



SALVO RTOS

РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

ВЕРСИЯ 3.2.2

ДЛЯ ВСЕХ ДИСТРИБУЦИЙ

SALVO RTOS USER'S MANUAL v.3.2.2

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



Знакомство

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 `salvo\demo`. 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 обычно доступные аппаратные средства (см. `salvo\demo\d4`), вы можете создать ваше собственное приложение, модифицируя и компилируя исходный код примера.

См. *Приложение С • 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.

Если вы не имеете сертифицированного компилятора, то может быть доступна его свободно распространяемая версия. См. `salvo\free\links`.

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, для 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

Авторские права © 1995-2003 Pumpkin, Inc.

Все права защищены международными соглашениями. Ни одна часть данного документа не может быть воспроизведена, сохранена в поисковых системах или передана в любой форме электронными, механическими, записывающими, фотокопировальными и другими средствами без предварительного разрешения компании Pumpkin.

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*.

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: 17:08, Четверг, 9 августа 2003 г.
Страниц: 577
Слов: 105089

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

Содержание.....	i
Рисунки	xv
Листинги	xvii
Таблицы.....	xix
Замечания о выпуске.....	xxi
Введение	xxi
Что нового	xxi
Замечания о выпуске.....	xxi
Версии инструментов третьих сторон.....	xxi
Процессоры и компиляторы	xxiii
Предисловие.....	xxv
Типографские соглашения.....	xxv
Стандартизированная схема нумерации	xxvi
Стиль кодирования Salvo	xxvii
Господство конфигурируемости.....	xxvii
Берегите драгоценные ресурсы	xxvii
Учитесь любить препроцессор	xxvii
Документировать, но не дублировать	xxvii
Мы не совершенны.....	xxvii
Глава 1 • Введение	1
Приглашение.....	1
Что такое Salvo?	2
Почему я должен использовать Salvo?	2
К какому типу относится RTOS Salvo?.....	3
Как выглядит программа Salvo?	3
Какие ресурсы требуются Salvo?	5
Чем отличается Salvo?.....	5
Что мне нужно для использования Salvo?	6
Какие процессоры и компиляторы поддерживает Salvo?.....	7
Как распространяется Salvo?	7
Что содержится в этом руководстве?	7
Глава 2 • Основы RTOS	9
Введение	9
Основные термины.....	10
Системы переднего и заднего плана.....	12
Реентерабельность	13
Ресурсы	13
Многозадачность и переключение контекста.....	14
Задачи и прерывания.....	14
Приоритетное и кооперативное планирование	15
Приоритетное планирование.....	15
Кооперативное планирование	16

Дополнительно о многозадачности.....	17
Структура задачи	17
Простая многозадачность	18
Приоритетная многозадачность	18
Состояния задачи	18
Задержки и таймер	20
Управляемая событиями многозадачность.....	22
События и межзадачные коммуникации.....	23
Семафоры	24
Флаги событий	24
Синхронизация задач.....	26
Ресурсы	27
Сообщения	29
Очереди сообщений	31
Сводка о взаимодействии задач и событий	31
Конфликты	31
Взаимоблокировка	31
Инверсия приоритетов	32
Эффективность RTOS.....	32
Реальный пример	33
Стандартный подход с суперциклом	33
Управляемый событиями подход RTOS.....	34
Шаг за шагом	35
Инициализация операционной системы.....	36
Структурирование задач	36
Приоритизация задач	37
Интерфейс с событиями	37
Добавление системного таймера.....	37
Запуск задач.....	37
Разрешение многозадачности.....	38
Объединение всего вместе	38
Различия в RTOS.....	40
Глава 3 • Инсталляция	43
Введение.....	43
Запуск инсталлятора	43
Сетевая инсталляция	49
Инсталляция Salvo на не-Wintel платформах	49
Завершенная инсталляция	50
Деинсталляция Salvo.....	50
Деинсталляция Salvo на не-Wintel машинах	51
Инсталляция с несколькими дистрибутивами Salvo.....	51
Поведение инсталлятора.....	51
Инсталляция нескольких дистрибутивов Salvo.....	51
Деинсталляция с несколькими дистрибутивами Salvo.....	52
Копирование файлов Salvo.....	52
Модифицирование файлов Salvo.....	52
Глава 4 • Учебник	53
Введение.....	53
Часть 1: Написание приложения Salvo	53
Инициализация Salvo и запуск многозадачности	53
Создание, запуск и переключение задач.....	54
Добавление функциональности задачам	57
Использование событий для повышения эффективности.....	59
Задержка задач	63
Передача сигналов из нескольких задач.....	66

Заключение	70
Материал для размышлений	70
Часть 2: Создание приложения Salvo	71
Рабочая среда	71
Создание директории проекта	71
Включение salvo.h	72
Конфигурирование вашего компилятора	72
Установка путей поиска	72
Библиотеки против исходных файлов	73
Использование библиотек	73
Использование исходных файлов	74
Установка опций конфигурации	74
Компоновка с объектными файлами Salvo	77
Глава 5 • Конфигурация	79
Введение	79
Процесс построения Salvo	79
Построение с библиотеками	79
Построение с исходными кодами	82
Преимущества различных типов построения	84
Обзор опций конфигурации	84
Опции конфигурации всех дистрибуций	85
OSCOMPILER: идентификация компилятора	85
OSEVENTS: установка максимального числа событий	85
OSEVENT_FLAGS: установка максимального числа флагов событий	86
OSLIBRARY_CONFIG: определение конфигурации прекомпилированной библиотеки	87
OSLIBRARY_GLOBALS: определение типа памяти для глобальных объектов Salvo в прекомпилированной библиотеке	87
OSLIBRARY_OPTION: определение опции прекомпилированной библиотеки	88
OSLIBRARY_TYPE: определение типа прекомпилированной библиотеки	88
OSLIBRARY_VARIANT: определение варианта прекомпилированной библиотеки	89
OSMESSAGE_QUEUES: установка максимального числа очередей сообщений	90
OSTARGET: идентификация целевого процессора	90
OSTASKS: установка максимального числа задач	91
OSUSE_LIBRARY: использование прекомпилированной библиотеки	91
Опции конфигурации дистрибуций с исходными кодами	92
OSBIG_SEMAPHORES: использование 16-битных семафоров	92
OSBYTES_OF_COUNTS: установка размера счетчиков	92
OSBYTES_OF_DELAYS: установка длины задержек	93
OSBYTES_OF_EVENT_FLAGS: установка размера флагов событий	93
OSBYTES_OF_TICKS: установка максимального счета импульсов системных часов	94
OSCALL_OSCREATEEVENT: управление прерываниями при создании событий	94
OSCALL_OSGETPRIOTASK: управление прерываниями при получении приоритета задачи	96
OSCALL_OSGETSTATETASK: управление прерываниями при получении состояния задачи	96
OSCALL_OSMSGQCOUNT: управление прерываниями при получении числа сообщений в очереди сообщений	96
OSCALL_OSMSGQEMPTY: управление прерываниями при проверке очереди сообщений на отсутствие сообщений	97
OSCALL_OSRETURNEVENT: управление прерываниями при чтении или проверке событий	97
OSCALL_OSSIGNALEVENT: управление прерываниями при передаче сигналов событий и манипуляция флагами событий	97
OSCALL_OSSTARTTASK: управление прерываниями при запуске задач	97
OSCLEAR_GLOBALS: явная очистка всех глобальных параметров	97
OSCLEAR_UNUSED_POINTERS: сброс неиспользуемых указателей Tcb и Ecb	98
OSCLEAR_WATCHDOG_TIMER(): определение инструкций сброса сторожевого таймера	98
OSCOLLECT_LOST_TICKS: конфигурирование системного таймера на максимальную гибкость	99
OSCOMBINE_EVENT_SERVICES: комбинирование кода обслуживания общего события	100

OSCTXSW_METHOD: идентификация методологии переключения контекста	100
OSCUSTOM_LIBRARY_CONFIG: выбор файла конфигурации библиотеки пользователя.....	101
OSDISABLE_ERROR_CHECKING: запрет проверки ошибок времени выполнения.....	101
OSDISABLE_FAST_SCHEDULING: конфигурирование циклического планирования	102
OSDISABLE_TASK_PRIORITIES: назначение всем задачам одинакового приоритета	103
OSENABLE_BINARY_SEMAPHORES: разрешение поддержки двоичных семафоров.....	103
OSENABLE_BOUNDS_CHECKING: разрешение проверки границ указателей времени выполнения	103
OSENABLE_CYCLIC_TIMERS: разрешение циклических таймеров.....	104
OSENABLE_EVENT_FLAGS: разрешение поддержки флагов событий	104
OSENABLE_EVENT_READING: разрешение поддержки чтения событий	105
OSENABLE_EVENT_TRYING: разрешение поддержки проверки событий.....	106
OSENABLE_FAST_SIGNALING: разрешение передачи сигналов быстрых событий.....	106
OSENABLE_IDLE_COUNTER: отслеживание холостого хода диспетчера	107
OSENABLE_IDLING_HOOK: вызов функции пользователя при холостом ходе.....	107
OSENABLE_INTERRUPT_HOOKS: вызов функции пользователя при управлении прерываниями	108
OSENABLE_MESSAGES: разрешение поддержки сообщений	108
OSENABLE_MESSAGE_QUEUES: разрешение поддержки очередей сообщений	109
OSENABLE_OSSCHED_DISPATCH_HOOK: вызов функции пользователя внутри диспетчера	109
OSENABLE_OSSCHED_ENTRY_HOOK: вызов функции пользователя внутри диспетчера...	110
OSENABLE_OSSCHED_RETURN_HOOK: вызов функции пользователя внутри диспетчера	110
OSENABLE_SEMAPHORES: разрешение поддержки семафоров.....	111
OSENABLE_STACK_CHECKING: контроль глубины Call-Return стека	111
OSENABLE_TCBEXT0 1 2 3 4 5: разрешение расширений Tcb.....	112
OSENABLE_TIMEOUTS: разрешение поддержки таймаутов	114
OSGATHER_STATISTICS: сбор статистики времени выполнения	114
OSINTERRUPT_LEVEL: определение уровня прерывания для сервисов вызываемых прерыванием	115
OSLOC_ALL: тип хранения всех объектов Salvo	115
OSLOC_COUNT: тип хранения счетчиков	116
OSLOC_CTCB: тип хранения указателя блока управления текущей задачей.....	116
OSLOC_DEPTH: тип хранения счетчиков глубины стека.....	117
OSLOC_ECB: тип хранения указателей блоков управления событиями и очередями.....	117
OSLOC_EFCB: тип хранения блоков управления флагами событий	117
OSLOC_ERR: тип хранения счетчиков ошибок.....	117
OSLOC_GLSTAT: тип хранения глобальных битов состояния	117
OSLOC_LOGMSG: тип хранения строки сообщения регистрации	118
OSLOC_LOST_TICK: тип хранения потерянных импульсов времени	118
OSLOC_MQCB: тип хранения блоков управления очередями сообщений	118
OSLOC_MSGQ: тип хранения очередей сообщений.....	118
OSLOC_PS: тип хранения предделителя таймера.....	118
OSLOC_TCB: тип хранения блоков управления задачами.....	119
OSLOC_SIGQ: тип хранения указателей очередей просигналивших событий	119
OSLOC_TICK: тип хранения счетчика импульсов системного времени	119
OSLOGGING: регистрация ошибок и предупреждений времени выполнения.....	119
OSLOG_MESSAGES: конфигурирование сообщений регистрации времени выполнения.....	120
OS_MESSAGE_TYPE: конфигурирование указателей сообщений.....	120
OSMPLAB_C18_LOC_ALL_NEAR: размещение всех объектов Salvo в банке быстрого доступа (только MPLAB-C18)	121
OSOPTIMIZE_FOR_SPEED: оптимизация размера кода или скорости.....	121
OSPIC18_INTERRUPT_MASK: конфигурирование режима прерываний PIC18	122
OSPRESERVE_INTERRUPT_MASK: управление поведением разрешения прерывания	123
OSRPT_HIDE_INVALID_POINTERS: OSRpt() не будет отображать недопустимые указатели	124
OSRPT_SHOW_ONLY_ACTIVE: OSRpt() отображает только текущую задачу и данные события	124
OSRPT_SHOW_TOTAL_DELAY: OSRpt() показывает суммарную задержку очереди задержек	124

OSRTNADDR_OFFSET: смещение (в байтах) для адреса возврата сохраненного переключением контекста.....	125
OSSCHED_RETURN_LABEL(): определение метки в OSSched()	125
OSSET_LIMITS: ограничение числа объектов времени выполнения Salvo	126
OSSPEEDUP_QUEUEING: ускорение операций с очередями	127
OSTIMER_PRESCALAR: конфигурирование предделителя для OSTimer()	127
OSTYPE_TCBEXT0 1 2 3 4 5: установка типа расширения Tcb.....	128
OSUSE_CHAR_SIZED_BITFIELDS: упаковка битовых полей в Chars	128
OSUSE_EVENT_TYPES: проверка типов событий во время выполнения.....	129
OSUSE_INLINE_OSSCHED: уменьшение глубины Call-Return стека задачи	129
OSUSE_INLINE_OSTIMER: устранение использования OSTimer() Call-Return стека.....	130
OSUSE_INSELIG_MACRO: уменьшение глубины вызовов Salvo	131
OSUSE_MEMSET: использование memset() (если доступно)	132
Другие символы	132
MAKE_WITH_FREE_LIB, MAKE_WITH_SE_LIB, MAKE_WITH_SOURCE, MAKE_WITH_STD_LIB, MAKE_WITH_TINY_LIB: использование salvocfg.h для нескольких проектов	132
SYSA B ... Z AA ...: идентификация тестовой системы Salvo.....	133
USE_INTERRUPTS: разрешение кода прерываний	134
Организация	135
Выбор правильных опций для вашего приложения	137
Предопределенные константы конфигурации	138
Устаревшие параметры конфигурации	139
Удалено из версии 3.2.2.....	139
Глава 6 • Частые вопросы (FAQ)	141
Общее	141
Что такое Salvo?	141
Имеется ли Salvo с открытым кодом, свободной, общедоступной версией?	141
Насколько компактна Salvo?	141
Почему я должен использовать Salvo?	141
Что я должен знать о Salvo Pro в сравнении с Salvo LE?	142
Что я могу делать с Salvo?	142
К какому типу RTOS относится Salvo?.....	143
Каковы минимальные требования Salvo?	143
На каких процессорах могут выполняться приложения Salvo?.....	143
Мой компилятор не применяет стек. Он размещает переменные, используя статическую оверлейную модель. Может ли он использоваться с Salvo?	143
Сколько задач и событий поддерживает Salvo?.....	143
Сколько уровней приоритета поддерживает Salvo?	144
Какие типы событий поддерживает Salvo?	144
Является ли Salvo Y2K совместимым?	144
Откуда происходит Salvo?	144
Начало работы.....	144
Где я могу найти примеры проектов, использующих Salvo?	144
Какие компиляторы вы рекомендуете для использования с Salvo?.....	144
Имеется ли учебник?.....	144
Какие имеются другие доступные документы кроме Руководства Пользователя Salvo?	145
Я имею ограниченный бюджет. Могу ли я использовать Salvo?.....	145
Я имею только ассемблер. Могу ли я использовать Salvo?	145
Эффективность	145
Как можно повысить эффективность моего приложения, используя Salvo?	145
Как работают задержки под Salvo?	146
Каковы преимущества наличия приоритетов задач?.....	146
Когда код Salvo фактически выполняется в моем приложении?	146
Как я могу выполнять быстрые, критические по времени операции под Salvo?	146
Память	147
Насколько Salvo увеличит затраты памяти ROM и RAM в моем приложении?.....	147
Насколько больше памяти RAM займет приложение при использовании библиотек?	147

Должен ли я волноваться об исчерпании памяти?	147
Если я определяю задачу или событие, но никогда их не использую, отнимет ли это память RAM?	148
Как много использует Salvo из call-return стека?	148
Почему я должен использовать указатели при работе с задачами? Почему я не могу использовать явно ID задач?	148
Как я могу избежать переинициализации переменных Salvo при пробуждении из режима sleep в PIC12C509 PICmicro MCU?	149
Библиотеки	150
Какие библиотеки включает Salvo?	150
Что находится в каждой библиотеке Salvo?	150
Почему так много библиотек?	150
Должен ли я использовать библиотеки или исходный текст при создании моего приложения?	150
В чем разница между свободными и стандартными библиотеками Salvo?	150
Мое приложение, основанное на библиотеках, использует больше памяти RAM, чем я могу позволить. Почему?	150
Я использую библиотеку. Почему мое приложение использует большее количество памяти RAM, чем компилируемое непосредственно из исходных файлов?	151
Я использую свободную библиотеку и получаю сообщение "#error: OSXYZ exceeds library limit - aborting". Почему?	151
Почему я не могу изменить функциональность библиотеки, добавляя опции конфигурации в мой файл salvocfg.h?	151
Библиотеки очень большие – намного больше, чем объем памяти ROM моего процессора. Повлияет ли это на мое приложение?	151
Я использую библиотеку. Могу ли я изменить банк, в котором размещаются переменные Salvo?	151
Конфигурация	151
Я подавлен количеством опций конфигурации. С чего я должен начать?	151
Должен ли я использовать все функциональные возможности Salvo?	152
Какие файлы я включаю в мой main.c?	152
Каково назначение OSENABLE_SEMAPHORES и подобных опций конфигурации?	152
Могу ли я собрать статистику времени выполнения с Salvo?	152
Как я могу очистить сторожевой таймер моего процессора с Salvo?	153
Я разрешил таймауты, и объемы занимаемой памяти RAM и ROM значительно возросли. Почему?	153
Таймер и синхронизация	153
Должен ли я устанавливать таймер?	153
Как мне установить таймер?	153
Я добавил таймер в мою ISR и теперь моя ISR большая и медленная. Что я должен сделать?	153
Как мне указать системную частоту в Salvo?	154
Как мне использовать предделитель таймера?	154
Я разрешил предделитель и установил его в 1, но не получил никакой разницы. Почему? ..	154
Какова точность системного таймера?	154
Каково время ожидания прерывания Salvo?	154
Как я могу определить задержки больше чем обеспечиваются 8 битами счетчика?	154
Как я могу достичь очень больших задержек в Salvo? Могу ли я сделать это, сохраняя минимальной память, занимаемую задачей?	155
Могу ли я определить таймаут при ожидании события?	155
Имеет ли Salvo функции для получения истекшего времени?	156
Как мне выбрать правильное значение для OSBYTES_OF_TICKS?	156
Мой процессор не имеет прерываний. Могу ли я использовать сервисы таймера Salvo?	156
Переключение контекста	157
Как мне узнать, когда я переключаю контекст в Salvo?	157
Почему я не могу переключать контекст из какого-либо другого места кроме уровня задачи?	157
Почему Salvo используют макросы для переключения контекста?	157
Могу ли я переключать контекст больше чем в одном месте задачи?	157

Когда я должен использовать метки переключения контекста?	158
Задачи и события	158
Что такое taskID?	158
Имеет ли значение, какой taskID я назначаю задаче?	158
Имеется ли в Salvo холостая задача?	158
Как я могу контролировать задачи в моем приложении?	159
Что происходит в диспетчере?	159
Что можно сказать о реентерабельном коде и Salvo?	159
Что такое "неявные" и "явные" функции задач OS?	159
Как мне установить бесконечный цикл в задаче?	159
Почему задачи должны использовать статические локальные переменные?	160
Занимает ли использование статических локальных переменных больше памяти, чем с другими RTOS?	160
Могут ли задачи иметь один и тот же приоритет?	160
Могу ли я иметь несколько экземпляров одной задачи?	161
Имеет ли значение порядок, в котором я запускаю задачи?	161
Как я могу уменьшить размер кода при старте задач?	161
Каково различие между задержанной и ждущей задачами?	162
Могу ли я создать задачу, немедленно ожидающую события?	162
Я запустил задачу, но она ни разу не выполнялась. Почему?	162
Что случится, если я забыл зациклить мою задачу?	162
Почему мои низкоприоритетные задачи начинают выполняться перед завершением запуска моей высокоприоритетной задачи?	163
Когда я передаю сигнал ждущей задаче, это занимает намного больше времени, чем переключение контекста для выполнения. Почему?	163
Могу ли я уничтожить задачу и создать новую в этом же месте?	163
Могут ли ожидать события больше чем одна задача?	163
Сохраняет ли Salvo порядок, в котором происходят события?	163
Может ли задача ожидать больше чем одно событие одновременно?	164
Как я могу реализовать флаги событий?	164
Что происходит при таймауте ожидания задачей события?	165
Почему моя высокоприоритетная задача увязла в ожидании, тогда как другие низкоприоритетные задачи выполняются?	165
Когда происходит событие и есть ожидающие его задачи, какие задачи станут готовыми? ..	165
Как я могу сообщить, что задача обнаруживает таймаут, ожидая событие?	166
Могу ли я создать событие изнутри задачи?	166
Какую информацию я могу передать задаче в сообщении?	167
Мое приложение использует сообщения и двоичные семафоры. Имеется ли какой-нибудь способ уменьшить код Salvo?	167
Почему требования к объему памяти RAM значительно увеличились, когда я разрешил очереди сообщений?	167
Могу ли я передавать сигнал о событии извне задачи?	167
Когда я передаю сигнал о сообщении, которое ожидает более чем одна задача, только одна задача станет готовой. Почему?	167
Я использую событие сообщения для передачи символьной переменной ждущей задаче, но я не получаю правильные данные, когда разыменовываю указатель. Что происходит?	168
Что происходит, когда не имеется задач в очереди готовых?	168
В каком порядке сообщения покидают очередь сообщений?	169
Что происходит, если событие передает сигнал прежде, чем какая-либо задача начинает ожидать его? Станет ли событие потерянным или будет обработано после того, как задача начнет ожидать его?	169
Что происходит, если событие передает сигнал несколько раз прежде, чем ждущая задача получает возможность выполниться и обработать это событие? Будет ли обработан только один последний сигнал а предыдущие потеряны? Или первый будет обработан а следующие сигналы потеряны?	169
Что более важно создать сначала, событие или задачу, которая его ждет? Важен ли порядок создания?	169
Если я больше не нуждаюсь в одном событии и хочу использовать промежуток для другого события. Могу ли я уничтожить событие?	169

Могу ли я использовать сообщения или очереди сообщений, чтобы передавать необработанные данные между задачами?	170
Как я могу проверить, имеется ли место в памяти для дополнительных сообщений в очереди сообщений, без передачи сигналов очереди сообщений?	170
Прерывания	170
Почему Salvo отключает все прерывания в течение критической секции кода?	170
Мне важно время ожидания прерывания. Могу ли я модифицировать Salvo, чтобы запрещать только некоторые прерывания в течение критических секций кода?	171
Насколько большие функции Salvo я могу вызывать из прерывания?	171
Почему моя процедура обработки прерывания вырастает и становится медленнее, когда я добавляю вызов OSTimer()?	171
Мое приложение не может позволить расходы для передачи сигналов из ISR. Как я могу обойти эту проблему?	172
Компиляция проектов	172
Какой уровень предупреждений я должен использовать при компиляции проектов Salvo? ..	172
Какой уровень оптимизации я должен использовать при компиляции проектов Salvo?	172
Разное	172
Может ли Salvo выполняться на 12-разрядном PICmicro с 2-х уровневый call-return стеком? ..	172
Изменит ли Salvo мой взгляд на программирование встроенных систем?	172
Глава 7 • Справочник	173
Архитектура времени выполнения	173
Правило #1: каждой задаче необходим переключатель контекста	173
Правило #2: переключатели контекста могут быть только в задачах	174
Правило #3: долговременные локальные переменные должны быть объявлены статическими	174
Сервисы пользователя	176
OS_Delay(): задержка текущей задачи и переключение контекста	177
OS_DelayTS(): задержка текущей задачи относительно временной отметки и переключение контекста	178
OS_Destroy(): уничтожение текущей задачи и переключение контекста	180
OS_Replace(): замена текущей задачи и переключение контекста	181
OS_SetPrio(): изменение приоритета текущей задачи и переключение контекста	183
OS_Stop(): остановка текущей задачи и переключение контекста	184
OS_WaitBinSem(): переключение контекста и ожидание текущей задачей двоичного семафора	185
OS_WaitEFlag(): переключение контекста и ожидание текущей задачей флага события	186
OS_WaitMsg(): переключение контекста и ожидание текущей задачей сообщения	189
OS_WaitMsgQ(): переключение контекста и ожидание текущей задачей в очереди сообщений	190
OS_WaitSem(): переключение контекста и ожидание текущей задачей семафора	191
OS_Yield(): переключение контекста	193
OSClrEFlag(): очистка битов флага события	193
OSCreateBinSem(): создание двоичного семафора	194
OSCreateCycTmr(): создание циклического таймера	195
OSCreateEFlag(): создание флага события	196
OSCreateMsg(): создание сообщения	197
OSCreateMsgQ(): создание очереди сообщений	199
OSCreateSem(): создание семафора	201
OSCreateTask(): создание и старт задачи	201
OSDestroyCycTmr(): уничтожение циклического таймера	203
OSDestroyTask(): уничтожение задачи	203
OSGetPrio(): получение приоритета текущей задачи	204
OSGetPrioTask(): получение приоритета указанной задачи	205
OSGetState(): получение состояния текущей задачи	205
OSGetStateTask(): получение состояния указанной задачи	206
OSGetTicks(): получение значения системного таймера	207
OSGetTS(): получение временной отметки текущей задачи	208

OSInit(): подготовка многозадачности.....	208
OSMsgQCount(): получение числа сообщений в очереди сообщений.....	209
OSMsgQEmpty(): проверка доступного места в очереди сообщений.....	210
OSReadBinSem(): безусловное получение двоичного семафора.....	211
OSReadEFlag(): безусловное получение флага события.....	212
OSReadMsg(): безусловное получение указателя сообщений.....	213
OSReadMsgQ(): безусловное получение указателя сообщений из очереди сообщений.....	213
OSReadSem(): безусловное получение семафора.....	215
OSResetCycTmr(): сброс циклического таймера.....	216
OSRpt(): показ состояния всех задач, событий, очередей и счетчиков.....	216
OSSched(): выполнение наиболее приоритетной готовой задачи.....	218
OSSetCycTmrPeriod(): установка периода циклического таймера.....	219
OSSetEFlag(): установка битов флага события.....	219
OSSetPrio(): изменение приоритета текущей задачи.....	221
OSSetPrioTask(): изменение приоритета задачи.....	222
OSSetTicks(): инициализация системного таймера.....	223
OSSetTS(): инициализация временной отметки текущей задачи.....	224
OSSignalBinSem(): сигнал двоичного семафора.....	225
OSSignalMsg(): передача сообщения.....	226
OSSignalMsgQ(): передача сообщения через очередь сообщений.....	228
OSSignalSem(): сигнал семафора.....	229
OSStartCycTmr(): старт циклического таймера.....	230
OSStartTask(): сделать задачу готовой.....	231
OSStopCycTmr(): останов циклического таймера.....	232
OSStopTask(): останов задачи.....	233
OSSyncTS(): синхронизация временной отметки текущей задачи.....	234
OSTimer(): использование таймера.....	235
OSTryBinSem(): получение двоичного семафора если доступно.....	236
OSTryMsg(): Получение сообщения если доступно.....	237
OSTryMsgQ(): получение сообщения из очереди сообщений если доступно.....	238
OSTrySem(): получение семафора если доступно.....	239
Дополнительные сервисы пользователя.....	240
OSAnyEligibleTasks(): проверка на готовые задачи.....	240
OScTcbExt0 1 2 3 4 5, OSTcbExt0 1 2 3 4 5(): получение расширения Tcb.....	241
OSCycTmrRunning(): проверка циклического таймера на работу.....	242
OSDi(), OSEi(): управление прерываниями.....	243
OSProtect(), OSUnprotect(): защита сервисов от разрушения прерываниями.....	243
OSTimedOut(): проверка на таймаут.....	244
OSVersion(), OSVERSION: получение версии в виде целого.....	246
Пользовательские макросы.....	247
_OSLabel(): определение метки для переключения контекста.....	247
OSECBP(), OSEFCBP(), OSMQCBP(), OSTCBP(): получение указателя на блок управления.....	247
Сервисы определяемые пользователем.....	249
OSDisableIntsHook(), OSEnableIntsHook(): средства еправления грерываниями.....	249
OSIdlingHook(): средства холостой функции.....	249
OSSchedDispatchHook(), OSSchedEntryHook(), OSSchedReturnHook(): средства диспетчера.....	250
Коды возврата.....	252
Типы определяемые Salvo.....	253
Переменные Salvo.....	257
Исходный код Salvo.....	259
Расположение функций Salvo.....	261
Сокращения используемые Salvo.....	263

Глава 8 • Библиотеки 267

Типы библиотек.....	267
Библиотеки для различных сред.....	267
Родные компиляторы.....	267
Неродные компиляторы.....	267

Использование библиотек	268
Переопределение установок RAM по умолчанию	268
Функциональность библиотеки	269
Типы	270
Модели памяти	270
Опции	270
Глобальные переменные	270
Конфигурации	270
Варианты	271
Ссылка на библиотеку	272
Рекомпиляция библиотек	272
GNU Make и bash Shell	273
Рекомпиляция библиотек Salvo	273
Среда Linux/Unix	273
Разные версии компилятора	273
Среда Win32	274
Настройка библиотек	274
Создание файла конфигурации собственной библиотеки	274
Создание собственной библиотеки	275
Использование собственной библиотеки при компиляции библиотеки	275
Пример – собственная библиотека с 16-разрядными задержками и ненулевым предделителем	275
Предохранение файлов пользователя salvoclcN.h	276
Восстановление стандартных библиотек	276
Собственные библиотеки для пользователей не-Salvo Pro	276
Описания Makefile	276
\salvo\src\Makefile	276
\salvo\src\Makefile2	276
\salvo\src\targets.mk	277
\salvo\src\makeXyz.bat	277

Глава 9 • Эффективность 279

Введение	279
Измерение эффективности	279
Примеры эффективности	279
Тестовые системы	279
Тестовые конфигурации	280
Тестовые программы	281
Эффективность времени компиляции	282
Размер кода (ROM)	282
Переменные (RAM)	283
Эффективность времени выполнения	284
Быстродействие сервисов пользователя	285
OS_Delay()	285
OS_Destroy()	285
OS_Prio()	285
OS_Stop()	285
OS_WaitBinSem()	285
OS_WaitMsg()	286
OS_WaitMsgQ()	286
OS_WaitSem()	286
OS_Yield()	286
OSCreateBinSem()	286
OSCreateMsg()	287
OSCreateMsgQ()	287
OSCreateSem()	287
OSCreateTask()	287
OSInit()	287

OSSched().....	288
OSSignalBinSem()	288
OSSignalMsg()	288
OSSignalMsgQ()	288
OSSignalSem()	289
OSStartTask()	289
OSTimer().....	289
Максимальные времена работы переменных	289
t_InsPrioQ	290
t_DelPrioQ	290
t_InsDelayQ.....	290
t_DelDelayQ	291
Влияние операций с очередями.....	292
Простые очереди.....	293
t_InsPrioQ.....	293
Конфигурации I и III	294
Конфигурации II и IV	294
Конфигурация V	294
t_DelPrioQ	294
Конфигурации I & III	294
Конфигурации II & IV	294
Конфигурация V	294
t_InsDelayQ	295
Конфигурации II & IV	295
Конфигурация V	295
t_DelDelayQ	296
Конфигурации II & IV	296
Конфигурация V	296
Другие операции с переменной скоростью	296
t_InitTcb	296
Конфигурация I	297
Конфигурация II	297
Конфигурация III	297
Конфигурация IV	297
Конфигурация V	297
t_InitEcb	297
Конфигурация I	297
Конфигурация II	297
Конфигурация III	297
Конфигурация IV	298
Конфигурация V	298
Глава 10 • Переносимость.....	299
Глава 11 • Советы и приемы	301
Введение	301
Ошибки времени компиляции.....	301
Я получаю много ошибок в самом начале	301
Мой компилятор не может найти salvo.h	301
Мой компилятор не может найти salvocfg.h	301
Мой компилятор не может найти некоторые заголовочные файлы для целевого процессора	302
Мой компилятор не может определить местонахождение некоторого сервиса Salvo.....	302
Мой компилятор выдает ошибку "неопределенный символ" для метки переключения контекста, которую я определил правильно	302
Мой компилятор что-то сообщает об OSIdlingHook	302
Мой компилятор не имеет инструментов командной строки. Могу ли я создавать библиотеки?	303

Ошибки времени выполнения.....	303
Ничего не происходит.....	303
Моя программа работает только при пошаговой отладке	304
Программа все еще не работает. Как я должен начать отладку?	304
Поведение моей программы все еще бессмысленно	305
Проблемы компилятора	305
Где я могу взять бесплатный компилятор Си?	305
Где я могу взять бесплатную утилиту make?	305
Где я могу взять Linux/Unix-подобную оболочку для моего Windows PC?	305
Мой компилятор ведет себя странно при компиляции из командной строки DOS, например "Программа выполнила недопустимую операцию и будет завершена".	305
Мой компилятор выдает ошибки повторного определения когда я компилирую мою программу с исходными файлами Salvo	306
Компилятор HI-TECH PICC	306
Выполнение HPDPIC под Windows 2000 Pro	306
Установка формата ошибки/предупреждения PICC под Windows 2000 Pro	306
Компоновщик сообщает об ошибках настройки	307
Размещение переменных в RAM	307
Ошибки компоновки при работе с библиотеками	307
Обход абсолютных путей имен файлов	308
Скомпилированный код не работает	308
Ошибки передачи указателей PIC17CXXX	308
Операторы while() и переключатели контекста	308
Генерация библиотек в HPDPIC	308
Проблемы с банками переменных Salvo в 12-разрядных устройствах	309
Работа с сообщениями Salvo	309
Добавление OSTimer() в процедуру обработки прерывания	309
Использование директивы pragma interrupt_level	310
Компилятор HI-TECH V8C	310
Симуляторы	310
Компилятор HI-TECH 8051C	311
Проблемы с инициализацией статических переменных и малой и средней моделями памяти.....	311
Компилятор IAR PICC	311
Заголовочные файлы, специфические для процессора	311
Прерывания.....	311
Компилятор Mix Power C	311
Необходимые опции компилятора	312
Приложение терпит сбой после добавления длинных исходных строк Си в задачу Salvo	312
Приложение терпит сбой после добавления сложных выражений в задачу Salvo	312
Приложение терпит сбой при компиляции с опцией /t	313
Приложение терпит сбой при использовании системы make	313
Компилятор Metrowerks CodeWarrior	313
Компилятор терпит внутреннюю фатальную ошибку при компиляции вашего исходного кода	313
Microchip MPLAB	314
Окно стека показывает вложенные прерывания	314
Управление размером вашего приложения	314
Работа с указателями сообщений.....	314

Приложение А • Источники..... 317

Публикации Salvo.....	317
Заметки о применении	317
Сборочные руководства.....	318
Руководства по применению компиляторов.....	318
Изучение Си	318
Керниган и Ритчи	318
Справочное руководство по Си	318

Power C	318
Ядра реального времени	319
μC/OS и MicroC/OS-II	319
CTask	319
Программирование встроенных систем	319
Вопросы RTOS	320
Инверсия приоритетов	320
Микроконтроллеры	320
PIC16	320
Приложение В • Другие источники	321
Интернет-ссылки на другие источники	321
Приложение С • Файлы и программы	323
Обзор	323
Тестовые системы	323
Проекты	324
Номенклатура	324
Исходные файлы	325
Предопределенные символы SYS	325
Типы файлов	325
Включенные проекты и программы	327
Демонстрационные программы	327
demo\d1\sya e f t	327
demo\d2\sya f h	328
demo\d3\sya j †	328
demo\d4\sya e f h †	328
Примеры программ	328
ex\ex1\sya e f h i p q r s t v w x aa	328
ex\ex2\sya	328
Шаблоны	329
tpl\te1	329
Тестовые программы	329
test\t1\sya b c d	329
test\t2\sya b c d	329
test\t3\sya b c d	329
test\t4\sya b c	329
test\t5\sya b c	330
test\t6\sya b c d	330
test\t7\sya b c d	330
test\t8\sya b c d	330
test\t9\sya b c d	330
test\t10\sya b c d	330
test\t11\sya	330
test\t12\sya	330
test\t13\sya	330
test\t14\sya	330
test\t15\sya	331
test\t16\sya	331
test\t17\sya	331
test\t18\sya	331
test\t19\sya	331
test\t20\sya	331
test\t21\sya	331
test\t22\sya	331
test\t23\sya	332
test\t24\sya	332

test\t25\sysa	332
test\t26\sysa	332
test\t27\sysa	332
test\t28\sysa	332
test\t29\sysa	332
test\t30\sysa	332
test\t31\sysa	332
test\t32\sysa	332
test\t33\sysa	332
test\t34\syself	332
test\t35\syso	332
test\t36\sysa	333
test\t37\sysf	333
test\t38	333
test\t39	333
test\t40-t47\sysa e f i l p q r s t.....	333
Учебные программы	334
tut\tu1\sysa e f h i l m p q r s t v w x y aa †	334
tut\tu2\sysa e f h i l m p q r s t v w x y aa †	334
tut\tu3\sysa e f h i l m p q r s t v w x y aa †	334
tut\tu4\sysa e f h i l m p q r s t v w x y aa †	334
tut\tu5\sysa e f h i l m p q r s t v w x y aa †	334
tut\tu6\sysa e f h i l m p q r s t v w x y aa †	335
Библиотечные файлы	335
lib*. *	335
Файлы других производителей.....	335
free\links*. *	335

Рисунки

Рисунок 1: Процессы переднего и заднего планов.....	12
Рисунок 2: Прерывания могут происходить во время выполнения задачи.....	15
Рисунок 3: Приоритетное планирование.....	15
Рисунок 4: Кооперативное планирование.....	16
Рисунок 5: Состояния задачи.....	19
Рисунок 6: Двоичные и счетные семафоры.....	24
Рисунок 7: Передача сигнала двоичному семафору.....	24
Рисунок 8: Ожидание двоичного семафора когда событие уже произошло.....	25
Рисунок 9: Сигнализация двоичного семафора когда задача ждет соответствующее событие.....	25
Рисунок 10: Синхронизация двух задач с флагами событий.....	26
Рисунок 11: Использование счетного семафора для реализации кольцевого буфера.....	28
Рисунок 12: Сигнализация сообщения с указателем на содержимое сообщения.....	30
Рисунок 13: Экран приглашения.....	43
Рисунок 14: Экран регистрации.....	44
Рисунок 15: Экран найдена предыдущая версия.....	44
Рисунок 16: Экран лицензионное соглашение Salvo.....	45
Рисунок 17: Экран выбор места установки.....	45
Рисунок 18: Экран тип установки.....	46
Рисунок 19: Экран выбор папки программ.....	47
Рисунок 20: Экран готовность к установке.....	47
Рисунок 21: Экран поддерживаемые компиляторы.....	48
Рисунок 22: Экран завершение.....	48
Рисунок 23: Содержание типичной директории Salvo.....	50
Рисунок 24: Папка меню старт.....	50
Рисунок 25: Запуск деинсталлятора.....	50
Рисунок 26: Экран подтверждения удаления файлов.....	51
Рисунок 27: Экран завершения деинсталляции.....	51
Рисунок 28: Построение приложения Salvo с библиотеками.....	81
Рисунок 29: Построение приложения Salvo с исходными кодами.....	83
Рисунок 30: Как вызвать OSCreateBinSem() когда OSCALL_OSCREATEEVENT установлен в OSFROM_BACKGROUND.....	95
Рисунок 31: Как вызвать OSCreateBinSem() когда OSCALL_OSCREATEBINSEM установлен в OSFROM_FOREGROUND.....	95
Рисунок 32: Как вызвать OSCreateBinSem() когда OSCALL_CREATEBINSEM установлен в OSFROM_ANYWHERE.....	96
Рисунок 33: Вывод программы расширения Tcb.....	113
Рисунок 34: Вывод на экран терминала OSRpt().....	218

Листинги

Листинг 1: Простая программа Salvo	4
Листинг 2: Требования к свойствам компилятора Си	6
Листинг 3: Ошибка реентерабельности с printf()	13
Листинг 4: Структура задачи в приоритетной многозадачности	17
Листинг 5: Структура задачи в кооперативной многозадачности	17
Листинг 6: Цикл задержки	20
Листинг 7: Задержка через RTOS	21
Листинг 8: Примеры событий	22
Листинг 9: Синхронизация задач при помощи двоичных семафоров	27
Листинг 10: Использование двоичного семафора для управления доступом к ресурсу	27
Листинг 11: Использование счетного семафора для управления доступом к ресурсу	28
Листинг 12: Передача сигнала сообщения с указателем	30
Листинг 13: Прием сообщения и операция над его содержимым	30
Листинг 14: Суперцикл торгового автомата	33
Листинг 15: Версия задачи ReleaseItem()	36
Листинг 16: Версия задачи CallPolice()	37
Листинг 17: Приоритизация задачи	37
Листинг 18: Создание события сообщения	37
Листинг 19: Вызов системного таймера	37
Листинг 20: Запуск всех задач	38
Листинг 21: Начало многозадачности	38
Листинг 22: Торговый автомат на основе RTOS	40
Листинг 23: Минимальное приложение Salvo	53
Листинг 24: Многозадачное приложение Salvo с двумя задачами	55
Листинг 25: Многозадачность с двумя нетривиальными задачами	58
Листинг 26: Многозадачность с использованием событий	60
Листинг 27: Многозадачность с задержкой	64
Листинг 28: Вызов OSTimer() с частотой системных часов	65
Листинг 29: Передача сигналов из нескольких задач	68
Листинг 30: Файл salvocfg.h для учебной программы	77
Листинг 31: Пример расширения Tcb	113
Листинг 32: Файл salvocfg.h для нескольких проектов	133
Листинг 33: Использование SYSA ... в main.c	134
Листинг 34: Использование SYSA ... SYSZ в salvocfg.h	134
Листинг 35: Использование USE_INTERRUPTS в isr.c	135
Листинг 36: Устаревшие параметры конфигурации	139
Листинг 37: Задачи без переключения контекста	173
Листинг 38: Задача с корректным переключением контекста	173
Листинг 39: Ошибочное переключение контекста вне задачи	174
Листинг 40: Задача, использующая долгоживущую локальную переменную	175
Листинг 41: Задача, использующая локальные переменные auto	175
Листинг 42: Файлы исходных кодов	260
Листинг 43: Расположение аункций в исходном коде	262
Листинг 44: Список сокращений	264
Листинг 45: Пример salvocfg.h для использования со стандартной библиотекой	268
Листинг 46: Пример salvocfg.h со стандартной библиотекой и уменьшенным числом задач	269
Листинг 47: Дополнение в salvocfg.h для уменьшения требуемой памяти с библиотеками Salvo ...	269
Листинг 48: Частичный список сервисов, которые могут быть вызваны из прерываний	272
Листинг 49: Создание одиночной библиотеки Single	273
Листинг 50: Создание библиотек Salvo для специфического компилятора	273
Листинг 51: Создание библиотек Salvo для специфического процессора	273
Листинг 52: Получение списка библиотечных целевых процессоров в Makefile	273
Листинг 53: Создание библиотек Salvo для компилятора IAR MSP430 C v2.x	274
Листинг 54: Пример файла конфигурации пользовательской библиотеки salvoclc4.h	275

Листинг 55: Создание пользовательской библиотеки Salvo с конфигурацией 4	275
Листинг 56: Пример salvocfg.h для создания пользовательской библиотеки конфигурации 4 и средств разработки Archelon/Quadravox AQ430	276
Листинг 57: Создание библиотек Salvo PICC для PICmicro в среде Win32 без рекурсивного Make.	277

Таблицы

Таблица 1: Поддерживаемые процессоры и компиляторы	xxiii
Таблица 2: Допустимые типы хранения / квалификаторы типов для объектов Salvo	116
Таблица 3: Опции конфигурации по категориям.....	135
Таблица 4: Опции конфигурации по требуемым свойствам.....	137
Таблица 5: Предопределенные символы.....	138
Таблица 6: Коды возврата	252
Таблица 7: Нормальные типы	254
Таблица 8: Нормальные типы указателей	255
Таблица 9: Квалифицированные типы	255
Таблица 10: Квалифицированные типы указателей	255
Таблица 11: Переменные Salvo.....	258
Таблица 12: Литерные коды библиотек Salvo.....	270
Таблица 13: Коды конфигурации для библиотек Salvo.....	270
Таблица 14: Общие свойства всех конфигураций библиотек Salvo	271
Таблица 15: Коды вариантов библиотек Salvo	272
Таблица 16: Обзор тестовых систем.....	279
Таблица 17: Свойства разрешенные в тестовых конфигурациях I-V	280
Таблица 18: Использование ROM и RAM для тестовых программ 1-5 в тестовых системах A и B..	281
Таблица 19: Скорости и времена переключения контекста для тест-программ 6-10 в тест-системах A-C	281
Таблица 20: Требования RAM для конфигураций I-V в тест-системах A-C	284
Таблица 21: Время выполнения OS_Delay()	285
Таблица 22: Время выполнения OS_Destroy().....	285
Таблица 23: Время выполнения OS_Prio().....	285
Таблица 24: Время выполнения OS_Stop().....	285
Таблица 25: Время выполнения OS_WaitBinSem()	285
Таблица 26: Время выполнения OS_WaitMsg()	286
Таблица 27: Время выполнения OS_WaitMsgQ()	286
Таблица 28: Время выполнения OS_WaitSem()	286
Таблица 29: Время выполнения OS_Yield()	286
Таблица 30: Время выполнения OSCreateBinSem().....	286
Таблица 31: Время выполнения OSCreateMsg()	287
Таблица 32: Время выполнения OSCreateMsgQ().....	287
Таблица 33: Время выполнения OSCreateSem().....	287
Таблица 34: Время выполнения OSCreateTask()	287
Таблица 35: Время выполнения OSInit()	287
Таблица 36: Время выполнения OSSched()	288
Таблица 37: Время выполнения OSSignalBinSem()	288
Таблица 38: Время выполнения OSSignalMsg()	288
Таблица 39: Время выполнения OSSignalMsgQ()	288
Таблица 40: Время выполнения OSSignalSem().....	289
Таблица 41: Время выполнения OSStartTask	289
Таблица 42: Время выполнения OSTimer().....	289
Таблица 43: Максимум t_InsPrioQ для 1-8 задач в конфигурациях I-V (простые очереди)	290
Таблица 44: Максимум t_DelPrioQ для 1-8 задач в конфигурациях I-V (простые очереди).....	290
Таблица 45: Максимум t_InsDelayQ для 1-8 задач в конфигурациях I-V (простые очереди, 8-бит задержки, w/OSSPEEDUP_QUEUEING)	290
Таблица 46: Максимум t_InsDelayQ для 1-8 задач в конфигурациях I-V (простые очереди, 16-бит задержки, w/OSSPEEDUP_QUEUEING)	291
Таблица 47: Максимум t_DelDelayQ для 1-8 задач в конфигурациях I-V (простые очереди, 8-бит задержки).....	291
Таблица 48: Максимум t_DelDelayQ для 1-8 задач в конфигурациях I-V (простые очереди, 16-бит задержки).....	291
Таблица 49: Пример времени выполнения операций с очередью	292

Таблица 50: t_InsPrioQ для конфигураций I & III	294
Таблица 51: t_InsPrioQ для конфигураций II & IV.....	294
Таблица 52: t_InsPrioQ для конфигурации V	294
Таблица 53: t_DelPrioQ для конфигураций I & III	294
Таблица 54: t_DelPrioQ для конфигураций II & IV	294
Таблица 55: t_DelPrioQ для конфигурации V	294
Таблица 56: t_InsDelayQ для конфигураций II и IV и 8-бит задержек	295
Таблица 57: t_InsDelayQ для конфигураций II и IV и 16-бит задержек	295
Таблица 58: t_InsDelayQ для конфигураций II и IV и 8-бит задержек, с OSSPEEDUP_QUEUEING....	295
Таблица 59: t_InsDelayQ для конфигураций II и IV и 16-бит задержек, с OSSPEEDUP_QUEUEING..	295
Таблица 60: t_InsDelayQ для конфигураций V и 8-бит задержек	295
Таблица 61: t_InsDelayQ для конфигурации V и 16-бит задержек	295
Таблица 62: t_InsDelayQ для конфигурации V и 8-бит задержек, с OSSPEEDUP_QUEUEING.....	296
Таблица 63: t_InsDelayQ для конфигурации V и 16-бит задержек, с OSSPEEDUP_QUEUEING.....	296
Таблица 64: t_DelDelayQ для конфигураций II, IV и 8-бит задержек.....	296
Таблица 65: t_DelDelayQ для конфигураций II, IV и 16-бит задержек.....	296
Таблица 66: t_DelDelayQ для конфигурации V и 8-бит задержек.....	296
Таблица 67: t_DelDelayQ для конфигурации V и 16-бит задержек.....	296
Таблица 68: t_InitTcb для конфигурации I.....	297
Таблица 69: t_InitTcb для конфигурации II.....	297
Таблица 70: t_InitTcb для конфигурации III.....	297
Таблица 71: t_InitTcb для конфигурации IV	297
Таблица 72: t_InitTcb для конфигурации V	297
Таблица 73: t_InitEcb для конфигурации I	297
Таблица 74: t_InitEcb для конфигурации II	297
Таблица 75: t_InitEcb для конфигурации III	298
Таблица 76: t_InitEcb для конфигурации IV	298
Таблица 77: t_InitEcb для конфигурации V	298
Таблица 78: Test System Names, Targets and Development Environments	323
Таблица 79: Configurations for Test Programs t40-t47	333

Замечания о выпуске

Введение

Что нового

Пожалуйста, обратитесь к файлу `salvo-whatsnew.txt` из комплекта дистрибутива за подробной информацией о том, что нового появилось в выпуске версии v3.2.2.

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 v3.2.2.

Версии инструментов третьих сторон

Пожалуйста, обратитесь к специфическим для дистрибутивов замечаниям (`salvo-release-targetname.txt`) для получения номеров версий инструментов третьих сторон (компиляторы, компоновщики, библиотеки и т.д.) для выпуска версии v3.2.2.

Процессоры и компиляторы

С момента выпуска версии v3.2.2, Salvo поддерживает следующие процессоры и компиляторы:

Таблица 1: Поддерживаемые процессоры и компиляторы

Процессоры	Компиляторы
Atmel AVR и MegaAVR	<ul style="list-style-type: none"> ImageCraft ICCAVR
Семейство Intel 8051 и его производные	<ul style="list-style-type: none"> HI-TECH 8051C Keil Cx51
Семейство Intel 80x86 и его производные	<ul style="list-style-type: none"> gcc (GNU C Compiler) (Cygwin special) Metrowerks CodeWarrior IDE
Семейства Microchip PIC12, PIC16 и PIC17 PICmicro	<ul style="list-style-type: none"> HI-TECH PICC
Семейство Microchip PIC18	<ul style="list-style-type: none"> HI-TECH PICC-18 IAR PIC18 C Microchip MPLAB-C18
Motorola M68HC11	<ul style="list-style-type: none"> ImageCraft ICC11
TI's MSP430	<ul style="list-style-type: none"> Archelon / Quadravox AQ430 IAR MSP430 C ImageCraft ICC430 Rowley Associates CrossStudio for MSP430
VAutomation V8-μRISC	<ul style="list-style-type: none"> HI-TECH V8C

Пожалуйста, обратитесь к специфическим для дистрибутивов замечаниям (`salvo-release-targetname.txt`) за номерами версий инструментов третьих сторон (компиляторы, компоновщики, библиотеки и т.д.) для выпуска v3.2.2. Если вы имеете названный компилятор более старой версии, чем указана в списке, вам может понадобиться обновить его для работы с Salvo. Свяжитесь с поставщиком компилятора для получения информации по обновлению.

Предисловие

Salvo 2.0 была первым коммерческим выпуском кооперативной приоритетной многозадачной RTOS компании Pumpkin Inc. Salvo 1.0 была внутренним выпуском, написанным на ассемблере и предназначенным специально для Microchip PIC17C756 PICmicro в частной внутренней системе сбора данных.

Salvo 1.0 обеспечивала большую часть базовых функций 2.0. Было решено расширить ее функциональность, переписав Salvo на Си. При этом возникли возможности многочисленных опций конфигурации и оптимизации, что сделало Salvo по сравнению с предшественницей 1.0 не только более мощной и гибкой, но также полностью переносимой.

Типографские соглашения

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

<p>Замечание: Исходный код Salvo использует шаг табуляции равный 2, т.е. символ табуляции эквивалентен 2-м символам пробела. (От переводчика: В оригинале шаг табуляции объявлен как 4. Переводчиком был использован шаг табуляции 2.)</p>

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` и/или `syse` и/или `sysf`.

Имена путей DOS и Windows используют символ `\`. Имена путей Linux и 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 полей¹, 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:

<code>salvo-lite-pic-2.2.0</code>	v2.2 Salvo Lite для PICmicro MCU, инсталлятор, выпуск
<code>salvo-le-8051-3.1.0-rc3</code>	v3.1.0 Salvo LE для 8051 семейства, инсталлятор, кандидат на выпуск #3
<code>salvo-pro-avr-3.2.2</code>	v3.2.2 Salvo Pro для AVR и MegaAVR, инсталлятор, выпуск

Выпуски Salvo обобщенно упоминаются по их номерам `MAJOR.MINOR`, например "выпуск 3.0".

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, `OSENABLE_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 для компиляции. Хотя сначала это может показаться насилием, 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 заявлено и без ошибок, вполне возможно, что мы могли пропустить 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 Salvo².

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, 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, идите и возьмите 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 тех же самых программных решений, используемых большими 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 размещаемая 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 здесь имеется. Нет стека? 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.

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

Что такое Salvo?

Salvo – мощная, высокоэффективная и недорогая Операционная Система Реального Времени – OCPB (Real Time Operating System – RTOS), требующая очень мало памяти и не требующая стека. 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.

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 гарантирования успеха. Salvo обеспечивает 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 очень похожей на любую другую, которая выполняется в многозадачной среде RTOS. Листинг 1 содержит исходный текст (с комментариями) 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, земля (общий), Rx (получение желательной температуры кресла 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, чтобы указать, что кресло имеет желаемую температуру.

```
#include <salvo.h>

typedef unsigned char t_boolean;
typedef unsigned char t_temp;

/* Salvo context-switching labels */
_OSLabel(TaskControl1)
_OSLabel(TaskStatus1)
_OSLabel(TaskStatus2)
_OSLabel(TaskStatus3)
_OSLabel(TaskStatus4)

/* 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 )
{
    for (;;)
    {
        warm = CtrlTemp(UserTemp(), SeatTemp());
        OS_Yield(TaskControl1);
    }
}

/* 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;

    for (;;)
    {
        OS_Delay(100, TaskStatus1);
        TX_PORT |= 0x01;
        OS_Delay(5, TaskStatus2);
        TX_PORT &= ~0x01;
        if (warm)
        {
            OS_Delay(UL TaskStatus3);
            TX_PORT |= 0x01;
            OS_Delay(UL TaskStatus4);
            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);

    for (;;)
        OSSched();
}

```

Листинг 1: Простая программа Salvo

Важно заметить, что, когда программа выполняется, управление температурой продолжается постоянно, тогда как TaskStatus() периодически задерживается. Вызов OS_Delay() мог бы выполнять программный цикл на некоторое время, и затем продолжить выполнение. Это было бы тратой ресурсов процессора, то есть циклов команд. Взамен, этого программа просто обращается к Salvo, чтобы приостановить генератор импульсов и гарантировать, что он продолжит работать 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. Вообразите насколько просто добавить большее количество задач к этому приложению, чтобы расширить функциональные возможности.

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

Какие ресурсы требуются Salvo?

Salvo не использует стек общего назначения. Это означает, что, даже если ваш процессор не имеет инструкций PUSH и POP или регистров стека, вы, вероятно, сможете использовать Salvo. Единственный стек, требуемый Salvo – это тот, который поддерживает вызов и возврат из функции, то есть так называемый call-return или аппаратный стек.

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.

Объем RAM, требуемый Salvo, также зависит от специфики вашей конфигурации. В RISC-приложении⁴, каждая задача требует 4...12 (обычно 7) байт, каждое событие – 3...4 байта⁵, 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 или 16 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, Salvo требует, чтобы вы обеспечили это одним прерыванием. Это прерывание не требуется самому 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 стек⁶. 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 задач⁷.

⁴ Серия PIC16 (например, PIC16C64). Указатели на ROM требуют 2 байта, указатели на RAM требуют 1 байт.

⁵ Очереди сообщений требуют дополнительной RAM.

⁶ Под "стеком" мы подразумеваем стек общего назначения, которым можно манипулировать (например, при помощи инструкций PUSH и POP). Salvo также требует call...return стек, иногда называемый "аппаратный стек".

⁷ Мы объясним этот термин позже, но сейчас это означает находиться в одной задаче и освобождать управление процессором так, чтобы могла выполняться другая задача.

- 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, 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 – 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.

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 Salvo из исходного текста, который использует многие свойства Си, 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 аргументов⁸ (`va_arg()`, и т.д.)

Листинг 2: Требования к свойствам компилятора Си

Ваш компилятор должен уметь использовать встроенный ассемблер. Чем более функционален 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).

⁸ Это не является абсолютно необходимым, но желательно. `va_arg()` является частью стандарта ANSI C.

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?

См. выше *Поддерживаемые Процессоры и Компиляторы*.

Как распространяется Salvo?

Salvo поставляется на CD-ROM или через Интернет как программа установки для Windows. Установка создаст группу папок с исходным кодом Salvo, примеры, демонстрационные программы, руководство и другие файлы.

Что содержится в этом руководстве?

Глава 1 • Введение.

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 • Установка 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.

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 содержит данные о размере и скорости Salvo в разных конфигурациях. Она также содержит советы о том, как оценить производительность Salvo в вашем частном приложении.

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 и Неполадки содержит информацию о проблемах, с которыми вы можете встретиться, и их решения.

Приложение А • Рекомендуемая Литература содержит ссылки на литературу о многозадачности и связанные с ней документы.

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

Приложение С • Описание Файлов и Программ содержит описания всех файлов и типов 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. Гэнсл⁹.*

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 каркас программного обеспечения, 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 специальные источники.

⁹ "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 программы¹⁰.

¹⁰ Некоторые ядра обеспечивают функции ввода-вывода и другой сервис, например управление памятью.

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 критические 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 отдельной 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 переключать основной цикл 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.

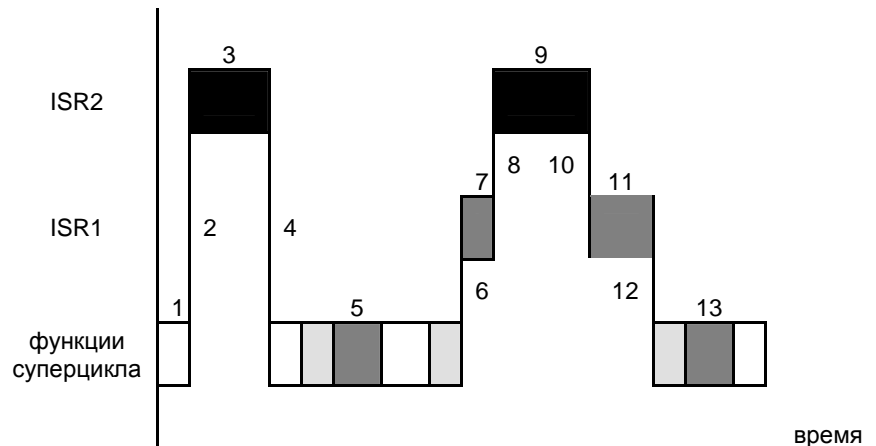


Рисунок 1: Процессы переднего и заднего планов

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 выполняться. 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. Этим способом задача А может быть прервана в любое время, чтобы позволить диспетчеру выполнить более приоритетную задачу В. Как только задача В завершена, задача А продолжается с того места, где была приостановлена. Эффект воздействия переключателя контекста на задачу А 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.

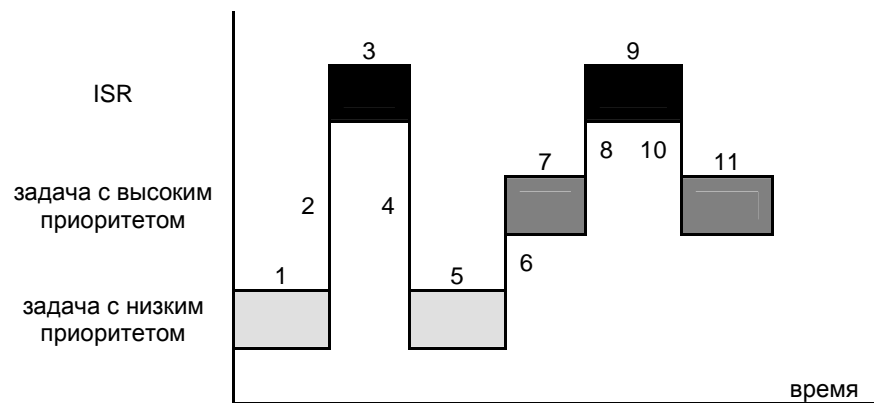


Рисунок 2: Прерывания могут происходить во время выполнения задачи

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

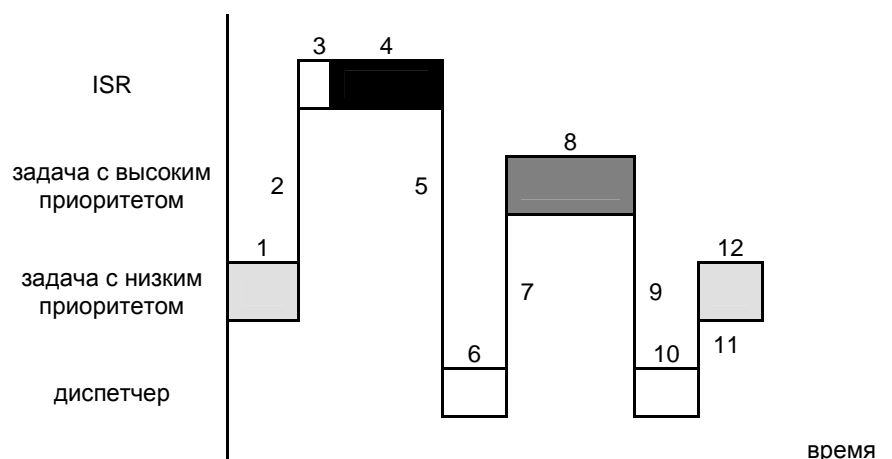


Рисунок 3: Приоритетное планирование

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* ($t_4 - t_2$). Время между концом ISR и возобновлением выполнения задачи – время *восстановления из прерывания* ($t_7 - t_5$). *Время ответа на событие системы* составляет ($t_7 - t_2$).

Кооперативное планирование

Кооперативный диспетчер более простой, чем его приоритетный аналог. 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, храня вызов переключателя контекста вне этого кода. Из-за его относительной простоты данный диспетчер имеет ряд преимуществ.

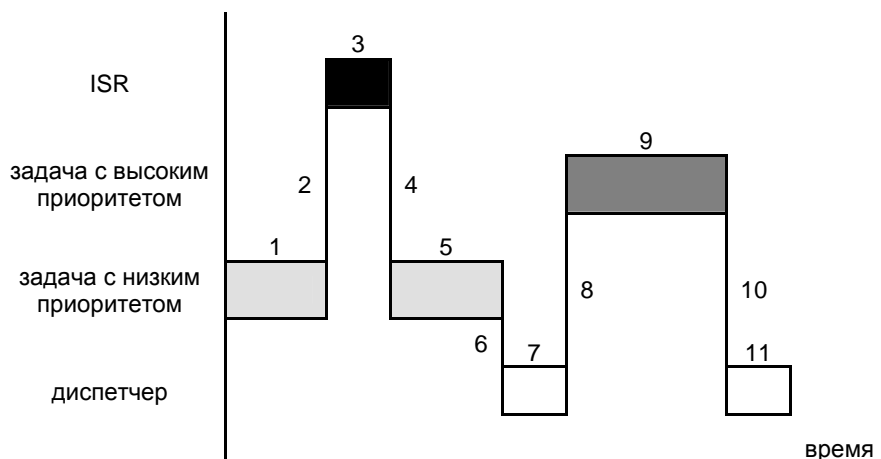


Рисунок 4: Кооперативное планирование

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 короткий ответ на прерывание и восстановление. Но реальный отклик хуже, так как высокоприоритетная готовая продолжаться 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 данный 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, остальное – дело диспетчера.

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();
for (;;)
{
    ...
}
```

Листинг 4: Структура задачи в приоритетной многозадачности

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

```
Initialize();
for (;;)
{
    ...
    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 держит 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.

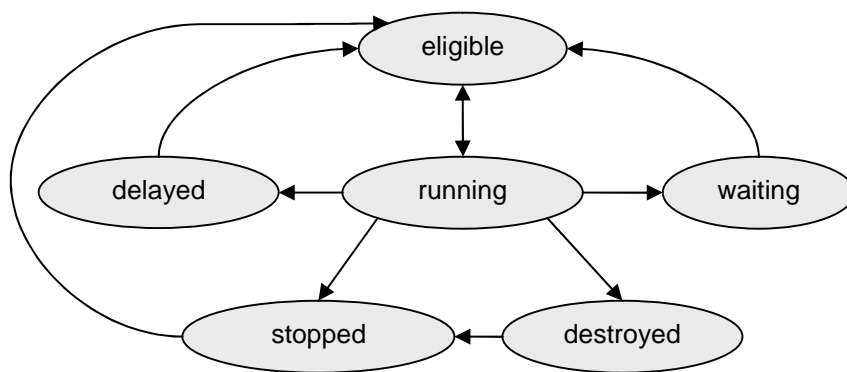


Рисунок 5: Состояния задачи

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 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 записывается для целей сброса.

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 имеют подобные ограничения. 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, даже готовая 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 завершения таймера задержки. Завершение 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 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 одного события, но того же самого события может ждать любое число задач. Если одна или более задач ждут события, и событие сигнализирует, самая высокоприоритетная задача или первая задача, ожидающая события, 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 схему¹¹ управления тем, какая задача становится готовой.

¹¹ 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 события, чтобы гарантировать, что в любое время система может ответить на событие так быстро, как только возможно. Это потому, что ждущие задачи не¹² 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 приоритета¹³. 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.

¹² Если они не ожидают с таймаутом, который требует таймера.

¹³ Приоритеты задач легко вводятся в основанную на событиях многозадачность.

Общими для всех этих межадачных коммуникаций являются два действия: передача *сигнала* (также называемая *регистрация* или *посылка*) и *ожидание* (также называемое *задержка* 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:

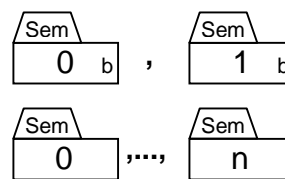


Рисунок 6: Двоичные и счетные семафоры

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*.

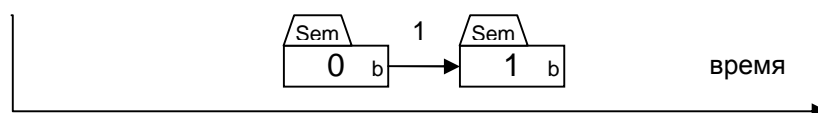


Рисунок 7: Передача сигнала двоичному семафору

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.

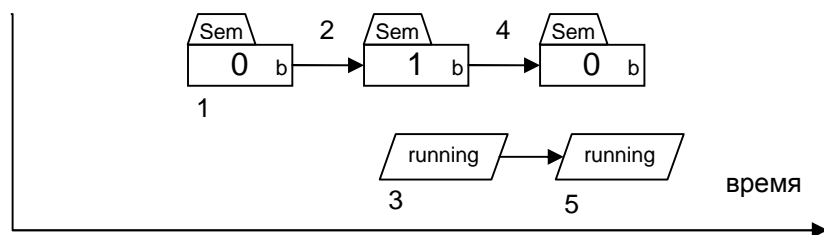


Рисунок 8: Ожидание двоичного семафора когда событие уже произошло

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 сигнализировать снова, а задача продолжит выполняться [5].

Замечание: Семафоры всегда инициализированы 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.

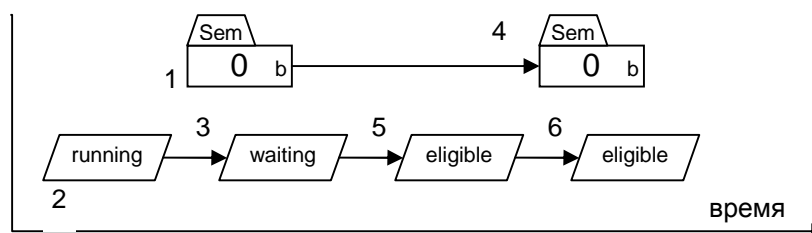


Рисунок 9: Сигнализация двоичного семафора когда задача ждет соответствующее событие

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, просигналил 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 сигналил 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.

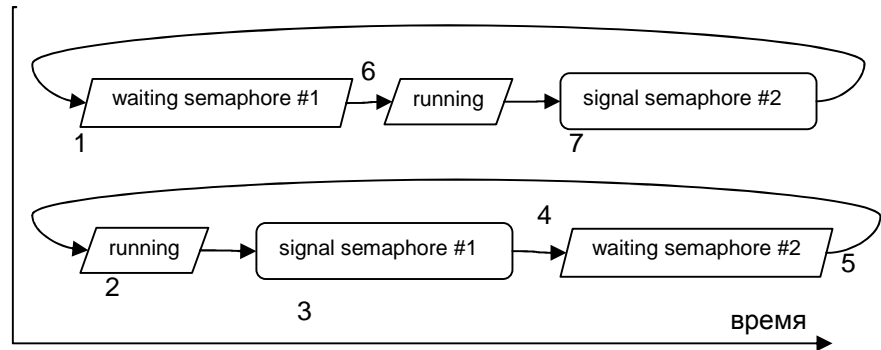


Рисунок 10: Синхронизация двух задач с флагами событий

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

LowerTask()
{
    for (;;)
    {
        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()
{
    for (;;)
    {
        ...
        prepare time & data string;
        wait binary semaphore;
        write time & date string to display;
        signal binary semaphore;
        ...
    }
}

TaskShowAlert()
{
    for (;;)
    {
        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.

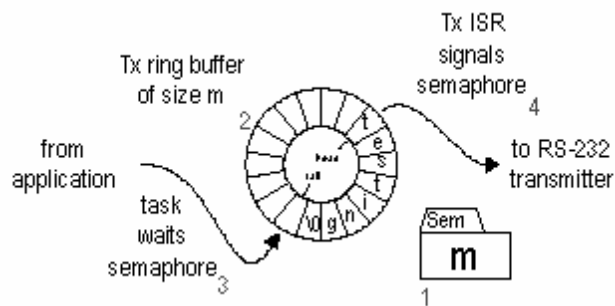


Рисунок 11: Использование счетного семафора для реализации кольцевого буфера

На Рисунке 11 счетный семафор инициализирован значением m [1] для представления числа мест, доступных в пустом кольцевом буфере [2]. Кольцевой буфер заполняется с хвоста¹⁴ задачей [3] и освобождается из головы процедурой обработки прерывания ISR [4]. Перед добавлением символа в буфер задача должна дождаться семафора. Если она заблокирована, 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 символы¹⁵, каждый символ будет извлечен из буфера прерыванием Tx ISR, передан и семафор увеличит свое значение передачей сигнала. Соответствующий псевдокод показан¹⁶ в Листинге 11.

```
initialize counting semaphore to m;

TaskFillTxBuffer()
{
    for (;;)
    {
        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

¹⁴ 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.

¹⁵ Обычно это означает разрешение прерываний по передаче.

¹⁶ 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 заблокирована (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*¹⁷.

Если сообщение инициализировано пустым содержанием, оно содержит *пустой указатель*. Пустой указатель имеет значение 0. По соглашениям, пустой указатель не указывает на что-нибудь – 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.

¹⁷ 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 и *p являются эквивалентом var.

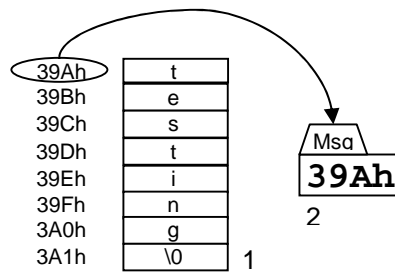


Рисунок 12: Сигнализация сообщения с указателем на содержимое сообщения

На Рисунке 12 строка символов языка Си¹⁸ [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 отправки 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 сообщения. Задача будет заблокирована до тех пор, пока сообщение не придет. 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()
{
  for (;;)
  {
    wait message containing string pointer p;
    while ((p) is not null)19
    {
      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, оно 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.

¹⁸ В языке Си символьная строка завершается символом NUL ('\\0').

¹⁹ "(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. Псылка 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 сигнализирует, то либо произошла ошибка, либо сигнализирующая 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 заблокированы, то есть они будут неопределенно долго ожидать.

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

- всегда загладевайте ресурсами в предопределенном порядке,
- загладевайте 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 вы изменили бы 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 выполняться 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, написанные для многозадачного приложения, выглядят подобно задачам, написанным для приложения с суперциклом. Наибольшее отличие состоит в структуре полной программы. Задачи для многозадачности не входят в какие-либо циклы или большие функции – все они являются независимыми функциями. `ReleaseItem()`, которая отпускает товар один раз, если набор условий был выполнен, на псевдокоде могла бы походить на следующее:

```
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 время как она ждет прибытия сообщения, `ReleaseItem()` находится в *ждущем* состоянии. Когда сообщение послано, `ReleaseItem()` становится готовой продолжать выполнение, и когда это случается, задача извлекает содержание сообщения (в данном случае, код для желаемого товара, например "B3") и отпускает его заказчику. `ReleaseItem()` не находится внутри какого-либо большего цикла и не вызывается какими-либо другими функциями (за исключением 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 многозадачности

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 назначению ей соответствующего приоритета.

Глава 3 • Инсталляция

Введение

Обычно Salvo поставляется в одной из двух форм: на CD-ROM, или 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* и *Salvo Application Notes*. Все файлы Salvo содержатся в сжатой и зашифрованной форме внутри инсталлятора. Действительный серийный номер требуется для Salvo SE, LE и Pro. Salvo Lite не требует серийного номера.

Замечание: В этом разделе предполагается, что вы инсталлируете Salvo на PC или PC совместимый компьютер с Microsoft Windows 98. Инсталляция в Windows 95, NT, 2000 и XP аналогична. Если вы инсталлируете Salvo на PC с Windows 3.1, инсталляция в основном такая же, за некоторыми исключениями, касающимися путей и имен файлов.

Запуск инсталлятора

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

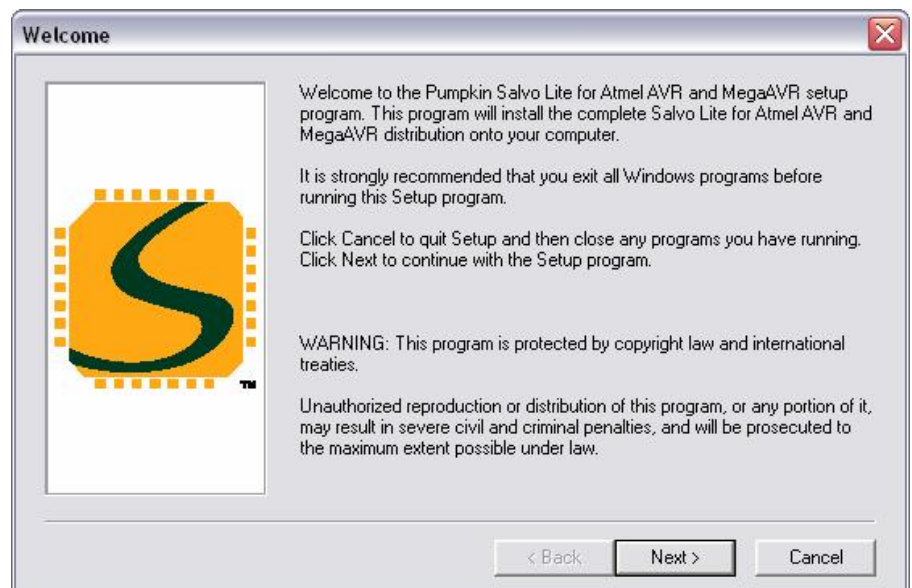


Рисунок 13: Экран приглашения

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. После нажатия кнопки **Next** появится экран Регистрации:

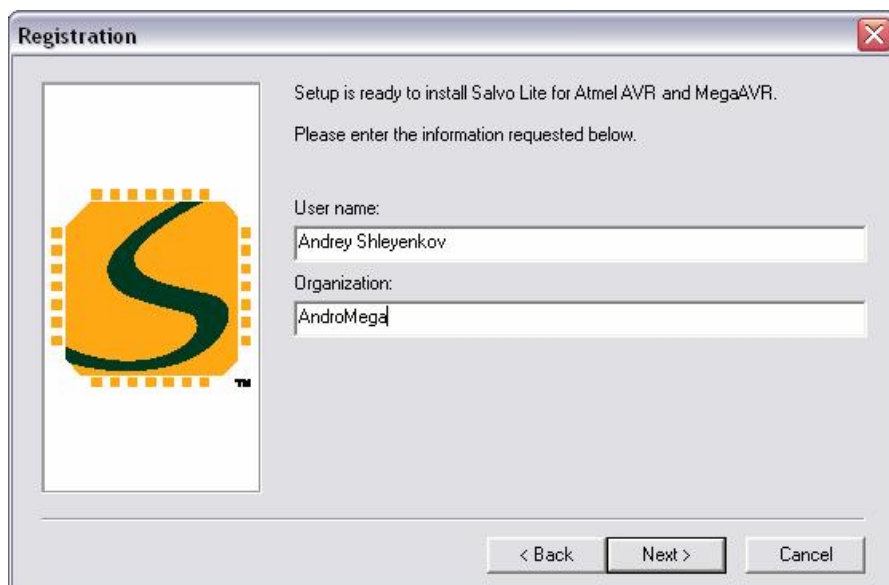


Рисунок 14: Экран регистрации

Введите ваше имя пользователя и организацию, к которой вы принадлежите (если существует). Salvo Lite не требует серийного номера, все другие дистрибутивы требуют. Серийный номер может быть найден внутри упаковки Salvo или предоставлен вам во время приобретения.

Замечание: Серийный номер является чувствительным к регистру алфавитных символов.

3. Если инсталлятор обнаружит предыдущую версию Salvo, уже установленную на вашем PC, перед продолжением он запросит вас о его удалении при помощи экрана Найдена Предыдущая Версия:



Рисунок 15: Экран найдена предыдущая версия

Установите переключатель Yes и нажмите кнопку **Next**.

4. Вы получите запрос на подтверждение удаления файлов. Нажмите кнопку **Yes**. Деинсталлятор удалит предыдущую версию Salvo этого же типа и сообщит вам о выполнении. Нажмите кнопку **OK**.

5. Появится экран Лицензионное Соглашение Salvo:

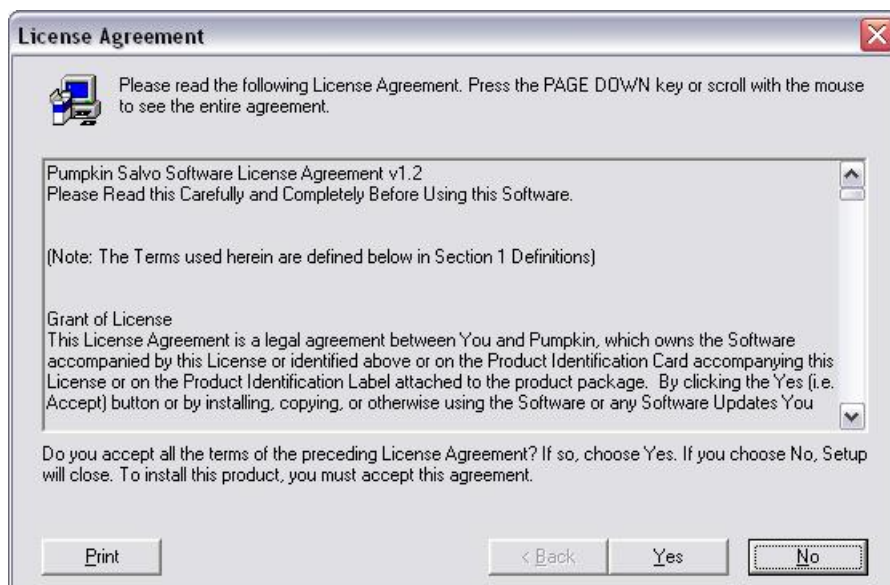


Рисунок 16: Экран лицензионное соглашение Salvo

Этот экран содержит *Лицензионное Соглашение Pumpkin Salvo*. Прочтите это соглашение внимательно. Этот документ будет включен в папку Salvo по окончании установки. Чтобы продолжить установку Salvo, вы должны принять условия Лицензии. Вы можете распечатать копию Лицензии, нажав кнопку **Print**. Чтобы принять Лицензию, нажмите кнопку **Yes**. Если вы не принимаете Лицензию, нажмите кнопку **No** и возвратите программный продукт²⁰.

6. Появится экран Выбор Места Установки:

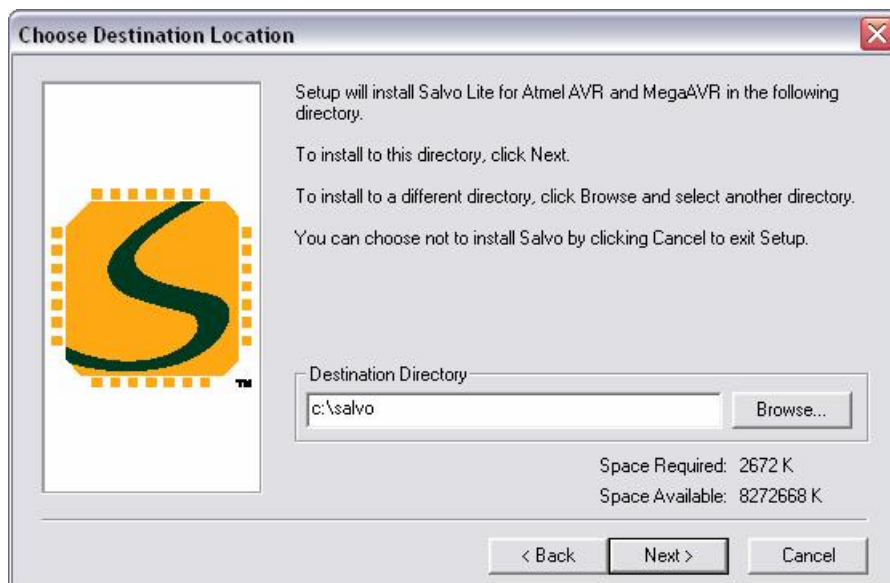


Рисунок 17: Экран выбор места установки

²⁰ Инструкции по возврату программного продукта содержатся в Лицензии и Руководстве Пользователя.

Этот экран позволяет вам определить каталог, в который будет установлена Salvo. Инсталлятор поместит в выбранном месте несколько²¹ каталогов, некоторые из которых содержат вложенные подкаталоги. Вы можете оставить каталог установки по умолчанию (c:\salvo) или изменить его, нажав на кнопку **Browse...** и выбрав другой каталог.

Замечание: Для избежания потенциальных проблем компилятора с длинными именами файлов и путей, мы рекомендуем выбирать целевой каталог, который располагается как можно ближе к корневому каталогу диска. Выбор чрезмерно глубоко вложенного каталога для установки (например, c:\My Projects\Programming\Tools\RTOS\Salvo) может вызвать проблемы с написанными под DOS и другими инструментальными средствами из-за чрезвычайно длинных имен путей файлов Salvo. Также, в именах путей нужно избегать пробелов.

7. Далее, после нажатия кнопки **Next**, появится экран Тип Установки:

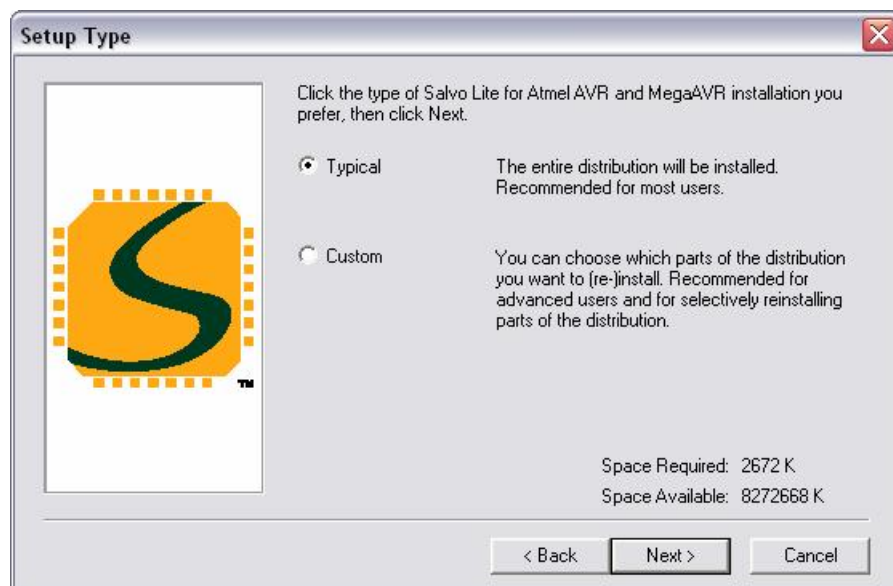


Рисунок 18: Экран тип установки

На этом экране вы можете выбрать один из двух типов установки Salvo. Большинство пользователей выберет установку **Typical**, которая устанавливает Salvo полностью. Выбирая **Custom**, вы полностью контролируете состав установки.

Совет: Если вы когда-либо случайно изменили или удалили один или несколько исходных файлов Salvo, вы можете использовать кнопку **Details** на экране Выбор Компонентов (Select Components) установки **Custom**, чтобы определить файлы, которые вы хотите восстановить или переустановить.

²¹ См. Рисунок 23 "Содержание Типичной Директории Salvo".

8. После выбора типа установки, нажмите кнопку **Next**. Появится экран Выбор Папки Программ:

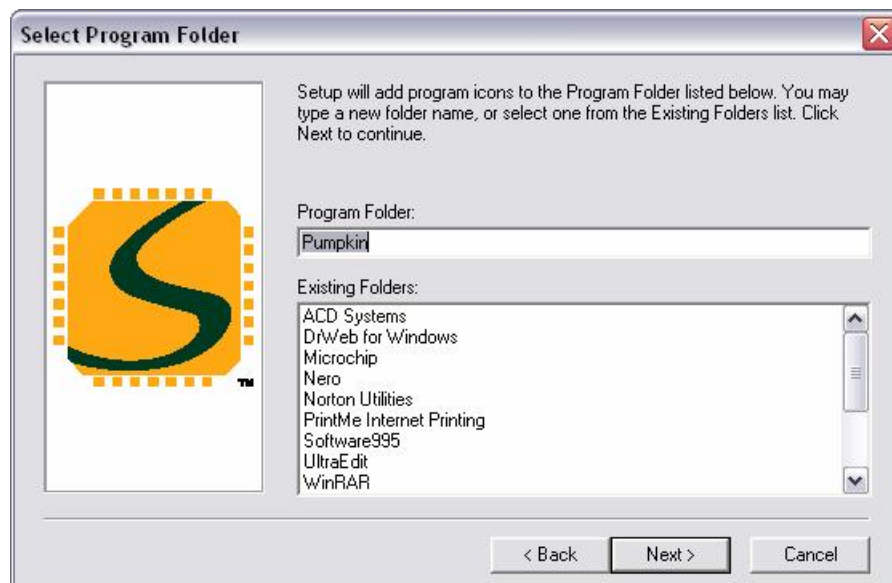


Рисунок 19: Экран выбор папки программ

9. Нажмите **Next** для продолжения. Появится экран Готовность к Установке:

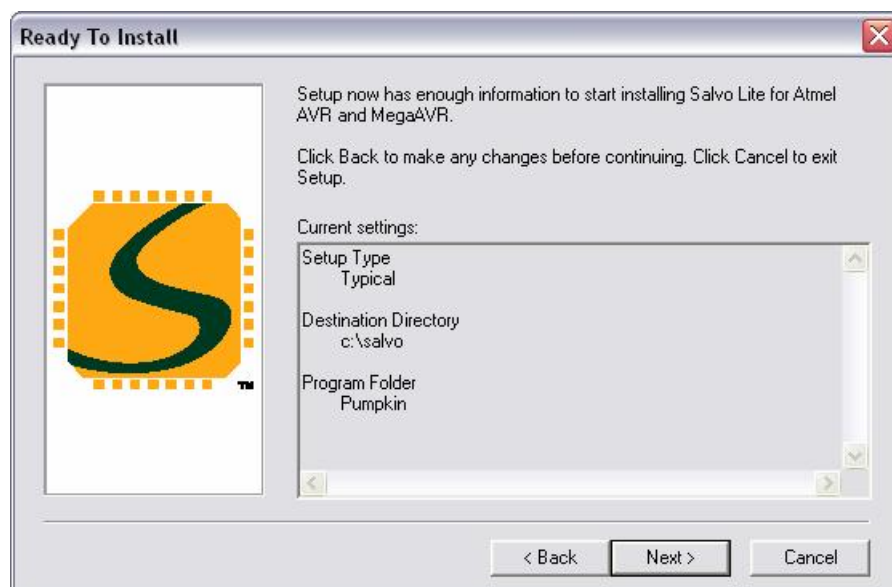


Рисунок 20: Экран готовность к установке

Проверьте правильность этих установок. Если есть ошибки, нажмите кнопку **Back** и сделайте необходимые изменения. Если все правильно, нажмите кнопку **Next**.

10. Дистрибутивы Salvo, поддерживающие несколько компиляторов, представляют экран Поддерживаемые Компиляторы:

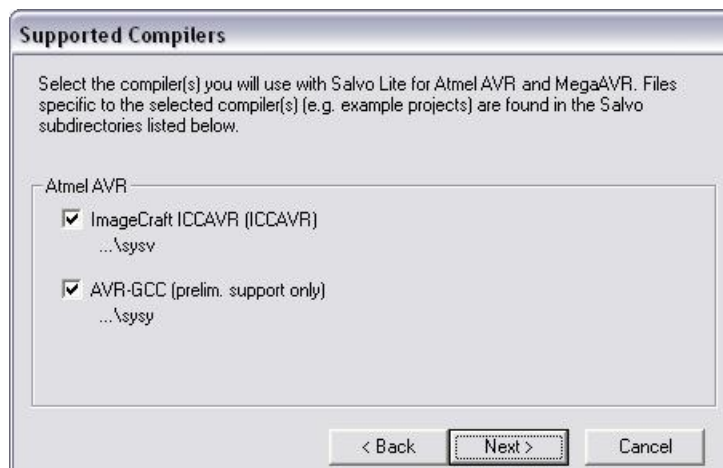


Рисунок 21: Экран поддерживаемые компиляторы

Если вы не желаете устанавливать файлы для компиляторов, которых не имеете, снимите их выбор и нажмите кнопку **Next**.

11. Инсталлятор поместит все файлы Salvo в соответствующие подкаталоги целевого каталога. Когда это выполнится, появится экран Завершение:

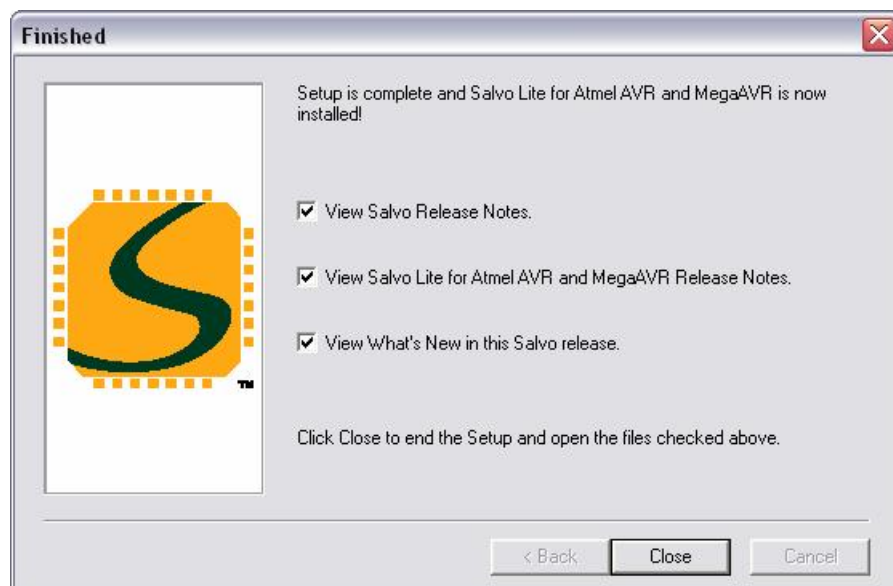


Рисунок 22: Экран завершение

12. Выберите, какие информационные файлы вы желаете прочитать, и нажмите кнопку **Close**. Эти файлы будут открыты средствами просмотра текстовых файлов вашего PC.

13. В заключение вы можете быть приглашены посетить один или несколько веб-сайтов Salvo и зарегистрировать ваше программное обеспечение.

Сетевая установка

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 установки должны выполнять условия *Лицензионного Соглашения Salvo*. См. Лицензию для получения подробной информации.

Установка 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:\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 установку. 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 установка

Директория Salvo после типовой установки должна выглядеть подобно следующей:

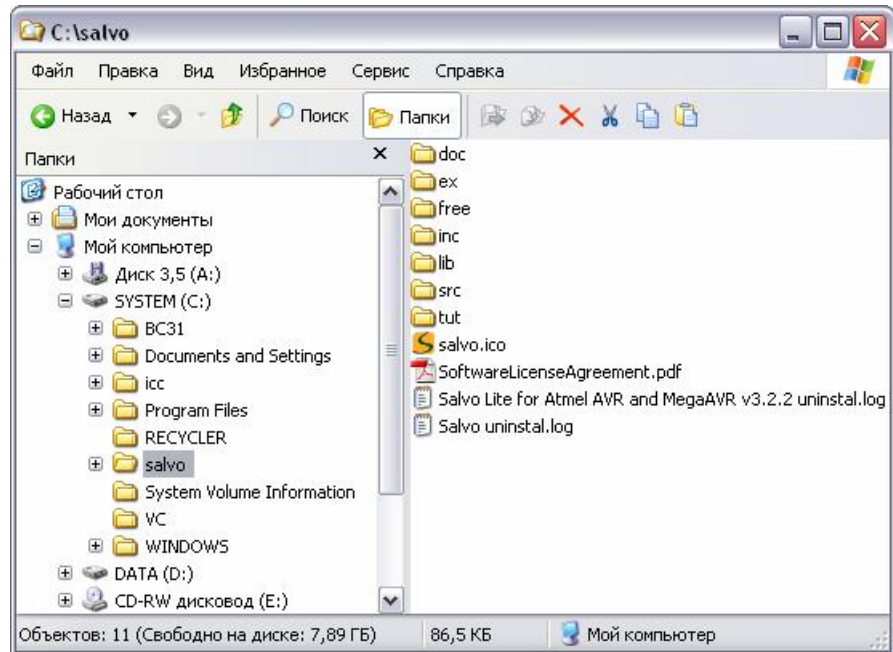


Рисунок 23: Содержание типичной директории Salvo

Программа установки также добавляет папку Pumpkin в Меню Старт:

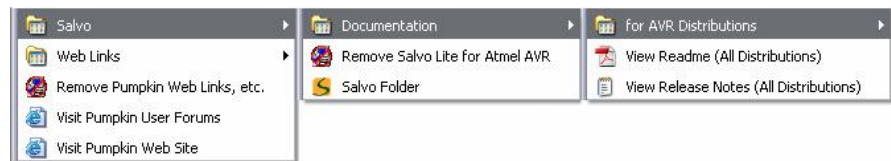


Рисунок 24: Папка меню старт

Также создаются ярлыки для папки Salvo, документации Salvo, деинсталлятора Salvo и др.

Деинсталляция Salvo

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

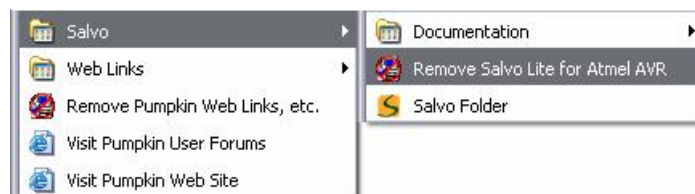


Рисунок 25: Запуск деинсталлятора

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

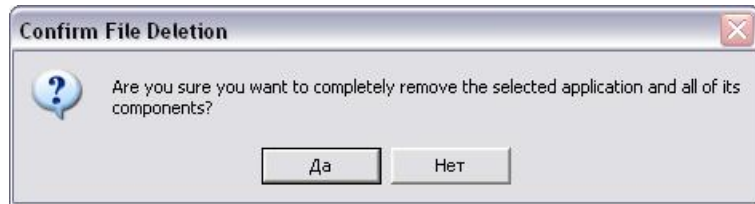


Рисунок 26: Экран подтверждения удаления файлов

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



Рисунок 27: Экран завершения деинсталляции

Нажмите кнопку **OK** для завершения 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.

Установка с несколькими дистрибутивами Salvo

Установщик Salvo поддерживает несколько дистрибутивов различных типов, установленных в один каталог (обычно `c:\salvo`)²². Например, можно иметь Salvo Lite для TI's MSP430 а также Salvo Pro для семейства 8051, установленные вместе в `c:\salvo`.

Поведение установщика

Установщики 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 установщиком.

Установка нескольких дистрибутивов Salvo

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

²² Как в Salvo v3.2.2.

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).

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 `salvo\tut`. Вы можете найти их полезными для лучшего понимания их работы.

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. Исходный код этой программы находится в файле `\salvo\tut\tut\main.c`.

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

int main( void )
{
    Init();

    OSInit();

    for (;;)
        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()`.

Замечание: `OSSched()` находится в цикле `for()` несмотря на отсутствие фигурных скобок.

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

Замечание: Функция `Init()` в `main()` предназначена для инициализации²³ устройства. Она и файл заголовка `main.h` ничего не делают с кодом Salvo, но присутствуют для полноты структуры программы.

OSInit()

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

OSSched()

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

Совет: Для наилучшего использования call-return стека 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.

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. Эта программа расположена в папке `salvo\tut\tu2\main.c`.

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

_OSLabel(TaskA1)
_OSLabel(TaskB1)

void TaskA( void )
{
    for (;;)
        OS_Yield(TaskA1);
}

void TaskB( void )
{
    for (;;)
        OS_Yield(TaskB1);
}
```

²³ Например, выбор осциллятора и настройка портов Ввода-Вывода микроконтроллера.

```
}

int main( void )
{
    Init();

    OSInit();

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

    for (;;)
        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 задачей (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 создает²⁴. 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.

²⁴ 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_".

Совет: Каждый переключатель контекста Salvo требует явной уникальной метки. Простой способ создать метку состоит в использовании Salvo макро OSLabel() с именем метки, которое вы определяете. Затем, используйте эту метку как аргумент в переключателе контекста. Таково назначение меток TaskA1 и TaskB1 из примера выше. TaskA1 – метка первого переключателя контекста в пределах TaskA(). Вы можете выбрать альтернативное соглашение по обозначениям, такое как TaskA_label1, и т.п.

OSCreateTask()

Для создания задачи, вызовите OSCreateTask() со *стартовым адресом задачи*, *указателем tcb* и *приоритетом* как с параметрами. Стартовый адрес – это обычно начало задачи, определенное 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() – краткий²⁵ 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.

²⁵ OSTcbArea[n - 1] является более длинным путем.

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()`.

Возможно, это пока не очевидно, но Листинг 24 также иллюстрирует другую основную концепцию RTOS – это состояния задачи. В Salvo все задачи стартуют 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 `for()`, 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 – какой бы высокий приоритет не имела задача, она будет работать в следующий раз, при 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.

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. Эта программа расположена в `\salvo\tut\tu3\main.c`.

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

_OSLabel(TaskCount1)
_OSLabel(TaskShow1)

unsigned int counter;

void TaskCount( void )
{
    for (;;)
    {
        counter++;

        OS_Yield(TaskCount1);
    }
}

void TaskShow( void )
```

```
{
    InitPORT()

    for(;;)
    {
        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;

    for (;;)
        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 инициализируется²⁶ 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-битовый порт (PORT) с 8 подключенными к нему светодиодами LED. Это продолжается неопределенно долго.

Подробности

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

В этом примере TaskCount() запускается первой²⁷. Вначале инкрементируется счетчик, затем контекст переключится через OS_Yield(). Эта макрокоманда отмечает, где находится выполнение программы в TaskShow() (в конце for() цикла), и затем возвратит выполнение программы диспетчеру. Диспетчер тогда проверит TaskCount(), чтобы увидеть готова ли она еще продолжать выполнение. В данном случае, так как мы не делали никаких изменений состояния, она заработает снова, когда станет самой готовой задачей.

²⁶ Строго говоря, эта инициализация необязательна, т.к. ANSI компилятор установит счетчик в 0 перед main().

²⁷ Потому-что была запущена первой и обе задачи имеют равный приоритет.

Диспетчер завершает свою работу и вызывается снова, потому что он находится в бесконечном `for()` 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 инициализируется²⁸. Затем `TaskShow()` входит в бесконечный цикл в первый раз, `PORT` инициализирован в `0x00` (счетчик теперь равен `0x0001`), и опять `OS_Yield()` возвращает выполнение программы диспетчеру после отметки, где "возвращаться" в `TaskShow()`. `TaskShow()` также остается готовой продолжать работу.

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 `for()` 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 `for()` 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, это создает 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. Все же мы знаем, что задействованы две отдельных задачи, так что мы называем эту программу многозадачное приложение. Это пока не очень мощное приложение, и его функции могли бы быть дублированы 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.

Использование событий для повышения эффективности

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

Листинг 26 показан ниже. Мы использовали некоторые директивы `#define` препроцессора для улучшения читабельности. Эта программа расположена в `\salvo\tut\tu4\main.c`.

```
#include "main.h"
```

²⁸ В этом примере, каждый вывод порта `PORT` может конфигурироваться как вход или как выход. При включении питания, все выходы конфигурированы как входы, следовательно необходимо сконфигурировать их как выходы через `InitPORT()`. `InitPORT()` также инициализирует 8-битовый порт ввода/вывода в `0x00`.

```

#include <salvo.h>

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

_OSLabel(TaskCount1)
_OSLabel(TaskShow1)

unsigned int counter;

void TaskCount( void )
{
    for (;;)
    {
        counter++;

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

        OS_Yield(TaskCount1);
    }
}

void TaskShow( void )
{
    InitPORT();

    for (;;)
    {
        OS_WaitBinSem(BINSEM_UPDATE_PORT_P,
                      OSNO_TIMEOUT, TaskShow1);

        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;

    for (;;)
        OSSched();
}

```

Листинг 26: Многозадачность с использованием событий

В Листинге 26 мы общаемся между двумя задачами, чтобы обновлять порт только тогда, когда обновление 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 `esb` и начальным значением. Двоичный семафор создан без задач, ждущих его. Двоичный семафор должен быть создан прежде, чем он может сигнализировать или ожидаться.

OSSignalBinSem()

Двоичный семафор передает сигнал через `OSSignalBinSem()`. Если ни одна из задач не ждет 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, пока он не передаст сигнал, а задача с наивысшим приоритетом, ожидает двоичный семафор. Поэтому больше чем одна задача может ждать специфическое событие.

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 при вызове `OS_WaitBinSem()` в зависимости от значения двоичного семафора может произойти переключение контекста.

Совет: Вы должны всегда определять таймаут²⁹ 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.

²⁹ Параметр таймута 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.

Подробности

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

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

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 готового на ожидание. Теперь есть только одна готовая задача – TaskCount(), и диспетчер перезапускает ее многократно.

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% времени³⁰, она работает только в редких случаях, когда изменяется старший байт счетчика. Остальную часть времени диспетчер управляет TaskCount().

Должно быть, очевидно, что вызовы OS_WaitBinSem() и OSSignalBinSem() несколько повышают функциональные возможности. 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.

³⁰ Измерено на Test System A.

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 нежелательного вмешательства из RTOS.

Задержка задач

Один элемент, отсутствующий в предыдущем примере – понятие работы в реальном времени. 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.

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 (LED) на младшем бите PORT с частотой 1Hz, создав и запустив задачу, которая задерживает себя на 500ms после переключения бита порта, делая это повторно. Эта программа расположена в \salvo\tut\tu5\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        10 /* task priorities */
#define PRIO_SHOW         10 /* ' ' ' ' */
#define PRIO_BLINK        2  /* ' ' ' ' */
#define BINSEM_UPDATE_PORT_P OSECBP(1) /* binSem
#1 */

unsigned int counter;

_OSLabel(TaskCount1)
_OSLabel(TaskShow1)
_OSLabel(TaskBlink1)

void TaskCount( void )
{
    for (;;)
    {
        counter++;
        if ( !(counter & 0x01FF) )
            OSSignalBinSem(BINSEM_UPDATE_PORT_P);
    }
}
```

```

        OS_Yield(TaskCount1);
    }
}

void TaskShow( void )
{
    for (;;)
    {
        OS_WaitBinSem(BINSEM_UPDATE_PORT_P,
                      OSNO_TIMEOUT, TaskShow1);

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

void TaskBlink( void )
{
    InitPORT();

    for (;;)
    {
        PORT ^= 0x01;

        OS_Delay(50, TaskBlink1);
    }
}

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;

    OSEi();

    for (;;)
        OSSched();
}

```

Листинг 27: Многозадачность с задержкой

Для этого дополнительно потребуются прерывания для вызова OSTimer() с желаемой частотой системных часов 100Hz. Код, выполняющий это, расположен в \salvo\tut\tul\sysa\isr.c³¹.

```

#include <salvo.h>

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

void interrupt IntVector( void )

```

³¹ IntVector() также используется в ту6 далее. IntVector() (и следовательно содержание isr.c) специфично для контроллера и компилятора.

```
{
    if ( TOIE && 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 вышеупомянутом 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 OS Ei() должен предшествовать старту многозадачности. Так как задержки определены в единицах системных часов, задача мигания задерживается на 50*10ms, или на 500ms.

OSTimer()

Чтобы использовать сервис задержки 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 вызывая его с периодической 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 у `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 контролирует задержанные задачи один раз за вызов `OSTimer()`, и не зависит от числа задержанных задач³².

Это показывает, что системный таймер полезен по разным 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.

Рассмотрите код в Листинге 29, чтобы увидеть, что случается, когда мы понижаем приоритет всегда работающей задачи `TaskCount()`, и имеем `TaskShow()`, осуществляющую все записи в `PORT`. Эта программа расположена в `\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;
```

³² За исключением случая, когда одна или более задержек задач истекают одновременно

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

_OSLabel(TaskCount1)
_OSLabel(TaskShow1)
_OSLabel(TaskBlink1)
_OSLabel(TaskBlink2)

void TaskCount( void )
{
    counter = 0;

    for (;;)
    {
        counter++;

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

        OS_Yield(TaskCount1);
    }
}

void TaskShow( void )
{
    OStypeMsgP msgP;

    InitPORT();

    for (;;)
    {
        OS_WaitMsg(MSG_UPDATE_PORT_P,
                  &msgP,
                  OSNO_TIMEOUT,
                  TaskShow1);

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

void TaskBlink( void )
{
    OStypeErr err;

    for (;;)
    {
        OS_Delay(50, TaskBlink1);

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

```
        if ( err == OSERR_EVENT_FULL )
        {
            OS_SetPrio(PRIO_SHOW+1, TaskBlink2);
            OSSignalMsg(MSG_UPDATE_PORT_P,
                (OStypeMsgP) &CODE_B);
            OS_SetPrio(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);

    OSEi();

    for (;;)
        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 объединяли ваш код используя predefined типы, например – OStypeMsgP для указателей сообщения. Вы должны использовать predefined типы Salvo везде, где возможно. См. *Главу 7 • Справочник* для получения дополнительной информации о predefined типах Salvo.

OSSignalMsg()

Чтобы сигнализировать о сообщении `OSSignalMsg()`, вы должны определить и `esb` 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 типу его разыменование для отправки и получения в сообщении соответствующей информации.

В Листинге 29, возможен следующий сценарий: Немедленно после того, как `TaskCount()` сигнализирует о сообщении, задержка `TaskBlink()` истекает, и `TaskBlink()` становится готовой. Так как `TaskBlink()` имеет самый высокий приоритет, сообщение будет все еще присутствовать, когда `TaskBlink()` сигнализирует сообщением. Поэтому `OSSignalMsg()` возвратит ошибку. Светодиод LED на выводе `PORT` будет не в состоянии 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 разумным³³.

³³ Альтернативное решение этой проблемы состояло бы в том, чтобы использовать очередь сообщений с местом для двух сообщений.

В более сложном приложении, например электроника автомобиля, можно представить `TaskShow()` замененной 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()` ждала бы этого сообщения. Каждое сообщение указало 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 отправки. 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.

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, требованиях времени вашего приложения и событиях, управляющих его поведением. Тогда соедините 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 • Установка*, 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 построить выполняемую 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 выполняемой 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 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 непосредственно³⁴, и вы не должны излишне дублировать никаких файлов 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 версии библиотек Salvo для разных проектов.

³⁴ Исходные файлы Salvo устанавливаются как файлы только-для-чтения.

Замечание: Полные проекты для всех учебных программ могут быть найдены в \salvo\tut\tu1-tu6.

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:\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.

Конфигурирование вашего компилятора

Для успешной компиляции вашего Salvo приложения, 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 \salvo\inc

- `salvocfg.h` расположенный в каталоге вашего текущего проекта

Вы можете также нуждаться в определении каталога исходных файлов Salvo (`\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 – для опытных пользователей Salvo Pro.

Совет: Вы можете найти *Рисунок 28: Построение Приложения Salvo с Библиотеками* и *Рисунок 29: Построение Приложения Salvo с Исходными Кодами* 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 путем 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 `\salvo\tut\tu1-tu6`. Дополнительную информацию см. в *Приложении С • 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-битовых 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 напечатаете 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_P35  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.

Вы не обязаны использовать все события, для которых вы выделяете память, и при этом вы не должны использовать их соответствующие указатели esb (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, которые из свойств Salvo вы планируете использовать в вашем приложении. Многие из особенностей 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.

³⁵ Суффикс `p` должен напоминать вам, что объект – указатель на что-либо.

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 tut\tu6

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
```

Затем, \salvo\tut\tu6 использует сообщения как средство межзадачных коммуникаций. Код сообщения запрещен по умолчанию, и мы разрешаем его следующим образом:

```
#define OSENABLE_MESSAGES      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: Директива #define в salvocfg.h может 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 \salvo\tut\tu6. Это показано (с удаленными комментариями Си³⁶) в Листинге 30.

```
#define OSBYTES_OF_DELAY      1
#define OSENABLE_MESSAGES      TRUE
```

³⁶ И без дополнительных опций конфигурации, которые соответствуют таковым в связанной свободно распространяемой библиотеке.


```
#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 компоуя 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, содержащиеся в компоуемых 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.

Размер каждого скомпилированного объектного модуля сильно зависит от опций конфигурации, которые вы выбираете. Кроме того, вы можете разумно выбрать, 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 `\salvo\tut\tu1-tu6`. См. *Приложение С • Описания Файлов и Программ* для получения дополнительной информации.

