтобы увидеть, что случается, когда мы понижаем приоритет всегда работающей задачи `TaskCount()`, и имеем `TaskShow()`, обрабатывающую все записи в `PORT`. Эта программа находится в `\Pumpkin\Salvo\Tut\Tu6\main.c`.

```
#include "main.h"
#include <salvo.h>

#define TASK_COUNT_P       OSTCBP(1)/* task #1 */
#define TASK_SHOW_P        OSTCBP(2)/* "" #2 */
#define TASK_BLINK_P       OSTCBP(3)/* "" #3 */
#define PRIO_COUNT         12 /*task priorities*/
#define PRIO_SHOW          10 /* "" */
#define PRIO_BLINK         2 /* "" */
#define MSG_UPDATE_PORT_P  OSECBP(1) /* sem #1 */

unsigned int counter;

char CODE_B = 'B';
char CODE_C = 'C';

void TaskCount( void )
{
  counter = 0;

  while (1) {
    counter++;

    if ( !(counter & 0x01FF) ) {
      OSSignalMsg(MSG_UPDATE_PORT_P,
                  (OStypeMsgP) &CODE_C);
    }

    OS_Yield();
  }
}

void TaskShow( void )
{
  OStypeMsgP msgP;

  InitPORT();

  while (1) {
    OS_WaitMsg(MSG_UPDATE_PORT_P,
               &msgP,
               OSNO_TIMEOUT);
```

```
    if ( *(char *)msgP == CODE_C ) {
      PORT = (PORT & ~0xFE)|((counter >> 8)&0xFE);
    }
    else {
      PORT ^= 0x01;
    }
  }
}

void TaskBlink( void )
{
  OStypeErr err;

  while (1) {
    OS_Delay(50);

    err = OSSignalMsg(MSG_UPDATE_PORT_P,
                      (OStypeMsgP) &CODE_B);

    if ( err == OSERR_EVENT_FULL ) {
      OS_SetPrio(PRIO_SHOW + 1);
      OSSignalMsg(MSG_UPDATE_PORT_P,
                  (OStypeMsgP) &CODE_B);
      OSSetPrio(PRIO_BLINK);
    }
  }
}

void main( void )
{
  Init();

  OSInit();

  OSCreateTask(TaskCount, TASK_COUNT_P, PRIO_COUNT);
  OSCreateTask(TaskShow, TASK_SHOW_P, PRIO_SHOW);
  OSCreateTask(TaskBlink, TASK_BLINK, PRIO_BLINK);

  OSCreateMsg(MSG_UPDATE_PORT_P, (OStypeMsgP) 0);

  Enable_interrupts();

  while (1) {
    OSSched();
  }
}
```

Листинг 29: Передача сигналов из нескольких задач

В Листинге 29 мы сделали два изменения относительно предыдущей программы. Первое – `TaskShow()` теперь делает все записи в `PORT`. Обе задачи `TaskCount()` и `TaskBlink()` посылают уникальное сообщение в `TaskShow()` (символ `'C'` для "счета" или `'B'` для "мигания", соответственно), которое она интерпретирует, чтобы показать счетчик в порте или переключить младший бит порта. Второе – мы понизили приоритет `TaskCount()`, создавая ее с низким приоритетом.

OSCreateMsg()

`OSCreateMsg()` используется для инициализации сообщения. Salvo имеет определенный тип для сообщений, и требует, чтобы вы инициализировали сообщение должным образом. Сообщение создано без ожидающих его задач. Сообщение должно быть создано прежде, чем оно может быть передано или будет ожидаться.

Замечание: Сервисы Salvo требуют, чтобы вы сопрягали их с вашим кодом, используя предопределенные типы, например – `OStypeMsgP` для указателей сообщения. Вы должны использовать предопределенные типы Salvo везде, где можно. См. *Главу 7 • Справочник* для получения дополнительной информации о предопределенных типах Salvo.

OSSignalMsg()

Чтобы сигнализировать о сообщении `OSSignalMsg()`, вы должны определить и ecb указатель и указатель на содержание сообщения. Если ни одна из задач не ждет сообщение, то сообщение получает указатель, и если сообщение еще не определено, то в этом случае происходит ошибка. Если одна или более задач ждут сообщение, то самая высокоприоритетная ждущая задача становится готовой. Вы должны корректно типизировать указатель сообщения так, чтобы он мог быть должным образом разыменован задачей, ждущей сообщение.

OS_WaitMsg()

Задача ждет сообщение через `OS_WaitMsg()`. Сообщение возвращается задаче через указатель сообщения. Чтобы извлечь содержание сообщения, вы должны привести указатель к указателю на определенный тип, соответствующий тому, на что указывает указатель сообщения.

OS_SetPrio()

Задача может изменить свой приоритет и переключить контекст немедленно после использования `OS_SetPrio()`.

OSSetPrio()

Задача может изменить свой приоритет, используя `OSSetPrio()`. Новый приоритет вступит в силу, как только задача вызовет диспетчер.

Подробности

`TaskShow()` – теперь единственная задача, записывающая в `PORT`. Единственное сообщение – все, что требуется для передачи уникальной информации от двух различных задач (работающих полностью различно) в `TaskShow()`. В данном случае, сообщение – указатель на 1-байтную константу. Так как сообщения содержат указатели, требуется соответствующее их типу разыменование для посылки и получения в сообщении соответствующей информации.

В Листинге 29, возможен следующий сценарий. Сразу после того, как `TaskCount()` сигнализирует о сообщении, задержка `TaskBlink()` истекает, и `TaskBlink()` становится готовой. Так как `TaskBlink()` имеет высший приоритет, а сообщение все еще присутствует, когда `TaskBlink()` сигнализирует сообщением, `OSSignalMsg()` возвратит ошибку. Светодиод на выводе `PORT` будет не сможет переключиться...

Этот пример иллюстрирует использование *возвращаемого значения* сервисом Salvo. Проверяя на вышеупомянутое ошибочное условие, мы можем гарантировать надлежащие результаты, временно понижая приоритет `TaskBlink()` и уступая управление диспетчеру перед передачей сигнала сообщения снова. `TaskShow()` временно будет задачей самого высокого приоритета, и она будет "требовать" сообщения. Пока `TaskCount()` не сигнализирует сообщением быстрее чем один раз за каждые три переключения контекста, это решение остается разумным[35].

В более сложном приложении, например электроника автомобиля, можно представить замену `TaskShow()` другой задачей, которая использует приборную панель, разделенную на различные области. Четыре задачи контролировали бы информацию (например, обороты в минуту, скорость, давление масла и температуру воды) и передавали ее, сигнализируя сообщением всякий раз, когда параметр изменился. `TaskShow()` ожидала бы этого сообщения. Каждое сообщение указывало бы, где показать параметр, какой цвет использовать (например, красный при превышении температуры) и новое значение параметра. Так как визуальные приборы вообще имеют низкую скорость обновления информации, `TaskShow()` могла бы работать с более низким приоритетом, чем задачи посылки. Эти задачи работали бы с более высоким приоритетом, чтобы обработать информацию, которую они собирают без несвоевременного вмешательства в медленную задачу отображения. Например, задача контроля давления масла могла бы работать с самым высоким приоритетом, так как потеря давления масла означает определенную неисправность двигателя. При наличии возможностей индикации в задаче вместо подлежащей вызову функции, вы можете настроить производительность вашей программы, назначая соответствующий приоритет каждой из используемых задач.

Понижая приоритет `TaskCount()` мы изменили поведение нашего приложения. Обновления `PORT` теперь имеют приоритет перед инкрементированием счетчика. Это означает, что обновления `PORT` произойдут быстрее после того, как сообщение принято. Счетчик теперь инкрементируется только тогда, когда больше нет никакой иной работы. Вы можете кардинально и очевидно изменить поведение вашей программы, изменяя только приоритет при создании задачи.

Заключение

Как пользователь Salvo вы не должны волноваться о планировании, состояниях задач, управлении событиями или межзадачными коммуникациями. Salvo сделает это для вас автоматически и эффективно. Вы должны только создать и использовать задачи и события надлежащим образом, чтобы получить все эти и другие функциональные возможности.

Замечание: *Глава 7 • Справочник* содержит рабочие примеры с комментированным исходным текстом Си для каждого сервиса Salvo. Обратитесь к ним за дополнительной информацией об использовании задач и событий.

[35] Альтернативное решение этой проблемы состояло бы в том, чтобы использовать очередь сообщений с местом для двух сообщений.

Материал для размышлений

Теперь, когда вы пишете код с задачами и структурами на основе событий, которые обеспечивает Salvo, вы можете найти полезным или даже необходимым изменить способ создания новых программ. Вместо того чтобы волноваться о том, сколько ресурсов процессора, ISR, глобальных переменных и циклов команд потребует приложение, сфокусируйтесь на конкретных задачах, их приоритетах и целях, временных требованиях вашего приложения, событиях, управляющих его поведением. Объедините все это с должным образом установленными приоритетами задач, которые используют события, чтобы управлять их выполнением и общаться в вашей программе.

Часть 2: Создание приложения Salvo

Замечание: Если вы еще не сделали этого, пожалуйста, следуйте инструкциям в *Главе 3 • Инсталляция*, чтобы установить все компоненты Salvo на ваш компьютер. Вы можете также найти полезным справиться с *Главой 5 • Конфигурация* и *Главой 7 • Справочник* для получения дополнительной информации о некоторых из тем, упоминаемых ниже. Наконец, вы должны изучить *Salvo Application Note*, которые описывают построение приложений с вашим компилятором. Ищите в *Salvo Compiler Reference Manual* особенности вашего компилятора.

Теперь, когда вы знакомы с тем, как написать приложение Salvo, пришло время получить выполнимую программу. Ниже вы найдете общие инструкции по построению приложения Salvo.

Рабочая среда

Salvo распространяется как пакет файлов исходного кода, объектных, библиотечных и других файлов поддержки. Так как Salvo Pro обеспечивает полный исходный код, Salvo может быть скомпилирован на разных платформах. Необходимо иметь опыт работы с вашим редактором / компилятором / интегрированной средой разработки (IDE), чтобы скомпилировать приложение Salvo.

Вы должны быть знакомы с концепциями включения файла в другой файл, компиляции файлов, компоновки файлов, работы с библиотеками, создания выполнимой программы, отладки продукта вашего компилятора, и помещения вашей программы в память.

Обратитесь к документации вашего редактора / компилятора / IDE за тем, как включить файлы в исходный текст, компилировать исходный код, компоновать объектные модули, и связывать их с библиотеками.

Многие IDE поддерживают автоматическое создание утилиты *make*. Вы, вероятно, найдете это очень полезным, работая с Salvo. Если вы не имеете утилиты make, вы можете захотеть узнать о ее получении. Коммерческие, бесплатные и условно бесплатные утилиты *make* существуют для командной строки DOS и Windows 95 / 98 / 2000 / NT.

Создание директории проекта

При создании приложения Salvo вы будете включать исходные файлы Salvo в ваш исходный код, и вероятно также скомпонуете их с объектными или библиотечными файлами Salvo. Мы настоятельно рекомендуем, чтобы вы не изменяли никакие файлы Salvo непосредственно[36], и вы не должны излишне дублировать никакие файлы Salvo. Если вы не намереваетесь делать изменения в исходных кодах Salvo, вы не должны изменять ни одного файла Salvo.

Создавая рабочую директорию для каждого нового приложения Salvo, которое вы пишете, вы сможете:

- минимизировать занимаемый объем жесткого диска,
- эффективнее управлять вашими файлами,
- изменять одно приложение без воздействия на другие,
- компилировать уникальные версии библиотек Salvo для разных проектов.

Замечание: Полные проекты для некоторых учебных программ могут быть найдены в `\Pumpkin\Salvo\Tut`.

Включение salvo.h

Главный заголовочный файл Salvo – `salvo.h` должен быть включен в каждый из ваших исходных файлов, использующих Salvo. Вы можете сделать это, вставляя

```
#include <salvo.h>
```

в каждый из ваших исходных файлов, который вызывает сервис Salvo. Вы можете также нуждаться в конфигурировании ваших инструментов разработки чтобы добавить основной каталог Salvo (обычно `C:\Pumpkin\Salvo`) к *пути включаемых файлов* системы вашего инструментария - см. *Установка путей поиска*, ниже.

Замечание: Использование

```
#include "salvo.h"
```

не рекомендуется.

Совет: Если вы включаете заголовочный файл проекта (например, `myproject.h`) во все ваши исходные файлы, вы можете включить `salvo.h` в него.

Включение `salvo.h` автоматически включит ваш специфический файл версии проекта `salvocfg.h` (см. *Установка опций конфигурации*, ниже). Вы не должны включать `salvocfg.h` в какой-либо из ваших исходных файлов, включения только в `salvo.h` вполне достаточно.

Совет: `salvo.h` имеет встроенную "защиту включения", которая предотвратит проблемы ссылок многократного включения `salvo.h`, содержащихся в одном исходном файле.

[36] Исходные файлы Salvo инсталлируются как файлы только-для-чтения.

Конфигурирование вашего компилятора

Для успешной компиляции вашего приложения Salvo, вы должны сконфигурировать ваш компилятор для использования с исходными файлами и библиотеками Salvo. Вы имеете несколько доступных вам опций для объединения вашего кода с исходным текстом Salvo, чтобы построить приложение.

Установка путей поиска

Сначала, вы должны определить соответствующие пути поиска так, чтобы компилятор мог найти необходимые включаемые (*.h) и исходные (*.c) файлы Salvo.

> **Совет:** Все поддерживаемые Salvo компиляторы поддерживают явные пути поиска. Поэтому вы *никогда* не должны копировать файлы Salvo из их исходных каталогов в каталог вашего проекта, используя компилятор, находящий их на основании того факта, что они находятся в текущем каталоге.

По меньшей мере, ваш компилятор должен знать, где найти следующие файлы:

- `salvo.h` расположенный в `\Pumpkin\Salvo\Inc`
- `salvocfg.h` расположенный в каталоге вашего текущего проекта

Вы можете также нуждаться в определении каталога исходных файлов Salvo (`\Pumpkin\Salvo\Src`), если вы планируете включить исходные файлы Salvo в ваши собственные исходные файлы (см. ниже).

Библиотеки против исходных файлов

Различные методы включения Salvo в ваше приложение описаны ниже. Компоновка с библиотеками Salvo – самый простой метод, но имеет ограничения. Включение исходных файлов Salvo в ваш проект – самый гибкий метод, но не такой простой, и требует Salvo Pro. Создание пользовательских библиотек Salvo из исходных файлов может иметь смысл для опытных пользователей Salvo Pro.

> **Совет:** Вы можете найти *Рисунок 25: Построение приложения с библиотеками Salvo* и *Рисунок 26: Построение приложения с исходными кодами Salvo* полезными для понимания процесса построения приложения Salvo.

Использование библиотек

Точно так же как библиотечные функции компилятора Си, например `rand()` в стандартной библиотеке (`stdlib.h`) или `printf()`в стандартной библиотеке ввода / вывода (`stdio.h`), Salvo имеет функции (называемые *пользовательскими сервисами*) содержащиеся в библиотеках. В отличие от функций библиотеки компилятора, пользовательские сервисы Salvo гибко конфигурируемы, то есть их поведением можно управлять на основании функциональных возможностей, которые вы желаете получить в вашем приложении. Каждая библиотека Salvo содержит пользовательские функции, скомпилированные для специфического набора *опций конфигурации*. Существует много различных библиотек Salvo.

Замечание: Опции конфигурации – инструменты *времени компиляции*, используемые для конфигурирования исходного кода Salvo и генерации библиотек. Поэтому функциональные возможности прекомпилированной библиотеки *не могут быть изменены* через опции конфигурации. Чтобы изменить функциональные возможности библиотеки, они должны быть перекомпилированы заново с Salvo Pro и новыми опциями конфигурации.

Чтобы облегчить начало работы, все дистрибуции Salvo содержат библиотеки с уже включенным большинством функциональных возможностей Salvo. Как новичок, вы должны начать строить ваши приложения с использованием библиотек. Таким образом вы не должны беспокоиться о несметном числе опций конфигурации.

Совет: Самый легкий и самый быстрый способ создать рабочее приложение состоит в том, чтобы связать ваш исходный текст с соответствующей библиотекой Salvo. Специфические для компилятора *Salvo Application Notes* в деталях описывают как создать приложение для каждого компилятора.

Полные проекты на основе библиотек для всех учебных программ могут быть найдены в `\Pumpkin\Salvo\Tut`. Дополнительную информацию см. в *Приложении C • Описания файлов и программ*.

Использование исходных файлов

Salvo конфигурируется прежде всего, чтобы минимизировать размер пользовательских сервисов и таким образом уменьшить объем памяти ROM. Кроме того, его конфигурируемость помогает в уменьшении используемой RAM. Без этого, пользовательские сервисы Salvo и переменные могли бы быть слишком большими для применения во многих приложениях. Все это составляет его преимущества и недостатки. С одной стороны, вы можете точно настроить Salvo, чтобы использовать только необходимый объем ROM и RAM в вашем приложении. С другой стороны, это может быть препятствием изучению работы всех других вариантов конфигурации.

Есть некоторые случаи, когда лучше создавать ваше приложение, добавляя исходные файлы Salvo как узлы к вашему проекту. Когда вы используете этот метод, вы можете изменить опции конфигурации и перестроить приложение так, чтобы иметь изменения, которые дают эффект только в исходном коде Salvo. Только Salvo Pro включает исходный код. Оставшаяся часть этой главы охватывает этот подход.

Установка опций конфигурации

Система Salvo гибко конфигурируема. Вы будете должны создать и использовать файл конфигурации `salvocfg.h` для каждого нового приложения, которое вы пишете. Этот простой текстовый файл используется, чтобы выбрать опции конфигурации Salvo времени компиляции, которые воздействуют на то, сколько задач и событий ваше приложение может использовать. Все опции конфигурации имеют значения по умолчанию – большинство из них может быть приемлемым для вашего приложения.

Замечание: Всякий раз, когда вы переопределяете опцию конфигурации в `salvocfg.h`, вы *должны* перекомпилировать все исходные файлы Salvo в вашем приложении.

Примеры ниже предполагают, что вы создаете и редактируете `salvocfg.h` текстовым редактором. Каждая опция конфигурации устанавливается директивой `#define` языка Си. Например, чтобы сконфигурировать Salvo для поддержки 16-битных задержек, вы добавили бы

```
#define OSBYTES_OF_DELAYS     2
```

в файл `salvocfg.h` вашего проекта. Без этой специальной строки, эта опция конфигурации автоматически устанавливается по умолчанию (в данном случае, 8-битовые задержки).

Замечание: Имя и значение опции конфигурации чувствительны к регистру символов. Если вы введете имя неправильно, то нужная опция будет заменена значением по умолчанию Salvo.

Идентификация компилятора и целевого процессора

Обычно, Salvo автоматически определяет, какой компилятор и целевой процессор вы используете. Это делается обнаружением предопределенных символов, обеспечиваемых компилятором.

Определение числа задач

Память для внутренних структур задач Salvo распределяется в момент компиляции. Вы должны определить в `salvocfg.h`, сколько задач вы хотели бы поддерживать в вашем приложении, например:

```
#define OSTASKS     4
```

Вы не обязаны использовать все задачи, для которых вы резервируете память, и при этом вы не должны использовать их соответствующие указатели tcb (нумеруемые последовательно от `OSTCBP(1)` до `OSTCBP(OSTASKS)`). Если вы попытаетесь сослаться на задачу, для которой не была выделена память, то пользовательский сервис Salvo возвратит код предупреждения.

Совет: В Salvo на задачи ссылаются по их tcb указателям. Рекомендуется, чтобы вы использовали описательные обозначения в вашем коде, чтобы обращаться к вашим задачам. Это наиболее легко сделать при использовании директивы `#define` в главном заголовочном файле (*.h) вашего проекта. Вашу программу будет легче понять, называя сервисы задач Salvo значащими именами как, например:

```
#define TASK_CHECK_TEMP_P[37]   OSTCBP(1)
#define TASK_MEAS_SPEED_P     OSTCBP(2)
#define TASK_DISP_RPM_P       OSTCBP(3)
```

Определение числа событий

Память для внутренних структур событий Salvo также выделяется при компиляции. Вы должны определить в `salvocfg.h`, сколько событий вы хотели бы поддерживать в вашем приложении, например:

```
#define OSEVENTS     3
```

События включают семафоры (бинарные и счетные), сообщения и очереди сообщений.

[37] Суффикс **P** должен напоминать вам, что объект – указатель на что-либо.

Вы не обязаны использовать все события, для которых вы выделяете память, и при этом вы не должны использовать их соответствующие указатели ecb (нумеруемые последовательно от `OSECBP(1)` до `OSECBP(OSEVENTS)`). Если вы попытаетесь сослаться на событие, для которого не была выделена память, то пользовательский сервис Salvo возвратит код предупреждения.

Если ваше приложение не использует события, оставьте `OSEVENTS` неопределенным в вашем `salvocfg.h` или установите его в 0.

Совет: Вы также должны использовать описательные имена для событий. См. совет выше о том, как это сделать.

Определение других опций конфигурации

Вам может также потребоваться определить другие опции конфигурации, в зависимости от того, какие из свойств Salvo вы планируете использовать в вашем приложении. Многие из свойств Salvo не доступны, пока они не разрешены через опцию конфигурации. Это сделано, чтобы минимизировать размер кода, который Salvo добавляет к вашему приложению. Для маленьких проектов, маленький `salvocfg.h` может быть адекватным. Для больших проектов и более сложных приложений, вы будете должны выбрать соответствующие опции конфигурации для всех свойств, которые вы желаете использовать. Другие опции конфигурации включают:

- размер задержек, счетчиков и т.п. в байтах,
- размер семафоров и указателей сообщений,
- директивы памяти, специфические для компилятора.

Совет: Если вы пытаетесь использовать свойство Salvo, вызывая функцию Salvo, и ваш компилятор выдает сообщение об ошибке, говорящее, что он не может найти функцию, это может быть потому, что функция не разрешена опцией конфигурации.

В сложном приложении, могли бы быть некоторые из дополнительных опций конфигурации:

```
#define OSBYTES_OF_DELAYS     3
#define OSTIMER_PRESCALER     20
#define OSLOC_ECB             bank3
```

Значения опций будут числовыми константами, или предопределенными константами (например, `TRUE` и `FALSE`), или определениями, предусмотренными для использования компилятора (например, `bank3`, используемое компилятором HI_TECH PICC, чтобы определить местонахождение переменных в специфическом банке памяти).

Пример salvocfg.h – приложение Salvo Tut5

Поскольку учебная программа относительно проста, только несколько опций конфигурации должны быть определены в `salvocfg.h`. Начиная с пустого `salvocfg.h`, мы начинаем все конфигурации с их значений по умолчанию.

Для трех задач и одного события, мы будем нуждаться в следующих директивах `#define`.

```
#define OSTASKS      3
#define OSEVENTS     1
```

Далее `\Pumpkin\Salvo\Tut\Tut5` использует двоичные семафоры, как средство межзадачных коммуникаций. Код двоичных семафоров запрещен по умолчанию, и мы разрешаем его следующим образом:

```
#define OSENABLE_BINARY_SEMAPHORES     TRUE
```

Наконец, так как мы используем задержки, мы должны определить размер возможных задержек.

```
#define OSBYTES_OF_DELAYS     1
```

Эта опция конфигурации должна быть определена, потому что по умолчанию Salvo не поддерживает задержки, что сводит требования RAM к минимуму. Так как `TaskBlink()` задерживает себя на 50 тактов системных часов, один байт – это все, что требуется. С байтом для задержек, каждая задача может задержать себя на 255 тактов системных часов единственным вызовом `OS_Delay()`.

Замечание: Директивы `#define` в `salvocfg.h` могут появляться в любом порядке.

Этот четырехстрочный `salvocfg.h` типичен для маленькой и средней по размеру программы умеренной сложности. Полный файл конфигурации Salvo для этой программы может быть найден в `\Pumpkin\Salvo\Tut\Tut5`. С удаленными комментариями Си[38] он показан в Листинге 30.

```
#define OSBYTES_OF_DELAY              1
#define OSENABLE_BINARY_SEMAPHORES    TRUE
#define OSEVENTS                      1
#define OSTASKS                       3
```

Листинг 30: Файл salvocfg.h для учебной программы

Компоновка с объектными файлами Salvo

Вы можете создать приложение, компилируя и затем компонуя ваше приложение с некоторыми или всеми исходными файлами `*.c` Salvo. Этот метод рекомендуется для большинства приложений, и совместим с утилитами *make*. Это является относительно прямым, но имеет неудобство, что ваш финальный выполнимый файл может содержать все функциональные возможности Salvo, содержавшиеся в компонуемых файлах, независимо от того, использует ли ваше приложение их или нет.

Замечание: Некоторые компиляторы способны к "разумной компоновке", посредством чего функции, которые связаны, но не используются, не попадают в выполнимый файл. В этой ситуации нет никакой проблемы с компоновкой вашего приложения со всеми исходными файлами Salvo.

Глава 7 • Справочник содержит описания всех сервисов пользователя Salvo и исходных файлов Salvo, которые содержат их. Как только вы используете сервис в вашем коде, вы будете также должны скомпоновать его с соответствующим исходным файлом. Это обычно делается в IDE компилятора, добавляя исходные файлы Salvo в ваш проект. Если вы будете использовать сервис, не добавляя файла, то вы получите ошибку компоновки при построении вашего проекта.

[38] И без дополнительных опций конфигурации, которые соответствуют таковым в связанной свободно распространяемой библиотеке.

Размер каждого компилированного объектного модуля сильно зависит от опций конфигурации, которые вы выбираете. Кроме того, вы можете разумно выбирать, какие модули компилировать и компоновать, например, если не планируется использовать динамические приоритеты задач в вашем приложении, вы можете изменить `salvocfg.h` соответственно и опустить `prio.c` для сокращения размера кода.

Совет: Специфические для компиляторов *Salvo Application Notes* описывают подробно, как создать приложения для каждого компилятора.

Полные, базирующиеся на исходном коде проекты для всех учебных программ могут быть найдены в `\Pumpkin\Salvo\Tut`. См. *Приложение C • Описания файлов и программ* для получения дополнительной информации.