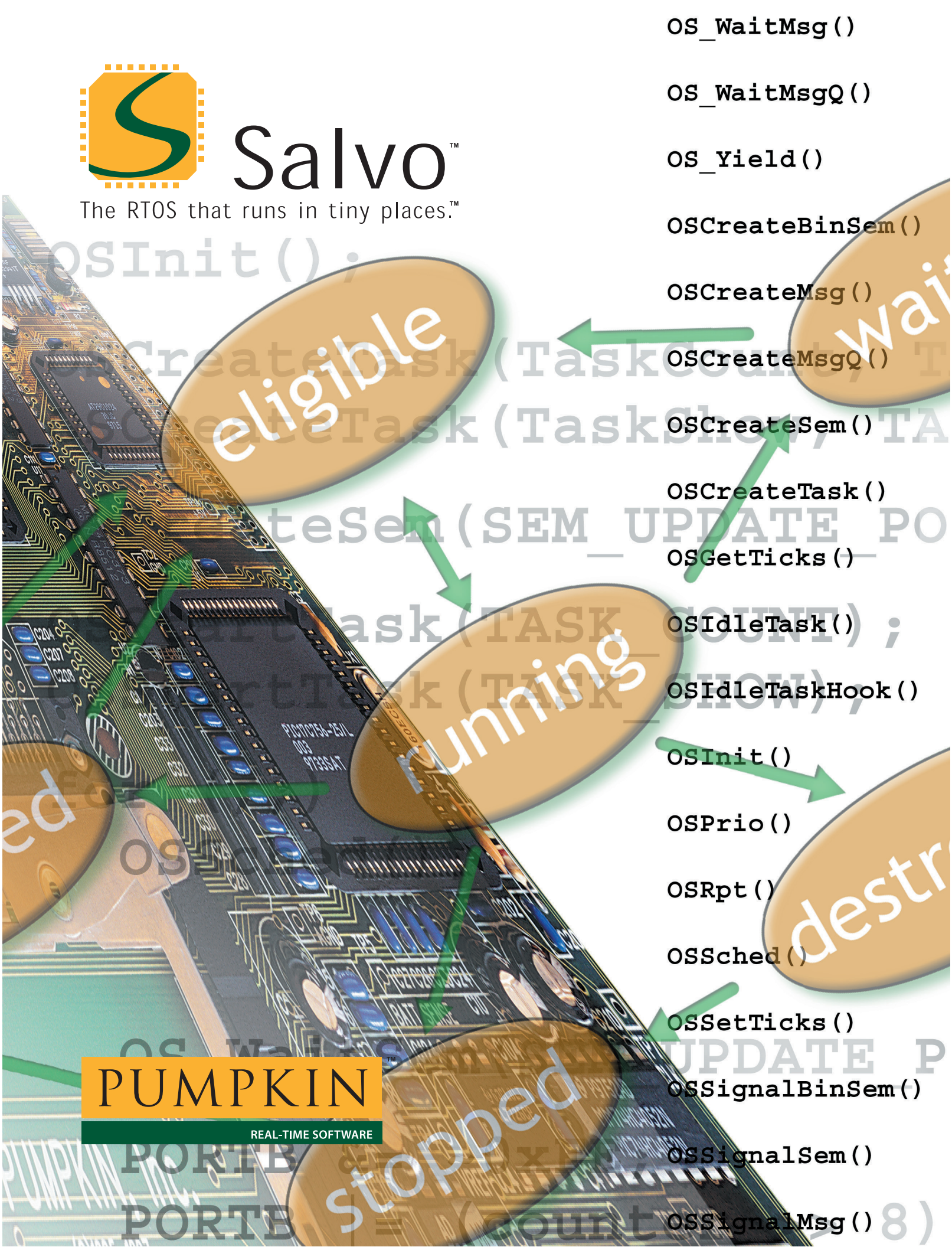
Salvo™
The RTOS that runs in tiny places.™
OS_WaitMsg()
OS_WaitMsgQ()
OS_Yield()
OSCreateBinSem()
OSCreateMsg()
OSCreateMsgQ()
OSCreateSem()
OSCreateTask()
OSGetTicks()
OSIdleTask()
OSIdleTaskHook()
OSInit()
OSPrio()
OSRpt()
OSSched()
OSSetTicks()
OSSignalBinSem()
OSSignalSem()
OSSignalMsg()
eligible
running
stopped
PUMPKIN
REAL-TIME SOFTWARE

(inside front cover)

Salvo™

The RTOS that runs in tiny places.™

User Manual

version 4.2.2

for all distributions

PUMPKIN™

REAL-TIME SOFTWARE

Quick Start Guide

Thanks for purchasing Salvo, The RTOS that runs in tiny places.™ Pumpkin is dedicated to providing powerful, efficient and low-cost embedded programming solutions. We hope you'll like what we've made for you.

If this is the first time you've encountered Salvo, please review *Chapter 1 • Introduction* to get a flavor for what Salvo is, what it can do, and what other tools you'll need to use it successfully. See *Chapter 2 • RTOS Fundamentals* if you haven't used an RTOS before. Then try the steps below in the order listed.

Note You don't need to purchase Salvo to run the demo programs, try the tutorial or use the freeware libraries to build your own multitasking Salvo application – they're all part of Salvo Lite, the freeware version of Salvo.

Running on Your Hardware

If you have a compatible target environment, you can run one of the standalone Salvo example applications contained in `Pumpkin\Salvo\Example` on your own hardware. Open the demo's project, build it, download or program it into your hardware, and let it run. Most demo programs provide real-time feedback. If it's a Salvo Lite demo and uses commonly available hardware, you can even build your own application by modifying the source and rebuilding it.

See *Appendix C • File and Program Descriptions* for more information on the demo programs.

Trying the Tutorial

Chapter 4 • Tutorial builds a multitasking, event-driven Salvo application in six easy steps. The tutorial will familiarize you with Salvo's terminology, user services, and the process of building a working application. A set of tutorial projects is included with every Salvo distribution for embedded targets, enabling you to build each tutorial application by simply loading and building the project in the appropriate development environment.

Salvo Lite

A compiler that's certified for use with Salvo is all you need to use Salvo Lite, the freeware version of Salvo. You can write your own, small multitasking application with calls to Salvo services and link it to the freeware libraries. See *Chapter 4 • Tutorial* and the *Salvo Application Note* for your compiler and/or target for more information.

Salvo LE

Salvo LE adds the standard Salvo libraries to Salvo Lite. This means that the numbers of tasks, events, etc. in your application are limited only by the available RAM.

Salvo Pro

With Salvo Pro, you'll have full access to all its source code, standard libraries, test programs and priority support. If you haven't done so already, try the tutorial in *Chapter 4 • Tutorial* as a first step towards creating your own application. Then use the configuration options in *Chapter 5 • Configuration* and the services outlined in *Chapter 7 • Reference*, along with their examples, to fine-tune Salvo to your application's requirements. If you run into problems or have questions, you'll find lots of useful information in *Chapter 6 • Frequently Asked Questions (FAQ)* and *Chapter 11 • Tips, Tricks and Troubleshooting*.

Getting Help

Some of the best resources for new and experienced Salvo users are the Salvo User Forums, hosted on Pumpkin's web site, http://www.pumpkininc.com/. Check there for up-to-date information on the latest Salvo releases.

Contact Information & Technical Support

Contacting Pumpkin

Pumpkin's mailing address and phone and fax numbers are:

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

Time Zone: GMT–0800 (Pacific Standard Time)

Connecting to Pumpkin's Web Site

Use your web browser to access the Pumpkin web site at

- http://www.pumpkininc.com/

Information available on the web site includes

• Latest News
• Software Downloads & Upgrades
• *User Manuals*
• *Compiler Reference Manuals*
• *Application Notes*
• *Assembly Guides*
• *Release Notes*
• User Forums

Salvo User Forums

Pumpkin maintains User Forums for Salvo at Pumpkin's web site. The forums contain a wealth of practical information on using Salvo, and is visited by Salvo users as well as Pumpkin technical support.

How to Contact Pumpkin for Support

Pumpkin provides online Salvo support via the Salvo Users Forums on the Pumpkin World Wide Web (WWW) site. Files and information are available to all Salvo users via the web site. To access the site, you'll need web access and a browser (e.g. Netscape, Opera, Internet Explorer).

Internet (WWW)

The Salvo User Forums are located at:

- http://www.pumpkininc.com

and are the *preferred* method for you to post your pre-sales, general or technical support questions.

Email

Normally, we ask that you post your technical support questions to the Salvo User Forums on our website. We monitor the forums and answer technical support questions on-line.

In an emergency, you can reach technical support via email:

- support@pumpkininc.com

We will make every effort to respond to your email requests for technical support within 1 working day. Please be sure to provide as much information about your problem as possible.

Mail, Phone & Fax

If you were unable to find an answer to your question in this manual, check the Pumpkin website and the Salvo user Forums (see below) for additional information that may have been recently

posted. If you are still unable to resolve your questions, please contact us directly at the numbers above.

What To Provide when Requesting Support

Registered users requesting Salvo technical support should supply:

- The Salvo version number
- The compiler name and version number
- The user's source code snippet(s) in question
- The user's `salvocfg.h` file
- All other relevant files, details, etc.

Small code sections can be posted directly to the Salvo User Forums – see the on-line posting FAQ on how to use the UBB code tags (`[code]` and `[/code]`) to preserve the code's formatting and make it more legible.

If the need arises to send larger code sections, or even a complete, buildable project, please compress the files and email them directly to Salvo Technical support (see below). Please be sure to provide all necessary files to enable Technical Support to build your Salvo application locally in an attempt to solve your problem. Keep in mind that without the appropriate target system hardware, support in these cases is generally limited to non-runtime problem solving. Technical Support will keep all user code in strictest confidence.

Salvo User Manual

Pumpkin, Inc.
750 Naples Street
San Francisco, CA 94112 USA

tel: 415-584-6360
fax: 415-585-7948
web: www.pumpkininc.com
email: sales@pumpkininc.com

Disclaimer

Pumpkin, Incorporated ("Pumpkin") has taken every precaution to provide complete and accurate information in this document. However, due to continuous efforts being made to improve and update the product(s), Pumpkin and its Licensor(s) shall not be liable for any technical or editorial errors or omissions contained in this document, or for any damage, direct or indirect, from discrepancies between the document and the product(s) it describes.

The information is provided on an as-is basis, is subject to change without notice and does not represent a commitment on the part of Pumpkin, Incorporated or its Licensor(s).

Trademarks

The Pumpkin name and logo, the Salvo name and logo, the CubeSat Kit name and logo, "The RTOS that runs in tiny places." and "Don’t leave Earth without It." are trademarks of Pumpkin, Incorporated.

The absence of a product or service name or logo from this list does not constitute a waiver of Pumpkin's trademark or other intellectual property rights concerning that name or logo.

All other products and company names mentioned may be trademarks of their respective owners. All words and terms mentioned that are known to be trademarks or service marks have been appropriately capitalized. Pumpkin, Incorporated cannot attest to the accuracy of this information. Use of a term should not be regarded as affecting the validity of any trademark or service mark.

This list may be partial.

Patent Information

The software described in this document is manufactured under one or more of the following U.S. patents:

Patents Pending

Life Support Policy

Pumpkin, Incorporated's products are not authorized for use as critical components in life support devices or systems without the express written approval of the president of Pumpkin, Incorporated. As used herein:

- 1) Life support devices or systems are devices or systems which, (a) are intended for surgical implant into the body, or (b) support or sustain life, and whose failure to perform, when

properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in significant injury to the user.

-
- 2) A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.

Refund Policy and Limited Warranty on Media

Pumpkin wants you to be happy with your Salvo purchase. That's why Pumpkin invites you to test drive Salvo before you buy. You can download and evaluate the fully functional Salvo freeware version Salvo Lite from the Salvo web site. If you have questions while you are using Salvo Lite, please don't hesitate to consult the Salvo User Forums, contact our support staff at support@pumpkininc.com, or contact Pumpkin directly.

Because of this free evaluation practice, and because the purchased version contains the complete source code for Salvo, Pumpkin does not offer refunds on software purchases.

Pumpkin will replace defective distribution media or manuals at no charge, provided you return the item to be replaced with proof of purchase to Pumpkin during the 90-day period after purchase. More details can be found in Section 11 Limited Warranty on Media of the Pumpkin Salvo License.

Documentation Creation Notes

This documentation was produced using Microsoft Word, Creative Softworx Capture Professional, CorelDRAW!, Adobe Photoshop, Adobe Illustrator and Adobe Acrobat.

Document name:	SalvoUserManual.doc (a Master document)
Template used:	User's Manual - Template (TT).dot
Last saved on:	16:06, Thursday, June 3, 2010
Total pages:	528
Total words:	97163

Credits

Author:	Andrew E. Kalman
Artwork:	Laura Macey, Elizabeth Peartree, Andrew E. Kalman
C-language Advice:	Russell K. Kadota, Clyde Smith-Stubbs, Dan Henry
Compiler Advice:	Matthew Luckman, Jeffrey O'Keefe, Paul Curtis, Richard Man

Pumpkin Salvo Software License Agreement v1.2

Please Read this Carefully and Completely Before Using this Software.

(Note: The Terms used herein are defined below in Section 1 Definitions)

Grant of License

This License Agreement is a legal agreement between You and Pumpkin, which owns the Software accompanied by this License or identified above or on the Product Identification Card accompanying this License or on the Product Identification Label attached to the product package. By clicking the Yes (i.e. Accept) button or by installing, copying, or otherwise using the Software or any Software Updates You agree to be bound by the terms of this License. If You do not agree to the terms of this License, Pumpkin is unwilling to license the Software to You, and You must not install, copy, or use the Software, including all Updates that You received as part of the Software. In such event, You should click the No (i.e. Decline) button and promptly contact Pumpkin for instructions on returning the entire unused Software and any accompanying product(s) for a refund. By installing, copying, or otherwise using an Update, You agree to be bound by the additional License terms that accompany such Update. If You do not agree to the terms of the additional License terms that accompany the Update, disregard the Update and the additional License terms that accompany the Update. In this event, Customer's rights to use the Software shall continue to be governed by the then-existing License.

1 Definitions

"License" means this document, a license agreement.

"You" means an individual or a legal entity exercising rights under, and complying with all of the terms of, this License or a future version of this License. For legal entities, "You" includes any entity that controls, is controlled by, or is under common control with You. For purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares or beneficial ownership of such entity.

"Pumpkin" means Pumpkin, Incorporated and its Supplier(s).

"Original Code" means Source Code of computer software that is described in the Source Code Notice (below) as Original Code, and which, at the time of its release under this License is not already Covered Code governed by this License.

"Source Code" means the preferred form of the Covered Code for making modifications to it, including all modules it contains, plus any associated interface definition files, scripts used to control compilation and installation of an Executable, or a list of source code differential comparisons against either the Original Code or another well known, available Covered Code of Your choice.

"Covered Code" means the Original Code or Modifications or the combination of the Original Code and Modifications, in each case including portions thereof.

"Executable" means Covered Code in any form other than Source Code.

"Application" means computer software or firmware that is created in combination with Covered Code.

"Software" means the proprietary computer software system owned by Pumpkin that includes but is not limited to software components (including, but not limited to Covered Code), product documentation and associated media, sample files, extension files, tools, utilities and miscellaneous technical information, in whole or in part.

"Update" means any Software Update.

"Larger Work" means a work that combines Covered Code or portions thereof with code not governed by the terms of this License.

"Modifications" means any addition to or deletion from the substance or structure of either the Original Code or any previous Modifications. When Covered Code is released as a series of files, a Modification is (i) any addition to or deletion from the contents of a file containing Original Code or previous Modifications, or (ii) any new file that contains any part of the Original Code or Previous Modifications.

"Support" means customer support.

"Prerelease Code" means portions of the Software identified as prerelease code or "beta" versions.

2 Copyright

The Software, including all applicable rights to patents, copyrights, trademarks and trade secrets, is the sole and exclusive property of Pumpkin, Incorporated and its Licensor(s) and is provided for Your exclusive use for the purposes of this License. The Software is protected by United States copyright laws and international treaty provisions. Therefore, You must treat the Software like any other copyrighted material, except that You may either (i) make one copy of the Software in machine readable form solely for backup or archival purposes, or (ii) transfer the Software to a hard disk, provided You keep the original solely for backup and archival purposes. Additionally, only so long as the Software is installed only on the permanent memory of a single computer and that single computer is used by one user for at least 80% of the time the computer is in use, that same user may also make a copy of the Software to use on a portable or home computer which is primarily used by such user. As an express condition of this License, You must reproduce and include on each copy any copyright notice or other proprietary notice that is on the original copy of the Software supplied by Pumpkin. You may not copy the printed materials accompanying the Software.

3 Source Code License

3.1 The Software is licensed, not sold, to You by Pumpkin for use only under the terms of this License, and Pumpkin reserves any rights not expressly granted to You. Except where explicitly identified as such, the Software is neither "shareware" nor "freeware" nor "communityware." The Software contains intellectual property in the form of Source Code, algorithms and other manifestations. You own the media on which the Software is recorded or fixed, but Pumpkin, Incorporated and its Licensor(s) retains ownership of the Software, related documentation and fonts.

3.2 Pumpkin grants You the use of the Software only if You have registered the Software with Pumpkin by returning the registration card or by other means specified by Pumpkin.

3.3 Pumpkin grants You a non-exclusive, worldwide License, subject to third-party intellectual property claims, (i) to use and modify ("Utilize") the Software (or portions thereof) with or without Modifications, or as part of a Larger Work, on a single computer for the purpose of creating, modifying, running, debugging and testing Your own Application and any of its updates, enhancements and successors, and (ii) under patents now or hereafter owned or controlled by Pumpkin, to Utilize the Software (or portions thereof), but solely to the extent that any such patent is reasonably necessary to enable You to Utilize the Software (or portions thereof) and not to any greater extent that may be necessary to Utilize further Modifications or combinations. To use ("Use") the Software means that the Software is either loaded in the temporary memory (i.e. RAM) of a computer or installed on the permanent memory of a computer (i.e. hard disk, etc.). You may Use the Software on a network, provided that a licensed copy of the software has been acquired for each person permitted to access the Software through the network. You may also Use the Software in object form only (i.e. as an Executable) on a single, different computer or computing device (e.g. target microcontroller or microprocessor, demonstration or evaluation board, in-circuit emulator, test system, prototype, etc.).

3.4 Any supplemental software code or other materials provided to You as part of Pumpkin's Support shall be considered part of the Software and subject to the terms and conditions of this License. With respect to technical information You provide to Pumpkin as part of the Support, Pumpkin may use such information for its business purposes, including product support and development. Pumpkin will not utilize such technical information in a form that personally identifies You without Your permission.

3.5 The Software shall be deemed accepted by You upon payment of the Software by You and shall not be granted a refund of any license fees for the Software, except for Your rights defined in this License.

4 Software Distribution Obligations

4.1 You may not under any circumstances release or distribute the Source Code, with or without Modifications, or as part of a Larger Work, without Pumpkin's express written permission.

4.2 You may distribute the Software in Executable form only and as part of a Larger Work only (i.e. in conjunction with and as part of Your Application. Additionally, You must (i) not permit the further redistribution of the Software in any form by Your customers, (ii) include a valid copyright notice in Your application (where possible - if it is not possible to put such a notice in Your Application due to its structure, then You must include such a notice in a location (such as a relevant directory file) where a user would be likely to look for such a notice), (iii) include the existing copyright notice(s) in all Pumpkin Software used in Your Application, (iv) agree to indemnify, hold harmless and defend Pumpkin from and against any and all claims and lawsuits, including attorney's fees, that arise or result from the use or distribution of Your Application, (v) otherwise comply with the terms of this License, and (vi) agree that Pumpkin reserves all rights not expressly granted.

4.3 You may freely distribute the demonstration programs (identified as "Demo") that are part of the Software as long as they are accompanied by this License.

4.4 The freeware version (consisting of pre-compiled libraries, a limited number of source code files, and various other files and documentation) and identified as "Freeware" is governed by this license, with the following exceptions: The sole exception shall be for a Larger Work created exclusively with the freeware libraries that are part of the Software; in this case Pumpkin automatically grants You the right to distribute Your Application freely.

4.5 You may not under any circumstances, other than those explicitly mentioned in Sections 4.2, 4.3 and 4.4 above, release or distribute the Covered Code, with or without Modifications, or as part of a Larger Work, without Pumpkin's express written permission.

5 Other Restrictions

5.1 You may not permit other individuals to use the Software except under the terms of this License.

5.2 You may not rent, lease, grant a security interest in, loan or sublicense the Software; nor may You create derivative works based upon the Software in whole or in part.

5.3 You may not translate, decompile, reverse engineer, disassemble (except and solely to the extent an applicable statute expressly and specifically prohibits such restrictions), or otherwise attempt to create a human-readable version of any parts of the Software supplied exclusively in binary form.

5.4 If the Software was licensed to You for academic use, You may not use the software for commercial product development.

5.5 You may not remove any designation mark from any supplied material that identifies such material as belonging to or developed by Pumpkin.

5.6 You may permanently transfer all of Your rights under this License, provided You retain no copies, You transfer all of the Software (including all component parts, the media and printed materials, any upgrades, and this License), You provide Pumpkin notice of Your name, company, and address and the name, company, and address of the person to whom You are transferring the rights granted herein, and the recipient agrees to the terms of this License and pays to Pumpkin a transfer fee in an amount to be determined by Pumpkin and in effect at the time in question. If the Software is an upgrade, any transfer must include all prior versions of the Software. If the Software is received as part of a subscription, any transfer must include all prior deliverables of Software and all other subscription deliverables. Upon such transfer, Your License under this Agreement is automatically terminated.

5.7 You may use or transfer the Updates to the Software only in conjunction with Your then-existing Software. The Software and all Updates are licensed as a single product and the Updates may not be separated from the Software for use at any time.

6 Termination

This License is effective until terminated. This License will terminate immediately without notice from Pumpkin or judicial resolution if You fail to comply with any provision of this License, and You may terminate this License at any time. Upon such termination You must destroy the Software, all accompanying written materials and all copies thereof. Provisions which, by their nature, must remain in effect beyond the termination of this License shall survive.

7 Multiple Media

Even if this Pumpkin product includes the Software on more than one medium (e.g., on both a CD-ROM and on magnetic disk(s); or on both 3.5 inch disk(s) and 5.25 inch disk(s)), You are only licensed to use one copy of the Software as described in Section 2.3. The restrictions contained herein apply equally to hybrid media that may contain multiple versions of the Software for use on different operating systems. Regardless of the type of media You receive, You may only use the portion appropriate for Your single user computer / workstation. You may not use the Software stored on the other medium on another computer or common storage device, nor may You rent, lease, loan or transfer it to another user except as part of a transfer pursuant to Section 5.7.

8 Prerelease Code

Prerelease Code may not be at the level of performance and compatibility of the final, generally available product offering, and may not operate correctly and may be substantially modified prior to first commercial shipment. Pumpkin is not obligated to make this or any later version of the Prerelease Code commercially available. The grant of license to use Prerelease Code expires upon availability of a commercial release of the Prerelease Code from Pumpkin.

9 Export Law Assurances

You may not use or otherwise export or re-export the Software except as authorized by United States law and the laws of the jurisdiction in which the Software was obtained. In particular, but without limitation, the Software may not be exported or re-exported to (i) into (or to a national or resident of) any U.S. embargoed country or (ii) to anyone on the U.S. Treasury Department's list of Specially Designated Nations or the U.S. Department of Commerce's Table of Denial Orders. By using the Software You represent and warrant that You are not located in, under control of, or a national or resident of any such country or on any such list.

10 U.S. Government End Users

If You are acquiring the Software and fonts on behalf of any unit or agency of the United States Government, the following provisions apply. The Government agrees that the Software and fonts shall be classified as "commercial computer software" and "commercial computer software documentation" as such terms are defined in the applicable provisions of the Federal Acquisition Regulation ("FAR") and supplements thereto, including the Department of Defense ("DoD") FAR Supplement ("DFARS"). If the Software and fonts are supplied for use by DoD, it is delivered subject to the terms of this Agreement and either (i) in accordance with DFARS 227.7202-1(a) and 227.7202-3(a), or (ii) with restricted rights in accordance with DFARS 252.227-7013(c)(1)(ii) (OCT 1988), as applicable. If the Software and fonts are supplied for use by any other Federal agency, it is restricted computer software delivered subject to the terms of this Agreement and (i) FAR 12.212(a); (ii) FAR 52.227-19; or (iii) FAR 52.227-14(ALT III), as applicable.

11 Limited Warranty on Media

Pumpkin warrants for a period of ninety (90) days from Your date of purchase (as evidenced by a copy of Your receipt) that the media provided by Pumpkin, if any, on which the Software is recorded will be free from defects in materials and workmanship under normal use. Pumpkin will have no responsibility to replace media damaged by accident, abuse or misapplication. PUMPKIN'S ENTIRE LIABILITY AND YOUR SOLE AND EXCLUSIVE REMEDY WILL BE, AT PUMPKIN'S OPTION, REPLACEMENT OF THE MEDIA, REFUND OF THE PURCHASE PRICE OR REPAIR OR REPLACEMENT OF THE SOFTWARE. ANY IMPLIED WARRANTIES ON THE MEDIA, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF DELIVERY. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS, AND YOU MAY ALSO HAVE OTHER RIGHTS THAT VARY BY JURISDICTION.

12 Disclaimer of Warranty

THIS LIMITED WARRANTY IS THE ONLY WARRANTY PROVIDED BY PUMPKIN. PUMPKIN EXPRESSLY DISCLAIMS ALL OTHER WARRANTIES AND/OR CONDITIONS, ORAL OR WRITTEN, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE WITH REGARD TO THE SOFTWARE AND ACCOMPANYING WRITTEN MATERIALS, AND NONINFRINGEMENT. PUMPKIN DOES NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE SOFTWARE WILL MEET YOUR REQUIREMENTS, OR THAT THE OPERATION OF THE SOFTWARE WILL BE UNINTERRUPTED OR ERROR-FREE, OR THAT DEFECTS IN THE SOFTWARE WILL BE CORRECTED. FURTHERMORE, PUMPKIN DOES NOT WARRANT OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE SOFTWARE OR RELATED DOCUMENTATION IN TERMS OF THEIR CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE. AS A RESULT, THE SOFTWARE IS LICENSED "AS-IS", AND YOU THE LICENSEE EXPRESSLY ASSUME ALL LIABILITIES AND RISKS, FOR USE OR OPERATION OF ANY APPLICATION PROGRAMS YOU MAY CREATE WITH THE SOFTWARE, INCLUDING WITHOUT LIMITATION, APPLICATIONS DESIGNED OR INTENDED FOR MISSION CRITICAL APPLICATIONS AND HIGH-RISK ACTIVITIES, SUCH AS THE OPERATION OF NUCLEAR FACILITIES, PACEMAKERS, DIRECT LIFE SUPPORT MACHINES, WEAPONRY, AIR TRAFFIC CONTROL, AIRCRAFT NAVIGATION OR COMMUNICATIONS SYSTEMS, FACTORY CONTROL SYSTEMS, ETC., IN WHICH THE FAILURE OF THE SOFTWARE COULD LEAD DIRECTLY TO DEATH, PERSONAL INJURY, OR SEVERE PHYSICAL OR ENVIRONMENTAL DAMAGE. NO PUMPKIN DEALER, DIRECTOR, OFFICER, EMPLOYEE OR AGENT IS AUTHORIZED TO MAKE ANY MODIFICATION, EXTENSION, OR ADDITION TO THIS WARRANTY. BECAUSE SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF IMPLIED WARRANTIES, THE ABOVE LIMITATION MAY NOT APPLY TO YOU. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS, AND YOU MAY ALSO HAVE OTHER RIGHTS THAT VARY BY JURISDICTION.

13 Limitation of Liabilities, Remedies and Damages

TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT WILL PUMPKIN, INCORPORATED, OR ANY OF ITS LICENSORS, SUPPLIERS, DIRECTORS, OFFICERS, EMPLOYEES OR AGENTS (COLLECTIVELY "PUMPKIN AND ITS SUPPLIER(S)") BE LIABLE TO YOU FOR ANY CONSEQUENTIAL, INCIDENTAL, INDIRECT OR SPECIAL DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION AND THE LIKE, OR ANY OTHER PECUNIARY LOSS), WHETHER FORESEEABLE OR UNFORESEEABLE, ARISING OUT OF THE USE OF OR INABILITY TO USE THE SOFTWARE OR ACCOMPANYING WRITTEN MATERIALS, REGARDLESS OF THE BASIS OF THE CLAIM AND EVEN IF PUMPKIN AND ITS SUPPLIER(S) HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. THIS LIMITATION WILL NOT APPLY IN CASE OF PERSONAL INJURY ONLY WHERE AND TO THE EXTENT THAT APPLICABLE LAW REQUIRES SUCH LIABILITY. BECAUSE SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES, THE ABOVE LIMITATIONS MAY NOT APPLY TO YOU. IN NO EVENT SHALL PUMPKIN AND ITS SUPPLIER(S)' TOTAL LIABILITY TO YOU FOR ALL

DAMAGES, LOSSES AND CAUSES OF ACTION (WHETHER IN CONTRACT, TORT (INCLUDING NEGLIGENCE), PRODUCT LIABILITY OR OTHERWISE) EXCEED $50.00.

PUMPKIN SHALL BE RELIEVED OF ANY AND ALL OBLIGATIONS WITH RESPECT TO THIS SECTION FOR ANY PORTIONS OF THE SOFTWARE THAT ARE REVISED, CHANGED, MODIFIED, OR MAINTAINED BY ANYONE OTHER THAN PUMPKIN.

14 Complete Agreement, Controlling Law and Severability

This License constitutes the entire agreement between You and Pumpkin with respect to the use of the Software, the related documentation and fonts, and supersedes all prior or contemporaneous understandings or agreements, written or oral, regarding such subject matter. No amendment to or modification of this License will be binding unless in writing and signed by a duly authorized representative of Pumpkin. The acceptance of any purchase order placed by You is expressly made conditional on Your assent to the terms set forth herein, and not those in Your purchase order. This License will be construed under the laws of the State of California, except for that body of law dealing with conflicts of law. If any provision of this License shall be held by a court of competent jurisdiction to be contrary to law, that provision will be enforced to the maximum extent permissible, and the remaining provisions of this License will remain in full force and effect. The application of the United Nations Convention on Contracts for the International Sale of Goods is expressly excluded. Any law or regulation that provides that the language of a contract shall be construed against the drafter shall not apply to this License. In the event of any action to enforce this Agreement, the prevailing party shall be entitled to recover from the other its court costs and reasonable attorneys' fees, including costs and fees on appeal.

15 Additional Terms

Nothing in this License shall be interpreted to prohibit Pumpkin from licensing under terms different from this License any code which Pumpkin otherwise would have a right to License.

This License does not grant You any rights to use the trademarks or logos that are the property of Pumpkin, Inc., even if such marks are included in the Software. You may contact Pumpkin for permission to display the above-mentioned marks.

Pumpkin may publish revised and/or new versions of this License from time to time. Each version will be given a distinguishing version number.

Should You have any questions or comments concerning this License, please do not hesitate to write to Pumpkin, Inc., 750 Naples Street, San Francisco, CA 94112 USA, Attn: Warranty Information. You may also send email to support@pumpkininc.com.

Source Code Notice

The contents of this file are subject to the Pumpkin Salvo License (the "License"). You may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.pumpkininc.com, or from warranty@pumpkininc.com.

Software distributed under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for specific language governing the warranty and the rights and limitations under the License.

Contents

Figures

Listings

Tables

Release Notes

Introduction

What's New

Please refer to the distribution's `salvo-whatsnew.txt` file for more information on what's new in the v4.2.2 release.

Release Notes

Please refer to the general (`salvo-release.txt`) and distribution-specific (`salvo-release-`*`targetname`*`.txt`) release notes for more information on release-related changes and updates in the v4.2.2 release.

Third-Party Tool Versions

Please refer to the distribution-specific (`salvo-release-`*`targetname`*`.txt`) release notes for the version numbers of third-party tools (compilers, linkers, librarians, etc.) in the v4.2.2 release.

Supported Targets and Compilers

As of v4.2.2, Salvo supports a variety of 8-, 16- and 32-bit targets and compilers:

Please refer to the distribution-specific (`salvo-release-`*`targetname`*`.txt`) release notes for the version numbers of third-party tools (compilers, linkers, librarians, etc.) in the v4.2.2 release. If you have a named compiler that is older than the ones listed, you may need to upgrade it to work with Salvo. Contact the compiler vendor for upgrade information.

Preface

Historical Information

Pumpkin, Inc.'s Salvo v1 was an internal release, written in assembly language and targeted specifically for the Microchip PIC17C756 PICmicro® MCU in a proprietary, in-house data acquisition system. This 1998 version provided much of the basic functionality that would later makes its way into the later Salvo releases.

After a market analysis Pumpkin, Inc. decided to expand on Salvo v1's functionality by rewriting it in C. In doing so, opportunities arose for many configuration options and optimizations, to the point where not only was the C version more powerful and flexible than its assembly-language predecessor, but it was completely portable, too.

In 2000, Salvo v2 became the first commercial release of Pumpkin, Inc.'s cooperative priority-based multitasking RTOS. It was targeted at the entire range of Microchip PICmicro® MCUs.

In 2002, Salvo v3 was released. This marked the expansion of the Salvo RTOS into new embedded targets, like the 8-bit 8051 and the 16-bit MSP430.

Salvo 4 was released in 2005. Not only did this release mark the first Salvo support for 32-bit embedded targets, but it also included many of the lessons learned over the previous six years in terms of usability and maximum configurability for high performance. Salvo 4 is the first Salvo release to remove all non-instruction-set hardware dependencies from the core Salvo code. This gives end-users complete flexibility in configuring Salvo for maximum real-time performance.

Typographic Conventions

Various text styles are used throughout this manual to improve legibility. Code examples, code excerpts, path names and file names are shown in a `monospaced font`. New and particularly useful terms, and terms requiring emphasis, are shown *italicized*.

User input (e.g. at the DOS command line) is shown in this manner. Certain terms and sequence numbers are shown in **bold**. Important notes, cautions and warnings have distinct borders around them:

Note Salvo source code uses tab settings of 4, i.e. tabs are equivalent to 4 spaces.

The letters xyz are used to denote one of several possible names, e.g. `OSSignalXyz()` refers to `OSSignalBinSem()`, `OSSignalMsg()`, `OSSignalMsgQ()`, `OSSignalSem()`, etc. Xyz is case-insensitive.

The symbol | is used as a shorthand to denote multiple, similar names, e.g. `sysa|e|f` denotes `sysa` and/or `syse` and/or `sysf`.

DOS and Windows pathnames use '\'. Linux and Unix pathnames use '/'. They are used interchangeably throughout this document.

Standardized Numbering Scheme

Salvo employs a *standardized numbering scheme* for all software releases. The version/revision numbering scheme uses multiple fields[1] as shown below:

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

where

- `distribution` refers to Salvo Lite, tiny, SE, LE or Pro
- `target` refers to the target processor(s) supported in the distribution
- `MAJOR` changes when major features (e.g. array mode) are added.
- `MINOR` changes when minor features (e.g. new user services) are added to or changed.
- `SUBMINOR` changes during alpha and beta testing and when support files (e.g. new *Salvo Application Notes*) are added.
- `PATCH` is present and changes each time a bug fix is applied and/or new documentation is

[1] The final field is present only on patches.

added. PATCH may also be used for release candidates, e.g. rc4.

All MAJOR.MINOR.SUBMINOR versions are released with their own, complete installer. -PATCH may be used on complete installers or on minimal installers or archives that add new or modified files to an existing Salvo code and documentation installation.

Examples include:

• salvo-lite-pic-2.2.0	v2.2 Salvo Lite for PICmicro® MCUs installer, released
• salvo-le-8051-3.1.0-rc3	v3.1.0 Salvo LE for 8051 family installer, release candidate #3
• salvo-pro-msp430-4.1.0	version 4.1.0 Salvo Pro for TI's MSP430 installer, released

Salvo releases are generically referred to by their MAJOR.MINOR numbering, i.e. "the 3.0 release."

The Salvo Coding Mindset

Configurability Is King

Salvo is extremely configurable to meet the requirements of the widest possible target audience of embedded microcontrollers. It also provides you, the user, with all the necessary header files, user hooks, predefined constants, data types, useable functions, etc. that will enable you to create your own Salvo application as quickly and as error-free as possible.

Conserve Precious Resources

The Salvo source code is written first and foremost to use as few resources as possible in the target application. Resources include RAM, ROM, stack call…return levels and instruction cycles. *Most of Salvo's RAM- and ROM-hungry functionality is disabled by default.* If you want a particular feature (e.g. event flags), you must enable it via a *configuration option* (e.g. `OSENABLE_EVENT_FLAGS`) and re-make your application. This allows you to manage the Salvo code in your application from a single point – the Salvo configuration file `salvocfg.h`.

Learn to Love the Preprocessor

Salvo makes heavy use of the C preprocessor and symbols predefined by the compiler, Salvo and/or the user in order to configure the source code for compilation. Though this may seem somewhat daunting at first, you'll find that it makes managing Salvo projects much simpler.

Document, But Don't Duplicate

Wherever possible, neither source code nor documentation is repeated in Salvo. This makes it easier for us to maintain and test the code, and provide accurate and up-to-date information.

We're Not Perfect

While every effort has been made to ensure that Salvo works as advertised and without error, it's entirely possible that we may have overlooked a problem or failed to catch a mistake. Should you find what you think is an error or ambiguity, please contact us so that we can resolve the issue(s) as quickly as possible and enable you to continue coding your Salvo applications worry-free.[2]

Note We feel that it should not be necessary for you to modify the source code to achieve functionality close to what Salvo already provides. We urge you to contact us first with your questions before modifying the source code, as we cannot support modified versions of Salvo. In many instances, we can both propose a solution to your problem, and perhaps also incorporate it into the next Salvo release.

[2] See Pumpkin Salvo Software License Agreement for more information.

Chapter 1 • Introduction

Welcome

In the race to innovate, time-to-market is crucial in launching a successful new product. If you don't take advantage of in-house or commercially available software foundations and support tools, your competition will. But cost is also an important issue, and with silicon (as in real life) prices go up as things get bigger. If your design can afford lots memory and maybe a big microprocessor, too, go out and get those tools. That's what everybody else is doing …

But what if it can't?

What if you've been asked to do the impossible – fit complex, real-time functionality into a low-cost microcontroller and do it all on a tight schedule? What if your processor has only a few KB of ROM and even less RAM? What if the only tools you have are a compiler, some debugging equipment, a couple of books and your imagination? Are you really going to be stuck again with state machines, jump tables, complex interrupt schemes and code that you can't explain to anyone else? After a while, that won't be much fun anymore. Why should you be shut out of using the very same software frameworks the big guys use?

They say that true multitasking needs plenty of memory, and it's not an option for your design. But is that really true?

Not any more. Not with Salvo. Salvo is full-blown multitasking in a surprisingly small memory space – it's about as big as `printf()`![3] Multitasking, priorities, events, a system timer – it's all in there. No interrupts available? That's not a problem, either. You'll get more functionality out of your processor quicker than you ever thought possible. And you can put Salvo to work for you right away.

[3] Comparison based on implementations with full `printf()` functionality.

What Is Salvo?

Salvo is a proven, powerful, high-performance and royalty-free real-time operating system (RTOS) that requires very little program and data memory, and no task stacks. It is an easy-to-use software tool to help you quickly create powerful, reliable and sophisticated applications (programs) for embedded systems.

Salvo was designed from the ground up for use in microprocessors and microcontrollers with severely limited resources, and will typically require from 5 to 100 times less memory than other RTOSes. In fact, Salvo's memory requirements are so minimal that it will run where no other RTOS can.

Salvo is ROMable, easily scaleable and extremely portable. It runs on just about any processor, from a PIC to a Pentium.

Why Should I Use Salvo?

If you're designing the next hot embedded product, you know that time-to-market is crucial to guarantee success. Salvo provides a powerful and flexible framework upon which you can quickly build your application.

If you're faced with a complex design and limited processing resources, Salvo can help you make the most of what's available in your system.

And if you're trying to cost-reduce or add functionality to an existing design, Salvo may be what you need because it helps you leverage the processing power you already have.

Before Salvo, embedded systems programmers could only dream of running an RTOS in their low-end processors. They were locked out of the benefits that an RTOS can bring to a project, including reducing time-to-market, managing complexity, enhancing robustness and improving code sharing and re-use. They were unable to take advantage of the many well-established RTOS features designed to solve common and recurring problems in embedded systems programming.

That dream is now a reality. With Salvo, you can stop worrying about the underlying structure and reliability of your program and start focusing on the application itself.

What Kind of RTOS Is Salvo?

Salvo is a purely event-driven cooperative multitasking RTOS, with full support for event and timer services. Multitasking is priority-based, with sixteen separate priority levels supported. Tasks that share the same priority will execute in a round-robin fashion. Salvo provides services for employing semaphores (binary and counting), messages, message queues and event flags for intertask communications and resource management. A full complement of RTOS functions (e.g. context-switch, stop a task, wait on a semaphore, etc.) is supported. Timer functions, including delays, timeouts and cyclic timers, are also supported.

Salvo is written in ANSI C, with a very small number of processor-specific extensions, some of which are written in native assembly language. It is highly configurable to support the unique demands of your particular application.

While Salvo is targeted towards embedded applications, it is universally applicable and can also be used to create applications for other types of systems (e.g. 16-bit DOS applications).

What Does a Salvo Program Look Like?

A Salvo program looks a lot like any other that runs under a multitasking RTOS. Listing 1 shows (with comments) the source code for a remote automotive seat warmer with user-settable temperature. The microcontroller is integrated into the seat, and requires just four wires for communication with the rest of the car's electronics – power, ground, Rx (to receive the desired seat temperature from a control mounted elsewhere) and Tx (to indicate status). The desired temperature is maintained via `TaskControl()`. `TaskStatus()` sends, every second, either a single 50ms pulse to indicate that the seat has not yet warmed up, or two consecutive 50ms pulses to indicate that the seat is at the desired temperature.

```
#include <salvo.h>

typedef unsigned char t_boolean;
typedef unsigned char t_temp;

/* Local flag.                                  */
t_boolean warm = FALSE;

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

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

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

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

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

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

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

Listing 1: A Simple Salvo Program

It's important to note that when this program runs, temperature control continues while `TaskStatus()` is delayed. The calls to `OS_Delay()` do not cause the program to loop for some amount of time and then continue. After all, that would be a waste of processor resources (i.e. instruction cycles). Instead, those calls simply

instruct Salvo to suspend the pulse generator and ensure that it resumes running after the specified time period. `TaskControl()` runs whenever `TaskStatus()` is suspended.

Apart from creating a simple Salvo configuration file and tying Salvo's timer to a 10ms periodic interrupt in your system, the C code above is all that is needed to run these two tasks concurrently. Imagine how easy it is to add more tasks to this application to enhance its functionality.

See *Chapter 4 • Tutorial* for more information on programming with Salvo.

What Resources Does Salvo Require?

The amount of ROM Salvo requires will depend on how much of Salvo you are using. A minimal multitasking application on an 8-bit RISC processor might use a few hundred instructions. A full-blown Salvo application on the same processor will use around 1K instructions.

In conventional RTOSes, a large amount of RAM is dedicated to the individual task stacks. Since Salvo does not need or maintain task tasks, its RAM requirements are commensurately (much) smaller, since C compilers may use the stack for local/auto variables, function parameters, etc. Also because of this fundamental design aspect of Salvo, Salvo can run on targets that have limited hardware call/return stacks[4] instead of a more common general-purpose stack.

The amount of RAM Salvo requires is also dependent on your particular configuration. In an 8-bit RISC application,[5] each task will require 4-12 (typically 7) bytes, each event 3-4 bytes,[6] and 4-6 more bytes are required to manage all the tasks, events and delays. That's it!

In all cases, the amount of RAM required is primarily dependent on the size of pointers (i.e. 8, 16 or 32 bits) to ROM and RAM in

---

4 A hardware call/return stack is used only to store the caller function's return address, and is limited to some depth (e.g. 16 levels on PIC17 processors). A hardware call/retun stack cannot be used for local (auto variables), for example. Additionally, processors with hardware call/return stacks do not implement PUSH and POP instructions.

5 PIC16 series (e.g. PIC16C64). Pointers to ROM take two bytes, and pointers to RAM take one byte.

6 Message queues require additional RAM.

your application, i.e. it's application-dependent. In some applications (e.g. CISC processors) additional RAM may be required for general-purpose register storage.

If you plan to use the delay and timeout services, Salvo requires that you call its timer service at a regular rate. While there are non-interrupt-driven ways of achieving this, this requirement is often satisfied by calling the timer service via a single interrupt. However, this interrupt need not be dedicated to Salvo – it can be used for your own purposes, too.

The number of tasks and events is limited only by the amount of available memory.

See *Chapter 6 • Frequently Asked Questions (FAQ)* for more information.

How Is Salvo Different?

Salvo is a cooperative RTOS that doesn't need a stack.[7] Virtually all other RTOSes use a stack, and many are preemptive as well as cooperative. This means that compared to other RTOSes, Salvo differs primarily in these ways:

- • Salvo is a cooperative RTOS, so you must explicitly manage task switching[8].
- • Task switching can only occur at the task level, i.e. directly inside your tasks, and not from within a function called by your task, or elsewhere. This is due to the absence of a general-purpose stack and the concomitant ability of the RTOS to save task & state information on the stack. This may have a small impact on the structure of your program.
- • Compared to other cooperative or preemptive RTOSes, which need lots of RAM memory (usually in the form of a general-purpose stack), Salvo needs very little. For processors without much RAM, Salvo may be your only RTOS choice.

---

7 By "doesn't need a stack" we mean that Salvo does not need RAM to store the data that a conventional RTOS usually stores on the (general-purpose) stack, including return addresses, local/auto variables, saved registers, and other (usually compiler-dependent) task-specific data.

8 We'll explain this term later, but for now it means being in one task and relinquishing control of the processor so that another task may run.

Salvo is able to provide most of the performance and features of a full-blown RTOS while using only a fraction as much memory. With Salvo you can quickly create powerful, fast, sophisticated and robust multitasking applications.

What Do I Need to Use Salvo?

A working knowledge of C is recommended. But even if you're a C beginner, you shouldn't have much difficulty learning to use Salvo.

Some knowledge of RTOS fundamentals is useful, but not required. If working with an RTOS is new to you, be sure to review *Chapter 2 • RTOS Fundamentals*.

You will need a good ANSI-C-compliant compiler for the processor(s) you're using. It must be capable of compiling the Salvo source code, which makes use of many C features, including (but not limited to):

- • arrays,
- • unions,
- • bit fields,
- • structures,
- • static variables,
- • multiple source files,
- • indirect function calls,
- • multiple levels of indirection,
- • passing of all types of parameters,
- • multiple bytes of parameter passing,
- • extensive use of the C preprocessor,
- • pointers to functions, arrays, structures, unions, etc., and
- • support for variable arguments lists[9] (via `va_arg()`, etc.)

Listing 2: C Compiler Feature Requirements

A compiler with the ability to perform in-line assembly is a plus. The more fully-featured the in-line assembler, the better.

[9] This is not absolutely necessary, but is desireable. `va_arg()` is part of the ANSI C standard.

Lastly, your compiler should be capable of compiling to object (`*.o`) modules and libraries (`*.lib`), and linking object modules and libraries together to form a final executable (usually `*.hex`).

We recommend that you use a compiler that is already certified for use with Salvo. If your favorite compiler and/or processor are not yet supported and it meets Salvo's requirements, you can probably do a port to them in a few hours. *Chapter 10 • Porting* will guide you through the process. Always check with the factory for the latest news concerning supported compilers and processors.

Which Processors and Compilers does Salvo Support?

Please visit Pumpkin's website for up-to-date information.

How Is Salvo Distributed?

Salvo is supplied on downloadable over the Internet as a Windows 95 / 98 / ME / NT / 2000 / XP installer program. After you install Salvo onto your computer you will have a group of subdirectories that contain the Salvo source code, Salvo libraries, Salvo examples, and various other support files.

What Is in this Manual?

Chapter 1 • Introduction is this chapter.

Chapter 2 • RTOS Fundamentals is an introduction to RTOS programming. If you're only familiar with traditional "superloop" or "foreground / background" programming architectures, you should definitely review this chapter.

Chapter 3 • Installation covers how to install Salvo onto your computer.

Chapter 4 • Tutorial is a guide to using Salvo. It contains examples to introduce you to all of Salvo's functionality and how to use it in your application. Even programmers familiar with other RTOSes should still review this chapter.

Chapter 5 • Configuration explains all of Salvo's configuration parameters. Beginners and experienced users need this information to

optimize Salvo's size and performance to their particular application.

Chapter 6 • Frequently Asked Questions (FAQ) contains answers to many frequently asked questions.

Chapter 7 • Reference is a guide to all of Salvo's user services (callable functions).

Chapter 8 • Libraries lists the available freeware and standard libraries and explains how to use them.

Chapter 9 • Performance has actual data on the size and speed of Salvo in various configurations. It also has tips on how to characterize Salvo's performance in your particular system.

Chapter 10 • Porting covers the issues you'll face if you're porting Salvo to a compiler and/or processor that is not yet formally certified or supported by Salvo.

Chapter 11 • Tips, Tricks and Troubleshooting has information on a variety of problems you may encounter, and how to solve them.

Appendix A • Recommended Reading contains references to multitasking and related documents.

Appendix B • Other Resources has information on other resources that may be useful to you in conjunction with Salvo.

Appendix C • File and Program Descriptions contains descriptions of all of the files and file types that are part of a Salvo installation.

Chapter 2 • RTOS Fundamentals

Note If you're already familiar with RTOS fundamentals you may want to skip directly to *Chapter 3 • Installation*.

Introduction

- *"I've built polled systems. Yech. Worse are applications that must deal with several different things more or less concurrently, without using multitasking. The software in both situations is invariably a convoluted mess. Twenty years ago, I naïvely built a steel thickness gauge without an RTOS, only to later have to shoehorn one in. Too many asynchronous things were happening; the in-line code grew to outlandish complexity."* Jack G. Ganssle[10]

Most programmers are familiar with traditional systems that employ a looping construct for the main part of the application and use interrupts to handle time-critical events. These are so-called *foreground / background* (or *superloop*) systems, where the interrupts run in the *foreground* (because they take priority over everything else) and the main loop runs in the *background* when no interrupts are active. As applications grow in size and complexity this approach loses its appeal because it becomes increasingly difficult to characterize the interaction between the foreground and background.

An alternative method for structuring applications is to use a software framework that manages overall program execution according to a set of clearly defined rules. With these rules in place, the application's performance can be characterized in a relatively straightforward manner, regardless of its size and complexity.

Many embedded systems can benefit from using an approach involving the use of multiple, concurrent tasks communicating amongst themselves, all managed by a kernel, and with clearly-

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

defined run-time behavior. This is the RTOS approach to programming. These and other terms are defined below.

Note This chapter is only a quick introduction to the operation and use of an RTOS. *Appendix A • Recommended Reading* contains references for further, in-depth reading.

Basic Terms

A *task* is a sequence of instructions, sometimes done repetitively, to perform an action (e.g. read a keypad, display a message on an LCD, flash an LED or generate a waveform). In other words, it's usually a small program inside a bigger one. When running on a relatively simple processor (e.g. Z80, 68HC11, PIC), a task may have all of the system's resources to itself regardless of how many tasks are used in the application.

An *interrupt* is an internal or external hardware event that causes program execution to be suspended. Interrupts must be enabled for an interrupt to occur. When this occurs, the processor vectors to a user-defined *interrupt service routine (ISR)*, which runs to completion. Then program execution picks up where it left off. Because of their ability to suspend program execution, interrupts are said to run in the foreground, and the rest of the program runs in the background.

A task's *priority* suggests the task's importance relative to other tasks. It may be fixed or variable, unique or shared with other tasks.

A *task switch* occurs when one task suspends running and another starts or resumes running. It may also be called a *context switch*, because a task's context (generally the complete contents of the stack and the values of the registers) is usually saved for re-use when the task resumes.

Preemption occurs when a task is interrupted and another task is made ready to run. An alternative to a preemptive system is a *co-operative* system, in which a task must voluntarily relinquish control of the processor before another task may run. It is up to the programmer to structure the task so that this occurs. If a running task fails to cooperate, then no other tasks will execute, and the application will fail to work properly.

Preemptive and cooperative context switching are handled by a *kernel*. Kernel software manages the switching of tasks (also called *scheduling*) and intertask communication. A kernel generally ensures that the highest-priority eligible task is the task that's running (preemptive scheduling) or will run next (cooperative scheduling). Kernels are written to be as small and as fast as possible to guarantee high performance in the overlying application program.[11]

A *delay* is an amount of time (often specified in milliseconds) during which a task's execution can be suspended. While suspended, a task should use as few of the processor's resources as possible to maximize the performance of the overall application, which is likely to include other tasks that are not concurrently suspended. Once the delay has *elapsed* (or *expired*), the task resumes executing. The programmer specifies how long the delay is, and how often it occurs.

An *event* is an occurrence of something (e.g. a key was pressed, an error occurred or an expected response failed to occur) that a task can wait for. Also, just about any part of a program can signal the occurrence of an event, thus letting others know that the event happened.

Intertask communication is an orderly means of passing information from one task to another following some well-established programming concepts. *Semaphores*, *messages*, *message queues* and *event flags* can be used to pass information in one form or another between tasks and, in some cases, ISRs.

A *timeout* is an amount of time (often specified in milliseconds) that a task can wait for an event. Timeouts are optional – a task can also wait for an event indefinitely. If a task specifies a timeout when waiting for an event and the event doesn't occur, we say that a timeout has occurred, and special handling is invoked.

A task's *state* describes what the task is currently doing. Tasks change from one state to another via clearly defined rules. Common task states might be *ready / eligible*, *running*, *delayed*, *waiting*, *stopped* and *destroyed / uninitialized*.

The *timer* is another piece of software that keeps track of elapsed time and/or real time for delays, timeouts and other time-related services. The timer is only as accurate as the timer clock provided by your system.

---

[11] Some kernels also provide I/O functions and other services such as memory management. Those are not discussed here.

A system is *idling* when there are no tasks to run.

The *operating system (OS)* contains the kernel, the timer and the remaining software (called *services*) to handle tasks and events (e.g. task creation, signaling of an event). One chooses a *real-time operating system (RTOS)* when certain operations are critical and must be completed correctly and within a certain amount of time. An RTOS-enabled *application* or *program* is the end product of combining your tasks, ISRs, data structures, etc, with an RTOS to form single program.

Now let's examine all these terms, and some others, in more detail.

Foreground / Background Systems

The simplest program structure is one of a *main loop* (sometimes called a *superloop*) calling functions in an ordered sequence. Because program execution can switch from the main loop to an ISR and back, the main loop is said to run in the background, whereas the ISRs run in the foreground. This is the sort of programming that many beginners encounter when learning to program simple systems.

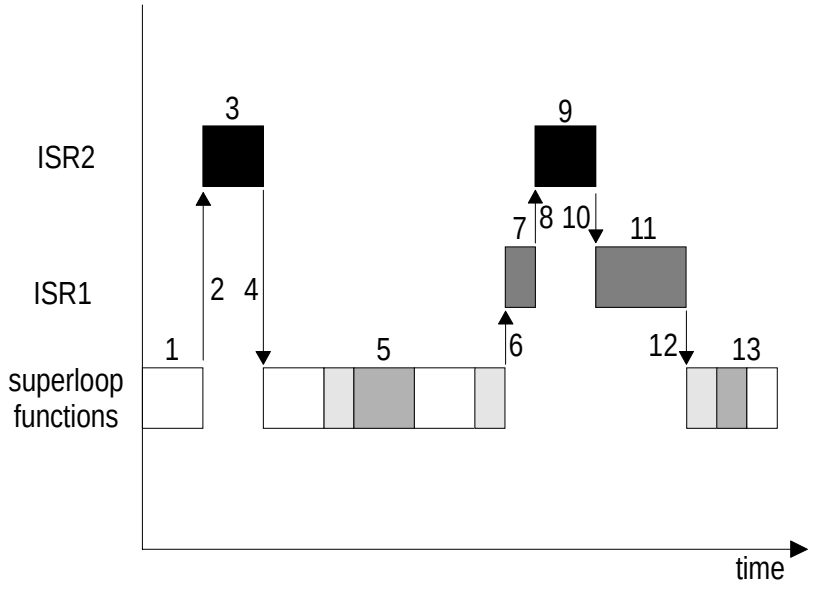

Figure 1: Foreground / Background Processing

In Figure 1 we see a group of functions repeated over and over [1, 5, 13] in a main loop. Interrupts may occur at any time, and even at multiple levels. When an interrupt occurs (high-priority interrupt at [2] and [8], low-priority interrupt at [6]), processing in the function

is suspended until the interrupt is finished, whereupon the program returns to the main loop or to a previous interrupted ISR. The main loop functions are executed in strictly serial manner, all at the same priority, without any means of changing when or even if the function should execute. ISRs must be used in order to respond quickly to external events, and can be prioritized if multiple interrupt levels are supported.

Foreground / background systems are relatively simple from a programming standpoint as long as there is little interaction amongst the functions in the main loop and between them and the ISRs. But they have several drawbacks: Loop timing is affected by any changes in the loop and/or ISR code. Also, the *response* of the system to inputs is poor because information made available by an ISR to a function in the loop cannot be processed by the function until its turn to execute. This rigidly sequential nature of program execution in the super loop affords very little flexibility to the programmer, and complicates time-critical operations. State machines may be used to partially solve this problem. As the application grows, the loop timing becomes unpredictable, and a variety of other complicating factors arise.

Reentrancy

One such factor is *reentrancy*. A reentrant function can be used simultaneously in one or more parts of an application without corrupting data. If the function is not written to be reentrant, simultaneous calls may corrupt the function's internal data, with unpredictable results in the application. For example, if an application has a non-reentrant `printf()` function and it is called both from main loop code (i.e. the background) and also from within an ISR (i.e. the foreground), there's an excellent chance that every once in a while the resultant output of a call to

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

from within the main loop and a call to

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

from within an ISR at the same time might be

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

Listing 3: Reentrancy Errors with printf()

This is clearly in error. What has happened is that the first instance of `printf()` (called from within the main loop) got as far as printing the first 9 characters ("`Here we a`") of its string argument before being interrupted. The ISR also included a call to `printf()`, which re-initialized its local variables and succeeded in printing its entire 36-character string ("`Now we … interrupt.\n`"). After the ISR finished, the main-loop `printf()` resumed where it had left off, but its internal variables reflected having successfully written to the end of a string argument, and no further output appeared necessary, so it simply returned and the main loop continued executing.

Note Calling non-reentrant functions as if they were reentrant rarely results in such benign behavior.

Various techniques can be employed to avoid this problem of a non-reentrant `printf()`. One is to disable interrupts before calling a non-reentrant function and to re-enable them thereafter. Another is to rewrite `printf()` to only use local variables (i.e. variables that are kept on the function's stack). The stack plays a very important role in reentrant functions.

Resources

A *resource* is something within your program that can be used by other parts of the program. A resource might be a register, a variable or a data structure, or it might be something physical like an LCD or a beeper. A *shared resource* is a resource that may be used by more than one part of your program. If two separate parts of a program are contending for the same resource, you'll need to manage this by *mutual exclusion*. Whenever a part of your program wants to use the resource it must obtain exclusive access to it in order to avoid corrupting it.

Multitasking and Context Switching

Many advantages can be realized by splitting a foreground / background application into one with multiple, independent tasks. In order to *multitask*, such that all tasks appear to run concurrently, some mechanism must exist to pass control of the processor and its resources from one task to another. This is the job of the *scheduler*, part of the kernel that (among its other duties) suspends one task and resumes another when certain conditions are met. It does this by storing the program counter for one task and restoring the pro-

gram counter for another. The faster the scheduler is able to switch tasks, the better the performance of the overall application, since the time spent switching tasks is time spent without any tasks running.

A context switch must appear transparent to the task itself. The task's "world view" before the context switch that suspends it and after the context switch that resumes it must be the same. This way, task A can be interrupted at any time to allow the scheduler to run a higher-priority task, task B. Once task B is finished, task A resumes where it left off. The only effect of the context switch on task A is that it was suspended for a potentially long time as a result of the context switch. Hence tasks that have time-critical operations must prevent context switches from occurring during those critical periods.

From a task's perspective, a context switch can be "out of the blue", in the sense that the context switch was forced upon it for reasons external to the task, or it can be intentional due to the programmer's desire to temporarily suspend the task to do other things.

Most processors support general-purpose stacks and have multiple registers. Just restoring the appropriate program counter will not be enough to guarantee the continuity of a task's execution. That's because the stack and the register values will be unique to that task at the moment of the context switch. A context switch saves the entire task's context (e.g. program counter, registers, stack contents). Most processor architectures require that memory must be allocated to each task to support context switching.

Tasks and Interrupts

As is the case with foreground / background systems, multitasking systems often make extensive use of interrupts. Tasks must be protected from the effects of interrupts, ISRs should be as fast as possible, and interrupts should be enabled most of the time. Interrupts and tasks coexist simultaneously – an interrupt may occur right in the middle of a task. The disabling of interrupts during a task should be minimized, yet interrupts will have to be controlled to avoid conflicts between tasks and interrupts when shared resources are accessed by both.

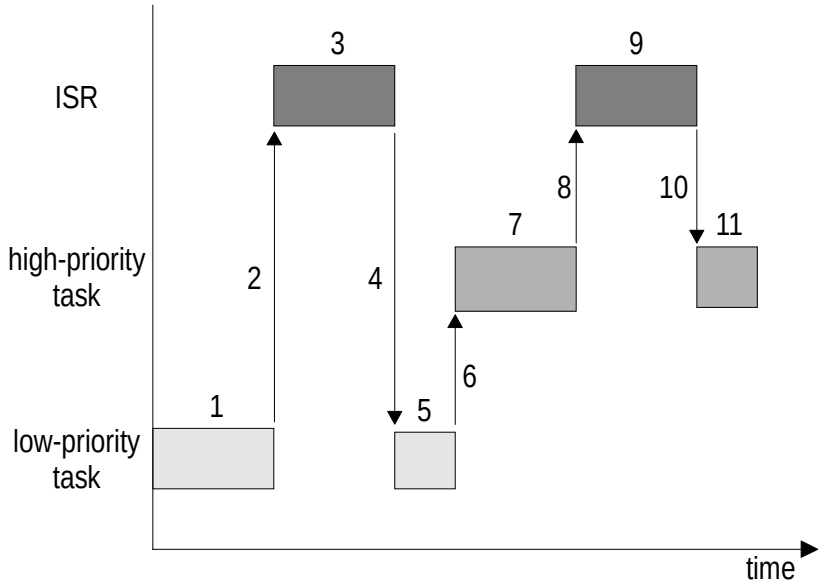

Figure 2: Interrupts Can Occur While Tasks Are Running

In Figure 2 a low-priority task is running [1] when an interrupt occurs [2]. In this example, interrupts are always enabled. The interrupt [3] runs to completion [4], whereupon the low-priority task [5] resumes its execution. A context switch occurs [6] and the high-priority task [7] begins executing. The context switch is handled by the scheduler (not shown). The high-priority task is also interrupted [8-10] before continuing [11].

Interrupt latency is defined as the maximum amount of time that interrupts are disabled, plus the time it takes to execute the first instruction of an ISR. In other words, it's the worst-case delay between when an interrupt occurs and when the corresponding ISR begins to execute.

Preemptive vs. Cooperative Scheduling

There are two types of schedulers: preemptive and cooperative. A *preemptive scheduler* can cause the current task (i.e. the task that's currently running) to be preempted by another one. Preemption occurs when a task with higher priority than the current task becomes eligible to run. Because it can occur at any time, preemption requires the use of interrupts and stack management to guarantee the correctness of the context switch. By temporarily disabling preemption, programmers can prevent unwanted disruptions in their programs during critical sections of code.

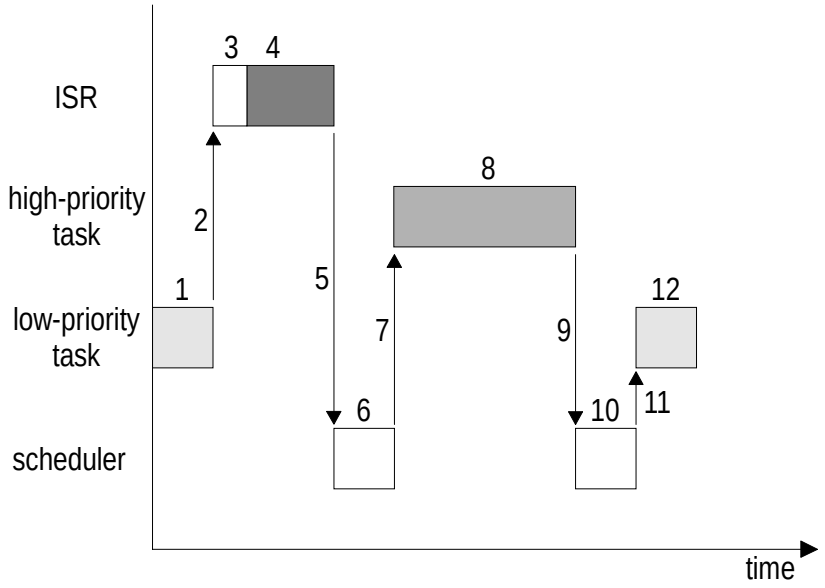

Figure 3: Preemptive Scheduling

Preemptive Scheduling

Figure 3 illustrates the workings of a preemptive scheduler. A low-priority task [1] is running when an external event occurs [2] that triggers an interrupt. The task's context and some other information for the scheduler are first saved [3] in the ISR, and the interrupt is serviced [4]. In this example the high-priority task is waiting for this particular event and should run as soon as possible after the event occurs. When the ISR is finished [5], it proceeds to the scheduler [6], which starts [7] the high-priority task [8]. When it is finished, control returns to the scheduler [9, 10], which then restores the low-priority task's context and allows it to resume where it was interrupted [11, 12].

Preemptive scheduling is very stack-intensive. The scheduler maintains a separate stack for each task so that when a task resumes execution after a context switch, all the stack values that are unique to the task are properly in place. These would normally be return addresses from subroutine calls, and parameters and local variables (for a language like C). The scheduler may also save a suspended task's context on the stack, since it may be convenient to do so.

Preemptive schedulers are generally quite complex because of the myriad of issues that must be addressed to properly support context switching at any time. This is especially true with regard to the handling of interrupts. Also, as can be seen in Figure 3, a certain

time lag exists between when an interrupt happens and when the corresponding ISR can run. This, plus the interrupt latency, is the *interrupt response* time (t_4 - t_2). The time between the end of the ISR and the resumption of task execution is the *interrupt recovery* time (t_7 – t_5). The system's *event response time* is shown as (t_7 - t_2).

Cooperative Scheduling

A *cooperative scheduler* is likely to be simpler than its preemptive counterpart. Since the tasks must all cooperate for context switching to occur, the scheduler is less dependent on interrupts and can be smaller and potentially faster. Also, the programmer knows exactly when context switches will occur, and can protect critical regions of code simply by keeping a context-switching call out of that part of the code. With their relative simplicity and control over context switching, cooperative schedulers have certain advantages.

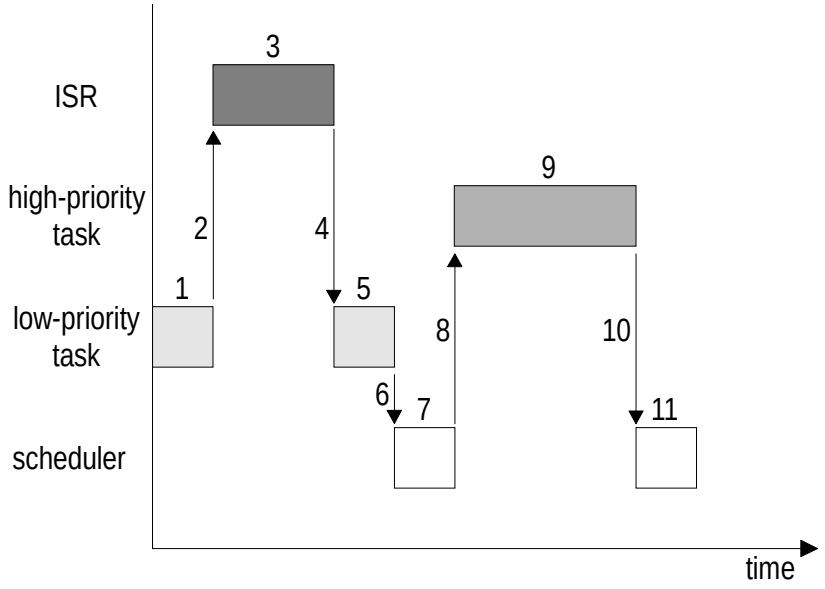

Figure 4: Cooperative Scheduling

Figure 4 illustrates the workings of a cooperative scheduler. As in the previous example, the high-priority task will run after the interrupt-driven event occurs. The event occurs while the low-priority task is running [1, 5]. The ISR is serviced [2-4] and the scheduler is informed of the event, but no context switch occurs until the low-priority task explicitly allows it [6]. Once the scheduler has a chance to run [7], it starts and runs the high-priority task to completion [8-10]. The scheduler [11] will then start whichever eligible task has the highest priority.

In comparison to the preemptive scheduling, cooperative scheduling has the advantage of shorter interrupt response and recovery times and greater overall simplicity. However, the responsiveness is worse because a high-priority eligible task cannot run until a lower-priority one has relinquished control of the processor via an explicit context switch.

More on Multitasking

You can think of tasks as little programs that run within a bigger program (your application). In fact, by using a multitasking RTOS your application can be viewed as a framework to define tasks and to control how and when they run. When your application is running, it means that a bunch of little programs (the tasks) are all running in a manner that makes it appear as if they execute simultaneously. Of course only one task can actually run at a particular instant. In order to take full advantage of the multitasking abilities of the RTOS, you want to define your tasks such that at any particular time, the processor is making the best use of its processing power by running whichever task is most important. Once your task priorities are correctly defined, the scheduler will take care of the rest.

Task Structure

What does a task in a multitasking application actually look like? A task is generally an operation that needs to occur over and over again in your application. The structure is really very simple, and consists of an optional initialization, and then a main loop that is repeated unconditionally. When used with a preemptive scheduler, a task might look like this:

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

Listing 4: Task Structure for Preemptive Multitasking

because a preemptive scheduler can interrupt a task at any time. With a cooperative scheduler a task might look like this:

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

```
}
```

Listing 5: Task Structure for Cooperative Multitasking

The only difference between the two versions is the need to explicitly call out the context switch in the cooperative version. In cooperative multitasking it's up to each task to declare when it is willing to potentially relinquish control of the processor to another task. Such context switches are usually unconditional – a trip through the scheduler may be required even if the current task is the only task eligible to run. In preemptive multitasking this would never occur, as the scheduler would force a context switch only when a higher-priority task had become eligible to run.

Note Context switches can occur multiple times inside a task, both in preemptive and cooperative multitasking systems.

Simple Multitasking

The simplest form of multitasking involves "sharing" the processor equally between two or more tasks. Each task runs, in turn, for some period of time. The tasks *round-robin*, or execute one after the other, indefinitely.

This has limited utility, and suffers from the problems of a super-loop architecture. That's because all tasks have equal, unweighted access to the processor, and their sequence of execution is likely to be fixed.

Priority-based Multitasking

Adding priorities to the tasks changes the situation dramatically. That's because by assigning task priorities you can guarantee that at any instant, your processor is running the most important task in your system.

Priorities can be static or dynamic. *Static priorities* are priorities assigned to tasks at compile time that do not change while the application is running. With *dynamic priorities* a task can change its priority during runtime.

Is should be apparent that if the highest-priority task were allowed to run continuously, then the system would no longer be multitasking. How can multiple tasks with different priorities coexist in a multitasking system? The answer lies in how tasks actually behave – they're not always running! Instead, what a certain task is doing

at any particular time depends on its *state* and on other factors, like *events*.

Task States

An RTOS maintains each task in one of a number of task *states*. Figure 5 illustrates the different states a task can be in, and the allowed transitions between states. *Running* is only one of several exclusive task states. A task can also be *eligible* to run, it can be *delayed*, it can be *stopped* or even *destroyed* / *uninitialized*, and it can be *waiting* for an event. These are explained below.

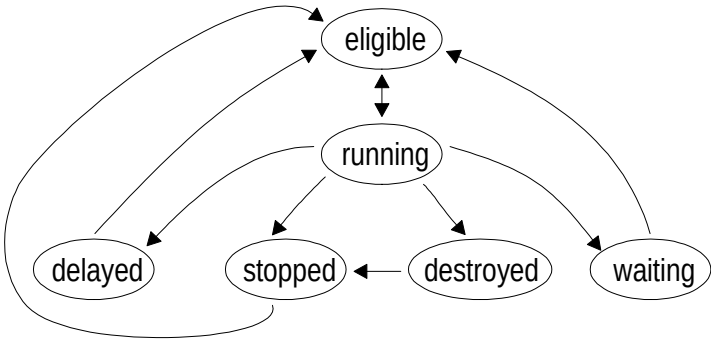

Figure 5: Task States

Before a task is created, it is in the uninitialized state. It returns to that state when and if it is destroyed. There's not much you can do with a destroyed task, other than create another one in its place, or recreate the same task again. A task *transitions* from the destroyed state to the stopped state when it is created via a call to the RTOS *service* that creates a task.

An eligible task is one that is ready to run, but can't because it's not the task with the highest priority. It will remain in this state until the scheduler determines that it is the highest-priority eligible task and makes it run. Stopped, delayed and/or waiting tasks can become eligible via calls to the corresponding RTOS services.

A running task will return to the eligible state after a simple context switch. However, it may transition to a different state if either the task calls an RTOS service that destroys, stops, delays or waits the task, or the task is forced into one of these states via a call to an RTOS service from elsewhere in your application.

A delayed task is one that was previously running but is now suspended and is waiting for a delay timer to expire. Once the timer has expired, the RTOS timer makes the task eligible again.

A stopped task was previously running, and was then suspended indefinitely. It will not run again unless it is (re-)started via a call to the RTOS service that starts a task.

A waiting task is suspended and will remain that way until the event it is waiting for occurs (See "Event-driven Multitasking" below).

It's typical for a multitasking application to have its various tasks in many different states at any particular instant. Periodic tasks are likely to be delayed at any particular instant. Low-priority tasks may be eligible but unable to run because a higher-priority task is already running. Some tasks are likely to be waiting for an event. Tasks may even be destroyed or stopped. It's up to the scheduler to manage all these tasks and guarantee that each tasks runs when it should. The scheduler and other parts of the RTOS ensure that tasks transition from one state to the next properly.

Note The heart of a priority-based multitasking application, the scheduler, is concerned with only one thing – running the highest-priority task that's eligible to run. Generally speaking, the scheduler interacts only with the running task and tasks that are eligible to run.

An RTOS is likely to treat all tasks in a particular state in the same manner, and thereby improve the performance of your application. For example, it shouldn't expend *any* processor cycles on tasks that are stopped or destroyed. After all, they're just "sitting there" and will remain so indefinitely, or until your program makes them eligible to run.

Delays and the Timer

Most embedded programmers are familiar with the simple delay loop construct, e.g.:

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

Listing 6: Delay Loop

The trouble with doing delays like the one shown in Listing 6 is that your application can't do any useful background processing while the loop is running. Sure, interrupts can occur in the foreground, but wouldn't it be nice to be able to do something else during the delay?

Another problem with the code in Listing 6 is that it is compiler-, processor- and speed-dependent. The compiler may or may not optimize the assembly instructions that make up this loop, leading to variations in the actual delay time. Changing the processor may change the delay time, too. And if you increase the processor clock, the delay will decrease accordingly. In order to circumvent these problems delay loops are often coded in assembly language, which severely limits code portability.

An RTOS provides a mechanism for tracking elapsed time through a system timer. This timer is often called in your application via a periodic interrupt. Each time it is called, the timer increments a counter that holds the number of elapsed *system ticks*. The current value of the system ticks is usually readable, and perhaps writeable too, in order to reset it.

The rate at which the timer is called is chosen to yield enough resolution to make it useful for time-based services, e.g. to delay a task or track elapsed time. A fluid level monitor can probably make do with a *system tick rate* of 1Hz (i.e. 1s system ticks), whereas a keypad reader might need a system tick rate of 100Hz (i.e. 10ms system ticks) in order to specify delays for the key debounce algorithm. An unduly fast system tick rate will result in substantial overhead and less processing power left over for your application, and should be avoided.

There must also be enough storage allocated to the system ticks counter to ensure that it will not overflow during the longest time period that you expect to use it. For example, a one-byte timer and a 10ms system tick period will provide a maximum specifiable task delay of 2.55s. In this example, attempting to calculate an elapsed time via the timer will result in erroneous results if successive reads are more than 2.55s apart. Task delays fall under similar restrictions. For example, a system with 10ms system ticks and support for 32-bit delays can delay a task up to a whopping 497 days!

Since the use of delays is common, an RTOS may provide built-in delay services, optimized to keep overhead to a minimum and to boost performance. By putting the desired delay inside a task, we can suspend the task while the delay is counting down, and then

resume the task once the delay has expired. Specifying the delay as a real amount of time will greatly improve our code's portability, too. The code for delaying a task via the RTOS looks quite different than that of Listing 6:

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

Listing 7: Delaying via the RTOS

In Listing 7, the call to the RTOS service `OS_Delay()` changes the state of the task from running to delayed. Since the task is no longer running, nor is it even eligible to run (remember, it's delayed), a context switch also occurs, and the highest-priority eligible task (if there is one) starts running.

In Listing 7 `OS_Delay()` also specifies a delay of 100 system ticks. If the system in has a system tick rate of 50Hz, then the task will be delayed for (100 ticks x 20ms) – two full seconds – before resuming execution once it becomes the highest-priority eligible task. Imagine how much processing other eligible tasks can do in two full seconds!

An RTOS can support multiple, simultaneously delayed tasks. It's up to the RTOS designer to maximize performance – i.e. minimize the overhead associated with the execution of the timer – regardless of how many tasks are delayed at any time. This timer overhead cannot be eliminated; it can only be minimized.

The *resolution* and *accuracy* of the system timer may be important to your application. In a simple RTOS, the resolution and the accuracy of the timer both equal the system tick period. For example, delaying a task by `n` system ticks will result in a delay ranging from just over `n-1` to just under `n` system ticks of real time (e.g. milliseconds). This is due to the asynchronous nature of the system timer – if you delay a task immediately after the (interrupt) call to the timer, the first delay tick will last nearly a full system tick. If, on the other hand, you delay a task immediately prior to a system tick, the first delay tick will be very short indeed.

Event-driven Multitasking

You may have noticed that a delayed task is actually waiting for something – it's waiting for its delay timer to expire. The expiration of a delay timer is an example of an *event*, and events may

cause a task to change state. Therefore events are used to control task execution. Examples of events include:

- • an interrupt,
- • an error occurring,
- • a timer timing out,
- • a periodic interrupt,
- • a resource being freed,
- • an I/O pin changing state,
- • a key on a keypad being pressed,
- • an RS-232 character being received or transmitted and
- • information being passed from one part of your application to another.

Listing 8: Examples of Events

In short, an event can be any action that occurs either internal or external to your processor. You associate an event with the rest of your application (primarily tasks, but also ISRs and background code) through the RTOS event services. The interaction between events and tasks follows certain simple rules:

- • *Creating* an event makes it available to the rest of your system. You cannot signal an event, nor can any task(s) wait on the event, until it has been created. Events can be created with different initial values.
- • Once an event has been created, it can be *signaled*. When an event is signaled, we say that the event has occurred. Events can be signaled from within a task or other background code, or from within an ISR. What happens next is dependent on whether there are one or more tasks waiting on the event.
- • Once an event has been created, one or more tasks can *wait* it. A task can only wait one event at a time, but any number of tasks can all wait the same event. If one or more tasks are waiting an event and the event is signaled, the highest-priority task or the first task to wait the event will become eligible to run, depending on how the RTOS implements this feature. If multiple waiting tasks share the same priority, the RTOS

will have a well-defined scheme[12] to control which task becomes eligible.

One reason for running tasks in direct response to events is to guarantee that at any time the system can respond as quickly as possible to an event. That's because waiting tasks consume no[13] processing power – they'll remaining waiting indefinitely, until the event they're waiting on finally occurs. Furthermore, you can tailor when the system acts on the event (i.e. run the associated task) based on its relative importance, i.e. based on the priority of the task(s) associated with the event.

The key to understanding multitasking's utility is to know how to structure the tasks in your application. If you're used to superloop programming, this may be difficult at first. That's because a common mode of thinking goes along the lines of "First I need to do this, then that, then the other thing, etc. And I must do it over and over again, checking to see if or when certain events have happened." In other words, the superloop system monitors events in a sequential manner and acts accordingly.

For event-driven multitasking programming, you may want to think along these lines: "What events are happening in my system, both internally and externally, and what actions do I take to deal with each event?" The difference here is that the system is purely *event-driven*. Events can occur repetitively or unpredictably. Tasks run in response to events, and a task's access to the processor is a function of its priority.[14] A task can react to an event as soon as there are no higher-priority tasks running.

Note Priorities are associated with tasks, not events.

In order to use events in your multitasking application, you must first ask yourself:

- • what does my system do?
- • how do I divide up its actions into separate tasks?
- • what does each task do?
- • when is each task done?
- • what are the events?

12 Generally LIFO or FIFO, i.e. the most recent task or the first task, respectively, to wait the event will become eligible when the event is signaled.

13 Unless they're waiting with a timeout, which requires the timer.

14 Task priorities are easily incorporated into event-based multitasking.

- • which event(s) cause each task to run?

Note Events need not be associated with tasks one-to-one. Tasks can interact with multiple events, and vice versa. Also, tasks that do not interact with any events are easily incorporated – but they are usually assigned low priorities, so that they only run when the system has nothing else to do.

Events and Intertask Communications

An RTOS will support a variety of ways to communicate with tasks. In event-based multitasking, for a task to react to an event, the event must trigger some sort of communication with the task. Tasks may also wish to communicate with each other. *Semaphores*, *messages* and *message queues* are used for intertask communication and are explained below.

Common to all these intertask communications are two actions: that of *signaling* (also called *posting* or *sending*) and *waiting* (also called *pending* or *receiving*). Each communication also requires an initialization (*creating*).

Note All operations involving semaphores, messages and message queues are handled through calls to the operating system.

Semaphores

There are two types of semaphores: *binary* semaphores and *counting* semaphores. A binary semaphore can take on only two values, 0 or 1. A counting semaphore can take on a range of values based on its size – for example, an 8-bit counting semaphore's value can range from 0 to 255. Counting semaphores can also be 16-bit or 32-bit. Figure 6 illustrates how we will represent semaphores and their values:

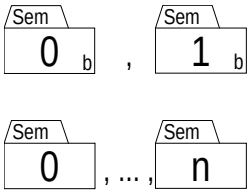

Figure 6: Binary and Counting Semaphores

Before it is used, a semaphore must be created with an initial value. The appropriate value will depend on how the semaphore is used.

Event Flags

Event flags are one such use for binary semaphores – they indicate the occurrence of an event. If a semaphore is initialized to 0, it means that the event has not yet occurred. When the event occurs, the semaphore is set to 1 by *signaling the semaphore*.

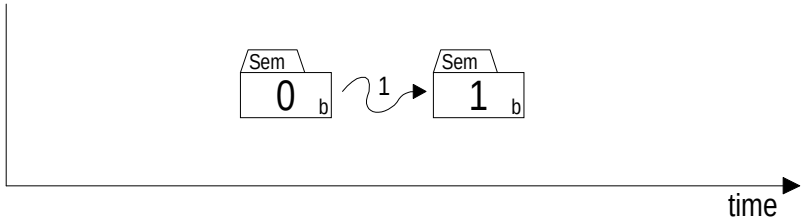

Figure 7: Signaling a Binary Semaphore

Figure 7 shows an ISR, task or other background code signaling [1] a binary semaphore. Once a semaphore (binary or counting) has reached its maximum value, further signaling is in error.

In addition to signaling a semaphore, a task can also *wait the semaphore*. Only tasks can wait semaphores – ISRs and other background code cannot. Figure 8 illustrates the case of an event having already occurred when the task waits the semaphore.

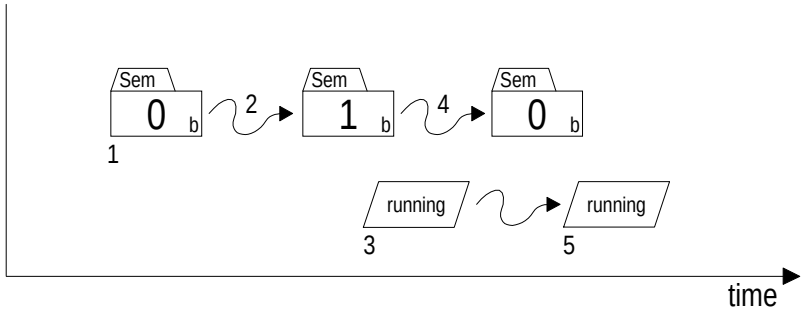

Figure 8: Waiting a Binary Semaphore When the Event Has Already Occurred

In Figure 8, the binary semaphore is initialized to 0 [1]. Some time later, the event occurs, signaling the semaphore [2]. When the task finally runs [3] and waits the semaphore, the semaphore will be reset [4] so that it can be signaled again and the task will continue running [5].

Note A semaphores is always initialized without any waiting tasks.

If the event has not yet occurred when the task waits the semaphore, then the task will be *blocked*. It will remain so (i.e. in the waiting state) until the event occurs. This is shown in Figure 9.

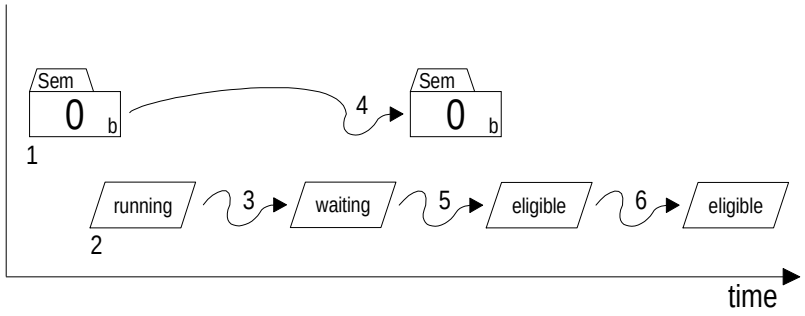

Figure 9: Signaling a Binary Semaphore When a Task is Waiting for the Corresponding Event

In Figure 9, an event has not yet been signaled [1] when a running task [2] waits the binary semaphore. Since the semaphore is not set, the task is blocked and must wait [3] indefinitely. The operating system knows that this task is blocked because it is waiting for a particular event. When the semaphore is eventually signaled from outside the task [4], the operating system makes the task eligible again [5] and it will run when it becomes the most eligible task [6]. The semaphore remains cleared because a task was waiting for it when it was signaled. Contrast this to Figure 7, where a semaphore is signaled with no tasks waiting for it.

It is also possible to combine event flags using the conjunctive (logical AND) or disjunctive (logical OR) combination of the event flags. The event is signaled when all (AND) or at least one (OR) of the event flags are set.

Note One or more tasks can concurrently wait an event. Which task becomes eligible depends on the operating system. For example, some operating systems may make the first task to wait the event eligible (FIFO), and others may make the highest-priority task eligible. Some operating systems are configurable to choose one scheme over the other.

Task Synchronization

Since tasks can be made to wait on an event before continuing, binary semaphores can be used as a means of *synchronizing* program

execution. Multitask synchronization is also possible – Figure 10 shows two tasks synchronizing their execution via two separate binary semaphores.

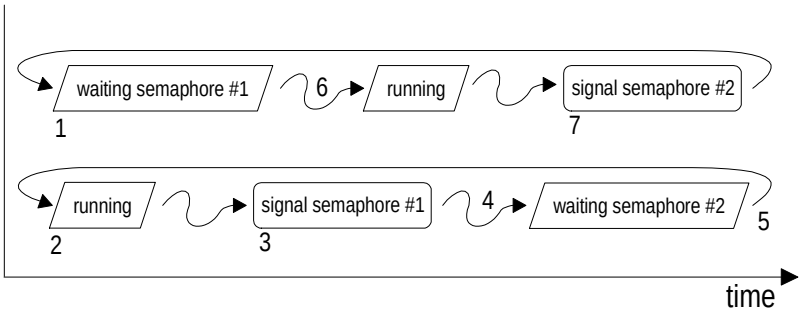

Figure 10: Synchronizing Two Tasks with Event Flags

In Figure 10, binary semaphores #1 and #2 are initialized to 0 and 1, respectively. The upper task begins by waiting semaphore #1, and is blocked [1]. The lower task begins running [2], and when it is ready to wait for the upper task it signals semaphore #1 [3] and then waits semaphore #2 [4], and is blocked [5] since it was initialized to 0. The upper task then begins running [6] since semaphore #1 was signaled, and when it is ready to wait for the lower task it signals semaphore #2 [7] and then waits semaphore #1, and is blocked [1]. This continues indefinitely. Listing 9 shows the pseudocode for this example.

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

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

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

Listing 9: Task Synchronization with Binary Semaphores

Resources

Semaphores can also be used to manage resources via *mutual exclusion*. The resource is available if the binary semaphore is 1, and is not available if it is 0. A task that wishes to use the resources must *acquire* it by successfully waiting the binary semaphore. Once it has acquired the resource, the binary semaphore is 0, and therefore any other tasks wishing to use the resource must wait until it is *released* (by signaling the binary semaphore) by the task that has acquired the resource.

```
initialize binary semaphore to 1;

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

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

Listing 10: Using a Binary Semaphore to Control Access to a Resource

In Listing 10 a binary semaphore is used to control access to a shared resource, a display (e.g. an LCD). In order to enable access to it, the semaphore must be initialized to 1. A task wishing to write to the display must acquire the resource by waiting the semaphore. If the resource is not available, the task will be blocked until the resource is released. After acquiring the resource and writing to the display, the task must then release the semaphore by signaling it.

Resources can also be controlled with counting semaphores. In this case, the value of the counting semaphore represents *how many of the resources* are available for use. A common example is that of a ring buffer. A ring buffer has space for `m` elements, and elements are added to and removed from it by different parts of an application. Figure 11 shows a scheme to transmit character strings via

RS-232 using a counting semaphore to control access to a ring buffer.

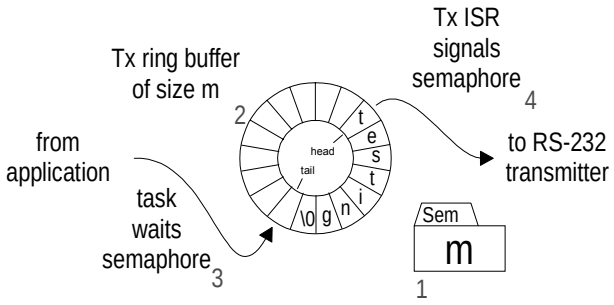

Figure 11: Using a Counting Semaphore to Implement a Ring Buffer

In Figure 11 a counting semaphore is initialized to `m` [1] to represent the number of spaces available in the empty ring buffer [2]. The ring buffer is filled at its tail[15] by the task [3] and emptied from its head by the ISR [4]. Before adding a character to the buffer the task must wait the semaphore. If it is blocked, it means that the buffer is full and cannot accept any more characters. If the buffer is not full, the semaphore is decremented, the task places the character at the tail of the buffer and increments the tail pointer. Once there are characters in the buffer[16], for each character the Tx ISR will remove it from the buffer, transmit it and increment the semaphore by signaling it. The corresponding pseudocode is shown[17] in Listing 11.

```
initialize counting semaphore to m;

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

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

15 The tail pointer points to the next available free space for insertion into the ring buffer. The head pointer points to the first available element for removal from the ring buffer.

16 This is usually signified by enabling transmit interrupts.

17 The control of Tx interrupts, which varies based on transmitter configurations, is not shown.

```
}
```

Listing 11: Using a Counting Semaphore to Control Access to a Resource

By using a task to fill the ring buffer, the application need not poll the buffer's status at regular intervals to determine when to insert new characters. Nor does the application need to wait in a loop for room to insert characters into the buffer. If only part of a string is inserted before the task is blocked (i.e. the string is larger than the available room in the buffer), the task will automatically resume inserting additional characters each time the ISR signals the counting semaphore. If the application sends strings infrequently, a low task priority will probably suffice. Otherwise a high task priority may be necessary.

Note The RAM required for the semaphore that is used to manage a resource is separate from the RAM allocated to the resource itself. The RTOS allocates memory for the semaphore – *the user must allocate memory for the resource*. In this example, 8-bit counting semaphores limit the size of the ring buffer to 256 characters. The semaphore will require one byte of RAM irrespective of the actual (user-declared) size of the ring buffer itself.

Messages

Messages provide a means of sending arbitrary information to a task. The information might be a number, a string, an array, a function, a pointer or anything else. Every message in a system can be different, as long as both the sender and the recipient of the particular message understand its contents. Even the type of message can even change from one message to the next, as long as the sender and recipient are aware of this! As with semaphores, the operating system provides the means to create, signal and wait messages.

In order to provide general-purpose message contents, when a message is sent and received, the actual content of the message is not the information itself, but rather a *pointer to the information*. A pointer is another word for the *address* (or location) *of the information*, i.e. it tells where to find the information. The message's recipient then uses the pointer to obtain the information contained in the message. This is called *dereferencing* the pointer.[18]

[18] In C, `&` is the address of operator, and `*` is the unary operator for indirection. Therefore if `var` is a variable and `p` points to it, then `p=&var` and `*p` is equal to `var`.

If a message is initialized to be empty, it contains a *null pointer*. A null pointer is a pointer with a value of 0. By convention, a null pointer doesn't point to anything; therefore it carries no other information with it. A null pointer cannot be dereferenced.

Signaling (i.e. sending) a message is more complex than signaling a semaphore. That's because the operating system's message-signaling function requires a message pointer as an argument. The pointer passed to the function must correctly point to the information you wish to send in the message. This pointer is normally non-zero, and is illustrated in Figure 12.

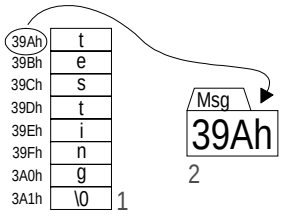

Figure 12: Signaling a Message with a Pointer to the Message's Contents

In Figure 12, a C-language character string[19] [1] is sent in a message [2] by signaling the message with a pointer. The string resides at a particular physical address. The message does *not* contain the first character of the string – it contains the address of the first character of the string (i.e. a pointer to the string), and the pointer's value is 39Ah. The pseudocode for sending this message is shown in Listing 12.

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

Listing 12: Signaling a Message with a Pointer

To receive a message's contents, a task must wait the message. The task will be blocked until the message arrives. The task then extracts the contents of the message (i.e. the pointer) and uses the pointer in whatever manner it chooses. In Listing 13, the receiving task capitalizes the string that the message points to.

```
TaskCaps()
{
  while (1) {
    wait message containing string pointer p;
```

19 In C, character strings end with the NUL character ('\0').

```
      while ((p) is not null) { [20]
        if ('a' <= (p) <= 'z')
        (p) = (p) - 32;
        increment p;
      }
    }
  }
```

Listing 13: Receiving a Message and Operating on its Contents

A message can contain at most one item of information (i.e. a pointer) at a time. If the message is empty, it can be signaled. If it's full, the message cannot be sent.

Messages can be used like binary semaphores. A message containing a null pointer is equivalent to a binary semaphore of value 0, and a message containing a non-zero pointer is equivalent to a binary semaphore of value 1. This is useful if binary semaphores are not explicitly supported by the RTOS.

Message Queues

Message queues are an extension of messages. A message queue can contain multiple messages (up to a predetermined number) at any time. Sending messages can continue until the message mailbox is full. A task that waits the message queue will receive messages until the message queue is empty.

An RTOS will need to allocate some additional RAM to manage each message queue. This RAM will be used to keep track of the number of messages in the message queue, and the order in which the messages exist in the message queue.

Summary of Task and Event Interaction

Here is a summary of the rules governing the interaction of tasks and events (i.e. semaphores, messages and message queues).

- • An events must be initialized. It is initialized without any waiting tasks.
- • A task cannot wait an event until the event has been initialized.
- • Multiple tasks can wait a single event.
- • A task can only wait one event at a time.

[20] "(pointer)" is pseudocode for "what is pointed to by the pointer."

- • A semaphore's value can range from 0 to its maximum value, depending on its size.
- • A message contains a pointer to some information.
- • Message queues can hold multiple messages at once.
- • An ISR, a task or other background code can signal an event.
- • Only a task can wait an event.
- • A task will be blocked (i.e. it will change to the waiting state) if the event it waits is not available.
- • Which waiting task becomes eligible when an event is signaled is dependent on how the operating system implements event services.
- • If an event has already been signaled, no task is waiting it, and it is signaled again, then either an error has occurred or the signaling task can be blocked. This is dependent on how the operating system implements event services.

Conflicts

A variety of conflicts may occur within a multitasking environment. They are described below.

Deadlock

Deadlock occurs with two or more tasks when each task is waiting for a resource controlled by another task. Since all of the affected tasks are waiting, there is no opportunity for any of the resources to become available. Therefore all the tasks will be deadlocked, i.e. they will wait indefinitely.

The solution is for all tasks wishing to acquire the resources to

- • always acquire the resources in a predetermined order,
- • acquire all the resources before continuing, and
- • release the resources in the opposite order.

Alternatively, by using a *timeout* one can break a deadlock. When attempting to acquire the resource, an optional time period can be

specified. If the resource is not acquired within that time period, the task continues, but with an error code that indicates that it timed out waiting for the resource. Special error handling may then be invoked.

Priority Inversions

Priority inversions occur when a high-priority task is waiting for a resource controlled by a low-priority task. The high-priority task must wait until the low-priority task releases the resource, whereupon it can continue. As a result, the priority of the high-priority task is effectively reduced to that of the low-priority task.

There are a variety of ways to avoid this problem (e.g. *priority inheritance*), most of which involve dynamically changing the priority of a task that controls a resource based on the priority of tasks wishing to acquire the resource.

RTOS Performance

The code to implement a multitasking RTOS may be larger than what's required in a superloop implementation. That's because each task requires a few extra instructions to be compatible with the scheduler. Even so, a multitasking application is likely to have much better performance and be more responsive than one with a superloop. That's because a well-written RTOS can take advantage of the fact that tasks that are not running often need not consume any processing power at all. This means that instead of spending instruction cycles testing flags, checking counters and polling for events, your multitasking application makes the most of the processor's power by using it directly where you need it most – on the highest-priority task that's eligible to run.

A Real-World Example

Let's look at an interesting example application – the controller for a remote soda-can vending machine. It must indicate (via LEDs on the buttons) if any selections are empty, handle the insertion of coins and bills, properly interpret the customer's selection, release the right item to the customer, and make change properly. A modern, microprocessor-controlled vending machine might also regulate internal temperatures (e.g. for soda cans), be connected to a network to relay out-of-stock information to a remote location, and be tied into a security system to deter vandalism. And of course all

of this has to be done without error regardless of how many unpredictable things the customer does in the quest to quench his or her hunger or thirst.

The Conventional Superloop Approach

The refrigerated, vandal-resistant vending machine in our example has a user interface consisting of an array of item-selection buttons and slots for bills and coins. The main loop for a pseudo-code version of a traditional superloop implementation might look like this:

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

  AcceptCurrency();

  flagSelectionGood = FALSE;
  ReadButtons();

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

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

Listing 14: Vending Machine Superloop

where some ISRs (not shown) are employed to do things like debounce the button presses. Listing 14 also shows neither the individual functions (e.g. `ReleaseItem()`) nor the global variables required to pass information between the functions, e.g. between `ReadButtons()` and `ReleaseItem()`.

Let's examine Listing 14 in more detail. In the superloop we call `ControlTemps()` once each time through the loop. On an 8-bit, 8MHz processor likely to be used in such an application, we might expect `ControlTemps()` to be called once every 200 microseconds when there's no user activity. This is a huge waste of processing power, as we know that we really only need to call it once a minute. We're calling `ControlTemps()` 5,000 times more often than necessary! While this may be acceptable in a vending machine, it's unlikely to be in a more demanding application.

One approach to fixing this would be to dedicate a periodic interrupt to set a globally visible bit every second. Then we could check this bit and call `ControlTemps()` when the bit is set high. This approach isn't too clever, because we're still doing an operation (testing the bit) every 200 microseconds. Another approach would be to move `ControlTemps()` completely into an ISR that's called every second, but that's ill-advised, especially if `ControlTemps()` is a large and complex function.

In our example, `ReleaseItem()` will run only when money's in the machine and a button has been pressed. In other words, it's waiting for an event – an event characterized by the presence of the proper amount of money AND a valid selection button being pressed.

As illustrated in Listing 14, foreground / background superloop software designs puts most of the required processing in a single main loop that the processor executes over and over again. External events and time-critical processing are handled in the foreground via ISRs. Note that no single operation in the superloop has priority over any other. The execution of the functions proceeds in a rigidly serial manner, with the use of many hierarchical loops. When adding more functionality to a system like this, the main loop is likely to grow larger and slower, perhaps more ISRs will be needed, and system complexity will increase in your attempt to keep everything working as a whole.

For instance, in the above example there's no way for the customer to cancel a purchase. How would you modify the code to handle this additional requirement? You could write an expanded state machine to handle various scenarios, or use lots of timer interrupts to control how often various functions can run. But do you think someone else would understand what you wrote? Or even you, two years from now?

The Event-Driven RTOS Approach

If we start to talk about understanding, modifying and maintaining foreground / background code of moderate to severe complexity, it loses its appeal. That's because there are no clear relationships among the various functions in the superloop, nor between the functions and the flag variables, nor between the ISRs and the super loop. Let's try a different, task- and event-based approach.

Here's a list of tasks we can identify from the example above:

- • Monitor and control internal temperature – `ControlTemps()`
- • Display empty bins via LEDs – `ShowEmpties()`
- • Accept or reject currency, and total it – `AcceptCurrency()`
- • Debounce and read buttons – `ReadButtons()`
- • Make change – `MakeChange()`
- • Release selected item to customer – `ReleaseItem()`
- • Attempt to protect the vending machine from vandalism – `CallPolice()`

Let's examine each of these tasks in a little more detail. We'll look at how important each one is, from 1 (most important) to 10 (least important), and when each task should run.

`ControlTemps()` is obviously important, as we want to keep the sodas cool. But it probably doesn't have to run more often than, say, once a minute, to accurately monitor and be able to control the temperature. We'll give it a priority of 4.

`ShowEmpties()` isn't too important. Moreover, the status of the empty bins only changes each time an item is released to the customer. So we'll give it a priority of 8, and we'd like it to run initially and once for every time an item is released.

`ReadButtons()` should have a reasonably high priority so that there's no noticeable lag when the customer presses the machine's buttons. Since button presses are completely asynchronous, we want to test the array of buttons regularly for activity. Let's give it a priority of 3, and run it every 40 milliseconds.

Since `AcceptCurrency()` is also part of the user interface, we'll give it the same priority as `ReadButtons()` and we'll run it every 20 milliseconds.

The machine's manufacturer does not consider `MakeChange()` to be all that important, so we'll give it a priority of 10. We'll link it to `ReleaseItem()`, since change must be made only after the selected item is delivered to the customer.

`ReleaseItem()` is interesting because we only need it once the proper amount of money has been accepted and an item button is pressed. To respond quickly we'll give it a priority of 2, and we'd

like it to run when the above combination of money and button press occurs.

The machine's manufacturer makes a big point of how vandal-resistant it is. It's even capable of detecting an attack (through built-in tilt sensors) and calling the local security service. We'll give `CallPolice()` the highest priority of 1, and we'll check the tilt sensors every 2 seconds for an attack.

Step By Step

Our vending machine example requires seven tasks with six different priorities, and a timer resolution of 20ms. To create this multitasking application from these functions, we'll need to:

- • initialize the operating system,
- • modify the structure of the tasks so as to be compatible with the operating system and the events,
- • create prioritized tasks from the task functions,
- • link the real-world events to events that the operating system understands,
- • create a system timer to keep track of elapsed time,
- • start the various tasks and
- • begin multitasking.

Initializing the Operating System

Initializing the operating system is usually straightforward, e.g.

```
InitializeMultitasking();
```

This creates the necessary (empty) structures the operating system will use to manage task execution and events. At this point all of the system's tasks are in the *uninitialized / destroyed* state.

Structuring the Tasks

The tasks written for a multitasking application look similar to those written for a superloop application. The big difference lies in the overall program structure. The multitasking tasks are not contained in any loops or larger functions – they're all independent

functions. `ReleaseItem()`, which releases an item once a set of conditions has been met, might look like this in pseudocode:

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

    Release(item);
  }
}
```

Listing 15: Task Version of ReleaseItem()

In Listing 15 `ReleaseItem()` waits forever for a (particular) message and does nothing until the message arrives. While it's waiting for the message to arrive, `ReleaseItem()` is in the *waiting* state. When the message is sent, `ReleaseItem()` becomes eligible to run, and when it runs, it extracts the contents of the message (in this case, a code for the desired item, e.g. "B3") and releases it to the customer. `ReleaseItem()` is not inside any larger loop, nor is it called by any other functions (except indirectly by the scheduler, below).

`CallPolice()` has a similar "stand-alone" look to it:

```
CallPolice()
{
  do forever {
    Delay(1000);

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

Listing 16: Task Version of CallPolice()

`CallPolice()` enters an infinite loop where it delays itself for 1000 x 20ms, or 2 seconds, and then sends a message to the police headquarters if the vending machine's tilt sensors detect an attack. It repeats this sequence indefinitely. While delayed, `CallPolice()` is in the *delayed* state.

Prioritizing the Tasks

An operating system call assigns a priority to a task, and prepares the task for multitasking. For example,

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

Listing 17: Prioritizing a Task

tells the operating system that it should give `ShowEmpties()`a priority of 8 and add it to the tasks whose execution it will manage. `ShowEmpties()` is now in the *stopped* state.

Interfacing with Events

In Listing 15, `ReleaseItem()` is using a message to handle an event – namely the release of an item. That message needs to be initialized:

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

Listing 18: Creating a Message Event

By initializing `messageSelection` to `empty` (i.e. no valid selection has been made), `ReleaseItem()` will only release an item once the required events (enough money inserted and appropriate button pressed) have occurred.

Adding the System Timer

An RTOS needs some way to keep track of real time – this is usually provided via some sort of timer function that the application must call at a regular, predefined rate. In this case that rate is 50Hz or every 20ms. Calling the system timer is often accomplished through an interrupt, e.g.:

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

Listing 19: Calling the System Timer

Starting the Tasks

Applications must create all of their tasks and events before any of them are actually used. By providing an explicit means of starting tasks, the RTOS enables you to manage system startup in a predictable way:

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

```
StartTask(ReadButtons());
StartTask(MakeChange());
StartTask(ReleaseItem());
StartTask(CallPolice());
```

Listing 20: Starting all Tasks

Since multitasking has not yet started, the order in which tasks are started is immaterial and is not in any way dependent on their priorities. At this point all of the tasks are in the *eligible* state.

Enabling Multitasking

Once everything is in place, events have been initialized and the tasks have been started (i.e. they are all ready to execute), multitasking can begin:

```
StartMultitasking();
```

Listing 21: Multitasking Begins

The scheduler will take the eligible task with the highest priority and run it – i.e. that task will be in the *running* state. From now on, the scheduler will ensure that the highest-priority task is the only one running at any time.

Putting It All Together

Listing 22 is a complete listing of the task- and event-driven vending machine application in pseudocode:

```
#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();
}
```

Listing 22: RTOS-based Vending Machine

The RTOS Difference

The program in Listing 22 has an entirely different structure than the superloop one in Listing 14. Several differences are immediately apparent:

- • It's somewhat longer – this is mainly due to the overhead of making calls to the operating system.
- • There are clearly-defined runtime priorities associated with each task.
- • The tasks themselves have simple structures and are easy to understand. Those that communicate with other tasks or ISRs use obvious mechanisms (e.g. semaphores and messages) to do so. Initialization can be task-specific.
- • The use of global variables is minimized.
- • There are no delay loops.
- • It's very easy to modify, add or delete a task without affecting the others.
- • The overall behavior of the application is largely dependent on the task priorities and intertask communication.

Perhaps most importantly, the RTOS handles the complexity of the application automatically – tasks run on a priority basis, task switching and state changes are handled automatically, delays require a minimum of processor resources, and the mechanisms of intertask communications are hidden from view.

There are other differences that become more apparent during runtime. If we were to look at a timeline showing task activity, we would see

- • Every 2 seconds `CallPolice()` wakes up to check for tampering and then returns to the delayed state,

- • Every second `ControlTemps()` wakes up to adjust the internal temperature and then returns to the delayed state,
- • Every 40ms `ReadButtons()` wakes up to debounce any button presses and then returns to the delayed state,
- • Every 20ms `AcceptCurrency()` wakes up to monitor the insertion of coins and bills and then returns to the delayed state, and
- • `ShowEmpties()`, `MakeChange()` and `ReleaseItem()` do nothing until a valid selection has been made, whereupon they briefly "come to life," deliver the selected item, refund any change and show full/empty item statuses, respectively, before returning to the waiting state.

In other words, for the vast majority of the time it's running, the vending machine's microcontroller has very little to do because the scheduler sees only delayed and waiting tasks. If the vending machine's manufacturer wanted to promote "Internet connectivity for enhanced stock management, remote querying and higher profits" as an additional feature, adding an extra task to transmit sales data (e.g. which sodas are purchased at what time and date and at what outside temperature) and run a simple web server would be as easy as creating another task to run in addition to the ones above and assigning it an appropriate priority.

Chapter 3 • Installation

Introduction

Salvo is provided in a self-extracting executable. Each installer will install all the files needed to build Salvo applications for the intended target and compiler, as well as additional files like *Salvo Compiler Reference Manuals*. All of the Salvo files are contained in compressed and encrypted form within the installer.

Note This section assumes you are installing Salvo onto a PC or PC compatible running Microsoft Windows XP. The installation for other Windows releases is similar.

Running the Installer

1. Launch the distribution-specific installer `salvo-lite|tiny|SE|LE|Pro-target-version.exe` on your PC. The Welcome screen appears:

Figure 13: Welcome Screen

Note Most of the installer's screens contain **Next**, **Back** and **Cancel** buttons. Click on the **Back** button for the previous screen. Click on the **Cancel** button to abort the installation.

2. After you click on the **Next** button, the Salvo License Agreement screen appears:

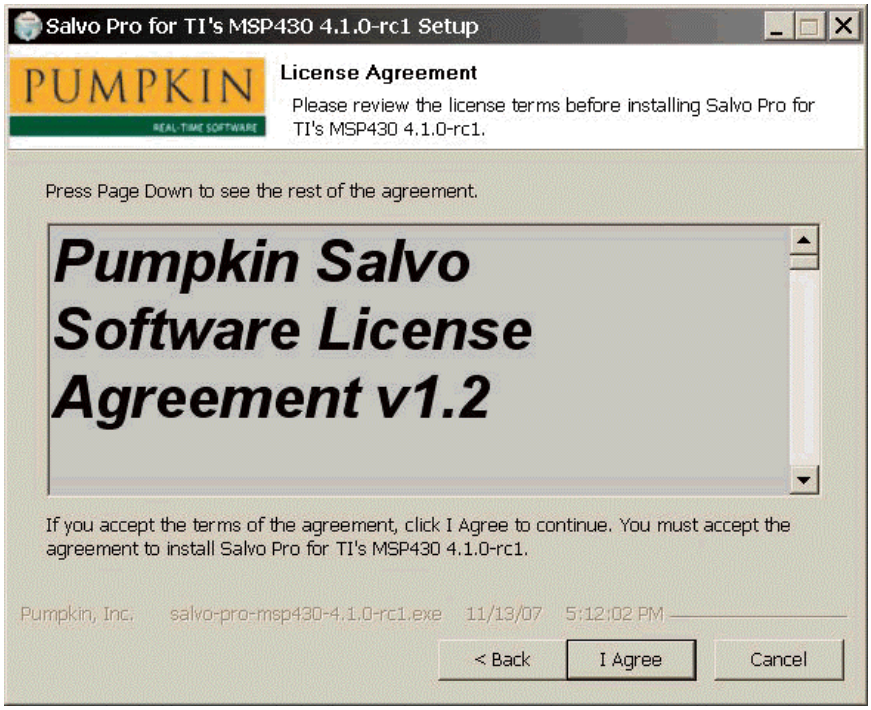

Figure 14: Salvo License Agreement Screen

This screen contains the *Pumpkin Salvo License Agreement*. Read this agreement carefully. This document is included in the Salvo folder once the installation is complete. You must accept the terms of the License in order to continue installing Salvo. To accept the License, click on the **I Agree** button. If you do not accept the License, click on the **Cancel** button and return the software.[21]

3. The Choose Components screen appears:

[21] Instructions on returning the software are contained in the License and in the User's Manual.

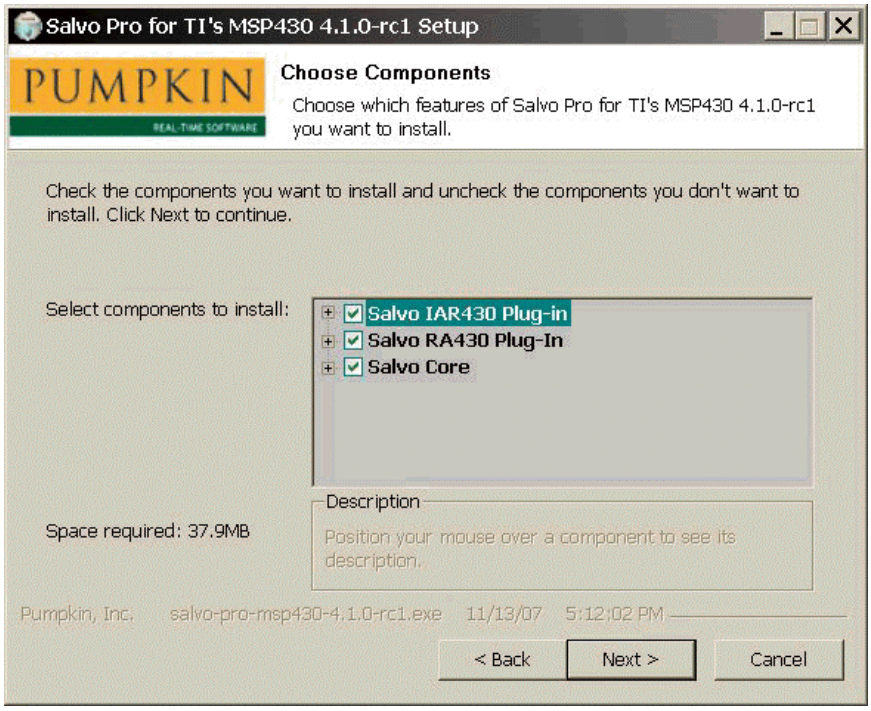

Figure 15: Choose Components Screen

Normally you should leave these selections at their default values. The components typically include the Salvo core (i.e. all the compiler- and target-independent components of Salvo), as well as compiler- and target-specific files.

Tip A description of the contents and function of each individually selectable component is available by expanding the tree and positioning the mouse over the component of interest.

This screen provides an elegant way of restoring individual Salvo components without doing a complete re-install. For example, if you accidentally delete a Salvo library for a particular compiler, you can choose to re-install just the Salvo libraries for said compiler via this screen.

Tip Installing components for toolsets you do not have installed does not normally cause problems. Therefore it is recommended that you leave all components selected (default).

4. The Choose Install Location screen appears:

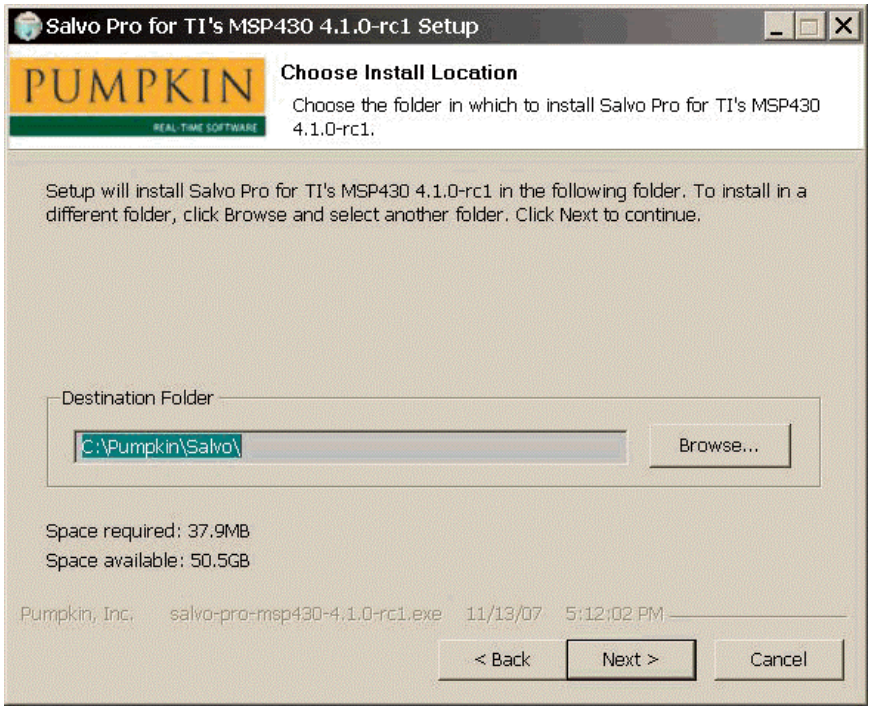

Figure 16: Choose Destination Location Screen

This screen allows you to set the directory where Salvo will be installed. The installer will place several[22] directories, some with nested subdirectories, in the destination directory. You can leave the destination directory at its default (`C:\Pumpkin\Salvo`) or you can change it by clicking on the **Browse…** button and selecting a different destination directory.

Tip In order to avoid potential compiler problems with long pathnames and spaces in path names, choosing the default path name is recommended. Choosing a deeply nested directory (e.g. `C:\Program Files\Pumpkin\My Projects\Programming\Tools \RTOS\Salvo\v4.1.0`) may cause problems with some tools due to exceedingly long pathnames for Salvo files. Also, spaces (' ') in pathnames should be avoided as some legacy compilers do not support them.

---

5. After clicking on the **Next** button the Choose Start Menu Folder screen appears:

22 See Figure 20: Typical Salvo Install Directory Contents.

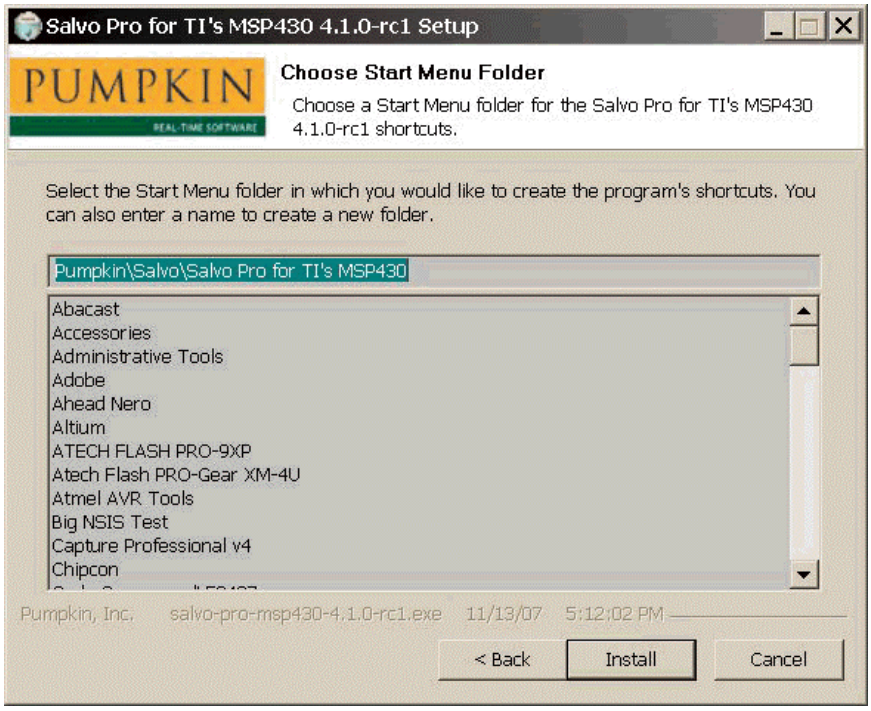

Figure 17: Choose Start Menu Folder Screen

6. Click on **Install** to continue. The installation begins, and ends with the Installation Complete screen:

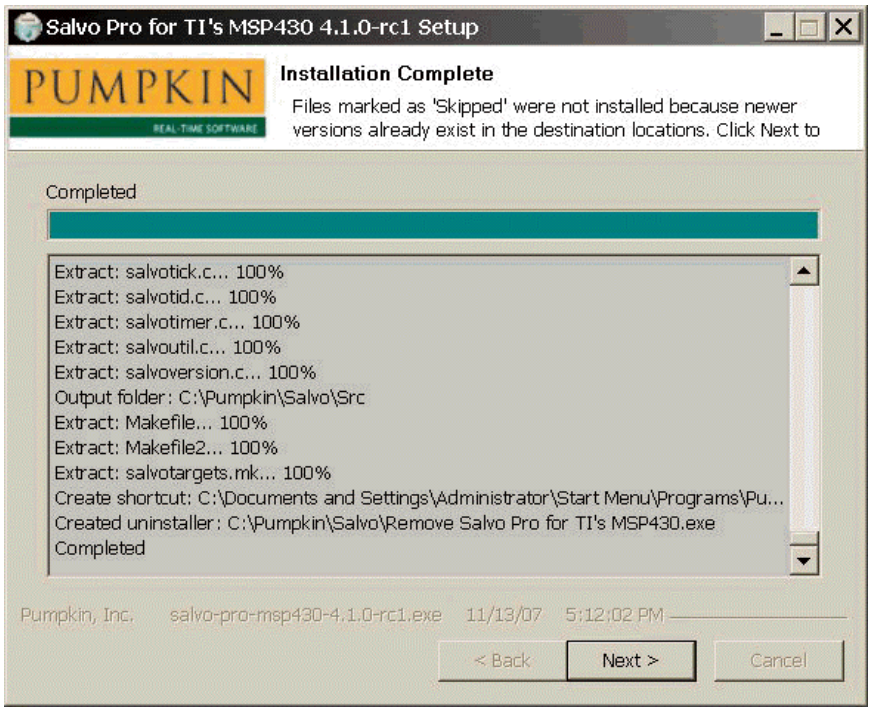

Figure 18: Installation Complete Screen

This screen lists all the files in the Salvo distribution installed to your PC. Individual files are marked as **Extract** (file was installed) or **Skipped** (file was skipped because a newer destination file with the same name was detected).

Tip The output of this screen's window is scrollable via the elevator on the right.

7. Once the installation of the files is completed, click on the **Next** button. You will be greeted with the Finish screen:

Figure 19: Finish Screen

Network Installation

If you are working in a networked environment with code sharing (e.g. for revision control) and need to install Salvo on a shared network disk, run the installer on a Wintel PC and choose a directory on a network drive as the destination directory. You may find it convenient to create the shortcuts in the Salvo Start Menu programs folder on each machine that is accessing Salvo over the network.

Note Network installations must comply with the terms of the *Salvo License Agreement*. See the License for more information.

Installing Salvo on non-Wintel Platforms

If you are developing Salvo applications on a non-Wintel platform, you will still need access to a Wintel machine in order to run the installer. The installer will place all of Salvo's files into the selected destination directory (the default is `C:\Pumpkin\Salvo`), with multiple subdirectories. You can then copy the entire subdirectory to another machine via a network or a mass storage device (e.g. Zip, Jaz, tape, etc.).

Note The Salvo License Agreement allows only one copy of the Salvo directories per installation. You must remove the entire Salvo directory from the Wintel machine after you have transported it to your non-Wintel development environment. See the License for more information.

Alternatively, if you are working in a networked environment with cross-platform file sharing, you can run the installer on a Wintel PC and select a (remote) directory on your non-Wintel platform as the destination directory for the installation. All of the Salvo files will be installed to the remote directory. After the installation is complete you may want to remove the Start Menu items from the Wintel PC if you will not be using them.

A Completed Installation

Your Salvo directory should look similar to this after a typical installation:

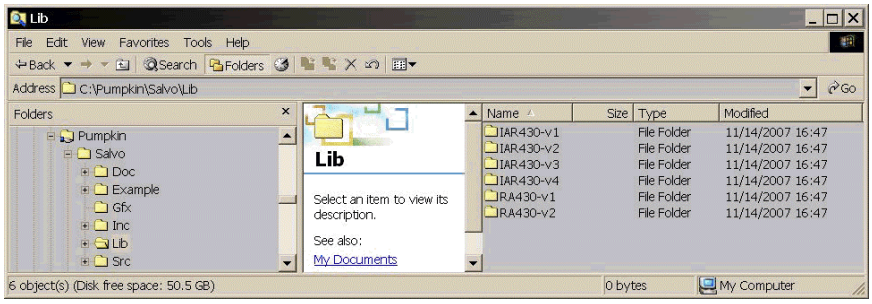

Figure 20: Typical Salvo Install Directory Contents (Lib Subdirectory View)

Uninstalling Salvo

The setup program automatically provides an uninstaller for each Salvo distribution. To use the uninstaller, run the appropriate Remove Salvo item as shown below:

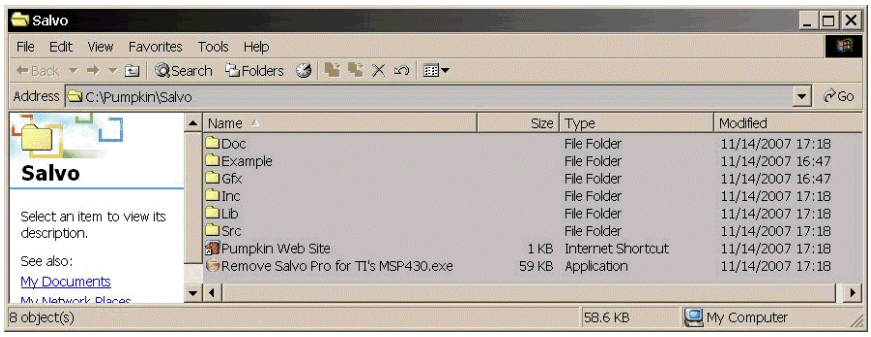

Figure 21: Location of the Uninstaller(s)

You will be greeted with a confirmation screen:

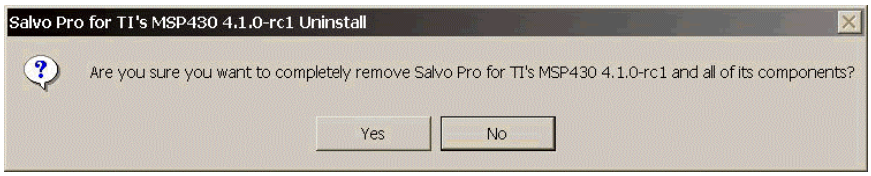

Figure 22: Confirming the Uninstall Operation

Click on the **Yes** button to begin uninstalling the specified Salvo distribution:

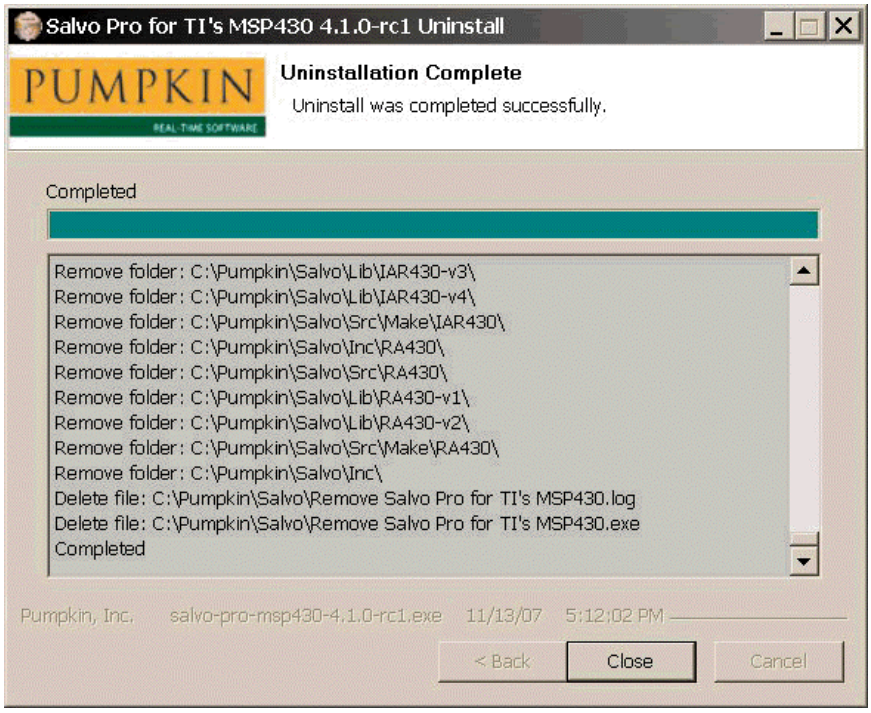

Figure 23: Uninstallation Complete Screen

Finally, the uninstaller will display the following screen upon successfully removing the specified Salvo distribution from your PC:

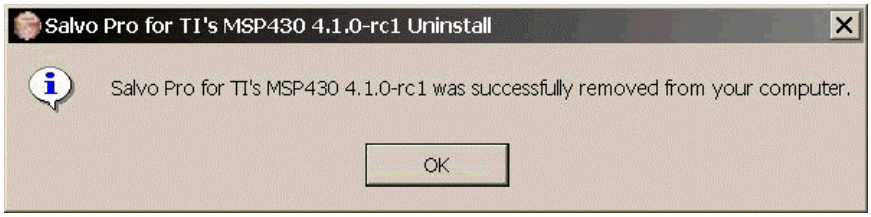

Figure 24: Uninstall Complete Screen

Click on the **OK** button to finish uninstalling Salvo.

Note The uninstaller will not remove any non-Salvo files in Salvo directories, nor will it delete any non-empty directories. If, after a Salvo uninstallation, files and/or directories still exist in the Salvo tree, you are advised to inspect those directories and delete the files and/or directories as required.

Uninstalling Salvo on non-Wintel Machines

If you are using Salvo on another platform (e.g. Linux), simply delete the Salvo destination directory and all of its subdirectories.

Installations with Multiple Salvo Distributions

The Salvo installer is designed to support multiple Salvo distributions of different types all in one directory (usually `C:\Pumpkin\Salvo`).[23] For example, you could have Salvo Lite for TI's MSP430 as well as Salvo Pro for 8051 family installed together in `C:\Pumpkin\Salvo`.

Installer Behavior

The Salvo installers replace files shared across all of the distributions only when the files to be installed are newer than the existing ones. When installed, a shared file is generally made read-only. Shared files include the target-independent Salvo header file and source files. Files that are unique to a distribution (e.g. project files and Salvo libraries) are always installed, i.e. overwritten by the installer.

Installing Multiple Salvo Distributions

Normally, no extra precautions are required when installing additional Salvo distributions onto a PC containing one or more existing Salvo distributions. By virtue of the installer's behavior, only the latest shared files should remain on the PC after each installer has finished.

Uninstalling with Multiple Salvo Distributions

Since an uninstaller will remove shared files, it is necessary to uninstall all of the Salvo distributions on the PC, and then re-install the desired ones.

Copying Salvo Files

Salvo users *are strongly discouraged* from copying any of Salvo's shared files to locations outside of the files' normal installation directories. Having duplicate Salvo files can lead to unpredictable behavior, and can greatly complicate debugging.

[23] As of Salvo v3.2.2.

Users with revision control systems who wish to add Salvo to their file repositories can do so by adding them in-place, and by retrieving them from a single source (e.g. a file server).

Modifying Salvo Files

Modifying Salvo's shared files can also lead to unpredictable behavior, and *is therefore strongly discouraged*. Generally speaking, only Salvo Pro users should modify Salvo's shared files, and only when a problem with the file(s) has been officially announced, and a solution provided. Once an updated Salvo distribution is available, it should automatically replace the modified file with an updated one.

Chapter 4 • Tutorial

Introduction

In this chapter we'll use a two-part, step-by-step tutorial to help you create a Salvo application from scratch. The first part is an introduction to using Salvo to write a multitasking program in C. In the second part we'll compile it to a working application.

Part 1: Writing a Salvo Application

Let's create a multitasking Salvo application step-by-step, introducing various concepts and Salvo features as we go. We'll start with a minimal application in C and build on it. We'll explain the purpose and use of each new Salvo feature, and describe in-depth what's happening in the application.

Tip Each one of the C listings below is provided as a complete application in the `Pumpkin\Salvo\Example\…\Tut\Tut5` directory of each Salvo distribution, complete with projects, source code and executables. You may find them useful to gain more insight into their operation.

Tut1: Initializing Salvo and Starting to Multitask

Each working Salvo application is a combination of calls to Salvo user services and application-specific code. Let's start using Salvo by creating a multitasking application.

A minimal Salvo application is shown in Listing 23.

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

int main( void )
{
  Init();

  OSInit();

  while (1) {
```

```
        OSSched();
    }
}
```

Listing 23: A Minimal Salvo Application

This elementary program calls two Salvo user services whose function prototypes are declared in `salvo.h`. `OSInit()` is called once, and `OSSched()` is called over and over again from within an infinite loop.

Tip These tutorials utilize a `while (1) { }` construct in C to create an infinite loop. The `for (;;) { }` and `do { } while (1)` constructs are functionally equivalent in terms of creating an infinite loop. All three are interchangeable for this puirpose.

Tip All user-callable Salvo functions are prefixed by "`OS`" or "`OS_`".

Note The `Init()` function in `main()` is provided for device initialization.[24] It and the header file `main.h` have nothing to do with the Salvo code per se, but are provided for completeness.

OSInit()

`OSInit()` initializes all of Salvo's data structures, pointers and counters, and must be called before any other calls to Salvo functions. Failing to call `OSInit()` first before any other Salvo routines may result in unpredictable behavior.

OSSched()

`OSSched()` is Salvo's multitasking scheduler. Only tasks which are in the eligible state can run, and each call to `OSSched()` results in the most eligible task running until the next context switch within that task. In order for multitasking to continue, `OSSched()` must be called repeatedly.

Tip In order to make best use of your processor's call ... return stack (whether hardware- or software-based), you should call `OSSched()` directly from `main()`.

In Depth

Since there are no tasks eligible to run, the scheduler in Listing 23 has very little to do.

---

[24] E.g. oscillator select and digital I/O crossbar select on Cygnal C8051F005 single-chip microcontroller.

Tut2: Creating, Starting and Switching tasks

Multitasking requires eligible tasks that the scheduler can run. A multitasking Salvo application with two tasks is shown in Listing 24.

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

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

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

int main( void )
{
  Init();

  OSInit();

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

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

Listing 24: A Multitasking Salvo Application with two Tasks

`TaskA()` and `TaskB()` do nothing but run and context switch over and over again. Since they both have the same priority (10), they run one after the other, continuously, separated by trips through the scheduler.

In order for multitasking to function properly, a running task must return control to the scheduler. This occurs via a context switch (or task switch) inside the task. Because it is designed to work without a stack, Salvo only supports context switching at the task level.

Warning A Salvo context switch at a call ... return level below that of the task (e.g. within a subroutine called by the task) will cause unpredictable behavior.

To multitask in Salvo, you must create and start tasks. Tasks are functions that consist of an optional initialization / preamble followed by an infinite loop containing at least one context switch. Salvo tasks cannot take any parameters. When the task is created via `OSCreateTask()`, you explicitly assign an unused *task control block* (*tcb*) to it and it is placed in the stopped state. A task can be created in many parts of your program. Tasks are often created prior to the start of multitasking, but they may also be created afterwards.

In order for a task to be able to run, it must be in the eligible state. `OSStartTask()` can make a stopped task eligible. However, in the interest of keeping the Salvo code size small, `OSCreateTask()` *automatically* starts the task that it has created.[25] Therefore a subsequent call to `OSStartTask()` is unnecessary if a Salvo task has been created normally. Once a task is made eligible, it will run by the scheduler as soon as it becomes the most eligible task, i.e. the eligible task with the highest priority.

Tip When a group of eligible tasks all share the same priority, they will execute one after the other in a round-robin fashion.

A stopped task can be started in many parts of your program. Tasks can only be started after they are created. A task may be started after multitasking begins.

OS_Yield()

Every task must context-switch at least once. `OS_Yield()` is Salvo's unconditional context switcher. A common place to find `OS_Yield()` would be at the bottom of, but still within, a task's infinite loop.

Note All Salvo user services with conditional or unconditional context switches are prefixed by "`OS_`".

OSCreateTask()

To create a task, call `OSCreateTask()` with a *task starting address*,[26] a *tcb pointer* and a *priority* as parameters. The starting address is usually the start of the task, specified by the task's name. Each task needs its own, unique tcb. The tcb contains all of the information Salvo needs to manage a task, like its start/resume address, state, priority, etc. There are `OSTASKS` tcbs available for use, numbered from `1` to `OSTASKS`. The Salvo `OSTCBP()` macro is a

[25] Optionally, the task can be left in the stopped state by using `OSDONT_START_TASK`.

[26] In C, this is equivalent to the name of the task (function).

shorthanded[27] way of specifying a pointer to a particular Salvo tcb, e.g. `OSTCBP(2)` is a pointer to the second tcb. The task priority is between 0 (highest) and 15 (lowest), and need not be unique to the task. Once created, a task is in the stopped state.

The default behavior for `OSCreateTask()` is to also start the Salvo task with the specified tcb pointer by making it eligible. It may be a while before the task actually runs, depending on the priority of the task, the states of any higher-priority tasks, and when the scheduler will run again.

Tip Many Salvo services return error codes that you can use to detect problems in your application. See *Chapter 7 • Reference* for more information.

In Depth

Listing 24 illustrates some of the basic concepts of an RTOS – tasks, task scheduling, task priorities and context switching. Tasks are functions with a particular structure – infinite loops are commonly used. A task will run whenever it is the most eligible task, and the scheduler decides which task is eligible based on the task priorities. Since Salvo is a cooperative RTOS, each task must relinquish control back to the scheduler or else no other tasks will have a chance to run. In this example, this is accomplished via `OS_Yield()`. In the following examples, we'll use other context switchers in place of `OS_Yield()`.

While it's perhaps not immediately apparent, Listing 24 also illustrates another basic RTOS concept – that of the task state. In Salvo, all tasks start out as destroyed – this is the state of an uninitialized task. Creating a task changes its state to stopped, and starting a task makes it eligible – i.e. it is now in the eligible state. When the task is actually executing it's said to be running. In this example, after being created and started, each task alternates between eligible and running over and over again. And there's a short time period during iteration of the main `for()` loop where neither task is running, i.e. they're both eligible – that's when the scheduler is running.

Task scheduling in Salvo follows two very simple rules: First, whichever task has the highest priority will run the next time the scheduler is called. Second, all tasks with the same priority will run in a round-robin manner as long as they are the most eligible tasks. This means that they will run one after the other until they have all run, and then the cycle repeats itself.

[27] `&OStcbArea[n-1]` is the longhanded way.

Tut3: Adding Functionality to Tasks

Listing 25 shows a multitasking application with two tasks that do more than just context switch. We'll use more descriptive task names this time.

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

unsigned int counter;

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

    OS_Yield();
  }
}

void TaskShow( void )
{
  InitPORT();

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

    OS_Yield();
  }
}

int main( void )
{
  Init();

  OSInit();

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

  counter = 0;

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

Listing 25: Multitasking with two Non-trivial Tasks

The two tasks in Listing 25 run independently of each other, and they both access a shared global variable, a 16-bit counter. The counter is initialized[28] before multitasking begins. The first task

[28] Strictly speaking, this initialization is unnecessary, as all ANSI compilers will set `counter` to 0 before `main()`.

increments the counter every time it has a chance to run. The other task takes the counter and outputs the upper 7 bits to an 8-bit port (`PORT`) with 8 LEDs connected to it. This goes on indefinitely.

Note Since Salvo is a cooperative RTOS, only one task can access the global variable `counter` at a time in this example.

In Depth

In Listing 25, neither task actually runs until multitasking begins with the call to the Salvo scheduler. Each time `OSSched()` is called, it determines which task is most eligible to run, and transfers program execution to that particular task. Since both tasks have the same priority, and are equally eligible to run, it is up to Salvo to decide which task will run first.

In this particular example, `TaskCount()` will run first.[29] It will start by incrementing the counter, and will then context-switch via `OS_Yield()`. This macro will make a note of where program execution is in `TaskCount()` (it's at the end of the `for()` loop), and then return program execution to the scheduler. The scheduler then examines `TaskCount()` to see if it's still eligible to continue running. In this case it is, because we made no changes to it, so it will run again when it becomes the most eligible task.

The scheduler finishes its work, and is then called again because it's in an infinite `for()` loop. This time, because Salvo round-robins tasks of equal priority, the scheduler decides that `TaskShow()` is the most eligible task, and makes it run. First, `PORT` is configured as an output port and initialized.[30] Then `TaskShow()` enters its infinite loop for the first time, `PORT` is initialized to 0x00 (the counter is now 0x0001), and once again `OS_Yield()` returns program execution to the scheduler after noting where to "return to" in `TaskShow()`. `TaskShow()` also remains eligible to run again.

After finishing its work, the scheduler is now called for the third time. Once again, `TaskCount()` is the most eligible task, and so it runs again. But this time it resumes execution where we last left it, i.e. at the end of the `for()` loop. Since it's an infinite loop, execution resumes at the top of the loop. `TaskCount()` increments the counter, and relinquishes control back to the scheduler.

---

29 Because it was started first, and both tasks have the same priority.

30 In this example, each pin on I/O port `PORT` can be configured as an input or as an output. At power-up, all pins are configured as inputs, hence the need to configure them as outputs via `InitPORT()`.`InitPORT()` also sets the 8-bit I/O port's initial value to 0x00.

The next time the scheduler is called, `TaskShow()` resumes where it left off, goes to the top of its `for()` loop, writes to `PORT`, and yields back to the scheduler. This entire process of resuming a task where it left off, running the task, and returning control back to the scheduler is repeated indefinitely, with each task running alternately with every call to the scheduler.

When the program in Listing 25 runs, it gives the outward appearance of two separate things occurring simultaneously. Both tasks are free-running, i.e. the faster the processor, the faster they'll run. A counter appears to be incremented and sent to a port simultaneously. Yet we know that two separate tasks are involved, so we refer to this program as a multitasking application. It's not very powerful yet, and its functionality could be duplicated in many other ways. But as we add to this application we'll see that using Salvo will allow us to manage an increasingly sophisticated system with a minimal coding effort, and we'll be able to maximize the system's performance, too.

Tut4: Using Events for Better Performance

The previous example did not use one of an RTOS' most powerful tools – intertask communications. It's also wasting processing power, since `TaskShow()` runs continuously, but `PORT` changes only once in every 512 calls to `TaskCount()`. Let's use intertask communication to make more efficient use of our processing power.

Listing 26 is shown below. We've used some `#define` preprocessor directives to improve legibility.

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

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

unsigned int counter;

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

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

```
      OSSignalBinSem(BINSEM_UPDATE_PORT_P);
    }

    OS_Yield();
  }
}

void TaskShow( void )
{
  InitPORT();

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

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

int main( void )
{
  Init();

  OSInit();

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

  OSCreateBinSem(BINSEM_UPDATE_PORT_P, 0);

  counter = 0;

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

Listing 26: Multitasking with an Event

In Listing 26 we communicate between two tasks in order to update the port only when an update is required. We'll use a binary semaphore to represent this event. We initialize it to 0, meaning the event has not yet occurred. `TaskCount()` signals the binary semaphore whenever the upper 7 bits of the counter change. `TaskShow()` waits for the event to occur, and then copies the upper 7 bits of the counter to `PORT`.

OSCreateBinSem()

`OSCreateBinSem()` creates a binary semaphore with the specified ecb pointer and initial value. A binary semaphore is created without any tasks waiting for it. A binary semaphore must be created before it can be signaled or waited.

OSSignalBinSem()

A binary semaphore is signaled via `OSSignalBinSem()`. If no task is waiting the binary semaphore, then it is simply incremented. If one or more tasks are waiting the binary semaphore, then the highest-priority waiting task is made eligible after signaling the binary semaphore.

OS_WaitBinSem()

A task will wait a binary semaphore until the binary semaphore is signaled. If the binary semaphore is zero when the tasks waits it, then the task switches to the waiting state and returns through the scheduler. It will keep waiting for the binary semaphore until the binary semaphore is signaled and the task is the highest-priority task waiting for the binary semaphore. That's because more than one task can wait for a particular event.

If, on the other hand, the binary semaphore is `1` when the task waits it, then the binary semaphore is reset to `0` and the task continues its execution without context switching.

Tip The "`OS_`" prefix in `OS_WaitBinSem()` should remind you that a context switch will *unconditionally* occur in every call to `OS_WaitBinSem()`, regardless of the value of the binary semaphore. If the binSem is set (i.e. equal to 1) and the task is the highest-priority eligible task, then execution will continue in the task. If not, execution in the task will resume at a later time when *both* of these conditions are met.

Tip You must always specify a timeout[31] when waiting a binary semaphore via `OS_WaitBinSem()`. If you want the task to wait forever for the binary semaphore to be signaled, use the predefined value `OSNO_TIMEOUT`.

Note In this example, `OS_WaitBinSem()` is used in place of `OS_Yield()`. In fact, the macro `OS_WaitBinSem()` includes a call to `OS_Yield()`. You do not need to call `OS_Yield()` when using a conditional context switcher like `OS_WaitBinSem()` – it does it for you.

In Depth

In order to improve the performance of our application, we'd like to update `PORT` only when the counter's upper 7 bits change. To do this we will use a signaling mechanism between the two tasks, called a binary semaphore. Here, the binary semaphore is a flag

---

[31] The timeout parameter is required regardless of whether or not your application is built with Salvo code (source files or libraries) that supports timeouts. This makes it possible to rebuild applications for timeouts without any user source code changes.

that's initialized to zero to mean that there's no need to update the port. When the binary semaphore is signaled, i.e. it is set to a value of `1`, it means that a `PORT` update is required.

Inter-task communication is achieved by using the binary semaphore to alert the waiting task (in this case, `TaskShow()`) that a `PORT` update is required. This is done in `TaskCount()` by calling `OSSignalBinSem()` with the parameter being a pointer to the binary semaphore, and by having `TaskShow()` wait the binary semaphore.

Note `TaskCount()` does not know which task(s) is(are) waiting on the binary semaphore, and `TaskShow()` does not know how the binary semaphore is signaled.

The first time `TaskShow()` runs through the scheduler it calls `OS_WaitBinSem()`. Since the binary semaphore was initialized to zero, `TaskShow()` yields control back to the scheduler and changes its state from eligible to waiting. Now there is only one eligible task, `TaskCount()`, and the scheduler runs it repeatedly.

When `TaskCount()` finally signals the binary semaphore, `TaskShow()` is made eligible again and will run once `TaskCount()` returns through the scheduler. After all, since the counter's upper 7 bits change only every 512 calls to `TaskCount()`, there's no point in running it more often than that. By using a binary semaphore, `TaskShow()` runs only when it needs to update `PORT`. The rest of the time, it is waiting and does not consume *any* processing power (instruction cycles).

The performance of this application is roughly twice as good (i.e. the counter increments at twice the speed) as that of Listing 25. That's because a waiting task consumes no processor power whatsoever while it waits – recall that the scheduler only runs tasks that are eligible. Since `TaskShow()` is waiting for the binary semaphore over 97% of the time,[32] it runs only on the rare occasion that the counter's upper byte has changed. The rest of the time, the scheduler is running `TaskCount()`.

It should be apparent that the calls to `OS_WaitBinSem()` and `OSSignalBinSem()` above implement some powerful functionality. In this example, these Salvo event services control when `TaskShow()` will run by using a binary semaphore for intertask communications. Here the binary semaphore is a simple flag (1 bit of

[32] Measured on Test System A.

information). Salvo supports the use of binary and counting semaphores, as well as other mechanisms, to pass more information (e.g. a count, or a pointer) from one task to another.

Listing 26 is a complete Salvo program – nothing is missing. There's nothing "running in the background", nothing checking to see if a waiting task should be made eligible, etc. In other words, there's no polling going on – all of Salvo's actions are event-driven, which contributes to its high performance. `TaskShow()` goes from waiting to eligible in the call to `OSSignalBinSem()`, and from running to waiting via `OS_WaitBinSem()`. With Salvo, you have complete control over what the processor is doing at any one time, and so you can optimize your program's performance without unwanted interference from the RTOS.

Tut5: Delaying a Task

One thing missing from the previous example is any notion of real-time performance – it just runs "open loop". If we add other tasks of equal or higher priority to the application, the rate at which the counter increments will decline. Let's look at how an RTOS can provide real-time performance by adding a task that runs at 2Hz, regardless of what the rest of the system is doing. We'll do this by repetitively delaying a task.

Being able to delay a task for a specified time period can be a very useful feature. A task will remain in the delayed state, ineligible to run, until the delay time specified has expired. It's up to the kernel to monitor delays and return a delayed task to the eligible state.

The application in Listing 27 blinks the LED on the least significant bit of `PORT` at 1Hz by creating and running a task which delays itself 500ms after toggling the port bit, and does this repeatedly. This program is located in `Pumpkin\Salvo\Example\…\Tut\Tut5` in every Salvo distribution.

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

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

```
unsigned int counter;

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

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

    OS_Yield();
  }
}

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

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

void TaskBlink( void )
{
  InitPORT();

  while (1) {
    PORT ^= 0x01;

    OS_Delay(50);
  }
}

void main( void )
{
  Init();

  OSInit();

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

  OSCreateBinSem(BINSEM_UPDATE_PORT_P, 0);

  counter = 0;

  enable_interrupts();

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

```
  }
}
```

Listing 27: Multitasking with a Delay

Additionally, interrupts are required to call `OSTimer()` at the desired system tick rate of 100Hz. The code to do this is located in `the source file tut5_sr.c that accompanies the project`. An example for the PIC16 is shown below:[33]

```
#include <salvo.h>

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

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

    OSTimer();
  }
}
```

Listing 28: Calling OSTimer() at the System Tick Rate

In order to use delays in a Salvo application, you must add the Salvo system timer to it. In the above example we've added a 10ms system timer by calling `OSTimer()` at a periodic rate of approximately 100Hz. The periodic rate is derived by a timer overflow, which causes an interrupt. Interrupts must be enabled in order for `OSTimer()` to be called – hence the call to `enable_interrupts()` just prior to starting multitasking. Since delays are specified in units of the system tick rate, the blink task is delayed by 50*10ms, or 500ms.

OSTimer()

In order to use Salvo delay services, you must call `OSTimer()` at a regular rate. This is usually done with a periodic interrupt. The rate at which your application calls `OSTimer()` will determine the resolution of delays. If the periodic interrupt occurs every 10ms, by calling `OSTimer()` from within the ISR you will have a system tick period of 10ms, or a rate of 100Hz. With a tick rate defined, you can specify delays to a resolution of one timer tick period, e.g. delays of 10ms, 20ms, ... 1s, 2s, ... are possible.

Note Salvo's timer features are highly configurable, with delays of up to 32 bits of system ticks, and with an optional prescalar.

---

[33] `IntVector()` is also used in tutorial # 6, below. `IntVector()` (and hence the contents of `tut5_isr.c`) are target- and compiler-specific.

Consult *Chapter 5 • Configuration* and *Chapter 6 • Frequently Asked Questions (FAQ)* for more information.

OS_Delay()

With `OSTimer()` in place and called repetitively at the system tick rate, you can now delay a task by replacing `OS_Yield()` with a call to `OS_Delay()`, which will force the context switch and delay the task for the number of system ticks specified. The task will automatically become eligible once the specified delay has expired.

In Depth

In Listing 27, each time `TaskBlink()` runs, it delays itself by 500ms and enters the delayed state upon returning to the scheduler. When `TaskBlink()`'s delay expires 500ms later it is automatically made eligible again, and will run after the current (running) task context-switches. That's because `TaskBlink()` has a higher priority than either `TaskCount()` or `TaskShow()`. By making `TaskBlink()` the highest-priority task in our application, we are guaranteed a minimum of delay (latency) between the expiration of the delay timer and when `TaskBlink()` toggles bit 0 of `PORT`. Therefore `TaskBlink()` will run every 500ms with minimal latency, irrespective of what the other tasks are doing.

Tip If `TaskBlink()` had the same priority as `TaskCount()` and `TaskShow()`, it would occasionally remain eligible (and would not run) while both `TaskCount()` and `TaskShow()` ran before it. Its maximum latency would increase. If `TaskBlink()` had a lower priority, it would never run at all.

The initialization of `PORT` was moved to `TaskBlink()` because of `TaskBlink()`'s priority. It will be the first task to run, and therefore `PORT` will be initialized as an output before `TaskShow()` runs for the first time.

Salvo monitors delayed tasks once per call to `OSTimer()`, and the overhead is independent of the number of delayed tasks.[34]

This illustrates that the system timer is useful for a variety of reasons. A single processor resource (e.g. a periodic interrupt) can be used in conjunction with `OSTimer()` to delay an unlimited number of tasks. More importantly, delayed tasks consume only a very small amount of processing power while they are delayed, much less than running tasks.

[34] Except when one or more task delays expire simultaneously.

Signaling from Multiple Tasks

A multitasking approach to programming delivers real benefits when priorities are put to good use and program functionality is clearly delineated along task lines.

Review the code in Listing 29 to see what happens when we lower the priority of the always-running task, `TaskCount()`, and have `TaskShow()` handle all writes to `PORT`. This program is located in `Pumpkin\Salvo\tut\tu6\main.c`.

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

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

unsigned int counter;

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

void TaskCount( void )
{
  counter = 0;

  while (1) {
    counter++;

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

    OS_Yield();
  }
}

void TaskShow( void )
{
  OStypeMsgP msgP;

  InitPORT();

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

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

```
    }
    else {
      PORT ^= 0x01;
    }
  }
}

void TaskBlink( void )
{
  OStypeErr err;

  while (1) {
    OS_Delay(50);

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

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

void main( void )
{
  Init();

  OSInit();

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

  OSCreateMsg(MSG_UPDATE_PORT_P, (OStypeMsgP) 0);

  enable_interrupts();

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

Listing 29: Signaling from Multiple Tasks

In Listing 29 we've made two changes to the previous program. First, `TaskShow()` now handles all writes to `PORT`. Both `TaskCount()` and `TaskBlink()` send a unique message to `TaskShow()` (the character ‘C' for "count" or ‘B' for "blink", respectively) which it then interprets to either show the counter on the port or

toggle the least significant bit of the port. Second, we've lowered the priority of `TaskCount()` by creating it with a lower priority.

OSCreateMsg()

`OSCreateMsg()` is used to initialize a message. Salvo has a defined type for messages, and requires that you initialize the message properly. A message is created without any tasks waiting for it. A message must be created before it can be signaled or waited.

Note Salvo services require that you interface your code using predefined types, e.g. `OStypeMsgP` for message pointers. You should use Salvo's predefined types wherever possible. See *Chapter 7 • Reference* for more information on Salvo's predefined types.

OSSignalMsg()

In order to signal a message with `OSSignalMsg()`, you must specify both a ecb pointer and a pointer to the message contents. If no task is waiting the message, then the message gets the pointer, unless the message is already defined, in which case an error has occurred. If one or more tasks are waiting the message, then the highest-priority waiting task is made eligible. You must correctly typecast the message pointer so that it can be dereferenced properly by whichever tasks wait the message.

OS_WaitMsg()

A task waits a message via `OS_WaitMsg()`. The message is returned to the task through a message pointer. In order to extract the contents of the message, you must dereference the pointer with a typecast matching what the message pointer is pointing to.

OS_SetPrio()

A task can change its priority and context-switch immediately thereafter using `OS_SetPrio()`.

OSSetPrio()

A task can change its priority using `OSSetPrio()`. The new priority will take effect as soon as the task yields to the scheduler.

In Depth

`TaskShow()` is now the only task writing to `PORT`. A single message is all that is required to pass unique information from two different tasks (which run at entirely different rates) to `TaskShow()`. In this case, the message is a pointer to a 1-byte constant. Since messages contain pointers, casting and proper dereferencing are required to send and receive the intended information in the message.

In Listing 29, the following scenario is possible: Immediately after `TaskCount()` signals the message, `TaskBlink()`'s delay expires and `TaskBlink()` is made eligible to run. Since `TaskBlink()` has the highest priority, the message will still be present when `Task-`

`Blink()` signals the message. Therefore `OSSignalMsg()` will return an error. The LED's PORT pin will fail to toggle …

This example illustrates the use of *return values* for Salvo services. By testing for the abovementioned error condition, we can guarantee the proper results by temporarily lowering `TaskBlink()`'s priority and yielding to the scheduler before signaling the message again. `TaskShow()` will temporarily be the highest-priority task, and it will "claim" the message. As long as `TaskCount()` does not signal messages faster than once every three context switches, this solution remains a robust one.[35]

In a more sophisticated application, e.g. a car's electronics, one can imagine `TaskShow()` being replaced with a task that drives a dashboard display divided into distinct regions. Four tasks would monitor information (e.g. rpm, speed, oil pressure and water temperature) and would pass it on by signaling a message whenever a parameter changed. `TaskShow()` would wait for this message. Each message would indicate where to display the parameter, what color(s) to use (e.g. red on overtemperature) and the parameter's new value. Since visual displays generally have low refresh rates, `TaskShow()` could run at a lower priority than the sending tasks. These tasks would run at higher priority so as to process the information they are sampling without undue interference from the slow display task. For example, the oil-pressure-monitoring task might run at the highest priority, since a loss of oil pressure means certain engine destruction. By having the display functionality in a task instead of in a callable function, you can fine-tune the performance of your program by assigning an appropriate priority to each of the tasks involved.

By lowering `TaskCount()`'s priority we've changed the behavior of our application. `PORT` updates now take precedence over the counter incrementing. This means that `PORT` updates will occur sooner after the message is signaled. The counter now increments only when there's nothing else to do. You can dramatically and predictably alter the behavior of your program by changing just the priority when creating a task.

Wrapping Up

As a Salvo user you do not have to worry about scheduling, tasks states, event management or intertask communication. Salvo han-

[35] An alternative solution to this problem would be to use a message queue with room for two messages in it.

dles all of that for you automatically and efficiently. You need only create and use the tasks and events in the proper manner to get all of this functionality, and more.

Note *Chapter 7 • Reference* contains working examples with commented C source code for every Salvo user service. Refer to them for more information on how to use tasks and events.

Food For Thought

Now that you're writing code with task- and event-based structures like the ones Salvo provides, you may find it useful or even necessary to change the way you approach new programs. Instead of worrying about how many processor resources, ISRs, global variables and clock cycles your application will require, focus instead on the tasks at hand, their priorities and purposes, your application's timing requirements and what events drive its overall behavior. Then put it all together with properly prioritized tasks that use events to control their execution and to communicate inside your program.

Part 2: Building a Salvo Application

Note If you have not done so already, please follow the instructions in *Chapter 3 • Installation* to install all of Salvo's components onto your computer. You may also find it useful to refer to *Chapter 5 • Configuration* and *Chapter 7 • Reference* for more information on some of the topics mentioned below. Lastly, you should review the *Salvo Application Note* that covers building applications with your compiler. Refer to your compiler's *Salvo Compiler Reference Manual* for particulars.

Now that you are familiar with how to write a Salvo application, it's time to build an executable program. Below you will find general instructions on building a Salvo application.

Working Environment

Salvo is distributed as a collection of source code files, object files, library files and other support files. Since all source code is provided in Salvo Pro, Salvo can be compiled on many development platforms. You will need to be proficient with your editor / com-

piler / integrated development environment (IDE) in order to successfully compile a Salvo application.

You should be familiar with the concepts of including a file inside another file, compiling a file, linking one or more files, working with libraries, creating an executable program, viewing the debugging output of your compiler, and placing your program into memory.

Please refer to your editor's / compiler's / IDE's documentation on how to include files into source code, compile source code, link to separate object modules, and compile and link to libraries.

Many IDEs support an automatic make-type utility. You will probably find this very useful when working with Salvo. If you do not have a make utility, you may want to investigate obtaining one. Both commercial and freeware / shareware make utilities exist, for command-line hosts (e.g. DOS) and Windows 95 / 98 / 2000 / NT.

Creating a Project Directory

In creating an application with Salvo you'll include Salvo source files in your own source code, and you'll probably also link to Salvo object files or Salvo libraries. We strongly recommend that you do not modify any Salvo files directly,[36] nor should you duplicate any Salvo files unnecessarily. Unless you intend to make changes to the Salvo source code, you should not change any of Salvo's files.

By creating a working directory for each new Salvo application you write, you'll be able to:

- • minimize hard disk usage,
- • manage your files better,
- • make changes to one application without affecting any others, and
- • compile unique versions of Salvo libraries for different projects.
-

Note Complete projects for certain tutorial programs can be found in `Pumpkin\Salvo\Tut`.

[36] Salvo source files are installed as read-only.

Including salvo.h

Salvo's main header file, `salvo.h`, must be included in each of your source files that use Salvo. You can do this by inserting

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

into each of your source files that calls Salvo services. You may also need to configure your development tools to add Salvo's home directory (usually `C:\Pumpkin\Salvo`) to your tools' *system include path* – see *Setting Search Paths*, below.

Note Using

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

is *not* recommended.

Tip If you include a project header file (e.g. `myproject.h`) in all of your source files, you may want to include `salvo.h` in it.

Including `salvo.h` will automatically include your project-specific version of `salvocfg.h` (see *Setting Configuration Options*, below). You should not include `salvocfg.h` in any of your source files – just including `salvo.h` is enough.

Note `salvo.h` has a built-in "include guard" which will prevent problems when multiple references to include `salvo.h` are contained in a single source file.

Configuring your Compiler

In order to successfully compile your Salvo application you must configure your compiler for use with the Salvo source files and libraries. You have several options available to you when combining your code with the Salvo source code in order to build an application.

Setting Search Paths

First, you must specify the appropriate search paths so that the compiler can find the necessary Salvo include (`*.h`) and source (`*.c`) files.

Tip All of Salvo's supported compilers support explicit search paths. Therefore you should *never* copy Salvo files from their source directories to your project directory in order to have the compiler find them by virtue of the fact that it's in the current directory.

At the very least, your compiler will need to know where to find the following files:

- • `salvo.h`, located in `Pumpkin\Salvo\inc`
- • `salvocfg.h`, located in your current project directory

You may also need to specify the Salvo source file directory (`Pumpkin\Salvo\Src`) if you have Salvo Pro and plan to include Salvo source files in your own source files (see below).

Using Libraries vs. Using Source Files

Different methods for incorporating Salvo into your application are outlined below. Linking to Salvo libraries is the simplest method, but has limitations. Including the Salvo source files in your project is the most flexible method, but isn't as simple, and requires Salvo Pro. Creating custom Salvo libraries from the source files is for advanced Salvo Pro users.

Tip You may find *Figure 25: Salvo Library Build Overview* and *Figure 26: Salvo Source-Code Build Overview* useful in understanding the process of building a Salvo application.

Using Libraries

Just like a C compiler's library functions – e.g. `rand()` in the standard library (`stdlib.h`) or `printf()` in the standard I/O library (`stdio.h`) – Salvo has functions (called *user services*) contained in libraries. Unlike a compiler's library functions, Salvo's user services are highly configurable – i.e. their behavior can be controlled based on the functionality you desire in your application. Each Salvo library contains user functions compiled for a particular set of *configuration options*. There are many different Salvo libraries.

Note Configuration options are *compile-time* tools used to configure Salvo's source code and generate libraries. Therefore the

functionality of a precompiled library *cannot be changed* through configuration options. To change a library's functionality, it must be regenerated (i.e. re-compiled) with Salvo Pro and new configuration options.

In order to facilitate getting started, all Salvo distributions contain libraries with most of Salvo's functionality already included. As a beginner, you should start by using the libraries to build your applications. This way, you don't have to concern yourself with the myriad of configuration options.

Tip The easiest and quickest way to create a working application is to link your source code to the appropriate Salvo library. The compiler-specific *Salvo Application Notes* describe in detail how to create applications for each compiler.

Complete library-based projects for all the tutorial programs can be found in `Pumpkin\Salvo\tut\tu1-tu6`. See *Appendix C • File and Program Descriptions* for more information.

Using Source Files

Salvo is configurable primarily to minimize the size of the user services and thus conserve ROM. Also, its configurability aids in minimizing RAM usage. Without it, Salvo's user services and variables might be too large to be of any use in many applications. All of this has its advantages and disadvantages – on the one hand, you can fine-tune Salvo to use just the right amount of ROM and RAM in your application. On the other hand, it can be a challenge learning how all the different configuration options work.

There are some instances where it's better to create your application by adding the Salvo source files as nodes to your project. When you use this method, you can change configuration options and re-build the application to have those changes take effect in the Salvo source code. Only Salvo Pro includes source files. The rest of this chapter covers this approach.

Setting Configuration Options

Salvo is highly configurable. You'll need to create and use a configuration file, `salvocfg.h`, for each new application you write. This simple text file is used to select Salvo's compile-time configuration options, which affect things like how many tasks and events

your application can use. All configuration options have default values – most of them may be acceptable to your application.

Note Whenever you redefine a configuration option in `salvocfg.h`, you *must* recompile all of the Salvo source files in your application.

The examples below assume that you are creating and editing `salvocfg.h` via a text editor. Each configuration option is set via a C-language `#define` statement. For example, to configure Salvo to support 16-bit delays, you would add

```
#define OSBYTES_OF_DELAYS 2
```

to your project's `salvocfg.h` file. Without this particular line, this configuration option would be automatically set to its default (in this case, 8-bit delays).

Note The name and value of the configuration option are case-sensitive. If you type the name incorrectly, the intended option will be overridden by the Salvo default.

Identifying the Compiler and Target Processor

Normally, Salvo automatically detects which compiler and target processor you are using. It does this by detecting the presence of certain predefined symbols provided by the compiler.

Specifying the Number of Tasks

Memory for Salvo's internal task structures is allocated at compile time. You must specify in `salvocfg.h` how many tasks you would like supported in your application, e.g.:

```
#define OSTASKS 4
```

You do not need to use all the tasks that you allocate memory for, nor must you use their respective tcb pointers (numbered from `OSTCBP(1)` to `OSTCBP(OSTASKS)`) consecutively. If you attempt to reference a task for which no memory was allocated, the Salvo user service will return a warning code.

Tip Tasks are referred to in Salvo by their tcb pointers. It's recommended that you use descriptive designations in your code to

refer to your tasks. This is most easily done by using the `#define` statement in your project's main header (`.h`) file, e.g.:

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

Your program will be easier to understand when calling Salvo task services with meaningful names like these.

Specifying the Number of Events

Memory for Salvo's internal event structures is also allocated at compile time. You must specify in `salvocfg.h` how many events you would like supported in your application, e.g.:

```
#define OSEVENTS 3
```

Events include semaphores (binary and counting), messages and message queues.

You do not need to use all the events that you allocate memory for, nor must you use their respective ecb pointers (numbered from `OSECBP(1)` to `OSECBP(OSEVENTS)`) consecutively. If you attempt to reference an event for which no memory was allocated, the Salvo user service will return a warning code.

If your application does not use events, leave `OSEVENTS` undefined in your `salvocfg.h`, or set it to 0.

Tip You should use descriptive names for events, too. See the tip above on how to do this.

Specifying other Configuration Options

You may also need to specify other configuration options, depending on which of Salvo's features you plan to use in your application. Many of Salvo's features are not available until they are enabled via a configuration option. This is done to minimize the size of the code that Salvo adds to your application. For small projects, a small `salvocfg.h` may be adequate. For larger projects and more complex applications, you will need to select the appropriate

[37] The `P` suffix is there to remind you that the object is a Pointer to something.

configuration option(s) for all the features you wish to use. Other configuration options include:

- • the size of delays, counters, etc. in bytes,
- • the size of semaphores and message pointers, and
- • memory-locating directives specific to the compiler.

Tip If you attempt to use a Salvo feature by calling a Salvo function and your compiler issues an error message suggesting that it can't find the function, this may be because the function has not been enabled via a configuration option.

In a sophisticated application, some of the additional configuration options might be:

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

The values for the options will either be numeric constants, predefined constants (e.g. `TRUE` and `FALSE`), or definitions provided for the compiler in use (e.g. `bank3`, used by the HI-TECH PICC compiler to locate variables in a particular bank of memory).

salvocfg.h Example – Salvo's Tut5 Application

Because the tutorial program is relatively simple, only a few configuration options need to be defined in `salvocfg.h`. By starting with an empty `salvocfg.h`, we begin with all configurations at their default values.

For three tasks and one event, we'll need the following `#define` directives.

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

Next, `Pumpkin\Salvo\Tut\Tut5` uses binary semaphores as a means of intertask communications. Binary Semaphore code is disabled by default, so we enable it with:

```
#define OSENABLE_BINARY_SEMAPHORES TRUE
```

Lastly, because we're using delays, we need to specify the size of possible delays.

```
#define OSBYTES_OF_DELAYS 1
```

This configuration option must be specified because Salvo defaults to no support for delays, which keeps RAM requirements to a minimum. Since `TaskBlink()` delays itself for 50 system ticks, a single byte is all that is required. With a byte for delays, each task could delay itself for up to 255 system ticks with a single call to `OS_Delay()`.

Note The `#defines` in `salvocfg.h` may appear in any order.

This four-line `salvocfg.h` is typical for small- to medium-sized programs of moderate complexity. The complete Salvo configuration file for this program can be found in `Pumpkin\Salvo\Tut\Tut5`. It is shown (with C comments removed[38]) in Listing 30.

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

Listing 30: salvocfg.h for Tutorial Program

Linking to Salvo Object Files

You can create an application by compiling and then linking your application to some or all of Salvo's `*.c` source files. This method is recommended for most applications, and is compatible with make utilities. It is relatively straightforward, but has the disadvantage that your final executable may contain all of the Salvo functionality contained in the linked files, regardless of whether your application uses them or not.

Note Some compilers are capable of "smart linking" whereby functions that are linked but not used do not make it into the final executable. In this situation there is no downside to linking your application to all of Salvo's source files.

[38] And without the additional configuration options that match those of the associated freeware library.

Chapter 7 • Reference contains descriptions of all the Salvo user services, and the Salvo source files that contain them. As soon as you use a service in your code, you'll also need to link to the appropriate source file. This is usually done in the compiler's IDE by adding the Salvo source files to your project. If you use the service without adding the file, you will get a link error when you make your project.

The size of each compiled object module is highly dependent on the configuration options you choose. Also, you can judiciously choose which modules to compile and link to – for example, if don't plan on using dynamic task priorities in your application, you can modify `salvocfg.h` appropriately and leave out `prio.c`, for a reduction in code size.

Tip The compiler-specific *Salvo Application Notes* describe in detail how to create applications for each compiler.

Complete source-code-based projects for certain tutorial programs can be found in `Pumpkin\Salvo\Tut`. See *Appendix C • File and Program Descriptions* for more information.

Chapter 5 • Configuration

Introduction

The Salvo source code contains configuration options that you can use to tailor its linkable object code to the specific needs of your application. These options are used to identify the compiler you're using and the processor you're compiling for, to configure Salvo for the number of tasks and events your application will require, and to enable or disable support for certain services. By selecting various configuration options you can fine-tune Salvo's abilities and performance to best match your application.

Note All configuration options are in the form of C preprocessor `#define` statements. They are therefore *compile-time* options. This means that they will not take effect until / unless you recompile each Salvo source code file that is affected by the configuration option.

The Salvo Build Process

Salvo applications are typically built in one of two ways – as a *library build*, or as a *source-code build*. Understanding Salvo's build process will aid in your understanding of how Salvo's configuration options are applied.

Note See your compiler's *Salvo Compiler Reference Manual* and the associated *Salvo Application Note(s)* for detailed information on creating and building Salvo projects.

Library Builds

Source-code builds are available in all Salvo distributions.

In a library build, a Salvo application is built from user source code (C and Assembly), from a precompiled Salvo library and from Salvo's `salvomem.c`. The user C source code makes calls to Salvo services that are contained in the Salvo library. Additionally, Salvo's global objects (i.e. its task control blocks, etc.) are in `Pump-`

kin\Salvo\Src\salvomem.c. Since the size of these objects is dependent on the application's numbers of tasks, events, etc., it must be re-compiled each time the project's Salvo configuration – defined in the project's salvocfg.h file – is changed.

Figure 25 presents an overview of the Salvo library build process.

In a library build, the configuration options in the project's salvocfg.h can only affect the user C source files and Salvo's salvomem.c. None of the Salvo services – contained in the Salvo library – are affected by the configuration options in salvocfg.h.

It is essential that the configuration options used to build the Salvo library match those applied to the user's C source files and to salvomem.c. Therefore part of the salvocfg.h for a library build (OSUSE_LIBRARY, OSLIBRARY_XYZ) is used to recreate the entire set of Salvo configuration options in place when the library was compiled. This is done automatically for the user by defining configuration options in salvolib.h based on the salvocfg.h settings, and by setting any undefined configuration options to their default values in salvo.h. The remaining configuration options in salvocfg.h simply set the sizes of Salvo's various global objects (e.g. the number of task control blocks). salvoclcN.h is included in the mix if a custom library is used.

For a successful library build, the chosen library must match the library options specified in salvocfg.h. See *Chapter 8 • Libraries* and your compiler's Salvo Compiler Reference Manual for more information on salvocfg.h for library builds.

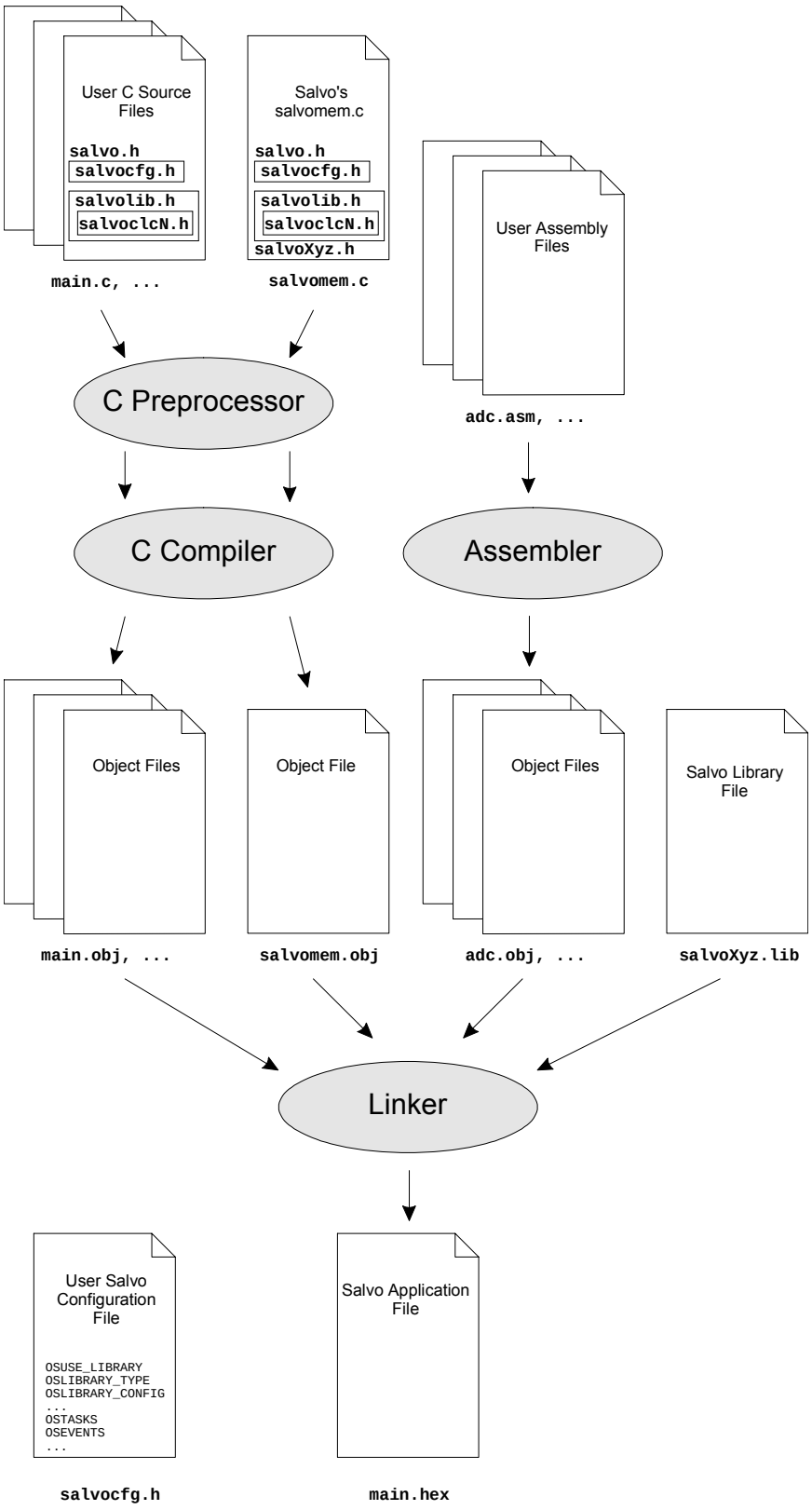

Figure 25: Salvo Library Build Overview

Source-Code Builds

Source-code builds are only available in Salvo Pro distributions.

In a source-code build, a Salvo application is built from user source code (C and Assembly) and from Salvo source code (C and Assembly, where applicable), including Salvo's `salvomem.c`. The user C source code makes calls to Salvo services that are contained in the Salvo source code. Again, Salvo's global objects (i.e. its task control blocks, etc.) are in `\Pumpkin\Salvo\Src\salvomem.c`. In a source-code build, all of Salvo's source-code modules must be re-compiled each time the project's Salvo configuration – defined in the project's `salvocfg.h` file – is changed.

Figure 26 presents an overview of the Salvo source-code build process.

In a source-code build, the configuration options in the project's `salvocfg.h` affect the user C source files and all of Salvo's C source files, where the desired user services are contained.

Each configuration option that the user wishes to set to a non-default value must be defined in `salvocfg.h`. All other configuration options are automatically set to their default values in `salvo.h`. As in a library build, certain configuration options (e.g. `OSTASKS`) set the sizes of Salvo's various global objects (e.g. the number of task control blocks).

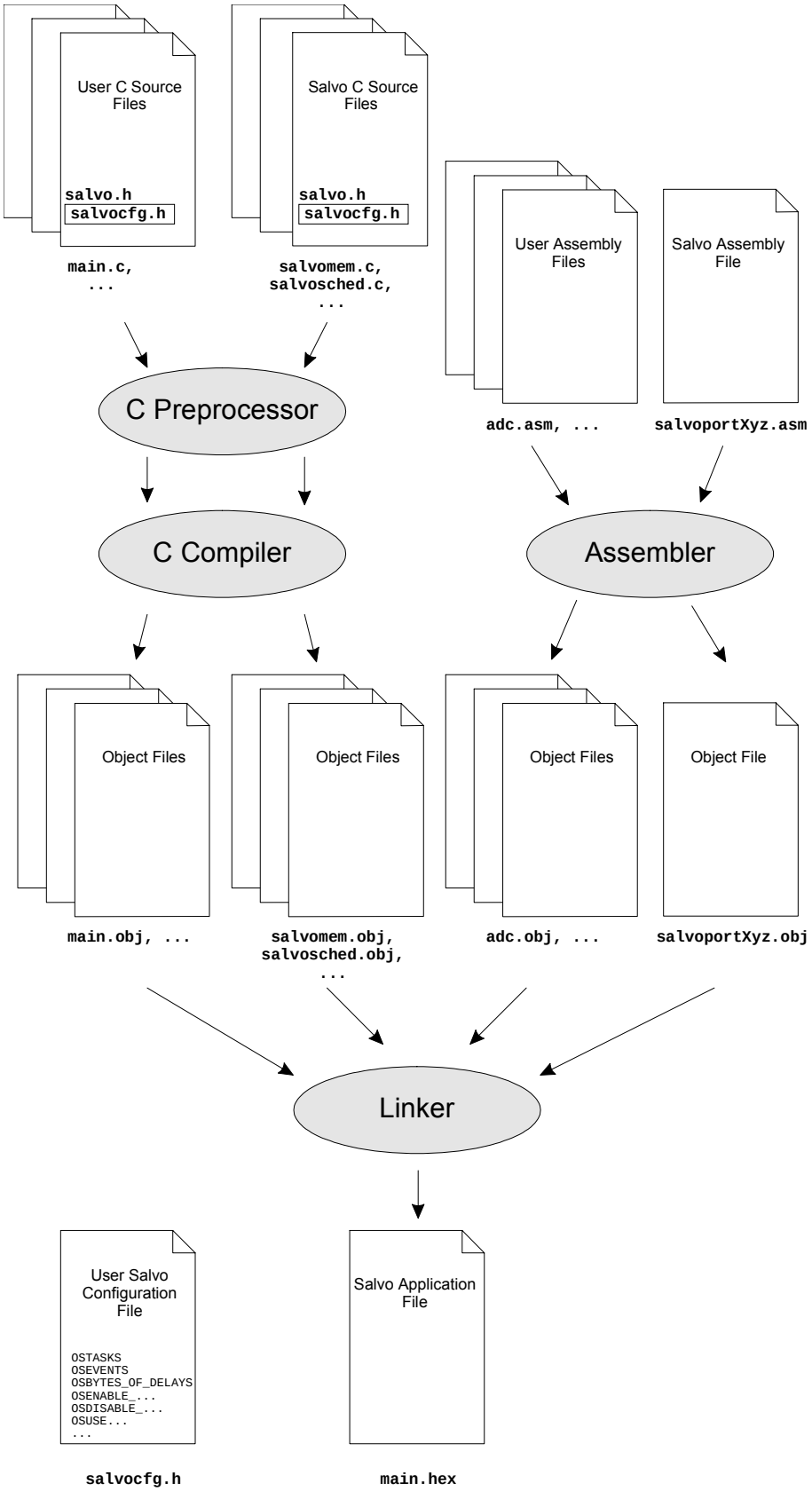

Figure 26: Salvo Source-Code Build Overview

Benefits of Different Build Types

Library builds have the advantage that all of the Salvo services are available in the library, and the linker will add only those necessary when building the application. The disadvantage is that if a different library configuration is required, both the `salvocfg.h` and the project file must be edited to ensure a match between the desired library and the library that linker sees.

With a source-code build, Salvo can be completely reconfigured just by simply adding or changing entries in `salvocfg.h`, and by adding the required Salvo source files to the project.

Note Salvo Pro is required for source-code builds.

Another benefit of library builds is that rebuilding a project within a Makefile-driven system is faster, since the library need not be rebuilt when allowable changes (e.g. changing the number of tasks) are made to `salvocfg.h`.

Configuration Option Overview

This section describes the Salvo configuration options. Each description includes information on:

- • the name of the configuration option,
- • the purpose of the configuration option,
- • the allowed values for the configuration option,
- • the default value for the configuration option,
- • the compile-time action that results from the configuration option,
- • related configuration options,
- • which user services are enabled by the configuration option,
- • how it affects memory requirements[39] and
- • notes particular to the configuration option.

You can fine-tune Salvo's capabilities, performance and size by choosing configuration options appropriate to your application.

[39] ROM requirements are described as small (e.g. a few lines of code in a single function) to considerable (e.g. a few lines of code in nearly every function).

Note All configuration options are contained in the user file `salvocfg.h`, and should not be placed in any other file(s). `salvocfg.h` should be located in the same directory as your application's source files. See *Chapter 4 • Tutorial* for more information on `salvocfg.h`.

Caution Whenever a configuration option is changed in `salvocfg.h`, you must recompile all of the Salvo files in your application. Failing to do so may result in unpredictable behavior or erroneous results.

Configuration Options for all Distributions

The configuration options described in this section can be used with:

- • Salvo Lite
- • Salvo tiny
- • Salvo SE
- • Salvo LE
- • Salvo Pro
- • Salvo Developer

and are listed in alphabetical order.

These configuration options affect the Salvo header (`*.h`) files, as well as `salvomem.c`.

OSCOMPILER: Identify Compiler in Use

Name:	OSCOMPILER
Purpose:	To identify the compiler you're using to generate your Salvo application.
Allowed Values:	see salvo.h
Default Value:	OSUNDEF, or automatically defined for certain compilers.
Action:	Configures Salvo source code for use with the selected compiler.
Related:	OSTARGET
Enables:	–
Memory Required:	n/a

Notes

This configuration option is used within the Salvo source code primarily to implement non-ANSI C directives like in-line assembly instructions and #pragma directives.

Salvo automatically detects the presence of nearly all of Salvo's supported compilers, and sets OSCOMPILER accordingly.[40] Therefore it is usually unnecessary to define OSCOMPILER in salvocfg.h.

If you are working with an as-yet-unsupported compiler, use OSUNDEF and refer to *Chapter 10 • Porting* for further instructions.

[40] OSCOMPILER can be overridden by setting it in salvocfg.h.

OSEVENTS: Set Maximum Number of Events

Name:	`OSEVENTS`
Purpose:	To allocate memory at compile time for event control blocks (ecbs), and to set an upper limit on the number of supported events.
Allowed Values:	`0` or greater.
Default Value:	`0`
Action:	Configures Salvo source code to support the desired number of events.
Related:	`OSENABLE_BINARY_SEMAPHORES`, `OSENABLE_EVENT_FLAGS`, `OSENABLE_EVENTS`, `OSENABLE_MESSAGES`, `OSENABLE_MESSAGE_QUEUES`, `OSENABLE_SEMAPHORES`, `OSEVENT_FLAGS`, `OSTASKS`, `OSMESSAGE_QUEUES`
Enables:	event-related services
Memory Required:	When non-zero, requires a configuration-dependent amount of RAM for each ecb.

Notes

Events (event flags, all semaphores, messages and message queues) are numbered from `1` to `OSEVENTS`.

Since event memory is allocated at compile time, the ecb memory will be used whether or not the event is actually created via `OSCreateBinSem|Eflag|Msg|MsgQ|Sem()`.

On a typical 8-bit processor, the amount of memory required by each event is 2-4 bytes[41] depending on which configuration options are enabled.

[41] For the purposes of these size estimates, pointers to ROM memory are assumed to be 16 bits, and pointers to RAM memory are assumed to be 8 bits. This is the situation for the PIC16 and PIC17 family of processors.

OSEVENT_FLAGS: Set Maximum Number of Event Flags

Name:	OSEVENT_FLAGS
Purpose:	To allocate memory at compile time for event flag control blocks (efcbs), and to set an upper limit on the number of supported event flags.
Allowed Values:	1 or greater.
Default Value:	1 if OSENABLE_EVENT_FLAGS is TRUE, 0 otherwise
Action:	Configures Salvo source code to support the desired number of event flags.
Related:	OSENABLE_EVENT_FLAGS, OSLOC_EFCB,
Enables:	-
Memory Required:	When non-zero, requires a configuration-dependent amount of RAM for each efcb.

Notes

This configuration parameter allocates RAM for event flag control blocks. Event flags require no other additional memory.

Event flags are numbered from 1 to OSEVENT_FLAGS.

Since event flag memory is allocated at compile time, the efcb memory will be used whether or not the event flag is actually created via OSCreateEFlag().

On a typical 8-bit processor, the amount of memory required by each event flag control block is represented by OSBYTES_OF_EVENT_FLAGS.

OSLIBRARY_CONFIG: Specify Precompiled Library Configuration

Name:	OSLIBRARY_CONFIG
Purpose:	To guarantee that an application's source files are compiled using the same salvocfg.h as was used to create the specified precompiled library.
Allowed Values:	OSA, OSD, OSE, OSM, OSS, OST, OSY
Default Value:	not defined
Action:	Sets the configuration options inside salvolib.h to match those used to generate the library specified.
Related:	OSLIBRARY_TYPE, OSLIBRARY_GLOBALS, OSLIBRARY_VARIANT, OSUSE_LIBRARY
Enables:	–
Memory Required:	n/a

Notes

OSLIBRARY_CONFIG is used in conjunction with OSLIBRARY_GLOBALS, OSLIBRARY_OPTION, OSLIBRARY_TYPE, OSLIBRARY_VARIANT and OSUSE_LIBRARY to properly specify the precompiled Salvo library you're linking to your project.

Library configurations might refer to, for example, whether the library is configured to support delays and/or events.

Please see your compiler's *Salvo Compiler Reference Manual* and *Chapter 8 • Libraries* for complete instructions on the use of OSLIBRARY_CONFIG.

See Also

OSUSE_LIBRARY.

OSLIBRARY_GLOBALS: Specify Memory Type for Global Salvo Objects in Precompiled Library

Name:	OSLIBRARY_GLOBALS
Purpose:	To guarantee that an application's source files are compiled using the same salvocfg.h as was used to create the specified precompiled library.
Allowed Values:	OSA …
Default Value:	not defined
Action:	Sets the configuration options inside salvolib.h to match those used to generate the library specified.
Related:	OSLIBRARY_TYPE, OSLIBRARY_CONFIG, OSLIBRARY_VARIANT, OSUSE_LIBRARY
Enables:	–
Memory Required:	n/a

Notes

OSLIBRARY_GLOBALS is used in conjunction with OSLIBRARY_CONFIG, OSLIBRARY_OPTION, OSLIBRARY_TYPE, OSLIBRARY_VARIANT and OSUSE_LIBRARY to properly specify the precompiled Salvo library you're linking to your project.

Library globals might refer to, for example, whether the library expects Salvo's global objects to be placed in internal or external RAM.

Please see your compiler's *Salvo Compiler Reference Manual* and *Chapter 8 • Libraries* for complete instructions on the use of OSLIBRARY_GLOBALS.

See Also

OSUSE_LIBRARY.

OSLIBRARY_OPTION: Specify Precompiled Library Option

Name:	OSLIBRARY_OPTION
Purpose:	To guarantee that an application's source files are compiled using the same salvocfg.h as was used to create the specified precompiled library.
Allowed Values:	OSA … or OSNONE
Default Value:	not defined
Action:	Sets the configuration options inside salvolib.h to match those used to generate the library specified.
Related:	OSLIBRARY_CONFIG, OSLIBRARY_GLOBALS, OSLIBRARY_VARIANT, OSUSE_LIBRARY
Enables:	–
Memory Required:	n/a

Notes

OSLIBRARY_OPTION is used in conjunction with OSLIBRARY_CONFIG, OSLIBRARY_GLOBALS, OSLIBRARY_TYPE, OSLIBRARY_VARIANT and OSUSE_LIBRARY to properly specify the precompiled Salvo library you're linking to your project.

Library options might refer to, for example, whether the library contains and/or supports embedded debugging information.

Please see your compiler's *Salvo Compiler Reference Manual* and *Chapter 8 • Libraries* for complete instructions on the use of OSLIBRARY_OPTION.

See Also

OSUSE_LIBRARY.

OSLIBRARY_TYPE: Specify Precompiled Library Type

Name:	OSLIBRARY_TYPE
Purpose:	To guarantee that an application's source files are compiled using the same salvocfg.h as was used to create the specified precompiled library.
Allowed Values:	OSF or OSL
Default Value:	not defined
Action:	Sets the configuration options inside salvolib.h to match those used to generate the library specified.
Related:	OSLIBRARY_CONFIG, OSLIBRARY_GLOBALS, OSLIBRARY_VARIANT, OSUSE_LIBRARY
Enables:	–
Memory Required:	n/a

Notes

OSLIBRARY_TYPE is used in conjunction with OSLIBRARY_CONFIG, OSLIBRARY_GLOBALS, OSLIBRARY_OPTION, OSLIBRARY_VARIANT and OSUSE_LIBRARY to properly specify the precompiled Salvo library you're linking to your project.

Library types normally refer to whether the library is a freeware library (OSF) or a standard library (OSL).

Please see your compiler's *Salvo Compiler Reference Manual* and *Chapter 8 • Libraries* for complete instructions on the use of OSLIBRARY_TYPE.

See Also

OSUSE_LIBRARY.

OSLIBRARY_VARIANT: Specify Precompiled Library Variant

Name:	OSLIBRARY_VARIANT
Purpose:	To guarantee that an application's source files are compiled using the same salvocfg.h as was used to create the specified precompiled library.
Allowed Values:	OSA … and OSNONE
Default Value:	not defined
Action:	Sets the configuration options inside salvolib.h to match those used to generate the library specified.
Related:	OSLIBRARY_CONFIG, OSLIBRARY_GLOBALS, OSLIBRARY_TYPE, OSUSE_LIBRARY
Enables:	–
Memory Required:	n/a

Notes

OSLIBRARY_VARIANT must be used in conjunction with OSLIBRARY_CONFIG, OSLIBRARY_GLOBALS, OSLIBRARY_OPTION, OSLIBRARY_TYPE and OSUSE_LIBRARY to properly specify the precompiled Salvo library you're linking to your project.

Library variants might refer to, for example, whether the library supports signaling events from within ISRs.

Not all libraries have variants. If a variant does not exist, set OSLIBRARY_VARIANT to OSNONE.

Please see your compiler's *Salvo Compiler Reference Manual* and *Chapter 8 • Libraries* for complete instructions on the use of OSLIBRARY_VARIANT.

See Also

OSUSE_LIBRARY.

OSMESSAGE_QUEUES: Set Maximum Number of Message Queues

Name:	`OSMESSAGE_QUEUES`
Purpose:	To allocate memory at compile time for message queue control blocks (mqcbs), and to set an upper limit on the number of supported message queues.
Allowed Values:	`1` or greater.
Default Value:	`1` if `OSENABLE_MESSAGE_QUEUES` is `TRUE`, `0` otherwise
Action:	Configures Salvo source code to support the desired number of message queues.
Related:	`OSENABLE_MESSAGE_QUEUES`, `OSLOC_MQCB`, `OSLOC_MSGQ`
Enables:	message-queue-related services
Memory Required:	When non-zero, requires a configuration-dependent amount of RAM for each mqcb.

Notes

This configuration parameter only allocates RAM for message queue control blocks. It does not allocate RAM for the message queues themselves – you must do that explicitly.

Message queues are numbered from `1` to `OSMESSAGE_QUEUES`.

Since message queue memory is allocated at compile time, the mqcb memory will be used whether or not the message queue is actually created via `OSCreateMsgQ()`.

On a typical 8-bit processor, the amount of memory required by each message queue control block is 6 bytes.

OSTARGET: Identify Target Processor

Name:	OSTARGET
Purpose:	To identify the processor you're using in your Salvo application.
Allowed Values:	see salvo.h
Default Value:	NONE
Action:	Configures Salvo source code for the target processor.
Related:	OSCOMPILER
Enables:	–
Memory Required:	n/a

Notes

This configuration option is used within the Salvo source code primarily to implement non-ANSI C directives like in-line assembly instructions and #pragma directives.

Nearly all of Salvo's supported compilers automatically override your settings and define OSTARGET based on the command-line arguments passed to the compiler to identify the processor. Therefore it is usually unnecessary to define OSTARGET in salvocfg.h.

If you are working with an as-yet-unsupported compiler, choose OSUNDEF. See *Chapter 10 • Porting* for more information.

OSTASKS: Set Maximum Number of Tasks and Cyclic Timers

Name:	`OSTASKS`
Purpose:	To allocate memory at compile time for task control blocks (tcbs), and to set an upper limit on the number of supported tasks and cyclic timers.
Allowed Values:	`1` or greater.
Default Value:	`0`
Action:	Configures Salvo source code to support the desired number of tasks and cyclic timers.
Related:	`OSEVENTS`
Enables:	general and task-related services
Memory Required:	When non-zero, requires a configuration-dependent amount of RAM for each tcb, and RAM for two tcb pointers.

Notes

Tasks and cyclic timers are numbered from `1` to `OSTASKS`. Each task and each cyclic timer requires one tcb.

Since task and cyclic timer memory is allocated and fixed at compile time, the tcb memory will be used whether or not the task is actually created via `OSCreateTask()` or the cyclic timer is created via `OSCreateCycTmr()`.

The amount of memory required by each task is dependent on several configuration options, and will range from a minimum of 4 to a maximum 12 bytes per task.[42]

[42] For the purposes of these size estimates, pointers to ROM memory are assumed to be 16 bits, and pointers to RAM memory are assumed to be 8 bits. This is the situation for the PIC16 and PIC17 family of processors.

OSUSE_LIBRARY: Use Precompiled Library

Name:	OSUSE_LIBRARY
Purpose:	To simplify linking to a precompiled Salvo library.
Allowed Values:	FALSE: you are not linking to a precompiled Salvo library. TRUE: you are linking to a precompiled Salvo library.
Default Value:	FALSE
Action:	If TRUE, the proper configuration options for the specified library will be used to build the application.
Related:	OSLIBRARY_CONFIG, OSLIBRARY_GLOBALS, OSLIBRARY_OPTION, OSLIBRARY_TYPE, OSLIBRARY_VARIANT
Enables:	–
Memory Required:	n/a

Notes

Salvo's configuration options are *compile-time options*. When linking to a precompiled library of Salvo services, the settings for your own application *must match* those originally used when the library was generated. OSUSE_LIBRARY, and the related OSLIBRARY_XYZ configuration options, take the guesswork out of creating a salvocfg.h header file for library builds.

Warning Failure to have matching configuration options may lead to compile- and link-time errors that can be difficult to interpret. Because of the large number of configuration options and their interrelationships, you *must* use OSUSE_LIBRARY and OSLIBRARY_XYZ when linking to precompiled Salvo libraries.

Configuration options used to create precompiled Salvo libraries differ from library to library. Please see your compiler's *Salvo Compiler Reference Manual* and *Chapter 8 • Libraries* for complete instructions on the use of OSUSE_LIBRARY and OSLIBRARY_XYZ.

Configuration Options for Source Code Distributions

The configuration options described in this section can only be used Salvo Pro and are listed in alphabetical order.

These configuration options affect the Salvo header (`*.h`) and source (`*.c`) files.

OSBIG_SEMAPHORES: Use 16-bit Semaphores

Name:	OSBIG_SEMAPHORES
Purpose:	To select 8- or 16-bit counting semaphores.
Allowed Values:	FALSE: Counting semaphores range from 0 to 255. TRUE: Counting semaphores range from 0 to 32,767.
Default Value:	FALSE
Action:	Changes the defined type OStypeSem from 8- to 16-bit unsigned integer.
Related:	–
Enables:	-
Memory Required:	When TRUE, requires an additional byte of RAM for each ecb.

Notes

This configuration option can be used to minimize the size of ecbs. Make OSBIG_SEMAPHORES TRUE only if your application requires 16-bit counting semaphores.

OSBIG_SEMAPHORES, when TRUE, will usually enlarge the size of ecbs by one byte on 8-bit targets.

OSBYTES_OF_COUNTS: Set Size of Counters

Name:	`OSBYTES_OF_COUNTS`
Purpose:	To allocate the RAM needed to hold the maximum possible value for counters used in Salvo, and to enable the code to run the counters.
Allowed Values:	`0`, `1`, `2`, `4`
Default Value:	`0`
Action:	If zero, disables all counters. If non-zero, enables the counters `OSctxSws` and `OSidleCtxSws`, and sets the defined type `OStypeCount` to be 8-, 16-, or 32-bit unsigned integer.
Related:	`OSGATHER_STATISTICS`
Enables:	–
Memory Required:	When non-zero, requires RAM for all enabled counters.

Notes

Salvo uses simple counters to keep track of context switches and notable occurrences. Once a counter reaches its maximum value it remains at that value.

OSBYTES_OF_DELAYS: Set Length of Delays

Name:	`OSBYTES_OF_DELAYS`
Purpose:	To enable delays and timeout services and to allocate the RAM needed to hold the maximum specified value (in system ticks) for delays and timeouts.
Allowed Values:	`0`, `1`, `2`, `4`
Default Value:	`0`
Action:	If zero, disables all delay and timeout services. If non-zero, enables the delay and timeout services, and sets the defined type `OStypeDelay` to be 8-, 16- or 32-bit unsigned integer.
Related:	`OSTIMER_PRESCALAR`
Enables:	`OS_Delay()`, `OSTimer()`
Memory Required:	When non-zero, requires 1, 2 or 4 additional bytes of RAM for each tcb and 1 tcb pointer in RAM.

Notes

Disabling delays and timeouts will reduce the size of the Salvo code considerably. It will also reduce the size of the tcbs by 2 to 6 bytes per tcb.

Use of `OSTIMER_PRESCALAR` in conjunction with `OSBYTES_OF_DELAYS` can provide for very long delays and timeouts while minimizing tcb memory requirements.

OSBYTES_OF_EVENT_FLAGS: Set Size of Event Flags

Name:	`OSBYTES_OF_EVENT_FLAGS`
Purpose:	To select 8-, 16- or 32-bit event flags.
Allowed Values:	`1`, `2`, `4`
Default Value:	`1`
Action:	Sets the defined type `OStypeEFlag` to 8-, 16- or 32-bit unsigned integer.
Related:	`OSENABLE_EVENT_FLAGS`
Enables:	–
Memory Required:	When event flags are enabled, requires 1, 2 or 4 bytes of RAM for each event flag control block (efcb) and additional ROM (code) dependent on the target processor.

Notes

You can tailor the size of event flags in your Salvo application via this configuration parameter.

Since each bit is independent of the others, it may be to your advantage to have a single, large event flag instead of multiple, smaller ones. For example, the RAM requirements for two 8-bit event flags will exceed those for a single 16-bit event flag since the former requires two event control blocks, whereas the latter needs only one.

OSBYTES_OF_TICKS: Set Maximum System Tick Count

Name:	`OSBYTES_OF_TICKS`
Purpose:	To enable elapsed time services and to allocate the RAM needed to hold the maximum specified system ticks value.
Allowed Values:	`0`, `1`, `2`, `4`
Default Value:	`0`
Action:	If zero, disables all elapsed time services. If non-zero, enables the services , and sets the defined type `OStypeTick` to be 8-, 16- or 32-bit unsigned integer.
Related:	`OSTIMER_PRESCALAR`
Enables:	`OSGetTicks()`, `OSSetTicks()`, `OSTimer()`
Memory Required:	When non-zero, requires RAM for the system tick counter.

Notes

Salvo uses a simple counter to keep track of system ticks. After it reaches its maximum value the counter rolls over to 0.

Elapsed time services based on the system tick are obtained through `OSGetTicks()` and `OSSetTicks()`.

`OSBYTES_OF_TICKS` must be greater or equal to `OSBYTES_OF_DELAYS`.

OSCALL_OSCREATEEVENT: Manage Interrupts when Creating Events

Name:	`OSCALL_OSCREATEEVENT`
Purpose:	For use on target processors without software stacks in order to manage for interrupts when calling event-creating services.
Allowed Values:	`OSFROM_BACKGROUND`: Your application creates events only in mainline code. `OSFROM_FOREGROUND`: Your application creates events only within interrupts. `OSFROM_ANYWHERE` Your application creates events both in mainline code and within interrupts. You must explicitly control interrupts around `OSCALL_OSCREATEEVENT` (see below).
Default Value:	`OSFROM_BACKGROUND`
Action:	Configures the interrupt control for all Salvo event-creating services.
Related:	`OSCALL_OSSIGNALEVENT`, `OSCALL_OSRETURNEVENT`
Enables:	–
Memory Required:	Small variations in ROM depending on its value.

Notes

`OSCALL_OSCREATEEVENT` is required *only* when using a compiler that does not maintain function parameters and auto variables on a software stack or in registers. Therefore this configuration parameter and all similar ones are only needed when using certain target processors and compilers.

Compilers that maintain function parameters and auto variables in a dedicated area of RAM usually do so because a software stack and stack pointers *do not exist* on the target processor. In order to minimize RAM usage, these compilers[43] *overlay* the parameter and variable areas of multiple functions as long as the functions do not occupy the same *call graph*. This is all done transparently – no user involvement is required.

The issue is complicated by wanting to call Salvo services from both mainline (background) and interrupt (foreground) code. In this case, each service needs its own parameter and auto variable

[43] E.g. the HI-TECH PICC and V8C compilers.

area separate from that of mainline-only services, and the user must "wrap" each mainline service with calls to disable and then re-enable interrupts[44] in order to avoid data corruption. See the examples below.

The control of interrupts in each event-creating service like `OSCreateBinSem()` depends on where it is called in your application. In Figure 27 interrupts will be disabled and re-enabled inside `OSCreateBinSem()`. This is referred to as *protecting a critical region of code*, and is typical of RTOS services. In this situation, `OSCALL_OSCREATEEVENT` must be set to `OSFROM_BACKGROUND`.

```
int main( void )
{
  …
  OSCreateBinSem(BINSEM1_P);
  …
}
```

Figure 27: How to call OSCreateBinSem() when OSCALL_OSCREATEEVENT is set to OSFROM_BACKGROUND

In Figure 28 `OSCreateBinSem()` must not change the processor's interrupt status, because re-enabling interrupts within an ISR can cause unwanted nested interrupts. In this situation, set `OSCALL_OSCREATEEVENT` to `OSFROM_FOREGROUND`.

```
interrupt myISR( void )
{
  …
  if (some_condition) {
    OSCreateBinSem(BINSEM2_P);
  }
  …
}
```

Figure 28: How to call OSCreateBinSem() when OSCALL_OSCREATEBINSEM is set to OSFROM_FOREGROUND

In Figure 29, `OSCreateBinSem()` is called from the background as well as the foreground. In this situation, `OSCALL_OSCREATEEVENT` must be set to `OSFROM_ANYWHERE` and `OSCreateBinSem()` must be preceded by `OSProtect()` and followed by `OSUnprotect()` wherever it's called in mainline (background) code.

```
int main( void )
{
  …
```

[44] See "Interrupt Levels" in the HI-TECH PICC and PICC-18 User's Guide.

```
  OSProtect();
  OSCreateBinSem(BINSEM1_P);
  OSUnprotect();
  …
  OSProtect();
  OSCreateBinSem(BINSEM2_P);
  OSUnprotect();
  …
}

interrupt myISR( void )
{
  …
  if (some_condition) {
    OSCreateBinSem(BINSEM2_P);
  }
  …
}
```

Figure 29: How to call OSCreateBinSem() when OSCALL_CREATEBINSEM is set to OSFROM_ANYWHERE

Failing to set `OSCALL_OSCREATEEVENT` properly to reflect where you are calling `OSCreateBinSem()` in your application may cause unpredictable results, and may also result in compiler errors.

With some compilers (e.g. HI-TECH PICC), `OSCALL_OSCREATEEVENT` also automatically enables certain special directives[45] in the Salvo source code to ensure proper compilation.

[45] E.g. `#pragma interrupt_level 0`, to allow a function to be called both from mainline code and from an interrupt. In this situation a function has "multiple call graphs."

OSCALL_OSGETPRIOTASK: Manage Interrupts when Returning a Task's Priority

`OSCALL_OSGETPRIOTASK` manages how interrupts are controlled in `OSGetPrio()` and `OSGetPrioTask()`.

See `OSCALL_OSCREATEEVENT` for more information on interrupt control for services that can be called from the foreground.

OSCALL_OSGETSTATETASK: Manage Interrupts when Returning a Task's State

`OSCALL_OSGETSTATETASK` manages how interrupts are controlled in `OSGetState()` and `OSGetStateTask()`.

See `OSCALL_OSCREATEEVENT` for more information on interrupt control for services that can be called from the foreground.

OSCALL_OSMSGQCOUNT: Manage Interrupts when Returning Number of Messages in Message Queue

`OSCALL_OSMSGQCOUNT` manages how interrupts are controlled in `OSMsgQCount()`.

See `OSCALL_OSCREATEEVENT` for more information on interrupt control for services that can be called from the foreground.

OSCALL_OSMSGQEMPTY: Manage Interrupts when Checking if Message Queue is Empty

`OSCALL_OSMSGQEMPTY` manages how interrupts are controlled in `OSMsgQEmpty()`.

See `OSCALL_OSCREATEEVENT` for more information on interrupt control for services that can be called from the foreground.

OSCALL_OSRETURNEVENT: Manage Interrupts when Reading and/or Trying Events

`OSCALL_OSRETURNEVENT` manages how interrupts are controlled in event-reading and event-trying services (e.g. `OSReadEFlag()` and `OSTrySem()`, respectively).

See `OSCALL_OSCREATEEVENT` for more information on interrupt control for event-reading and event-trying services.

OSCALL_OSSIGNALEVENT: Manage Interrupts when Signaling Events and Manipulating Event Flags

`OSCALL_OSSIGNALEVENT` manages how interrupts are controlled in event-signaling services (e.g. `OSSignalMsg()`), `OSClrEFlag()` and `OSSetEFlag()`.

See `OSCALL_OSCREATEEVENT` for more information on interrupt control for event-signaling services.

OSCALL_OSSTARTTASK: Manage Interrupts when Starting Tasks

`OSCALL_OSSTARTTASK` manages how interrupts are controlled in `OSStartTask()`.

See `OSCALL_OSCREATEEVENT` for more information on interrupt control for event-signaling services.

OSCLEAR_GLOBALS: Explicitly Clear all Global Parameters

Name:	OSCLEAR_GLOBALS
Purpose:	To guarantee that all global variables used by Salvo are explicitly initialized to zero.
Allowed Values:	FALSE, TRUE
Default Value:	TRUE
Action:	If TRUE, configures OSInit() to explicitly fill all global variables (e.g. queue pointers, tcbs, ecbs, etc.) with 0.
Related:	OSENABLE_EVENTS, OSENABLE_STACK_CHECKING
Enables:	OSInitTcb() and OSInitEcb() for some values of OSCOMPILER.
Memory Required:	When TRUE, requires a small amount of ROM.

Notes

All ANSI C compilers must initialize global variables to zero. OSInit() clears Salvo's variables by default. For those applications where ROM memory is extremely precious, this configuration option can be disabled, and your application may shrink somewhat as a result.

Caution If you disable this configuration option you must be absolutely sure that your compiler explicitly initializes all of Salvo's global variables to zero. Otherwise your application may not work properly. Even if your compiler does zero all global variables, keep in mind that OSInit() will no longer (re-)zero the global variables, and you will not be able to re-initialize Salvo via a call to OSInit().

OSCLEAR_UNUSED_POINTERS: Reset Unused Tcb and Ecb Pointers

Name:	OSCLEAR_UNUSED_POINTERS
Purpose:	To aid in debugging Salvo activity.
Allowed Values:	FALSE: Salvo makes no attempt to reset no-longer used pointers in tcbs and ecbs. TRUE: Salvo resets all unused tcb and ecb pointers to NULL.
Default Value:	FALSE
Action:	When TRUE, enables code to null unused tcb and ecb pointers.
Related:	OSBYTES_OF_DELAYS, OSENABLE_TIMEOUTS,
Enables:	–
Memory Required:	When TRUE, requires a small amount of ROM.

Notes

This configuration option is primarily of use to you if you are interested in viewing or debugging Salvo internals. It is much easier to understand the status of the queues, tasks and events if the unused pointers are NULLed.

Enabling this configuration option will add a few instructions to certain Salvo services.

OSCOLLECT_LOST_TICKS: Configure Timer System For Maximum Versatility

Name:	`OSCOLLECT_LOST_TICKS`
Purpose:	To avoid delay- and timeout-related tick errors due to poor task yielding behavior.
Allowed Values:	`FALSE`, `TRUE`
Default Value:	`TRUE`
Action:	Configures Salvo source code to log up to a maximum number of ticks in the timer for later delay and timeout processing in the scheduler.
Related:	`OSBYTES_OF_DELAYS`, `OSBYTES_OF_TICKS`, `OSENABLE_TIMEOUTS`
Enables:	–
Memory Required:	Target- and compiler-dependent. In most cases, should reduce ROM requirements slightly.

Notes

When `OSCOLLECT_LOST_TICKS` is `FALSE`, `OSTimer()` can log only a single tick per call for eventual processing in the scheduler `OSSched()`. If, for example, an application has tasks that fail to yield back to the scheduler within 2 system ticks, any tasks delayed or waiting with a timeout during this period will appear to have their delays or timeouts lengthened by the amount of time the poorly-behaved task(s) fails to yield to the scheduler.

When `OSCOLLECT_LOST_TICKS` is `TRUE`, `OSTimer()` can log up to 255 ticks for eventual processing in the scheduler. In the above example, the error in the delays or timeouts of simultaneously delayed or waiting tasks will be minimized.

`OSCOLLECT_LOST_TICKS` has no effect on the system's free-running system tick counter `OStimerTicks`, which is accessed via `OSGetTicks()` and `OSSetTicks()`.

OSCOMBINE_EVENT_SERVICES: Combine Common Event Service Code

Name:	OSCOMBINE_EVENT_SERVICES
Purpose:	To minimize code size with multiple event types enabled.
Allowed Values:	FALSE: All event services are implemented as separate, independent functions. TRUE: Event services use common code where possible.
Default Value:	FALSE
Action:	Changes the structure of the Salvo source code to produce minimum aggregate or individual size of event services.
Related:	–
Enables:	–
Memory Required:	When TRUE, reduces ROM requirements when event services for two or more event types are used.

Notes

The services for creating, signaling and waiting events contain common source code. When OSCOMBINE_EVENT_SERVICES is TRUE, event services use that common code, e.g. OSCreateBinSem() and OSCreateMsgQ() use the same underlying function. This means that the incremental increase in size of the object code is relatively small when another event type is enabled via OSENABLE_XYZ.

When OSCOMBINE_EVENT_SERVICES is FALSE, each event service is implemented as a separate, independent function, and some code is therefore duplicated. This is used when generating the Salvo freeware libraries for maximum versatility.

When creating an application using two or more event types, the aggregate size of all of the event services will be smaller when OSCOMBINE_EVENT_SERVICES is TRUE.

The C language va_arg() and related functions are required when OSCOMBINE_EVENT_SERVICES is TRUE.

Setting OSCOMBINE_EVENT_SERVICES to TRUE with HI-TECH 8051C and the small or medium memory models will prevent you from calling any allowed event services (e.g. OSSignalMsg()) from an ISR. This restriction is lifted in the large model.

OSCTXSW_METHOD: Identify Context-Switching Methodology in Use

Name:	`OSCTXSW_METHOD`
Purpose:	To configure the inner workings of the Salvo context switcher.
Allowed Values:	`OSRTNADDR_IS_PARAM`: `OSSaveRtnAddr()` is passed the task's return address as a parameter. `OSRTNADDR_IS_VAR`: `OSSaveRtnAddr()` reads the tasks's return address through a global variable. `OSVIA_OSCTXSW`: `OSCtxSw()` is used to return to the scheduler. `OSVIA_OSDISPATCH`: `OSCtxSw()` is used in conjunction with `OSDispatch()`.
Default Value:	Defined for each compiler and target in `portXyz.h`. If left undefined, default is `OSRTNADDR_IS_PARAM`.
Action:	Configures Salvo source code for use with the selected compiler and target processor.
Related:	`OSRTNADDR_OFFSET`
Enables:	–
Memory Required:	When set to `OSRTNADDR_IS_VAR`, requires a small amount of RAM. ROM requirements vary.

Notes

This configuration option is used within the Salvo source code to implement part of the context switcher `OS_Yield()`.

Warning Unless you are porting Salvo to an as-yet-unsupported compiler, do not override the value of `OSCTXSW_METHOD` in the porting file `salvoportXyz.h` appropriate for your compiler. Unpredictable results will occur.

If you are working with an as-yet-unsupported compiler, refer to the Salvo source code and *Chapter 10 • Porting* for further instructions.

OSCUSTOM_LIBRARY_CONFIG: Select Custom Library Configuration File

Name:	`OSCUSTOM_LIBRARY_CONFIG`
Purpose:	To simply the generation and use of custom Salvo libraries.
Allowed Values:	`0`, `1` through `20`[46]
Default Value:	`0` (i.e. no custom library is selected)
Action:	Configures Salvo source code to include the specified custom library configuration file.
Related:	`salvoclc1.h` through `salvoclc20.h`
Enables:	–
Memory Required:	n/a

Notes

`OSCUSTOM_LIBRARY_CONFIG` is used to ensure that the Salvo configuration for projects built with custom libraries matches the configuration that was in effect when the library was generated.

This configuration option need only be used when creating and using custom user libraries. There is no need to use `OSCUSTOM_LIBRARY_CONFIG` when the freeware or standard libraries supplied in a Salvo distribution are used.

See *Chapter 8 • Libraries* for detailed information on using `OSCUSTOM_LIBRARY_CONFIG`.

[46] Values in excess of 20 will result in an error message when building a Salvo library or application. Can be extended to larger values if need be – see `salvo/inc/salvolib.h`.

OSDISABLE_ERROR_CHECKING: Disable Runtime Error Checking

Name:	`OSDISABLE_ERROR_CHECKING`
Purpose:	To turn off runtime error checking.
Allowed Values:	`FALSE`: Error checking is enabled. `TRUE`: Error checking is disabled.
Default Value:	`FALSE`
Action:	Disables certain error checking in some Salvo user services.
Related:	–
Enables:	–
Memory Required:	When `FALSE`, requires ROM for error-checking.

Notes

By default, Salvo performs run-time error checking on certain parameters passed to user services, like task priorities.

This error checking can be costly in terms of code space (ROM) used. It can be disabled by setting `OSDISABLE_ERROR_CHECKING` to `TRUE`. However, this is never recommended.

Caution Disabling error checking is strongly discouraged. It should only be used as a last resort in an attempt to shrink code size, with the attendant knowledge that any run-time error that goes unchecked may result in unpredictable behavior.

OSDISABLE_FAST_SCHEDULING: Configure Round-Robin Scheduling

Name:	`OSDISABLE_FAST_SCHEDULING`
Purpose:	To alter execution sequence of tasks running in a round-robin manner.
Allowed Values:	`FALSE`: Fast scheduling is used. `TRUE`: Fast scheduling is not used.
Default Value:	`FALSE`
Action:	Changes the way in which eligible tasks returning to the scheduler are re-enqueued into the eligible queue.
Related:	–
Enables:	–
Memory Required:	When `TRUE`, requires a small amount of additional ROM.

Notes

By default, the Salvo scheduler immediately re-enqueues the current task upon its return to the scheduler if it is still eligible. This has a side effect on round-robin scheduling that is best illustrated by example.

If `OSDISABLE_FAST_SCHEDULING` is `FALSE` and the current task signals an event upon which another task of equal priority is waiting, then the scheduler will run the signaling task again *before* the waiting task.[47] On the other hand, if `OSDISABLE_FAST_SCHEDULING` is `TRUE` in this situation, then the scheduler will run the waiting task before the signaling task. In other words, the round-robin sequence of task execution matches the order in which the tasks are made eligible if `OSDISABLE_FAST_SCHEDULING` is set to `TRUE`.

Setting `OSDISABLE_FAST_SCHEDULING` to `TRUE` will have a small but significant negative impact on the context-switching speed of your application.

[47] This is indirectly related to the minimal stack depth required by `OSSignalXyz()` services.

OSDISABLE_TASK_PRIORITIES: Force All Tasks to Same Priority

Name:	`OSDISABLE_TASK_PRIORITIES`
Purpose:	To reduce code (ROM) size when an application does not require prioritized tasks.
Allowed Values:	`FALSE`: Tasks can have assigned priorities. `TRUE`: All tasks have same priority (0).
Default Value:	`FALSE`
Action:	Removes priority-setting and priority-dependent code from Salvo services.
Related:	–
Enables:	–
Memory Required:	When `FALSE`, requires ROM for management of task priorities.

Notes

By default, Salvo schedules task execution based on task priorities. Some savings in ROM size can be realized by disabling Salvo's priority-specific code. When `OSDISABLE_TASK_PRIORITIES` is set to `TRUE`, all tasks run at the same priority and round-robin.

OSENABLE_BINARY_SEMAPHORES: Enable Support for Binary Semaphores

Name:	`OSENABLE_BINARY_SEMAPHORES`
Purpose:	To control compilation of binary semaphore code via the preprocessor.
Allowed Values:	`FALSE`, `TRUE`
Default Value:	`FALSE`
Action:	If `FALSE`, binary semaphore services are not available. If `TRUE`, `OSCreateBinSem()`, `OSSignalBinSem()`and `OS_WaitBinSem()` are available.
Related:	`OSENABLE_EVENT_FLAGS`, `OSENABLE_MESSAGES`, `OSENABLE_MESSAGE_QUEUES`, `OSENABLE_SEMAPHORES`, `OSEVENTS`
Enables:	–
Memory Required:	When `TRUE`, requires ROM for binary semaphore services.

Notes

This configuration option is useful when controlling which parts of Salvo are to be included in an application. If you are including or linking to `salvobinsem.c` in your source code, you can control its compilation solely via this configuration option in `salvocfg.h`. This may be more convenient than, say, editing your source code or modifying your project.

OSENABLE_BOUNDS_CHECKING: Enable Runtime Pointer Bounds Checking

Name:	OSENABLE_BOUNDS_CHECKING
Purpose:	To check for out-of-range pointer arguments.
Allowed Values:	FALSE, TRUE
Default Value:	FALSE
Action:	If FALSE, pointer arguments are not bounds-checked. If TRUE, some services return an error if the pointer argument is out-of-bounds.
Related:	OSDISABLE_ERROR_CHECKING, OSSET_LIMITS
Enables:	–
Memory Required:	When TRUE, requires ROM for pointer bounds checking.

Notes

The result of passing an incorrect pointer to a service is unpredictable. Some protection can be achieved by bounds-checking the pointer to ensure that it is within a valid range of pointer values appropriate for the service. This can be useful when debugging an application that uses variables as placeholders for pointers instead of constants.

The utility of runtime pointer bounds checking is limited. Since valid pointers do not have successive addresses, the allowed range includes not only the valid pointer values but also all the other values within that range. Therefore runtime pointer bounds checking will only detect a small subset of invalid pointer arguments.

OSENABLE_BOUNDS_CHECKING is overridden (i.e. set to TRUE) when OSSET_LIMITS is set to TRUE.

OSENABLE_CYCLIC_TIMERS: Enable Cyclic Timers

Name:	`OSENABLE_CYCLIC_TIMERS`
Purpose:	To control compilation of cyclic timer code via the preprocessor.
Allowed Values:	`FALSE`, `TRUE`
Default Value:	`FALSE`
Action:	If `FALSE`, cyclic timer services are not available. If `TRUE`, cyclic timer services are available.
Related:	–
Enables:	–
Memory Required:	When `TRUE`, requires ROM and in some cases, tcb RAM.

Notes

This configuration option is useful when controlling which parts of Salvo are to be included in an application. If you are including or linking to any of the `salvocyclicN.c` source files in your source code, you can control their compilation solely via this configuration option in `salvocfg.h`. This may be more convenient than, say, editing your source code or modifying your project.

OSENABLE_EVENT_FLAGS: Enable Support for Event Flags

Name:	OSENABLE_EVENT_FLAGS
Purpose:	To control compilation of event flag code via the preprocessor.
Allowed Values:	FALSE, TRUE
Default Value:	FALSE
Action:	If FALSE, event flag services are not available. If TRUE, OSCreateEFlag(), OSClrEFlag(), OSSetEFlag()and OS_WaitEFlag() are available.
Related:	OSBYTES_OF_EVENT_FLAGS, OSENABLE_BINARY_SEMAPHORES, OSENABLE_MESSAGES, OSENABLE_MESSAGE_QUEUES, OSENABLE_SEMAPHORES, OSEVENTS, OSEVENT_FLAGS
Enables:	–
Memory Required:	When TRUE, requires ROM for event flag services.

Notes

This configuration option is useful when controlling which parts of Salvo are to be included in an application. If you are including or linking to salvoeflag.c in your source code, you can control its compilation solely via this configuration option in salvocfg.h. This may be more convenient than, say, editing your source code or modifying your project.

A value of 0 for OSEVENT_FLAGS automatically resets (overrides) OSENABLE_EVENT_FLAGS to FALSE.

OSENABLE_EVENT_READING: Enable Support for Event Reading

Name:	OSENABLE_EVENT_READING
Purpose:	To control compilation of event-reading code via the preprocessor.
Allowed Values:	FALSE, TRUE
Default Value:	FALSE
Action:	If FALSE, event-reading services are not available. If TRUE, OSReadBinSem(), OSReadEFlag(), OSReadMsg(), OSReadMsgQ()and OSReadSem() are available.
Related:	OSCALL_OSRETURNEVENT, OSENABLE_EVENT_TRYING
Enables:	–
Memory Required:	When TRUE, requires ROM for event-reading services.

Notes

If you use any event-reading services (e.g. OSReadMsg()), you must set OSENABLE_EVENT_READING to TRUE in salvocfg.h. If you do not use any event-reading services, leave it at is default value of FALSE.

This configuration option is useful when controlling which parts of Salvo are to be included in an application. If you are including Salvo event source code in your project, you can keep unused event-reading services out of your final object file solely via this configuration option in salvocfg.h. This may be more convenient than, say, editing your source code or modifying your project.

A value of TRUE for OSENABLE_EVENT_TRYING automatically sets (overrides) OSENABLE_EVENT_READING to TRUE.

OSENABLE_EVENT_TRYING: Enable Support for Event Trying

Name:	OSENABLE_EVENT_TRYING
Purpose:	To control compilation of event-trying code via the preprocessor.
Allowed Values:	FALSE, TRUE
Default Value:	FALSE
Action:	If FALSE, event-trying services are not available. If TRUE, OSTryBinSem(), OSTryMsg(), OSTryMsgQ()and OSTry-Sem() are available.
Related:	OSCALL_OSRETURNEVENT, OSENABLE_EVENT_READING
Enables:	–
Memory Required:	When TRUE, requires ROM for event-trying services.

Notes

If you use any event-trying services (e.g. OSTrySem()), you must set OSENABLE_EVENT_TRYING to TRUE in salvocfg.h. If you do not use any event-trying services, leave it at is default value of FALSE.

This configuration option is useful when controlling which parts of Salvo are to be included in an application. If you are including Salvo event source code in your project, you can keep unused event-trying services out of your final object file solely via this configuration option in salvocfg.h. This may be more convenient than, say, editing your source code or modifying your project.

A value of TRUE for OSENABLE_EVENT_TRYING automatically sets (overrides) OSENABLE_EVENT_READING to TRUE.

OSENABLE_FAST_SIGNALING: Enable Fast Event Signaling

Name:	`OSENABLE_FAST_SIGNALING`
Purpose:	To increase the rate at which events can be signaled.
Allowed Values:	`FALSE`, `TRUE`
Default Value:	`FALSE`
Action:	If `FALSE`, signaled events are processed[48] when the waiting task runs. If `TRUE`, signaled events are processed when the event is signaled.
Related:	–
Enables:	–
Memory Required:	When `TRUE`, requires a moderate amount of additional ROM, and extra tcb RAM for messages and message queues.

Notes

With `OSENABLE_FAST_SIGNALING` set to `FALSE`, when an event is signaled and a task was waiting the event, the event remains signaled until the waiting task runs. For example, when a binary semaphore is signaled with `TaskA()` waiting, `OSSignalBinSem()` will return `OSERR_EVENT_FULL` if called again before `TaskA()` runs. When `TaskA()` runs, the binary semaphore is reset to 0, and a subsequent call to `OSSignalBinSem()` will succeed. On the other hand, if `OSENABLE_FAST_SIGNALING` is `TRUE`, the binary semaphore will immediately return to zero when `TaskA()` is made eligible by `OSSignalBinSem()`, and thereafter the binary semaphore can be signaled again without error.

Fast signaling is useful when multiple tasks are waiting an event, or the same event is signaled in rapid succession. In these situations, `OSSignalXyz()` will succeed until no tasks are waiting the event and the event has been signaled.

[48] E.g. a semaphore is decremented.

OSENABLE_IDLE_COUNTER: Track Scheduler Idling

Name:	OSENABLE_IDLE_COUNTER
Purpose:	To count how many times the scheduler has been idle.
Allowed Values:	FALSE: Salvo does not keep track of how often the scheduler OSSched() is idle. TRUE: The OSidleCtxSw counter is incremented each time the scheduler is called with no eligible tasks, i.e. the system is idle.
Default Value:	FALSE
Action:	If TRUE, configures Salvo to track scheduler idling.
Related:	OSGATHER_STATISTICS, OSENABLE_IDLING_HOOK
Enables:	–
Memory Required:	When TRUE, requires a small amount of ROM, plus one byte of RAM.

Notes

If OSGATHER_STATISTICS, OSENABLE_COUNTS and OSENABLE_IDLE_COUNTER are all TRUE, and Salvo's idling hook function is enabled via OSENABLE_IDLING_HOOK, then the OSidleCtxSws counter will be incremented each time the scheduler is called and there are no tasks eligible to run. The percentage of time your application is spending idle can be obtained by:

idle time = (OSidleCtxSws / OSctxSws) x 100

OSENABLE_IDLING_HOOK: Call a User Function when Idling

Name:	`OSENABLE_IDLING_HOOK`
Purpose:	To provide a simple way of calling a user function when idling.
Allowed Values:	`FALSE`: No function is called when idling. `TRUE`: An external user hook function named `OSIdlingHook()` is called when idling.
Default Value:	`FALSE`
Action:	If `TRUE`, `OSSched()` calls `OSIdling-Hook()` when no tasks are eligible to run.
Related:	–
Enables:	–
Memory Required:	When `TRUE`, requires a small amount of ROM.

Notes

When you enable this both configuration, you must also define an external function `void OSIdlingHook(void)`. It will be called automatically when your Salvo application is idling.

OSENABLE_MESSAGES: Enable Support for Messages

Name:	`OSENABLE_MESSAGES`
Purpose:	To control compilation of message code via the preprocessor.
Allowed Values:	`FALSE`, `TRUE`
Default Value:	`FALSE`
Action:	If `FALSE`, message services are not available. If `TRUE`, `OSCreateMsg()`, `OSSignalMsg()` and `OS_WaitMsg()` are available.
Related:	`OSENABLE_BINARY_SEMAPHORES`, `OSENABLE_EVENT_FLAGS`, `OSENABLE_MESSAGE_QUEUES`, `OSENABLE_SEMAPHORES`, `OSEVENTS`
Enables:	–
Memory Required:	When `TRUE`, requires ROM for message services.

Notes

This configuration option is useful when controlling which parts of Salvo are to be included in an application. If you are including or linking to `salvomsg.c` in your source code, you can control its compilation solely via this configuration option in `salvocfg.h`. This may be more convenient than, say, editing your source code or modifying your project.

OSENABLE_MESSAGE_QUEUES: Enable Support for Message Queues

Name:	OSENABLE_MESSAGE_QUEUES
Purpose:	To control compilation of message queue code via the preprocessor.
Allowed Values:	FALSE, TRUE
Default Value:	FALSE
Action:	If FALSE, message services are not available. If TRUE, OSCreateMsgQ(), OSSignalMsgQ() and OS_WaitMsgQ() are available.
Related:	OSENABLE_BINARY_SEMAPHORES, OSENABLE_EVENT_FLAGS, OSENABLE_MESSAGES, OSENABLE_SEMAPHORES, OSEVENTS, OSMESSAGE_QUEUES
Enables:	–
Memory Required:	When TRUE, requires ROM for message queue services.

Notes

This configuration option is useful when controlling which parts of Salvo are to be included in an application. If you are including or linking to salvomsgq.c in your source code, you can control its compilation solely via this configuration option in salvocfg.h. This may be more convenient than, say, editing your source code or modifying your project.

A value of 0 for OSMESSAGE_QUEUES automatically resets (overrides) OSENABLE_MESSAGE_QUEUES to FALSE.

OSENABLE_OSSCHED_DISPATCH_HOOK: Call User Function Inside Scheduler

Name:	`OSENABLE_OSSCHED_DISPATCH_HOOK`
Purpose:	To provide a simple way of calling a user function from inside the scheduler.
Allowed Values:	`FALSE`: No user function is called from `OSSched()`. `TRUE`: An external, user-supplied function named `OSSchedDispatchHook()`is called within `OSSched()` immediately prior to the task being dispatched.
Default Value:	`FALSE`
Action:	If `TRUE`, you must define your own function to be called automatically each time the scheduler runs.
Related:	–
Enables:	–
Memory Required:	When `TRUE`, requires ROM for user function and function call.

Notes

This configuration option is provided for advanced users who want to call a function immediately prior to the most eligible task being dispatched by the scheduler.

Interrupts are normally disabled when `OSSchedEntryHook()` is called.

OSENABLE_OSSCHED_ENTRY_HOOK: Call User Function Inside Scheduler

Name:	`OSENABLE_OSSCHED_ENTRY_HOOK`
Purpose:	To provide a simple way of calling a user function from inside the scheduler.
Allowed Values:	`FALSE`: No user function is called from `OSSched()`. `TRUE`: An external, user-supplied function named `OSSchedEntryHook()`is called within `OSSched()` immediately upon entry.
Default Value:	`FALSE`
Action:	If `TRUE`, you must define your own function to be called automatically each time the scheduler runs.
Related:	–
Enables:	–
Memory Required:	When `TRUE`, requires ROM for user function and function call.

Notes

This configuration option is provided for advanced users who want to call a function immediately upon entry into the scheduler.

Interrupts are normally enabled when `OSSchedDispatchHook()` is called.

OSENABLE_OSSCHED_RETURN_HOOK: Call User Function Inside Scheduler

Name:	`OSENABLE_OSSCHED_RETURN_HOOK`
Purpose:	To provide a simple way of calling a user function from inside the scheduler.
Allowed Values:	`FALSE`: No user function is called from `OSSched()`. `TRUE`: An external, user-supplied function named `OSSchedReturnHook()`is called within `OSSched()` immediately after the dispatched task has returned to the scheduler.
Default Value:	`FALSE`
Action:	If `TRUE`, you must define your own function to be called automatically each time the scheduler runs.
Related:	–
Enables:	–
Memory Required:	When `TRUE`, requires ROM for user function and function call.

Notes

This configuration option is provided for advanced users who want to call a function immediately after the most eligible task has returned to the scheduler.

Interrupts are normally enabled when `OSSchedReturnHook()` is called.

OSENABLE_SEMAPHORES: Enable Support for Semaphores

Name:	`OSENABLE_SEMAPHORES`
Purpose:	To control compilation of semaphore code via the preprocessor.
Allowed Values:	`FALSE`, `TRUE`
Default Value:	`FALSE`
Action:	If `FALSE`, semaphore services are not available. If `TRUE`, `OSCreateSem()`, `OSSignalSem()` and `OS_WaitSem()` are available.
Related:	`OSENABLE_BINARY_SEMAPHORES`, `OSENABLE_EVENT_FLAGS`, `OSENABLE_MESSAGES`, `OSENABLE_MESSAGE_QUEUES`, `OSEVENTS`
Enables:	–
Memory Required:	When `TRUE`, requires ROM for semaphore services.

Notes

This configuration option is useful when controlling which parts of Salvo are to be included in an application. If you are including or linking to `salvosem.c` in your source code, you can control its compilation solely via this configuration option in `salvocfg.h`. This may be more convenient than, say, editing your source code or modifying your project.

OSENABLE_STACK_CHECKING: Monitor Call ... Return Stack Depth

Name:	`OSENABLE_STACK_CHECKING`
Purpose:	To enable the user to discern the maximum call ... return stack depth used by Salvo services.
Allowed Values:	`FALSE`: Stack depth checking is not performed. `TRUE`: Maximum and current stack depth is recorded.
Default Value:	`FALSE`
Action:	If `TRUE`, enables code in each function to monitor the current call ... return stack depth and record a maximum call ... return stack depth if it has changed.
Related:	`OSGATHER_STATISTICS`, `OSRpt()`
Enables:	–
Memory Required:	When `TRUE`, requires a considerable amount of ROM, plus two bytes of RAM.

Notes

Current and maximum stack depth are tracked to a maximum call ... return depth of 255.

Current stack depth is held in `OSstkDepth`. Maximum stack depth is held in `OSmaxStkDepth`.

Stack depth is only calculated for call ... returns within Salvo code and is not necessarily equal to the current hardware stack depth of your processor. However, for most applications they will be the same since `OSSched()` is usually called from `main()`.

OSENABLE_TCBEXT0|1|2|3|4|5: Enable Tcb Extensions

Name:	OSENABLE_TCBEXT0|1|2|3|4|5
Purpose:	To add user-definable variables to a task's control block.
Allowed Values:	FALSE: Named tcb extension is not enabled. TRUE: Named tcb extension is enabled.
Default Value:	FALSE
Action:	If TRUE, creates a user-definable and accessible object of type OStypeTcbExt0|1|2|3|4|5 within each tcb.
Related:	OSLOC_TCB, OSTYPE_TCBEXT0|1|2|3|4|5, OScTcbExt0|1|2|3|4|5, OStcbExt0|1|2|3|4|5
Enables:	tcbExt0|1|2|3|4|5 fields
Memory Required:	When TRUE, requires additional RAM per tcb.

Notes

Salvo's standard tcb fields are reserved for the management of tasks and events. In some instances it is useful to additional variables that are unique to the particular task. Salvo's *tcb extensions* are ideal for this purpose.

The default type for a tcb extension is `void *` (i.e. a void pointer). A tcb extension's type can be overridden to any type[49] by using the appropriate OSTYPE_TCBEXT0|1|2|3|4|5 configuration option.

Once enabled via OSENABLE_TCBEXT0|1|2|3|4|5, a tcb extension can be accessed through the OScTcbExt0|1|2|3|4|5 or OStcbExt0|1|2|3|4|5 macros.

OSLOC_TCB controls the storage type of tcb extensions. Tcb extensions are only initialized if / when OSInitTcb() is called, or by the compiler's startup code. Any desired mix of the tcb extensions can be enabled.

Consider the case of several identical tasks, all created from a single task function, which run concurrently. Each task is responsible for one of several identical communications channels, each with its own I/O and buffers. Enable a tcb extension of type pointer-to-

[49] Including structures, etc.

struct, and initialize it uniquely for each task. At runtime each task runs independently of the others, managing its own communications channel, defined by the struct. Since only one task function need be defined, substantial savings in code size can be realized.

The example in Listing 31 illustrates the use of a single, unsigned-char-sized tcb extension tcbExt1 that each of four identical tasks uses as an index into an array of offsets in the 4KB buffer the tasks share.

```
…

const unsigned offset[4] = { 3072,
                             2048,
                             1024,
                             0    };

void TaskBuff( void )
{
  while (1) {
    printf("Task %d's buffer ",
      OStID(OScTcbP, OSTASKS));
    printf("starts at %d\n",  offset[OScTcbExt1]);
    …
    OS_Yield();
  }
}

main()
{
  OSInit();

  OSCreateTask(TaskBuff, OSTCBP(2), 1);
  OSCreateTask(TaskBuff, OSTCBP(6), 1);
  OSCreateTask(TaskBuff, OSTCBP(7), 1);
  OSCreateTask(TaskBuff, OSTCBP(8), 1);

  OStcbExt1(OSTCBP(2)) = 0;
  OStcbExt1(OSTCBP(6)) = 1;
  OStcbExt1(OSTCBP(7)) = 2;
  OStcbExt1(OSTCBP(8)) = 3;

  for (i = 0; i < 4; i++) {
    OSSched();
  }
}
```

Listing 31: Tcb Extension Example

Each time TaskBuff() runs, it can obtain its offset into the 4KB buffer through OStcbExt1 for the current task, namely, itself. For this example, OSENABLE_TCBEXT1 was set to TRUE and

OSTYPE_TCBEXT1 was set to unsigned char in the project's salvocfg.h. The resulting output is shown in Figure 30.

```
Task 2's buffer starts at 3072
Task 6's buffer starts at 2048
Task 7's buffer starts at 1024
Task 8's buffer starts at 0
```

Figure 30: Tcb Extension Example Program Output

Tcb extensions can be used for a variety of purposes, including

- • Passing information via a pointer to a task at startup or during runtime.[50]
- • Avoiding the use of task-specific global variables accessed indirectly via OStID().
- • Embedding objects of any type in a task's tcb.

[50] This is useful because Salvo tasks must be declared as void Task (void), i.e. without any parameters.

OSENABLE_TIMEOUTS: Enable Support for Timeouts

Name:	`OSENABLE_TIMEOUTS`
Purpose:	To be able to specify an optional timeout when waiting for an event.
Allowed Values:	`FALSE`: Timeouts cannot be specified. `TRUE`: Timeouts can be specified.
Default Value:	`FALSE`
Action:	If `TRUE`, enables the passing of an extra parameter to specify a timeout when waiting for an event..
Related:	–
Enables:	`OSTimedOut()`
Memory Required:	When `TRUE`, requires a considerable amount of ROM, plus an additional byte of RAM per tcb.

Notes

By specifying a timeout when waiting for an event, the waiting task can continue if the event does not occur within the specified time period. Use `OSTimedOut()` to detect if a timeout occurred.

If timeouts are enabled, you can use the defined symbol `OSNO_TIMEOUT` for those calls that do not require a timeout.

See *Chapter 6 • Frequently Asked Questions (FAQ)* for more information on using timeouts.

OSGATHER_STATISTICS: Collect Run-time Statistics

Name:	`OSGATHER_STATISTICS`
Purpose:	To collect run-time statistics from your application.
Allowed Values:	`FALSE`: Statistics are not collected. `TRUE`: A variety of statistics are collected.
Default Value:	`FALSE`
Action:	If `TRUE`, enables Salvo code to collect run-time statistics from your application on the number of errors, warnings, timeouts, context switches and calls to the idle function.
Related:	`OSBYTES_OF_COUNTS`, `OSENABLE_STACK_CHECKING`
Enables:	–
Memory Required:	When `TRUE`, requires a small amount of ROM, plus RAM for counters.

Notes

The numbers of errors, warnings and timeouts are tracked to a maximum value of 255.

The maximum number of any counter is dependent on the value of `OSBYTES_OF_COUNTS`. If `OSBYTES_OF_COUNTS` is not defined or is defined to be 0, it will be redefined to 1.

Which statistics are collected is highly dependent on the related configuration options listed above.

If enabled via `OSLOGGING`, error and warning logging will occur regardless of the value of `OSGATHER_STATISTICS`.

OSINTERRUPT_LEVEL: Specify Interrupt Level for Interrupt-callable Services

Name:	`OSINTERRUPT_LEVEL`
Purpose:	To specify the interrupt level used in the Salvo source code. For use with these compilers: HI-TECH PICC and PICC-Lite HI-TECH PICC-18 HI-TECH V8C
Allowed Values:	`0-7` (depends on compiler)
Default Value:	`0`
Action:	
Related:	`OSCALL_OSXYZ`
Enables:	–
Memory Required:	–

Notes

Some compilers support an interrupt level feature. With `OSINTERRUPT_LEVEL` you can specify which level is used by Salvo services called from the foreground.

All affected Salvo services use the same interrupt level.

OSLOC_ALL: Storage Type for All Salvo Objects

Name:	OSLOC_ALL
Purpose:	To place Salvo objects anywhere in RAM.
Allowed Values:	See Table 1.
Default Value:	OSLOC_DEFAULT (in portxyz.h).
Action:	Set the memory storage type for all of Salvo's objects that aren't overridden by OSLOC_XYZ.
Related:	OSLOC_ALL, OSLOC_COUNT, OSLOC_CTCB, OSLOC_DEPTH, OSLOC_ECB, OSLOC_ERR, OSLOC_LOGMSG, OSLOC_MQCB, OSLOC_MSGQ, OSLOC_PS, OSLOC_SIGQ, OSLOC_TCB, OSLOC_TICK
Enables:	–
Memory Required:	n/a

Notes

Many compilers support a variety of storage types (also called *memory types*) for static objects. Depending on the target processor's architecture, it may be advantageous or necessary to place Salvo's variables into RAM spaces other than the default provided by the compiler.

OSLOC_ALL, when used alone, will locate all of Salvo's objects in the specified RAM space. OSLOC_ALL overrides all other undefined OSLOC_XYZ configuration parameters. To place all of Salvo's variables in RAM Bank 2 with the HI-TECH PICC compiler, use:

```
#define OSLOC_ALL bank2
```

in salvocfg.h. To place the event control blocks (ecbs) in data RAM, and everything else in external RAM with the Keil Cx51 compiler, use:

```
#define OSLOC_ALL xdata
#define OSLOC_ECB data
```

The storage types for *all* of Salvo's objects are set via OSLOC_ALL and the remaining OSLOC_XYZ (see below) configuration parameters. *Do not attempt to set storage types in any other manner* – compile- and / or run-time errors are certain to result.

Table 1 lists the allowable storage types / type qualifiers for Salvo objects for each supported compiler (where applicable). Those on separate lines can be combined, usually in any order.

compiler	storage types / type qualifiers
HI-TECH PICC	`bank1`, `bank2`, `bank3` `persistent`
HI-TECH PICC-18	`near` `persistent`
HI-TECH V8C	`persistent`
Keil Cx51	`data`, `idata`, `far`, `xdata`
Microchip MPLAB-C18	not supported – use `OSMPLAB_C18_LOC_ALL_NEAR` instead

Table 1: Allowable Storage Types / Type Qualifiers for Salvo Objects

See Also `OSLOC_XYZ`, *Chapter 11 • Tips, Tricks and Troubleshooting*

OSLOC_COUNT: Storage Type for Counters

Name:	`OSLOC_COUNT`
Purpose:	To place Salvo counters anywhere in RAM.
Allowed Values:	See Table 1.
Default Value:	`OSLOC_DEFAULT` (in `portxyz.h`).
Action:	Set storage type for Salvo counters.
Related:	`OSLOC_ALL`
Enables:	–
Memory Required:	n/a

Notes

`OSLOC_COUNT` will locate the context switch and idle context switch counters in the specified RAM area. Memory is allocated for these counters only when statistics are gathered.

To explicitly specify RAM Bank 0 with the HI-TECH PICC compiler, use:

```
#define OSLOC_COUNT
```

in `salvocfg.h`.

As with all `OSLOC_XYZ` configuration options, multiple type qualifiers can be used with `OSLOC_COUNT`. For example, to prevent HI-TECH PICC start-up code from re-initializing Salvo's counters in RAM bank 2, use:

```
#define OSLOC_COUNT bank2 persistent
```

See Also

Chapter 11 • Tips, Tricks and Troubleshooting

OSLOC_CTCB: Storage Type for Current Task Control Block Pointer

OSLOC_CTCB will locate the current task control block pointer in the specified RAM area. This pointer is used by OSSched().

See OSLOC_COUNT for more information on setting storage types for Salvo objects.

OSLOC_DEPTH: Storage Type for Stack Depth Counters

OSLOC_DEPTH will locate the 8-bit call ... return stack depth and maximum stack depth counters in the specified RAM area. Memory is allocated for these counters only when stack depth checking is enabled.

See OSLOC_COUNT for more information on setting storage types for Salvo objects.

See Also OSENABLE_STACK_CHECKING

OSLOC_ECB: Storage Type for Event Control Blocks and Queue Pointers

OSLOC_ECB will locate the event control blocks, the eligible queue pointer and the delay queue pointer in the specified RAM area. Memory is allocated for ecbs only when events are enabled. Memory is allocated for the delay queue pointer only when delays and/or timeouts are enabled.

See OSLOC_COUNT for more information on setting storage types for Salvo objects.

See Also OSEVENTS

OSLOC_EFCB: Storage Type for Event Flag Control Blocks

OSLOC_EFCB will locate the event flag control blocks – declared to be of type OSgltypeEfcb by the user – in the specified RAM area.

See OSLOC_COUNT for more information on setting storage types for Salvo objects.

OSLOC_ERR: Storage Type for Error Counters

OSLOC_ERR will locate the 8-bit error, warning and timeout counters in the specified RAM area. Memory is allocated for these counters only when logging is enabled.

See OSLOC_COUNT for more information on setting storage types for Salvo objects.

See Also OSENABLE_TIMEOUTS, OSGATHER_STATISTICS, OS_LOGGING

OSLOC_GLSTAT: Storage Type for Global Status Bits

OSLOC_GLSTAT will locate Salvo's global status bits in the specified RAM area. Memory is allocated for these bits whenever time functions are enabled.

See OSLOC_COUNT for more information on setting storage types for Salvo objects.

OSLOC_LOGMSG: Storage Type for Log Message String

OSLOC_LOGMSG will locate the character buffer used to hold log messages in the specified RAM area. This buffer is needed to create error, warning and descriptive informational messages.

See OSLOC_COUNT for more information on setting storage types for Salvo objects.

See Also OS_LOGGING, OSLOG_MESSAGES

OSLOC_LOST_TICK: Storage Type for Lost Ticks

OSLOC_LOST_TICK will locate the character buffer used to hold lost ticks in the specified RAM area. This buffer is used to avoid timing errors when the scheduler is not called rapidly enough.

See OSLOC_COUNT for more information on setting storage types for Salvo objects.

See Also OS_LOGGING, OSLOG_MESSAGES

OSLOC_MQCB: Storage Type for Message Queue Control Blocks

OSLOC_MQCB will locate the message queue control blocks (mqcbs) in the specified RAM area. Each message queue has an mqcb associated with it – however, message queues and mqcbs need not be in the same bank.

See OSLOC_COUNT for more information on setting storage types for Salvo objects.

OSLOC_MSGQ: Storage Type for Message Queues

OSLOC_MSGQ tells Salvo that the message queue buffers are located in the specified RAM area. By using the predefined Salvo qualified type OSgltypeMsgQP when declaring each buffer it will be automatically placed in the desired RAM bank.

See OSLOC_COUNT for more information on setting storage types for Salvo objects.

See Also OSMESSAGE_QUEUES

OSLOC_PS: Storage Type for Timer Prescalar

OSLOC_PS will locate the timer prescalar (used by OSTimer()) in the specified RAM area.

See OSLOC_COUNT for more information on setting storage types for Salvo objects.

See Also OSENABLE_PRESCALAR

OSLOC_TCB: Storage Type for Task Control Blocks

`OSLOC_TCB` will locate the task control blocks in the specified RAM area.

See `OSLOC_COUNT` for more information on setting storage types for Salvo objects.

OSLOC_SIGQ: Storage Type for Signaled Events Queue Pointers

`OSLOC_SIGQ` will locate the signaled events queue pointers in the specified RAM area. Memory is allocated for this counter only when events are enabled.

See `OSLOC_COUNT` for more information on setting storage types for Salvo objects.

OSLOC_TICK: Storage Type for System Tick Counter

`OSLOC_TICK` will locate the system tick counter in the specified RAM area. Memory is allocated for this counter only when ticks are enabled.

See `OSLOC_COUNT` for more information on setting storage types for Salvo objects.

See Also

`OSBYTES_OF_TICKS`

OSLOGGING: Log Runtime Errors and Warnings

Name:	OSLOGGING
Purpose:	To log runtime errors and warnings.
Allowed Values:	FALSE: Errors and warnings are not logged. TRUE: Errors and warnings are logged.
Default Value:	FALSE
Action:	Configures Salvo functions to log all errors and warnings that occur when during execution.
Related:	OSLOG_MESSAGES, OSRpt()
Enables:	–
Memory Required:	When TRUE, requires a considerable amount of ROM, plus RAM for the error and warning counters.

Notes

Most Salvo functions return an 8-bit error code. Additionally, Salvo can track run-time errors and warnings through the dedicated 8-bit counters OSerrs and OSwarns.

OSRpt() will display the error and warning counters if OSLOGGING is TRUE.

The value of OSLOGGING has no effect on the return codes for Salvo user services.

OSLOGGING is not affected by OSGATHER_STATISTICS.

See Also

OSRpt()

OSLOG_MESSAGES: Configure Runtime Logging Messages

Name:	`OSLOG_MESSAGES`
Purpose:	To aide in debugging your Salvo application.
Allowed Values:	`OSLOG_NONE`: No messages are generated. `OSLOG_ERRORS`: Only error messages are generated. `OSLOG_WARNINGS`: Error and warning messages are generated. `OSLOG_ALL`: Error, warning and informational messages are generated.
Default Value:	`OSLOG_NONE`
Action:	Configures Salvo functions to log in a user-understandable way all errors, warnings and/or general information that occurs when each function executes.
Related:	`OSLOGGING`
Enables:	–
Memory Required:	When `TRUE`, requires a considerable amount of ROM, plus RAM for an 80-character buffer, `OSlogMsg[]`.

Notes

Most Salvo functions return an 8-bit error code. If your application has the ability to `printf()` to a console, Salvo can be configured via this configuration option to report on errors, warnings and/or general information with descriptive messages. If an error, warning or general event occurs, a descriptive message with the name of the corresponding Salvo function is output via `printf()`. This can be useful when debugging your application, when modifying the source code or when learning to use Salvo.

Applications that do not have a reentrant `printf()` may have problems when reporting any errors. In these cases, set `OSLOG_MESSAGES` to `OSLOG_NONE`.

Stack depth for `printf()` is not tracked by Salvo – your application may have problems if there is insufficient stack depth beyond that used by Salvo.

`OSLOGGING` must be `TRUE` to use `OSLOG_MESSAGES`.

The value of `OSLOG_MESSAGES` has no effect on the return codes for Salvo user services.

OS_MESSAGE_TYPE: Configure Message Pointers

Name:	`OS_MESSAGE_TYPE`
Purpose:	Enable message pointers to access any area in memory. Compiler-dependent.
Allowed Values:	Any pointer type supported by the compiler.
Default Value:	`void`
Action:	Redefines the defined type `OStypeMsg`.
Related:	`OSCOMPILER`
Enables:	-
Memory Required:	Dependent on definition

Notes

Salvo's message pointers (of type `OStypeMsgP`), used by messages and message queues, are normally defined as void pointers, i.e. `void *`. A void pointer can usually point to anywhere in RAM or ROM. This is useful, for instance, if some of your message pointers point to constant strings in ROM as well as static variables (in RAM).

Some supported compilers require an alternate definition for message pointers in order to point to ROM and RAM together, or to external memory, etc. By redefining `OS_MESSAGE_TYPE`, message pointers can point to the memory of interest.

For example, for Salvo's message pointers to access both ROM and RAM with the HI-TECH PICC compiler, `OS_MESSAGE_TYPE` must be defined as `const` instead of `void`, because PICC's `const *` pointers can access both ROM and RAM, whereas its `void *` pointers can only access RAM.

Changing `OS_MESSAGE_TYPE` may affect the size of ecbs.

OSMPLAB_C18_LOC_ALL_NEAR: Locate all Salvo Objects in Access Bank (MPLAB-C18 Only)

Name:	`OSMPLAB_C18_LOC_ALL_NEAR`
Purpose:	To improve application performance by placing Salvo's global objects in access RAM.
Allowed Values:	`FALSE`: Salvo's global objects are placed in banked RAM. `TRUE`: Salvo's global objects are placed in access RAM.
Default Value:	`FALSE`
Action:	Declares all of Salvo's global objects to be of type near.
Related:	–
Enables:	–
Memory Required:	When `TRUE`, should reduce ROM requirements.

Notes

Salvo's `OSLOC_XYZ` configuration cannot be used with MPLAB-C18. Use `OSMPLAB_C18_LOC_ALL_NEAR` instead to place all of Salvo's global objects in access RAM for improved run-time performance.

OSOPTIMIZE_FOR_SPEED: Optimize for Code Size or Speed

Name:	`OSOPTIMIZE_FOR_SPEED`
Purpose:	To allow you to optimize your application for minimum Salvo code size or maximum speed.
Allowed Values:	`FALSE`: Salvo source code will compile for minimum size with existing configuration options. `TRUE`: Salvo source code will compile for maximum speed with existing configuration options.
Default Value:	`FALSE`
Action:	Takes advantage of certain opportunities to increase the speed of the Salvo code.
Related:	`OSENABLE_DELAYS`
Enables:	–
Memory Required:	When `TRUE`, requires small amounts of ROM and RAM.

Notes

Opportunities exist in the Salvo source code to improve execution speed at the cost of some additional lines of code or bytes of RAM. This configuration option enables you to take advantage of these opportunities.

This configuration option does not override other parameters that may also have an effect on code size.

This configuration option is completely independent of any optimizations your compiler may perform. The interaction between it and your compiler is of course unpredictable.

The interplay between execution speed and memory requirements is complex and is most likely to be unique to each application. For example, configuring Salvo for maximum speed may in some cases both increase speed and shrink ROM size, at the expense of some memory RAM.

OSPIC18_INTERRUPT_MASK: Configure PIC18 Interrupt Mode

Name:	`OSPIC18_INTERRUPT_MASK`
Purpose:	To allow you to control which PIC18 PICmicro interrupts are disabled during Salvo's critical sections.
Allowed Values:	`0xC0`, `0x80`, `0x40`, `0x00`
Default Value:	`0xC0` (all interrupts are disabled during critical sections).
Action:	Defines the interrupt-clearing mask that will be used in Salvo services that contain critical regions of code.
Related:	–
Enables:	–
Memory Required:	–

Notes

`OSPIC18_INTERRUPT_MASK` is currently supported for use with the IAR PIC18 and Microchip MPLAB-C18 compilers.

Microchip PIC18 PICmicro MCUs support two distinct interrupt modes of operation: one with two levels of interrupt priorities (`IPEN` is `1`), and one that is compatible with Microchip's mid-range PICmicro devices (`IPEN` is `0`). Depending on how your application calls Salvo services, it may be to your advantage to change `OSPIC18_INTERRUPT_MASK` to minimize interrupt latency.

When `OSPIC18_INTERRUPT_MASK` is set to `0xC0`, all interrupts (global / high-priority and peripheral / low-priority) are disabled during critical regions. Therefore a value of `0xC0` is compatible with both priority schemes and any method of calling Salvo services.

When `OSPIC18_INTERRUPT_MASK` is set to `0x80`, only global / high-priority interrupts are disabled during critical regions. Therefore a value of `0x80` should only be used in two cases: 1) in compatibility mode, and 2) in priority mode if Salvo services that can be called from the foreground / ISR level are called *exclusively from high-level interrupts*.

When `OSPIC18_INTERRUPT_MASK` is set to `0x40`, only peripheral / low-priority interrupts are disabled during critical regions. Therefore a value of `0x40` should only be used in priority mode if Salvo services that can be called from the foreground / ISR level are

called *exclusively from low-level interrupts*. A value of `0x40` must not be used in compatibility mode.

A value of 0x00 is permitted. However, it must only be used on applications that *do not use interrupts*.

Failure to use the correct value of `OSPIC18_INTERRUPT_MASK` for your application will lead to unpredictable runtime results.

See Microchip's PIC18 PICmicro databooks and your PIC18 compiler's *Salvo Compiler Reference Manual* for more information.

OSRPT_HIDE_INVALID_POINTERS: OSRpt() Won't Display Invalid Pointers

Name:	OSRPT_HIDE_INVALID_POINTERS
Purpose:	To make the output of OSRpt() more legible.
Allowed Values:	FALSE: All tcb and ecb pointer values will be displayed, regardless of whether or not they are valid. TRUE: Only those pointers which are valid are shown in the monitor.
Default Value:	TRUE
Action:	Configures OSRpt() to show or hide invalid pointers.
Related:	OSRPT_SHOW_ONLY_ACTIVE, OSRPT_SHOW_TOTAL_DELAY
Enables:	–
Memory Required:	When TRUE, requires a small amount of ROM.

Notes

In some cases, the pointer fields of tcbs and ecbs are meaningless. For example, if a task has been destroyed, the pointers in its tcb are invalid. By making OSRPT_HIDE_INVALID_POINTERS TRUE, OSRpt()'s output is simplified by removing unnecessary information. Invalid pointers are displayed as "n/a".

See *Chapter 7 • Reference* for more information on OSRpt().

OSRPT_SHOW_ONLY_ACTIVE: OSRpt() Displays Only Active Task and Event Data

Name:	`OSRPT_SHOW_ONLY_ACTIVE`
Purpose:	To remove unnecessary information from `OSRpt()`'s output.
Allowed Values:	`FALSE`: Show the contents of each tcb and ecb. `TRUE`: Show only the contents of each active tcb and ecb.
Default Value:	`TRUE`
Action:	Configures `OSRpt()` to show only tasks which are not destroyed and events which have already been created.
Related:	`OSRPT_HIDE_INVALID_POINTERS`, `OSRPT_SHOW_TOTAL_DELAY`
Enables:	–
Memory Required:	When `TRUE`, requires a small amount of ROM.

Notes

By showing neither the tcb contents of tasks in the destroyed state, nor the ecb contents of events which have not yet been created, `OSRpt()`'s output is simplified. However, if you wish to have all the tasks and events displayed by `OSRpt()`, set this configuration option to `FALSE`.

See *Chapter 7 • Reference* for more information on `OSRpt()`.

OSRPT_SHOW_TOTAL_DELAY: OSRpt() Shows the Total Delay in the Delay Queue

Name:	`OSRPT_SHOW_TOTAL_DELAY`
Purpose:	To aid in computing total delay times when viewing `OSRpt()`'s output.
Allowed Values:	`FALSE`: Only individual task delay fields are shown. `TRUE`: The total (cumulative) delay for all the tasks in the delay queue is computed and shown.
Default Value:	`TRUE`
Action:	Configures `OSRpt()` to compute and display the total delay of all delayed tasks.
Related:	`OSRPT_HIDE_INVALID_POINTERS`, `OSRPT_SHOW_ONLY_ACTIVE`
Enables:	–
Memory Required:	When `TRUE`, requires a small amount of ROM.

Notes

Task delays are stored in the delay queue in an incremental (and not absolute) scheme. When debugging your application it may be useful to be able to see the total delay of all tasks in the delay queue.

See *Chapter 7 • Reference* for more information on `OSRpt()`.

OSRTNADDR_OFFSET: Offset (in bytes) for Context-Switching Saved Return Address

Name:	`OSRTNADDR_OFFSET`
Purpose:	To configure the inner workings of the Salvo context switcher.
Allowed Values:	Any literal.
Default Value:	Defined for each compiler and target in `portXyz.h` whenever `OSCTXSW_METHOD` is `OSRTNADDR_IS_VAR`. If left undefined, default is 0.
Action:	Configures Salvo source code for use with the selected compiler and target processor.
Related:	`OSCTXSW_METHOD`
Enables:	–
Memory Required:	n/a

Notes

This configuration option is used within the Salvo source code to implement part of the context switcher `OS_Yield()`.

Warning Unless you are porting Salvo to an as-yet-unsupported compiler, do not override the value of `OSCTXSW_METHOD` in the porting file `salvoportXyz.h` appropriate for your compiler. Unpredictable results will occur.

If you are working with an as-yet-unsupported compiler, refer to the Salvo source code and *Chapter 10 • Porting* for further instructions.

OSSCHED_RETURN_LABEL(): Define Label within OSSched()

Name:	`OSSCHED_RETURN_LABEL`
Purpose:	To define a globally visible label for certain Salvo context switchers.
Allowed Values:	Undefined, or defined to be the instruction(s) required to create a globally visible label.
Default Value:	Defined but valueless.
Action:	Creates a globally visible label for use by the `goto` statement.
Related:	–
Enables:	–
Memory Required:	–

Notes

Salvo context switchers for certain compilers and/or target processors may be implemented with a `goto`-based approach rather than with a `call`-based approach. For those circumstances, a globally visible label within the scheduler `OSSched()` is required. By declaring a label via this configuration parameter, a context switcher will be able to "return" from a task to the appropriate part of the scheduler.

The preferred name for the label is `OSSchedRtn`.

For the Microchip 12-bit PICmicros (e.g. PIC16C57), which have only a 2-level hardware call…return stack, the following is used with the HI-TECH PICC compiler:

```
#define OSSCHED_RETURN_LABEL() { \
         asm("global _OSSchedRtn"); \
         asm("_OSSchedRtn:"); \
         }
```

This creates a globally visible label `OSSchedRtn` that can be jumped to from other parts of the program.

See the various `portxyz.h` compiler- and target-specific porting files for more information.

OSSET_LIMITS: Limit Number of Runtime Salvo Objects

Name:	OSSET_LIMITS
Purpose:	To limit the number of permissible Salvo objects when using the freeware libraries.
Allowed Values:	FALSE: The numbers of Salvo objects are limited only by their definitions in salvomem.c. TRUE: Salvo services reject operations on Salvo objects that are outside the limits set by the configuration parameters.
Default Value:	FALSE
Action:	Adds run-time bounds-checking on pointer arguments.
Related:	OSENABLE_BOUNDS_CHECKING
Enables:	Bounds-checking code sections in various Salvo services.
Memory Required:	When TRUE, requires some ROM.

Notes

Services involving Salvo objects (e.g. events) normally accept pointer arguments to any valid control blocks. However, when OSSET_LIMITS is TRUE, OSENABLE_BOUNDS_CHECKING is set to TRUE, and these services will only accept pointers that are within the control blocks as specified by configuration parameters (e.g. OSEVENTS) at compile time, and otherwise return an error code.

In other words, if OSSignalXyz() is compiled with OSSET_LIMITS as TRUE and OSEVENTS as 4, passing it an event control block pointer (ecbP) of OSECBP(5) or higher[51] will result in OSSignalXyz() returning an error code of OSERR_BAD_P.

All users should leave this option at its default value.

[51] ecbs are numbered from 1 to OSEVENTS.

OSSPEEDUP_QUEUEING: Speed Up Queue Operations

Name:	`OSSPEEDUP_QUEUEING`
Purpose:	To improve queueing performance.
Allowed Values:	`FALSE`: Use standard queueing algorithm. `TRUE`: Use fast queueing algorithm.
Default Value:	`FALSE`
Action:	Configures queueing routines for fastest performance.
Related:	–
Enables:	–
Memory Required:	When `TRUE`, requires a small amount of ROM and RAM.

Notes

It is possible to improve the speed of certain operations involving queues approximately 25% through the use of local variables in a few of Salvo's internal queueing routines.

Applications with minimal RAM should leave this configuration option at its default value.

See Chapter 9 • Performance for more information on queueing.

OSTIMER_PRESCALAR: Configure Prescalar for OSTimer()

Name:	`OSTIMER_PRESCALAR`
Purpose:	To allow you maximum flexibility in locating `OSTimer()` within your application.
Allowed Values:	`0`, `2` to (2^32)-1.
Default Value:	`0`
Action:	If non-zero, adds code and an 8- to 32-bit countdown timer to `OSTimer()` to implement a prescalar.
Related:	`OSBYTES_OF_DELAYS`, `OSBYTES_OF_TICKS`
Enables:	–
Memory Required:	When `TRUE`, requires a small amount of ROM, plus RAM for the prescalar.

Notes

If your application uses delays or timeouts, `OSTimer()` must be called at the desired system tick rate. This is typically every 10-100ms. If your processor has limited resources, it may be unacceptable to dedicate a (relatively slow) timer resource to `OSTimer()`. By using `OSTIMER_PRESCALAR` you can call `OSTimer()` at one rate but have it actually perform its timer-related duties at a much slower rate, as dictated by the value of `OSTIMER_PRESCALAR`.

Unlike some hardware prescalars, which provide powers-of-2 prescaling (e.g. 1:2, 1:4, ...), the Salvo timer prescalar is implemented with a simple countdown timer, and can therefore provide a prescalar rate anywhere from 1:2 to 1:(2^32)-1.

A prescalar value of 1 accomplishes nothing and should not be used.

Whenever `OSTimer()` is called and its prescalar has not reached 0, a minimum of housekeeping is performed. When the prescalar reaches zero, `OSTimer()` increments the system tick count (if enabled), and the scheduler processes delayed and/or timed-out tasks.

OSTYPE_TCBEXT0|1|2|3|4|5: Set Tcb Extension Type

Name:	OSTYPE_TCBEXT0|1|2|3|4|5
Purpose:	To allow you to change the type of a tcb extension.
Allowed Values:	Any valid C-language type.
Default Value:	void *
Action:	Redefines OStypeTcbExt0|1|2|3|4|5.
Related:	OSENABLE_TCBEXT0|1|2|3|4|5, OScTcbExt0|1|2|3|4|5, OStcbExt0|1|2|3|4|5
Enables:	–
Memory Required:	Dependent on definition – affects size of tcbs.

Notes

A tcb extension can be of any valid type, and can have memory type qualifiers applied to it so long as they do not conflict with existing OSLOC_XYZ configuration options.

To use tcb extensions, the associated OSENABLE_TCBEXT0|1|2|3|4|5 must be set to TRUE.

See the example for OSENABLE_TCBEXT0|1|2|3|4|5 for more information.

OSUSE_CHAR_SIZED_BITFIELDS: Pack Bitfields into Chars

Name:	OSUSE_CHAR_SIZED_BITFIELDS
Purpose:	To reduce the size of Salvo objects.
Allowed Values:	FALSE: Places Salvo bitfields into int-sized objects. TRUE: Places Salvo bitfields into char-sized objects.
Default Value:	FALSE
Action:	Alters the typedef for OStypeBitField.
Related:	–
Enables:	–
Memory Required:	When FALSE, reduces RAM requirements slightly.

Notes

ANSI C supports bitfields in structures. Multiple bits are combined into a single int-sized value, e.g.:

```
typedef struct {
  int field0:2;
  int field1:1;
  int field2:4;
  } bitfieldStruct;
```

Some compilers (e.g. HI-TECH PICC, Keil C51) allow the packing of bitfields into a single char-sized value in order to save memory. To use this feature, set OSUSE_CHAR_SIZED_BITFIELDS to TRUE. The Salvo type OStypeBitField will be of type char.

Not all compilers support this feature. If you are having problems compiling a Salvo application, set OSUSE_CHAR_SIZED_BITFIELDS to FALSE. The Salvo type OStypeBitField will then be of type int.

OSUSE_EVENT_TYPES: Check for Event Types at Runtime

Name:	OSUSE_EVENT_TYPES
Purpose:	To check for correct usage of an ecb pointer.
Allowed Values:	FALSE: Event-type error checking is not performed. TRUE: When using an event service (e.g. OSSignalSem()), Salvo verifies that the event being operated on is correct for the service.
Default Value:	TRUE
Action:	If TRUE, enables code to verify that the event type is what the service expects. This requires additional ROM, and a byte is added to each ecb (RAM).
Related:	–
Enables:	–
Memory Required:	When TRUE, requires a moderate amount of ROM.

Notes

Salvo uses event control block (ecb) pointers as handles to events. These pointers are passed as arguments to user event services (e.g. OS_WaitMsg()). A user might inadvertently pass an ecb pointer for one type of event (e.g. a semaphore) to a service for another type of event (e.g. OSSignalMsg()). The result would be unpredictable. Therefore an extra layer of error checking can be enabled to ensure that your application is protected against this sort of error.

Caution If you disable this configuration option you must be especially careful with event service arguments. The use of #define statements with descriptive names (e.g. SEM1_P, SEM_COM1_P, MSG12_P) for ecb pointers is highly recommended.

OSUSE_INLINE_OSSCHED: Reduce Task Call…Return Stack Depth

Name:	`OSUSE_INSELIG_MACRO`
Purpose:	To reduce the call…return stack depth at which Salvo tasks run.
Allowed Values:	`FALSE`, `TRUE`
Default Value:	`FALSE`
Action:	If `FALSE`, `OSSched()` is called as a function, and Salvo tasks run at a call…return stack depth of 1 greater than that of `OSSched()`. If `TRUE`, `OSSched()` is used in an inline form (i.e. macro), which reduces its call…return stack depth by 1.
Related:	`OSUSE_INLINE_OSTIMER`
Enables:	–
Memory Required:	When `FALSE`, a small amount of extra ROM and one additional call…return stack level are used by `OSSched()`. When `TRUE`, `OSSched()` uses less ROM and only one call…return stack level.

Notes

Normally, you will call Salvo's scheduler in your application like this:

```
main()
{
  …
  OSInit();
  …
  while (1) {
    OSSched();
  }
}
```

Since `OSSched()` calls Salvo tasks indirectly via function pointers, each task will run with two return addresses pushed onto the target processor's call…return stack: one inside of `OSSched()`, and one inside of `main()`.[52] This means that the call…return stack depth available to your functions called from within a Salvo task is equal to 2 less than the target processor's maximum call…return stack depth.

[52] This assumes that the compiler uses a `goto main()`, and calls all functions inside of `main()` from a call…return stack level of 0. Also, interrupts would add additional return addresses to the call…return stack.

If your target processor's call…return stack depth is limited, and you make deep, nested calls from within Salvo tasks or interrupt routines, you may want to reduce the call…return stack depth at which Salvo tasks run. By setting `OSUSE_INLINE_OSSCHED` to `TRUE`, and calling the scheduler like this:

```
main()
{
  …
  OSInit();
  …
  while (1) {
    #include "salvosched.c"
  }
}
```

you can make Salvo tasks run with one fewer return addresses on the call…return stack, thereby freeing up one call…return stack level for other functions.

OSUSE_INLINE_OSTIMER: Eliminate OSTimer() Call…Return Stack Usage

Name:	`OSUSE_INLINE_OSTIMER`
Purpose:	To enhance ISR performance and reduce Salvo's call…return stack usage.
Allowed Values:	`FALSE`, `TRUE`
Default Value:	`FALSE`
Action:	If `FALSE`, OSTimer() is called as a function from an ISR. If `TRUE`, uses a macro to perform the same operation.
Related:	`OSUSE_INLINE_OSTIMER`
Enables:	–
Memory Required:	When `FALSE`, a small amount of extra ROM and one call…return stack level are used by `OSTimer()`. When `TRUE`, `OSTimer()` uses less ROM and no call…return stack levels.

Notes

Normally you might call `OSTimer()` like this from your Salvo application:

```
void interrupt PeriodicIntVector ( void )
{
  …
  OSTimer();
}
```

This works for many applications. However, there may be disadvantages that arise when calling `OSTimer()` from an ISR. They include slower interrupt response time and larger code size due to the overhead of a call…return chain of instructions through `OSTimer()`and the need to save context during interrupts, and the consumption of one call…return stack level.

You can avoid all of these problems by setting `OSUSE_INLINE_OSTIMER` to `TRUE` and using `OSTimer()` like this:

```
void interrupt PeriodicIntVector ( void )
{
  …
  { #include "salvotimer.c" }
}
```

This will insert an in-line version of `OSTimer()` into your ISR.

OSUSE_INSELIG_MACRO: Reduce Salvo's Call Depth

Name:	OSUSE_INSELIG_MACRO
Purpose:	To reduce Salvo's maximum call depth and parameter RAM usage.
Allowed Values:	FALSE, TRUE
Default Value:	TRUE
Action:	If FALSE, uses a function to perform a common operation internal to Salvo. If TRUE, uses a macro to perform the same operation.
Related:	–
Enables:	–
Memory Required:	When FALSE, requires a small amount of ROM and may require extra RAM on the stack. When TRUE, requires a moderate amount of ROM.

Notes

If your processor is severely RAM-limited, you should leave this configuration option at its default value. For those processors that have a lot of RAM available (e.g. those with a general-purpose stack), then by setting OSUSE_INSELIG_MACRO to FALSE you should realize a reduction in code size at the expense of an additional call level and the RAM required to pass a tcb pointer as a parameter.

OSUSE_MEMSET: Use memset() (if available)

Name:	OSUSE_MEMSET
Purpose:	To take advantage of the presence of a working memset() library function.
Allowed Values:	FALSE, TRUE
Default Value:	FALSE
Action:	If FALSE, your code will use Salvo functions to clear global Salvo variables. If TRUE, memset() will be used to clear global Salvo variables.
Related:	OSLOC_XYZ
Enables:	–
Memory Required:	Requires some ROM when FALSE.

Notes

Compilers will often use the standard library function memset() to clear (zero) global variables in start-up code.

If your target processor has a linear organization for RAM, you should probably set OSUSE_MEMSET to TRUE.

If you target processor uses banked memory, memset() may not work correctly for certain settings of OSLOC_ECB and OSLOC_TCB. In these cases, you should set OSUSE_MEMSET to FALSE in order to use Salvo's explicit byte-by-byte structure clearing functions.

Organization

The configuration options are loosely organized as outlined below, by category.

Compiler in use:	OSCOMPILER
Target processor:	OSTARGET
Tasks and events:	OSBIG_SEMAPHORES, OSEABLE_BINARY_SEMAPHORES, OSENABLE_EVENT_READING, OSENABLE_EVENT_TRYING, OSENABLE_FAST_SIGNALING, OSENABLE_IDLE_COUNTER, OSENABLE_IDLING_HOOK, OSENABLE_MESSAGES, OSENABLE_MESSAGE_QUEUES, OSENABLE_SEMAPHORES, OSEVENTS, OSMESSAGE_QUEUES, OSMESSAGE_TYPE, OSTASKS, OSTASKS
Size-specific:	OSBYTES_OF_COUNTS, OSBYTES_OF_DELAYS, OSBYTES_OF_EVENT_FLAGS, OSBYTES_OF_TICKS
Time and ticks:	OSCOLLECT_LOST_TICKS, OSENABLE_TIMEOUTS, OSTIMER_PRESCALAR
Optimizations:	OSCLEAR_GLOBALS, OSOPTIMIZE_FOR_SPEED, OSSPEEDUP_QUEUEING, OSUSE_OSINSELIGQ_MACRO
Monitor and debugging:	OSCLEAR_UNUSED_POINTERS, OSENABLE_STACK_CHECKING, OSLOGGING, OSLOG_MESSAGES, OSRPT_HIDE_INVALID_POINTERS, OSRPT_SHOW_ONLY_ACTIVE, OSRPT_SHOW_TOTAL_DELAY
Error checking:	OSDISABLE_ERROR_CHECKING, OSUSE_EVENT_TYPES
Statistics:	OSGATHER_STATISTICS

Memory allocation and RAM banking:	OSLOC_ALL, OSLOC_COUNT, OSLOC_CTCB, OSLOC_DEPTH, OSLOC_ECB, OSLOC_ERR, OSLOC_LOGMSG, OSLOC_LOST_TICK, OSLOC_MQCB, OSLOC_MSGQ, OSLOC_PS, OSLOC_SIGQ, OSLOC_TCB, OSLOC_TICK, OSMPLAB_C18_LOC_ALL_NEAR, OSUSE_CHAR_SIZED_BITFIELDS, OSUSE_MEMSET
Interrupts:	OSCALL_OSCREATEEVENT, OSCALL_OSMSGQCOUNT, OSCALL_OSMSGQEMPTY, OSCALL_OSRETURNEVENT, OSCALL_OSSIGNALEVENT, OSCALL_OSSTARTTASK, OSINTERRUPT_LEVEL, OSTIMER_PRESCALAR
Porting:	OSCTXSW_METHOD, OSRTNADDR_OFFSET
Stack depth usage:	OSUSE_INLINE_OSSCHED, OSUSE_INLINE_OSTIMER
Code compression:	OSCOMBINE_EVENT_SERVICES
Linking to libraries:	OSCUSTOM_LIBRARY_CONFIG, OSLIBRARY_CONFIG, OSLIBRARY_GLOBALS, OSLIBRARY_TYPE, OSLIBRARY_VARIANT, OSUSE_LIBRARY
Hooks to user code:	OSENABLE_IDLING_HOOK, SENABLE_OSSCHED_DISPATCH_HOOK, OSENABLE_OSSCHED_ENTRY_HOOK, OSENABLE_OSSCHED_RETURN_HOOK
Scheduler behavior:	OSDISABLE_FAST_SCHEDULING
Extensions:	OSENABLE_TCBEXT0|1|2|3|4|5, OSTYPE_TCBEXT0|1|2|3|4|5
Cyclic Timers:	OSENABLE_CYCLIC_TIMERS

Table 2: Configuration Options by Category

Choosing the Right Options for your Application

You must select a compiler and a target when configuring Salvo for your application. Depending on how many Salvo services you wish to use in your application, you will also need to select and/or configure other options. Consult the table below for further information:

Multitasking:	OSTASKS

Using events:	OSENABLE_BINARY_SEMAPHORES, OSENABLE_EVENT_FLAGS, OSENABLE_FAST_SIGNALING, OSENABLE_MESSAGES, OSENABLE_MESSAGE_QUEUES, OSENABLE_SEMAPHORES, OSEVENTS
Using multiple event types:	OSCOMBINE_EVENT_SERVICES
Keeping unused code out of your application:	OSENABLE_EVENT_READING, OSENABLE_EVENT_TRYING
Delaying tasks:	OSBYTES_OF_DELAYS
Waiting on events with a timeout:	OSBYTES_OF_DELAYS
Setting the size of event flags:	OSBYTES_OF_EVENT_FLAGS
Keeping track of elapsed time:	OSBYTES_OF_TICKS, OSCOLLECT_LOST_TICKS
Counting the number of context switches:	OSBYTES_OF_COUNTS, OSGATHER_STATISTICS
Using 16-bit semaphores:	OSBIG_SEMAPHORES
Using ROM and RAM pointers:	OSMESSAGE_TYPE
Having an idle function:	OSENABLE_IDLING_HOOK, OSENABLE_IDLE_COUNTER
Checking call ... return stack depth:	OSENABLE_STACK_CHECKING, OSGATHER_STATISTICS
Collecting statistics:	OSGATHER_STATISTICS
Logging descriptive error, warning and status messages:	OSLOGGING, OSLOG_MESSAGES
Optimizing your application:	OSCLEAR_GLOBALS, OSOPTIMIZE_FOR_SPEED, OSSPEEDUP_QUEUEING
Making the most of limited resources:	OSTIMER_PRESCALAR
Avoiding event-type mismatches:	OSUSE_EVENT_TYPES
Learning how Salvo works:	OSCLEAR_UNUSED_POINTERS, OSRPT_HIDE_INVALID_POINTERS, OSRPT_SHOW_ONLY_ACTIVE, OSRPT_SHOW_TOTAL_DELAY

Porting to other compilers and / or target processors:	OSCTXSW_METHOD, OSRTNADDR_OFFSET, OSUSE_MEMSET
Minimizing Salvo's call…return stack usage:	OSUSE_INLINE_OSSCHED, OSUSE_INLINE_OSTIMER
Calling Salvo services from the background and the foreground:	OSCALL_OSCREATEEVENT, OSCALL_OSMSGQCOUNT, OSCALL_OSMSGQEMPTY, OSCALL_OSRETURNEVENT, OSCALL_OSSIGNALEVENT, OSCALL_OSSTARTTASK
Locating Salvo's variables in memory:	OSLOC_ALL, OSLOC_COUNT, OSLOC_CTCB, OSLOC_DEPTH, OSLOC_ECB, OSLOC_ERR, OSLOC_LOGMSG, OSLOC_LOST_TICK, OSLOC_MQCB, OSLOC_MSGQ, OSLOC_PS, OSLOC_SIGQ, OSLOC_TCB, OSLOC_TICK, OSMPLAB_C18_LOC_ALL_NEAR
Building an application with libraries:	OSCUSTOM_LIBRARY_CONFIG, OSLIBRARY_CONFIG, OSLIBRARY_GLOBALS, OSLIBRARY_TYPE, OSLIBRARY_VARIANT, OSUSE_LIBRARY
Running multiple tasks at same priority (round-robin):	OSDISABLE_FAST_SCHEDULING
Minimizing memory usage:	OSUSE_CHAR_SIZED_BITFIELDS
Extending task-specific functionality:	OSENABLE_TCBEXT0|1|2|3|4|5, OSTYPE_TCBEXT0|1|2|3|4|5
Using cyclic timers in place of tasks:	OSENABLE_CYCLIC_TIMERS

Table 3: Configuration Options by Desired Feature

Predefined Configuration Constants

Predefined symbols are listed with their values below.

FALSE	0
TRUE	1
OSLOG_NONE, OSLOG_ERRORS, OSLOG_WARNINGS, OSLOG_ALL	see OSLOG_MESSAGES

OSUNDEF, OSNONE	0
OSPIC12, OSPIC16, OSPIC17, OSPIC18, OSIX86, OSI8051, OSM68HC11, OSMSP430, OSVAV8, etc.	see OSTARGET
OSAQ_430, OSGCC, OSHT_8051C, OSHT_PICC, OSHT_V8C, OSIMAGECRAFT, OSMW_CW, OSMIX_PC, OSIAR_ICC, OSMPLAB_C18, OSKEIL_C51, etc.	see OSCOMPILER
OSFROM_BACKGROUND, OSFROM_FOREGROUND, OSFROM_ANYWHERE	see OSCALL_XYZ
OSRTNADDR_IS_PARAM, OSRTNADDR_IS_VAR, OSVIA_OSCTXSW, OSVIA_OSDISPATCH, etc.	see OSCTXSW_METHOD
OSALL_BITS, OSANY_BITS, OSEXACT_BITS	see OS_WaitEFlag()

Table 4: Predefined Symbols

Obsolete Configuration Parameters

Obsolete configuration parameters – id defined – are automatically caught during the preprocessing stage. Including them in your `salvocfg.h` will result in a compile-time error message indicating the name of the configuration option. Some error messages include instructions on alternate, renamed or related configuration options.

Chapter 6 • Frequently Asked Questions (FAQ)

General

What is Salvo?

Salvo is a powerful and feature-rich real-time operating system (RTOS) for single-chip microcontrollers with limited ROM and RAM. By imposing a few constraints on conventional RTOS programming, Salvo rewards you with the power of an RTOS without all of the RAM requirements.

Salvo is so small that it runs where other RTOSes can't. Its RAM requirements are minuscule, and it doesn't need much ROM, either.

Salvo is not a state machine. It is not a "a neat trick." It is not an app note. Salvo is all the RTOS code you need and more to create a high-performance embedded multitasking program in systems where kilobytes of ROM are a luxury and available RAM is measured in tens of bytes.

Is there a shareware / freeware / open source version of Salvo?

There is a freeware version called Salvo Lite.

Processor- and compiler-specific freeware libraries are provided as part of each Salvo Lite distribution. *Each freeware library supports a limited number of tasks and events.* All of the default functionality is included in the freeware libraries. If you need more tasks and/or events, or you need access to Salvo's advanced functionality, then you should consider purchasing Salvo LE or Pro.

Salvo Pro includes all source code. Source code is not included[53] in Salvo Lite or LE. Salvo is not open source.

Just how small is Salvo?

On a single-chip microcontroller, a typical[54] multitasking application might need around 1K ROM and around fifty bytes of RAM for all of Salvo's code and data.

Why should I use Salvo?

If you want to:

-
- • get your embedded product to market ahead of the competition,
- • add greater software functionality to your existing hardware design,
- • improve the real-time performance of a complex design,
- • not have to re-invent the wheel,
- • have a powerful framework to do multitasking programming,
- • control the increasing complexity of your applications,
- • minimize your hardware costs by using smaller and cheaper processors,
- • not be left behind by the multitasking / RTOS wave and/or
- • maximize the reliability of your complex applications

then Salvo is for you.

Low-cost single-chip microcontrollers are capable of hosting sophisticated real-time applications, but programming them to do so can be quite a challenge. Real-time kernels can simplify the design of complex software. They provide proven mechanisms to accomplish a variety of well-understood operations within predictable time frames. Unfortunately, most commercial real-time offerings require large amounts of ROM and RAM – requirements that are largely incompatible with these chips. Programmers of low-end

53 Except for a few specific files in certain freeware versions.

54 Microchip® PIC16C64 with five concurrent tasks and five events.

embedded processors have been at a disadvantage when developing non-trivial applications.

Salvo changes all of that. Now you can develop applications for inexpensive one-chip microcontrollers similar to how you would for a Pentium® in an embedded application.

Salvo will get your application up and running quickly. It provides you with a clean and easily-understood multitasking framework that uses a minimum of memory to get the job done.

What should I consider Salvo Pro over Salvo LE?

With Salvo Pro, you have the Salvo source code. With source code you have complete access to all of Salvo's configurability. This means that you can build custom Salvo libraries with Salvo Pro.

Plus, when your compiler is updated with support for new processors or with new optimizations, you can take advantage of the new compiler features without waiting for a Salvo libraries to be rebuilt and packaged into a new Salvo release.

Another advantage of having Salvo Pro is that it allows you to step through the Salvo code in C when symbolically debugging your application.

Additionally, if / when bugs are found and identified in the Salvo code, you can make changes locally without having to wait for a new Salvo release.

Lastly, some organizations demand access to source code for code reviews and code maintenance.

You can upgrade from Salvo LE to Salvo Pro at anytime.

What can I do with Salvo?

You can throw out any preconceived notions on how difficult or time-consuming embedded programming can be. You can stop dreaming about multiple, independent processes running concurrently in your application without crashing. You can reorganize your code and no longer worry about how a change in one area might affect another. You can add new functionality to your existing programs and know that it will integrate seamlessly. You can easily link external and internal events to program action.

Once you start creating applications with Salvo, you can focus on adding functionality to and improving the performance of your application by creating tasks and events tailored specifically to it. You can create multitasking applications where tasks pass information to other tasks and the rest of your application. You can prioritize the tasks so that your processor is spending its time doing what's most important, instead of unnecessary housekeeping chores. You can have events control how and when tasks run. You can worry a lot less about interrupts. You can write powerful, efficient and reliable multitasking applications with predictable real-time performance.

And you can do all of this a lot more quickly than you'd expect.

What kind of RTOS is Salvo?

Salvo is a priority-based, event-driven, cooperative, multitasking RTOS. It is designed to run on processors with severely limited resources (primarily ROM and RAM).

What are Salvo's minimum requirements?

Salvo requires a full-featured ANSI-C-compliant C compiler from a third party. Contact the factory or visit the website for a list of tested and/or approved compilers.

If you're not already reasonably proficient in C, you will need to review certain concepts (particularly pointers, if you plan on using messages and message queues) before beginning with Salvo. You don't need to be an expert C programmer to use Salvo.

What kind of processors can Salvo applications run on?

Salvo requires a processor with a hardware call…return stack of at least 4 levels and enough memory for Salvo's code and data. ROM and RAM requirements vary, and are controlled primarily by your application's source code and settings in the Salvo configuration file `salvocfg.h`.

My compiler doesn't implement a stack. It allocates variables using a static overlay model. Can it be used with Salvo?

Salvo has been implemented with this type of compiler, with conventional compilers (parameters and return addresses on the stack), and with compilers that take an in-between approach.

Where a general-purpose stack is present, Salvo's use of it is minimal.[55] It can run on stack-less processors as well as any processor with a stack, from a PICmicro® to a Pentium®.

How many tasks and events does Salvo support?

Salvo supports an unlimited number of tasks and events. The number of tasks and events in your application is limited only by available RAM. Salvo's default configuration supports up to 255 tasks, 255 events and 255 message queues.

How many priority levels does Salvo support?

Salvo supports 16 distinct priority levels. Tasks can share priority levels.

What kind of events does Salvo support?

Salvo supports binary semaphores, counting semaphores, event flags, messages and message queues. You can create ("init") events, signal ("post", "put", "unlock", "release", "send") events and have tasks wait ("pend", "get", "lock", "acquire", "receive") on each event.

Is Salvo Y2K compliant?

Yes. Salvo does not provide any functions for reporting or setting the absolute time of day and date (e.g. 10:22.36pm, Nov. 11, 1999). Therefore Salvo is by definition Y2K compliant.

[55] A stack pointer (SP) and/or PUSH and POP instructions are evidence of a general-purpose stack.

Where did Salvo come from?

Salvo 1.0 was originally developed in assembly language for use in a low-cost, high-performance multichannel racecar data acquisition system. Its appeal to a wider audience was quickly recognized, whereupon it was rewritten in C for greater portability and configurability.

Getting Started

Where can I find examples of projects that use Salvo?

Every Salvo distribution has `demo`, `tut` (tutorial) and `ex` (example) folders. Refer to *File and Program Descriptions* in the *Salvo User Manual* for a test system (e.g. `sysa`) that's similar to yours. Then search these folders in your Salvo installation for project files, source code (usually `main.c`) and configuration files (`salvocfg.h`).

Which compiler(s) do you recommend for use with Salvo?

As a matter of policy, we do not take any positions regarding the compilers we have certified for use with Salvo. The fact that we've certified a particular compiler should suggest to you that we consider it to be a production-level tool. When purchasing a compiler, we suggest you base your decision on the quality of its output, suitability to the task, flexibility, IDE (if included), debugging tools, support and price.

Unless otherwise noted in the *Salvo Compiler Reference Manuals*, compilers for the same target are generally interchangeable as far as Salvo is concerned.

Is there a tutorial?

Yes. An in-depth tutorial can be found in the *Salvo User Manual*.

Apart from the Salvo User Manual, what other sources of documentation are available?

The *Application Notes* contain information on a variety of topics. The *Salvo Compiler Reference Manuals* contain compiler-specific information.

I'm on a tight budget. Can I use Salvo?

You can use Salvo Lite, with its complete set of freeware libraries, to create fully functioning Salvo applications. You'll be limited to the numbers of tasks and events your application can support.

I only have an assembler. Can I use Salvo?

No. You will need a certified C compiler to use Salvo.

Performance

How can using Salvo improve the performance of my application?

If you're used to programming within the conventional foreground / background loop model, converting your application to a Salvo application may yield substantial performance benefits.

For example, it's not uncommon to write a program that polls something (say an I/O pin) repeatedly and performs a complicated and time-consuming action whenever the pin changes. You might have a timer interrupt which calls a subroutine to poll a port pin and XOR it against its previous value. If the pin changes, then you might set a bit in a global status byte, which is then tested every time through your main loop. If the bit is set, you disable interrupts, clear the status bit, reenable interrupts and then take an appropriate action.

The problem with this approach is that your program is consuming processor cycles while sampling information that remains unchanged for most of the time. The more infrequently the event (in this case, the change on I/O pin) occurs, the more inefficient your program is.

The solution is to employ an event-based approach by using Salvo. When a task is made to wait an event, and the event is not available (e.g. the I/O pin hasn't changed), then the task is put into a waiting state. From this time forward, until the event occurs, not a single processor cycle is expended on waiting for the event. Zip, zero, nada. When the event does finally occur, the task will process the event as soon as it is made to run by the scheduler. In other words, it's the event that drives all the other actions directly. With events driving your application, it can spend its time on the most important things, as defined by you, the programmer.

It's important that you understand the distinction between polled and event-based actions.

How do delays work under Salvo?

Salvo provides a simple means of delaying tasks. While a task is delayed, it consumes a minimum of processor resources, and your other (non-delayed) tasks can continue to run. The overhead to support one or more delayed tasks is the same. You can specify delays to the resolution of the system timer, which is under your control.

See the *Timer and Timing* section in this FAQ for more information.

What's so great about having task priorities?

The point of assigning priorities to tasks is to make the most of your processor's power by having it always doing what is most important at that particular instant in time.

For example, say you have an instrument whose primary purpose is to generate moderate-frequency waveforms. But you'd also like to monitor various analog voltages in the instrument to ensure no out-of-range conditions. By assigning the waveform-generating task a high priority, and the analog-sampling task a low priority, the Salvo application will automatically run the sampling task when there's no demand for the waveform to be generated. But while the waveform is being generated, the sampling task will not interfere.

All you have to do in Salvo is assign each task an appropriate priority, and ensure that each task context-switches often enough to allow other tasks to run as needed.

When does the Salvo code in my application actually run?

Salvo's code runs only when you explicitly call Salvo's user services within your application. In most cases it's pretty obvious when your processor is running Salvo code – for example, when you start a task by calling `OSCreateTask()` or `OSStartTask()`.

When the scheduler and timer actually run is perhaps a little less obvious. The scheduler runs as part of any context switch in your code, and it also runs when there are no tasks eligible to run. The timer runs whenever it is called at the periodic system timer rate, which is usually done via a periodic interrupt.

How can I perform fast, timing-critical operations under Salvo?

In order to control critical timing under any RTOS, follow these two rules: 1) give timing-critical tasks high priorities, and 2) use Salvo's flexible features to prevent or delay it from doing anything during a critical time period.

Since Salvo is a cooperative multitasking RTOS, during a timing-critical task there is only one source of potential interference – interrupts. Interrupts which might involve Salvo would be those that signal events and / or call the system timer `OSTimer()`. By preventing calls to Salvo services during timing-critical operations you can guarantee the proper operation of your system.

If, on the other hand, your application can tolerate the timing jitter that will occur if Salvo services are invoked during a critical period, then you may not have much to worry about. This is usually the case with operations whose frequency is much less (e.g. 1/50) than that of the system timer.

Memory

How much will Salvo add to my application's ROM and RAM usage?

Salvo's ROM requirements depend on how many of its functions you call, and its RAM requirements depend on how many tasks and resources you create. Salvo was specifically designed for processors with limited memory resources, and so it requires only a

small fraction of what a typical multitasking kernel would normally need.

The Salvo User's Manual contains specific information on memory requirements for a variety of representative test systems.

How much RAM will an application built with the libraries use?

Using a PIC16 library[56] that supports multitasking, delays, and events (binary and counting semaphores, as well as messages), an application will need

- ● 10 bytes of RAM for Salvo's global variables[57]
- ● 5 bytes of RAM per task
- ● 3 bytes of RAM event

The compiler will need some additional RAM to handle local variables, interrupt save and restore, etc. But the numbers above represent how little RAM Salvo needs to implement all its functionality.

Do I need to worry about running out of memory?

No. Salvo's RAM memory requirements are fixed at compile time. They are simply:

- #(tasks) x sizeof(task control block)
- + #(events) x sizeof(event control block)
- + #(tcb pointers[58]) x sizeof(tcb pointer)
- + #(message queues) x sizeof(message queue control block)
- + #(message queues) x sizeof(user-defined message queues)
- + sizeof(variables associated with configuration options)

These requirements do not change during runtime, and are not dependent on call depth, the status of any of the tasks, the values of

---

56 `sfp42Cab.lib`, for the PIC16F877 for use with the HI-TECH PICC compiler.

57 4 of the 10 bytes of global variables are for the 32-bit elapsed time counter, which can be disabled by doing a source-code build (no libraries).

58 2 or 3, depending on the configuration.

any of the events or any other multitasking-related issues. Once you define tasks and events in Salvo and your application has the memory to support them, you can do whatever you want without the fear of running out of memory.

Salvo cannot "run out of memory" during runtime.

If I define a task or event but never use it, is it costing me RAM?

Yes. The RAM memory is allocated at compile time.

How much call ... return stack depth does Salvo use?

Normal stack depth is 4, and in some instances Salvo can be configured to use a maximum call…return stack depth of 3. This means that no Salvo function will require a call-return stack more than 4 levels deep, not including interrupts. This is accomplished by setting the following configuration parameters in your `salvocfg.h`:

```
#define OSLOGGING                FALSE
#define OSUSE_INLINE_OSSCHED     TRUE
#define OSUSE_INLINE_OSTIMER     TRUE
#define OSUSE_OSINSELIGQ_MACRO   TRUE
```

and making the appropriate changes to your source code (see the configuration options' descriptions for more information). These options will configure Salvo to use in-line forms of various functions (thus saving one or more call…return stack levels) and to use simple function return codes without debug messages (saving another call…return stack level).

When calling Salvo functions (e.g. `OSSignalMsg()`) from ISRs, remember that ISRs are likely to run one or more stack levels deep, depending on when the interrupt is serviced. This will affect the maximum call ... return stack depth in your application.

By choosing `OSENABLE_STACK_CHECKING` Salvo will monitor the stack depth of all of its functions and report back the maximum stack depth reached. This is especially useful when simulating your application by running Salvo on a PC.

Note that the numbers above are based on Salvo's inherent call...return tree, and do not include any additional stack depth due

to how your compiler does certain things like indirect function calls.

Why must I use pointers when working with tasks? Why can't I use explicit task IDs?

Salvo user services originally took task, event and message queue IDs (simple integer constants) as parameters to refer to Salvo objects. The advantage of this approach was that it was very easy for beginners to understand, it easily accommodated run-time error checking, and the memory requirements (mainly when passing parameters) were minimal. However, it also had several severe disadvantages, including increased code size, lack of flexibility, poor run-time performance and increased call…return stack usage.

Salvo services now use pointers as parameters to refer to Salvo objects. Along with the attendant advantages that pointers bring with them, Salvo's syntax is more like other, larger RTOSes. Somewhat surprisingly, the memory requirements actually *decreased* for many target processors.

With the pointer-based approach, the simplest way to refer to a task is to use the `OSTCBP()` macro, which returns a pointer to the tcb of a particular task. This is a compile-time constant (it's an address of an array element), and on many targets[59] uses the same amount of memory as an 8-bit integer constant. Similar macros exist for events, message queues, etc. These macros allow you to refer to Salvo objects explicitly.

An alternative approach is to use a *handle*, a variable that contains a pointer to a particular task's tcb. This offers flexibility but has the disadvantage that it consumes extra RAM. For some applications handles can be very useful.

Using the C `#define` preprocessor directive for event IDs can substantially improve code legibility. For example, use:

```
/* pointer to display binSem. */
#define  BINSEM_DISP_P  OSECBP(3)

/* create display semaphore, init to 1. */
OSCreateSem(BINSEM_DISP_P, 1);
...
/* get display. */
OS_WaitSem(BINSEM_DISP_P, OSNO_TIMEOUT);
```

[59] E.g. PIC16 and PIC17 series of PICmicro MCUs.

```
...
/* release display. */
OSSignalSem(BINSEM_DISP_P);
```

to reference the binary semaphore that is used as a resource to control access to a display in a easy-to-read manner.

How can I avoid re-initializing Salvo's variables when I wake up from sleep on a PIC12C509 PICmicro MCU?

The PIC12C509 has a simple architecture (no interrupts, single reset vector) and always vectors to the last location in ROM when it wakes from sleep due to the watchdog timer or wake-on-pin-change. Normally, the startup code generated by the compiler will initialize all static and global variables immediately after any type of reset – power-on reset (POR) or otherwise. This will reset all of Salvo's variables to 0, equivalent to calling `OSInit()`.

Since you'd like to preserve the state of your multitasking system on wake-from-sleep, and not reset it, you must declare Salvo's variables to be of type persistent. This instructs the compiler to skip the initialization for these variables. If you are using HI-TECH PICC, the easiest way to declare Salvo's variables as persistent is to use the `OSLOC_ALL` configuration option, like this:

```
#define OSLOC_ALL bank1 persistent
```

This will place all of Salvo's variables in RAM bank 1, and will prevent the startup code (which is executed after every type of reset, not just POR) from resetting the variables to zero. If you use this method, you *must* call `OSInit()` after each POR (and not after other types of reset) in order to properly initialize Salvo.

Libraries

What kinds of libraries does Salvo include?

Every Salvo distribution includes the freeware Salvo libraries. Additionally, the Salvo LE and Pro include the standard Salvo libraries. There are many different library types, depending on how much functionality you need.

What's in each Salvo library?

Each Salvo library contains the default Salvo functionality for the particular library type. Additionally, each library is compiled for a default number of Salvo objects (tasks, events, etc.). Some libraries (notably those for targets with extremely limited RAM) have a subset of the normal functionality.

Why are there so many libraries?

Each library is generated with a particular compiler, target processor and library type in mind. As a result, a large number of libraries is required to span all the possible combinations.

Should I use the libraries or the source code when building my application?

If you don't have Salvo Pro, you'll have to use the libraries.

With Salvo Pro, you should use the standard libraries until you reach a situation where the configuration of the library no longer suits your application, e.g. you want 32-bit delays and the library supports only 8-bit delays. In that case, you can use the source code and some configuration options to build a custom Salvo library.

Alternatively, you can build a Salvo application wholly from the Salvo source code, bypassing the libraries altogether.

What's the difference between the freeware and standard Salvo libraries?

There is very little difference. The freeware libraries are limited to a maximum number of Salvo objects. The standard libraries support as many Salvo objects as you can fit in RAM.

My library-based application is using more RAM than I can account for. Why?

The default number of Salvo objects used by each library requires a certain amount of RAM, whether or not you use all of those objects. If your application uses fewer objects, you can reduce the

application's RAM requirements with a different set of configuration objects. See *Chapter 8 • Libraries* for more information.

I'm using a library. Why does my application use more RAM than one compiled directly from source files?

Each library is created with its own default configuration. Some configurations include Salvo features that require one or more bytes of RAM. For example, the library may be configured to support a single message queue as well as other event types. Each message queues requires its own message queue control block (mqcb), and RAM has been allocated for it in the library. Therefore even if you do not use message queues in your application when linking to a library, RAM is allocated for this (unused) message queue.

You can reduce some of the library's RAM requirements by overriding the RAM allocations. See *Chapter 8 • Libraries* for more information.

I'm using a freeware library and I get the message "#error: OSXYZ exceeds library limit – aborting." Why?

You've probably set `OSXYZ` to a number that exceeds the maximum value supported by the library. Remove `OSXYZ` from your `salvocfg.h` or upgrade to Salvo LE or Pro.

Why can't I alter the functionality of a library by adding configuration options to my salvocfg.h?

The configuration options affect a library only at compile time. Since the libraries are precompiled, changing configuration options in your `salvocfg.h` will have no effect on them. Choose a different library with the functionality you desire, or use the source code.

The libraries are very large – much larger than the ROM size of my target processor. Won't that affect my application?

No. Your compiler will extract only the modules that it needs from the library you're using. In fact, linking to libraries creates the smallest possible Salvo applications.

I'm using a library. Can I change the bank where Salvo variables are located?

No. On banked target processors, the locations of the Salvo variables are determined by the library. To "move" the variables to another bank, you'll need to build a custom library, or use the source files, set your own configuration options, and recompile.

Configuration

I'm overwhelmed by all the configuration options. Where should I start?

Nearly all of the configuration options are for Salvo Pro users doing source-code builds, or building custom libraries.

If you're using a Salvo library, the only configuration options you need are the ones that tell Salvo which kind of library you're using and how many Salvo objects you want in your application. You needn't worry too much about the others.

If you have Salvo Pro, or you want more objects than are supported by default in the standard libraries, you'll find various configuration options useful when tailoring Salvo to your application. Start with the default configurations (no configuration options in your `salvocfg.h`), which are described in *Chapter 5 • Configuration*. Then modify your `salvocfg.h` as you enable Salvo functionality that differs from the default.

Three good places to get acquainted with the configuration options and how they're used are the tutorial, example and demonstration programs in the standard Salvo distribution. By examining the programs and their corresponding `salvocfg.h` files you should be able to develop a feel for when to use a particular configuration

option. These programs are found in \salvo\tut, \salvo\ex and \salvo\demo.

Do I have to use all of Salvo's functionality?

You can use as little or as much as you like. Only those portions that you use will be incorporated into (i.e. will take up ROM and RAM in) your final executable. By choosing configuration options you can control how much functionality Salvo delivers to your application.

What file(s) do I include in my main.c?

In terms of Salvo services, all you need to include is salvo.h. For some target processors, including salvo.h is enough to automatically include the necessary processor-specific header files. If not, you'll also need to include target-specific header files in all of your source files – see your compiler's documentation for more information.

What is the purpose of OSENABLE_SEMAPHORES and similar configuration options?

Salvo Pro users who compile their applications by linking multiple Salvo source files may find this type of configuration option useful. That's because entire modules can be disabled simply setting the configuration option to FALSE in salvocfg.h instead of changing the setup to your compiler / project / IDE.

Can I collect run-time statistics with Salvo?

By enabling OSGATHER_STATISTICS Salvo will track and report the number of context switches, warnings, errors, timeouts and calls to the idle function (if enabled).

How can I clear my processor's watchdog timer with Salvo?

Good coding practice dictates that watchdog timers only be cleared from a single place within an application. An excellent place to do so is from within Salvo's scheduler, and by default, this is what

Salvo does. Therefore, if a task fails to release control back to the scheduler, the watchdog will time out, indicating a fault.

Salvo Pro users can clear the processor's watchdog timer from another location by redefining `OSCLEAR_WATCHDOG_TIMER()` in `salvocfg.h` to do nothing, and clearing the watchdog timer elsewhere in their code.

I enabled timeouts and my RAM and ROM grew substantially– why?

Salvo makes the most efficient use of RAM and ROM based on the configuration options you've chosen. Adding support for timeouts requires an additional amount of RAM for each task, and extra code in ROM, in order to support a task's ability to wait on an event with a timeout. RAM- and ROM-wise, this is probably the most "expensive" Salvo configuration option.

Timer and Timing

Do I have to install the timer?

If you want to make any use of Salvo's time-based functions (task delays, timeouts when waiting for a resource, elapsed time, etc.) you must install the timer. Simple multitasking and support for events do not require the timer, but delays and timeouts do.

Salvo Pro users can configure `OSBYTES_OF_DELAYS` to a non-zero value appropriate for the application in order to use Salvo's delay and timeout features in a source-code build. Similarly, configuring `OSBYTES_OF_TICKS` to a non-zero value in a source-code build enables the use of Salvo's elapsed time features.

How do I install the timer?

In your application you must call `OSTimer()` at the tick rate you feel is appropriate for your application. Usually this is done by creating a periodic interrupt at the desired tick rate, and having the associated ISR call `OSTimer()`. `OSTimer()` must be called in only one place in your application.

I added the timer to my ISR and now my ISR is huge and slow. What should I do?

See "Why did my interrupt service routine grow and become slower when I added a call to `OSTimer()`" in this FAQ.

How do I pick a tick rate for Salvo?

The ideal Salvo "tick" rate is dependent on the application, and hence is configurable. Rates on the order of 10-100Hz are commonly used. The tick rate defines the timer resolution in Salvo, but does not directly affect the latency of a task made ready-to-run. The context-switching rate is independent of the tick rate. A faster tick rate requires more processor, but it gives better timer resolution, and may require additional memory for the delay fields in the task blocks.

Once you've chosen a tick rate, you must configure your system to call `OSTimer()` each time the tick occurs. This is usually done via a periodic interrupt.

How do I use the timer prescalar?

A linear prescalar for the Salvo timer is provided to create a slower Salvo "tick" rate independent of the timer to which the Salvo timer is chained. For example, on a 4MHz system with a hardware timer that generates interrupts at a 500 Hz rate (i.e. every 2 ms), by defining `OSTIMER_PRESCALAR` to 5 the desired Salvo tick rate will be 100Hz (i.e. every 10ms). The maximum value for the prescalar is (2^32)-1, and to disable it altogether simply set it to 0 (the default).

I enabled the prescalar and set it to 1 but it didn't make any difference. Why?

The Salvo timer prescalar is enabled if `OSTIMER_PRESCALAR` is set to a number greater than or equal to 1, resulting in prescalar rates of 1:1, 1:2, 1:3, ... 1:(2^32)-1. A prescalar value of 1 will add a few instructions to `OSTimer()` and will require a byte of RAM storage for `OStimerPS`, but it will not change the rate at which `OSTimer()` is called, since the prescalar rate is 1:1. In order to change the rate at which `OSTimer()` is called in your application, choose a value for the timer prescalar that is 2 or greater.

What is the accuracy of the system timer?

As long as the system tick rate is slow enough to give Salvo's system timer `OSTimer()` enough time to do its job, the system timer will have no more than 1 timer tick of inaccuracy.

What is Salvo's interrupt latency?

Salvo must disable interrupts while certain internal operations are being performed. Every effort has been made to minimize Salvo's interrupt latency. However, because of Salvo's configurability it's difficult to provide a general answer to this question. Your best bet is to create your own test programs with Salvo Lite to test Salvo's interrupt latency.

What if I need to specify delays larger than 8 bits of ticks?

You have three options. You can call `OS_Delay()` multiple times (sequentially, or in a loop) to create longer delays.

With Salvo Pro, you can change the configuration parameter `OSBYTES_OF_DELAYS` to use 16- or 32-bit delays instead of 8-bit delays. This will consume an additional 1 or 3 bytes of RAM per task, respectively.

Or you can make use of the `OSTIMER_PRESCALAR` configuration parameter with Salvo Pro. However, this approach will reduce the resolution of the system timer.

How can I achieve very long delays via Salvo? Can I do that and still keep task memory to a minimum?

The maximum delay and timeout length is user-configurable as (2^(n x 8))-1, where n is the size in bytes for the task's delay field. For example, if 16-bit delays are selected, delays and timeouts of up to 65535 clock ticks are possible. Since all tasks have the same-size delay field, the total amount of RAM memory dedicated to holding the delays is

sizeof(delay field) x #(tasks).

If your application uses delays and timeouts sparingly, but requires a very long timeout, you can use a small value for `OSBYTES_OF_DELAYS` (e.g. 1, for 1 byte / 8 bits / maximum count of 255) and

nest the call within a local loop to achieve a multiple of the maximum timeout supported by Salvo. For example, using

```
for (i = 0; i <= TIMEOUT_MULTIPLE; i++) {
  OS_WaitSem(SEM_NAME_P, MAX_TIMEOUT);
  if ( !OSTimedOut() )
    break;
}
if (OSTimedOut()) {
  /* loop is over, are we here because of a    */
  /*  timeout or did we wait the semaphore     */
  /*  successfully?                            */
}
```

within a task (where the loop counter `i` is static) will result in a maximum timeout of `TIMEOUT_MULTIPLE x MAX_TIMEOUT`. With a looping construct like this a timeout or delay can be made arbitrarily long at the cost of only a single static variable local to the task of interest.

Note that many target processors do math efficiently only for their native data size. Therefore Salvo's timer code will grow substantially on an 8-bit PICmicro if you use 32-bit delays.

An alternative method is to use Salvo's timer prescalar. This method will affect all Salvo delays and timeouts, system-wide. In order to use Salvo's delays and timeouts `OSBYTES_OF_DELAYS` must be non-zero. In order to use the timer prescalar, `OSTIMER_PRESCALAR` must be set to a non-zero value.

Can I specify a timeout when waiting for an event?

Yes. When waiting for an event you can specify an optional timeout in system ticks. `OSENABLE_TIMEOUTS` must be `TRUE` in order to wait with timeouts.

Does Salvo provide functions to obtain elapsed time?

Yes. Salvo provides two elapsed time functions, `OSGetTicks()` and `OSSetTicks()`. These functions get and set, respectively, the current number of timer ticks since the free-running timer ticks counter rolled over. To use these elapsed time functions, the configuration parameter `OSBYTES_OF_TICKS` must be non-zero.

In this example, a task waits for a message, and once obtained, calculates the amount of elapsed time in timer ticks (OSBYTES_OF_TICKS is defined to be 4 in salvocfg.h):

```
...
static OStypeMsgP msgP;
static OStypeTick elapsedTicks;
...
while (1) {
  ...
  OSSetTicks(0);
  OS_WaitMsg(MSG_ID, &msgP, OSNO_TIMEOUT);
  elapsedTicks = OSGetTicks();
  printf("%lu ticks have passed\n", elapsedTicks);
  ...
}
```

How do I choose the right value for OSBYTES_OF_TICKS?

Salvo uses a free-running counter to monitor system ticks. This counter is incremented by 1 each time the system timer OSTimer() is called by your application.[60] The size of this counter, and hence the rollover period, is controlled by the configuration parameter OSBYTES_OF_TICKS.

Since system ticks are used only for obtaining elapsed time and statistics, your choice for the value of OSBYTES_OF_TICKS is entirely dependent on the longest elapsed time you wish to be able to measure accurately.

For example, let's assume that you have written your application to have an effective tick rate of 100Hz by enabling Salvo's system timer, choosing an appropriate value for OSTIMER_PRESCALAR, and calling OSTimer() from inside a timer-interrupt ISR. If OSBYTES_OF_TICKS were defined to be 2, the longest time interval you could measure would be (65535/100) seconds, or just under 11 minutes. If more than 11 minutes elapse before calling OSGetTicks(), the reported elapsed time will be the actual elapsed time modulo 11 minutes, an erroneous result.

[60] For every OSTIMER_PRESCALAR calls to OSTimer() if OSTIMER_PRESCALAR is nonzero.

My processor has no interrupts. Can I still use Salvo's timer services?

Yes. As long as you have some form of a timer, you can use `OSTimer()`. For example, you can monitor a free-running counter for overflow, and each time this occurs, you can call `OSTimer()`. This results in a system tick period equal to the timer overflow period. You can lengthen this period by using Salvo's timer prescalar. As long as you check often enough not to miss an overflow, you'll have an accurate system timer.

See *How can I avoid re-initializing Salvo's variables when I wake up from sleep on a PIC12C509 PICmicro MCU?*, above, for an example of how to do this.

Context Switching

How do I know when I'm context switching in Salvo?

All Salvo with an "OS_" prefix (e.g. `OS_Yield()`)cause a context switch. Context switches do not occur anywhere else in Salvo.

Why can't I context switch from something other than the task level?

Because Salvo is designed to run on processors with minimal amounts of RAM memory and no general-purpose stack, it does not presume that a stack is available to store context-switching information. Without it, there's no way to store the return addresses for the function calls nested within the task. If you were to context-switch from a function nested within a task, upon returning from that function the processor's program counter would be undefined.

Why does Salvo use macros to do context switching?

Context switching in Salvo is an inherently in-line action, and is not generally conducive to the use of functions or subroutines. The context-switching macros use function calls wherever possible to keep code size to a minimum.

Can I context switch in more than one place per task?

There is no limit on how many context switches you write into a given task.

For example, you could add several unconditional context switches (`OS_Yield()`) to the main loop of a low-priority yet long (in terms of lines of code) task. This way, if a higher-priority task needs to run, it will have several opportunities to run for each full path taken through the low-priority task's loop. For example,

```
void TaskLong( void )
{
  while (1) {
    ...
    /* give other tasks a chance to run.       */
    OS_Yield(TaskLong1);
    ...
    /* let's take a break to let higher-       */
    /*  tasks run.                             */
    OS_Yield(TaskLong2);
    ...
    /* we're about to hog the processor for a  */
    /*  while, so let's yield in case another  */
    /*  more important task is ready to run.   */
    OS_Yield(TaskLong3);
    ....
  }
}
```

When must I use context-switching labels?

Prior to Salvo v4, Salvo required context-switching labels.

Unless otherwise specified for a particular target and compiler, Salvo no longer requires context-switching labels.

Use of context-switching labels where they are not required will generate an error message.

Tasks & Events

What are taskIDs?

TaskIDs are just integers used to refer to a task. They are numbered from 1 to `OSTASKS`. There's a one-to-one mapping between a task's taskID and the task control block (tcb) assigned to it. You'll

rarely use taskIDs when writing your Salvo application. Instead, Salvo uses pointers as handles to tasks. For example, the pointer to the task with taskID 3 is `OSTCBP(3)`.

Does it matter which taskID I assign to a particular task?

No. The only rule to follow is that each task needs its own, unique taskID, and hence its own, unique tcb. A task's priority is independent of its taskID.

Is there an idle task in Salvo?

Salvo has a built-in facility for automatically calling a user-defined function when the system is idling. `OSIdlingHook()` is enabled via the configuration option `OSENABLE_IDLING_HOOK`.

If you prefer, you can create your own idle task with the lowest possible priority (`OSLOWEST_PRIO`). Be sure that no other tasks have this priority. Then, your idle task will run whenever none of the other tasks are eligible.

You can context-switch inside an idle task of your own making, but you cannot context-switch inside the built-in idling hook function. This is an important distinction. Which one you use will depend on what sort of functionality you want to occur when the system is idling. The scheduler must perform a context switch each time the idle task runs. Overall performance is better when using the idling hook function, since no real context switch is performed when calling `OSIdlingHook()`.

How can I monitor the tasks in my application?

Salvo provides a task monitor function that you can link to your application. The monitor is intended to work with a simple ASCII terminal program. The monitor can display the status of all tasks and events, and can control tasks. See `OSRpt()` for more information.

What exactly happens in the scheduler?

Salvo's scheduler `OSSched()` performs three major functions each time it is called. First, it processes the event queue, if events are in use. This means that for every event that had a waiting task when it

was signaled, the scheduler makes that task eligible to run. Next, it processes the delay queue. Any tasks that timed out while being delayed or waiting with a timeout will be made eligible to run. Finally, the scheduler runs the most eligible task. Interrupts are enabled and disabled at various times in the scheduler.

What about reentrant code and Salvo?

An RTOS requires a call…return stack, but Salvo works without a general-purpose stack. Therefore none of its functions are reentrant. In order to avoid problems with reentrancy, 1) do not directly call a task from anywhere within your program – let the scheduler handle it, and 2) carefully observe the restrictions on calling Salvo services from ISRs. By explicitly controlling interrupts and/or setting certain configuration parameters, you can call certain Salvo services from mainline, task and interrupt levels all in a single application.

What are "implicit" and "explicit" OS task functions?

The explicit OS functions require that you specify a task number as a parameter. A good example is `OSCreateTask()`, which creates and starts a specified task. Explicit OS task function names contain the word "Task". Implicit OS functions like `OS_Delay()` operate only on the current task, i.e. the task that is running. Once a task is running, most or all of the OS functions called are likely to be implicit ones, i.e. they operate on the current task.

How do I setup an infinite loop in a task?

A simple way in C is to use the following syntax:

```
void Task ( void )
{
  /* initialization code. */
  ...

  while (1) {
    /* body of task. */
    ...
  }
}
```

Note that somewhere in the for loop the task needs to return to the scheduler (e.g. via `OS_Yield()`) to make the highest-priority eligible task run.

Why must tasks use static local variables?

Static variables are assigned their own unique address in RAM, and may not be visible to other tasks. By declaring a task's variables as static you are guaranteeing that they will remain unchanged while the task is not running. This is the only way to preserve the variable from one context switch to the next. If the variable were not static (i.e. if it were an auto variable) it's likely that it would be changed by other tasks, functions or ISRs, and unpredictably.

It is safe to use auto variables in tasks[61] as long as the task does not require that the value of the variable be maintained in the task from one context switch to the next. For example, if a simple `for()` loop is used to repeatedly call a function, and then the task context switches, as long as the loop index is initialized each time, it should not pose a problem.

```
int i;

while (1) {
  for (i = 0; i < 5; i++) {
    WriteControlReg(0x55);
    WriteControlReg(0xAA);
  }
  ...
  OS_Yield(here);
}
```

Doesn't using static local variables take more memory than with other RTOSes?

No, it doesn't. The RAM required for saving persistent local variables in a Salvo application is the same as the RAM required to save auto local variables in conventional RTOSes.[62] In each situation, RAM must be permanently[63] allocated to the variable.

Can tasks share the same priority?

When Salvo is configured to use queues, there's no reason why more than one task cannot share the same priority. Tasks of equal

---

61 Some implementations (e.g. Salvo on x86-based machines with the Mix Software Power C compiler) do not permit the use of auto variables.

62 In a conventional RTOS, local auto variables are by their very nature stored on the stack, or in the task's context save area (if the local auto variable was in a register to begin with).

63 I.e. as long as the task is active.

priority will round-robin (execute one after the another in a circular queue) whenever they are the highest-priority eligible tasks. However, in many applications it is more efficient to give each task a unique priority.

When Salvo is configured to use arrays, each task must have a unique priority.

If an idle task is used in your Salvo application, it should be the only task with the lowest priority (`OSLOWEST_PRIO`). Other tasks should use priorities between `OSHIGHEST_PRIO` and `OSLOWEST_PRIO-1`.

Can I have multiple instances of the same task?

Yes. A Salvo task is essentially an address in your program at which your application will resume execution when the scheduler sends it there. You can configure two or more Salvo tasks to point to the same place in your program. For example,

```
void TaskDelayFiveTicks( void )
{
  while (1) {
    OS_Delay(5, here);
  }
}

...
OSCreateTask(TaskDelayFiveTicks, OSTCBP(5), 8);
OSCreateTask(TaskDelayFiveTicks, OSTCBP(6), 9);
...
while (1) {
  OSSched();
}
```

will create two Salvo tasks with different priorities, each of which delays itself for 5 system ticks over and over. Note that without reentrancy, the utility of multiple instances of the same task is limited. Note also that all static variables in the task function will be "shared" by each instance of the Salvo task.

Does the order in which I start tasks matter?

No. To start a task, it must have been created first. Creating a task initializes the fields in its task control block, but leaves it ineligible to run. Starting a task makes it eligible and places it in the eligible queue. Tasks are positioned within the eligible queue based on

their priority. A task will first execute based on its priority, not on when it was started.

If you start several tasks of equal priority together, they will begin executing in the order they were started. If they remain at these same priorities, they will continue to round robin.

By using `OSSetPrio()` or `OS_Prio()` to change the current task's priority you can control the order in which tasks execute.

How can I reduce code size when starting tasks?

You may face this question of you are explicitly starting tasks separately from when they are created (by using `OSDONT_START_TASK` with `OSCreateTask()`). Each task is referred to by its tcb pointer, which is specified in the call to `OSCreateTask()`. You can reduce the number of calls to `OSStartTask()` by placing it in a loop in order to start multiple tasks at once, e.g.

```
char i;
...
for (i = 1; i <= OSTASKS; i++) {
  OSStartTask(OSTCBP(i));
}
```

will start all of your tasks with just a single call to `OSStartTask()`, thereby reducing the size of your application.

What is the difference between a delayed task and a waiting task?

A task that is delayed is simply inactive for a specified number of system ticks. It will then rejoin the eligible tasks when the delay timer has expired. A task that is waiting will wait until an event occurs. If the event never occurs, then the task is never made eligible again, unless a timeout was specified when the task was made to wait. If the timeout timer expires before the event occurs, the task is made eligible and carries with it a flag that indicates that a timeout occurred. Your application program can handle this flag at the task level.

In order to delay tasks, `OSTimer()` must be called at the system tick rate from your application. This run-time overhead is independent of the number of tasks still delayed. Waiting tasks, on the

other hand, do not require the existence of `OSTimer()`,[64] and require no processing power whatsoever while they are waiting.

Can I create a task to immediately wait an event?

Not with a single service call. A task can only wait an event by calling `OS_WaitXyz()` while running. One way to start your application with a bunch of tasks waiting for event(s) is to create them with the highest priority (guaranteeing that they will run before all others) and create the events with initial values of 0. When each task runs, have it change its priority to the desired run-time priority with `OSSetPrio()` (not `OS_Prio()`!), and have it wait the event. When the events are signaled, the waiting tasks will run.

I started a task but it never ran. Why?

You may have incorrectly specified one or more parameters when calling the relevant Salvo services – check the function return codes to see if any errors were reported. A common error when using the freeware libraries is to create a task with a tcb pointer that exceeds `OSTCBP(OSTASKS)`.

If Salvo was initialized via `OSInit()`, the task was successfully created and started via `OSCreateTask()`, the scheduler `OSSched()` is active, and no other task has destroyed or stopped the task in question, then it probably had a lower priority than the other tasks running, and hence never ran. Try elevating the task's priority. Use the Salvo monitor `OSRpt()` to view the current status of all the tasks.

What happens if I forget to loop in my task?

You'll get some rather odd results. If your application doesn't crash immediately, the original task may leave its own function and continue through your code until it reaches a context switch, and will thereafter resume execution after that context switch, which will be part of another task! So you may have inadvertently created a second instance of another task by failing to keep execution within the intended task.

---

64 Unless they were made to wait with a timeout.

Why did my low-priority run-time tasks start running before my high-priority startup task completed?

It's common to use delays in a startup task (responsible for configuring peripherals like LCDs, for instance). The other tasks ran because the high-priority startup task was delayed. Regardless of its priority, whenever a task is delayed or waiting for an event, other lower-priority tasks are free to run.

If your application needs a startup task that uses delays, and if it's imperative that no other tasks run before the startup task is complete, then one elegant method is to initially create all the tasks but only start the startup task, and then start the other tasks at the end of the startup task. You can even "reuse" the startup task's tcb by destroying the startup task and creating a new task with the same tcb.

When I signaled a waiting task, it took much longer than the context switching time to run. Why?

A task that is made eligible will only run when it becomes the highest-priority eligible task. Other eligible tasks with higher priorities will run first, and will continue to run if they remain eligible. Also, interrupt service routines (ISRs) have the highest priorities of all.

Can I destroy a task and (re-) create a new one in its place?

Yes. As long as a task is destroyed, a new one can be created in its place. A Salvo task is really just a means of executing a function in ROM. Creating and starting a task allows that function to execute along with the other tasks in a priority-based scheme.

Before destroying any task you must ensure that:

- • it is not waiting for any event,
- • is it in the delayed queue and
- • has not acquired any resources that other tasks might need.

It is up to you to ensure that the above conditions are met. If you are to use `OSDestroy()` in a particular task that accesses resources, you must release all resources before destroying the task. Failing to do so would block any other tasks waiting for the resource previ-

ously owned by the now-destroyed task. Only if those tasks were waiting with a timeout would they ever run again.

Can more than one task wait on an event?

Yes. Up to all of the defined tasks can wait on a single event simultaneously.

Does Salvo preserve the order in which events occur?

Yes.

Can a task wait on more than one event at a time?

Yes, but not simultaneously. At any time a task can only be waiting on a single event. It can wait on more than one event sequentially (e.g. first on one, then on the other), but not simultaneously.

In this example, a task first waits for an error message (a string), then waits for a resource (an LCD display) to become available. Once it receives the error message and obtains exclusive access to the display, it writes the message to the display, waits one second, releases the display for others to use, and then returns to waiting for another message.

```
void TaskShowErrMsg( void )
{
  static OStypeMsgP msgP;
  static OStypeMsgP msgP2;

  while (1) {
    OS_WaitMsg(MSG_ERROR_STRING_P, &msgP,
      OSNO_TIMEOUT);
    OS_WaitMsg(MSG_LCD_DISPLAY_P, &msgP2,
      OSNO_TIMEOUT);
    DispStringOnLCD((char *) msgP);
    OS_Delay(ONE_SECOND);
    OSSignalMsg(MSG_LCD_DISPLAY_P, (OStypeMsgP)
1);
  }
}
```

By first acquiring the display resource and later releasing it,[65] the user is guaranteed to see the error message for at least one second. The error message will remain on the LCD display until this or an-

[65] In this example, `MSG_LCD_DISPLAY` is being used as a binary semaphore.

other task obtains the LCD display resource via `OS_WaitMsg(MSG_LCD_DISPLAY, ...)` and writes a new string to it via `DispStringOnLCD()`.

How can I implement event flags?

Event flags are used to synchronize tasks to the occurrence of multiple events. Two types of synchronization are possible – *conjunctive synchronization*, where the task can only proceed once all of the events it's waiting on have occurred (i.e. logical AND), and *disjunctive synchronization*, where the task can proceed as soon as any of the events it's waiting on has occurred (i.e. logical OR).

You can use Salvo's built-in event flag (eFlag) services(this is the preferred method), or you can implement simple flags using binary semaphores. See the Reference chapter in the *Salvo User Manual* for more info on Salvo's event flag services.

To implement conjunctive synchronization (i.e. the logical AND of multiple events) using binary semaphores, the task must wait on multiple events in sequential order. In the example below, the task waits for the occurrence of all three events (signified by binary semaphores) before proceeding.

```
…
OS_WaitBinSem(BINSEM1_P, OSNO_TIMEOUT,
  WaitForSync1);
OS_WaitBinSem(BINSEM2_P, OSNO_TIMEOUT,
  WaitForSync2);
OS_WaitBinSem(BINSEM3_P, OSNO_TIMEOUT,
  WaitForSync3);
…
```

The order in which the events occur (i.e. when each event is signaled) is unimportant. As long as the task is the highest-priority task waiting on each event, once all of the events have been signaled the task will proceed.

To implement disjunctive synchronization (i.e. the logical OR of multiple events) using binary semaphores, the task must wait on a single event that can be signaled from multiple locations in your application.

```
…
OS_WaitBinSem(BINSEM4_P, OSNO_TIMEOUT,
  WaitForSync4);
…
```

In this case the task can proceed as soon as any part of your application has signaled the event. Subsequent event signaling will not affect the task's execution until the next time it waits on the event.

What happens when a task times out waiting for an event?

If the task does not acquire the resource within the timeout period, it will be removed from the event queue (and the waiting queue) and made eligible to run again. When it runs, a timeout flag will be available at the task level to indicate that a timeout occurred. The Salvo user service `OSTimedOut()` returns `TRUE` when this flag is set, `FALSE` otherwise. The timeout flag is cleared when the task returns to the scheduler.

If a task times out waiting for an event, even if the event subsequently occurs before the task runs again, the timeout flag will remain until the task runs and returns to the scheduler. The event will also remain until a task waits on it.

Why is my high-priority task stuck waiting, while other low-priority tasks are running?

The unavailability of an event always takes precedence over a task's priority. Therefore, regardless of its priority, a task that waits on an event that is not available will become a waiting task, and it will remain a waiting task until either a) the event happens and the task is the highest-priority task waiting for the event, or b) a timeout (if specified) occurs.

This situation may simply be due to the fact that the event never occurred, or it may be due to priority inversion.

When an event occurs and there are tasks waiting for it, which task(s) become eligible?

The highest-priority waiting task becomes eligible. Only a single task will become eligible, regardless of how many tasks of equal priority are waiting for the event. All of Salvo's queues are priority queues. Additionally, tasks of equal priorities are inserted into the priority queues (i.e. they are enqueued) on a FIFO basis. For example, if a task of the highest priority is enqueued into a priority queue that already contains a task of highest priority, the task being enqueued will be enqueued after the existing task. In other words,

the first task to be enqueued with a particular priority will be the first task to be dequeued when tasks of that particular priority reach the head of the queue.

There is one exception to this behavior – namely, event flags. When an event flag is signaled, all the tasks waiting on said event flag will be made eligible.

How can I tell if a task timed out waiting for an event?

The macro `OSTimedOut()` is provided to detect timeouts. It returns `TRUE` if the current task has timed out waiting for an event, and `FALSE` otherwise. `OSTimedOut()` is only valid while the current task is running.

Can I create an event from inside a task?

Yes. You can create an event or a task anywhere in your code, as long as you have previously allocated the required memory at compile time. Keep in mind that operating on an event that is not yet defined can cause unpredictable behavior. For example, suppose you have two tasks, one to create and signal a resource, and one that waits for it:

```
void Task1( void )
{
  OSCreateSem(SEM1_P, 0);  /* init to 0 */

  while (1) {
    ...
    OSSignalSem(SEM1_P);
    ...
  }
}

void Task2( void )
{
  while (1) {
    ...
    OS_WaitSem(SEM1_P, OSNO_TIMEOUT);
    ...
  }
}
```

If your `main()` looks like this:

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

```
  OSCreateTask(Task1, TASK1_P, 3);
  OSCreateTask(Task2, TASK2_P, 1);
  while (1) {
    OSSched();
  }
}
```

you will have unpredictable results because `Task2()` will attempt to wait the semaphore `SEM1` before `Task1()` can create it. That's because `Task2()` has a higher priority than `Task1()`, and will therefore run first when the `OSSched()` starts dispatching tasks.

To avoid this, you can either ensure that the task that creates the resource has a higher priority than any task that uses it, or you can create the resource before beginning multitasking via `OSSched()`.

If you plan on creating events or tasks from within an ISR, you must configure `salvocfg.h` appropriately to avoid interrupt-related issues.

What kind of information can I pass to a task via a message?

Messages are application-specific – that is, a message contains whatever you want it to contain. Examples include characters, numbers, strings, structures and pointers. Messages are passed via pointer, and the default type for a Salvo message pointer is `OStypeMsgP`, which is usually a void pointer. Since a void pointer can point to anything, in order to obtain the information in the message, you'll need to typecast the pointer's contents to the message's inherent type.

The only restriction on Salvo messages is that all the messages in a particular message queue should point to the same type of information.

My application uses messages and binary semaphores. Is there any way to make the Salvo code smaller?

Yes, use messages with values of `(OStypeMsgP) 0` and `(OStypeMsgP) 1` instead of binary semaphores with values of `0` and `1`, respectively. This way you can use `OSCreateMsg()`, `OSSignalMsg()` and `OS_WaitMsg()` exclusively.

Why did RAM requirements increase substantially when I enabled message queues?

Each message queue requires both an ecb and a message queue control block (mqcb) of fixed size. The number of ecbs and mqcbs are determined by `OSEVENTS` and `OSMESSAGE_QUEUES`, respectively. Additionally, each message queue also requires RAM for the actual queue. Message queues are the only events that require this extra memory.

Can I signal an event from outside a task?

Yes. Events can be signaled and created from mainline code (e.g. from within tasks, functions or inside `main()`), and from within interrupts. The default Salvo configuration expects events to be created and signaled from mainline code. In order to create or signal tasks from interrupts and/or interrupts and mainline code, the configuration parameters appropriate to the event's user service (e.g. `OSSignalMsg()`) must be defined.

When I signal a message that has more than one task waiting for it, why does only one task become eligible?

A task waits for a message when the corresponding mailbox is empty. Signaling a message will fill the mailbox. The mailbox remains full (i.e. contains a single message) until the task that was waiting on the message runs, i.e. until the task becomes the highest-priority task and is dispatched by the scheduler. Put another way, signaling a message fills the mailbox, and running the task that's waiting on the message empties it. If the task never becomes eligible to run, the mailbox will remain full, and signaling it with a message will result in an error.

I'm using a message event to pass a character variable to a waiting task, but I don't get the right data when I dereference the pointer. What's going on?

Let's say you're trying to pass a character to a task via a message. To send the message you might write:

```
char tempVar;
...
tempVar = '!';
OSSignalMsg(MSG_CHAR_TO_TASK_P,
  (OStypeMsgP) &tempVar);
```

```
...
```

to send a '!' to the task that's waiting for the message `MSG_CHAR_TO_TASK`, which might look like this:

```
static OStypeMsgP msgP;
static char msgReceived;

while (1) {
  OS_WaitMsg(&msgP, MSG_CHAR_TO_TASK_P,
    OSNO_TIMEOUT);
  msgReceived = *(char *) msgP;
  switch (msgReceived) {
    case '!':
      printf("Received '!'\n");
      break;

    default:
      printf("Received anything but '!'\n");
  }
}
```

Because tasks obtain messages via pointers, the element referenced by the message pointer must remain unchanged until `OS_WaitMsg()` succeeds. In the example above, if the global or auto variable `tempVar` is assigned another value before the waiting task has a chance to obtain the message, the waiting task will receive a message quite different from what was intended. A safer solution would be to signal the message with a pointer to a character constant:

```
const char BANG = '!';
...
OSSignalMsg(MSG_CHAR_TO_TASK_P,
  (OStypeMsgP) &BANG);
...
```

This way, no matter how long it takes for the receiving task to run and obtain the message, it is guaranteed to be the '!' character.

What happens when there are no tasks in the eligible queue?

The scheduler loops in a very tight loop, with interrupts enabled, when there are no tasks eligible to run. As soon as a task is made eligible, either through the actions of `OSTimer()` or an interrupt signaling an event, the scheduler will cause it to run.

In what order do messages leave a message queue?

Each message queue operates on a FIFO (first-in, first-out) basis.

What happens if an event is signaled before any task starts to wait it? Will the event get lost or it will be processed after task starts to wait it?

The event will not be lost, and the highest-priority task to wait the event will get it, i.e. will remain eligible after `OS_WaitXyz()` instead of going to the waiting state.

What happens if an event is signaled several times before waiting task gets a chance to run and process that event? Will the last one signal be processed and previous lost? Or the first will be processed and the following signals lost?

That depends on the event – if it's a binary semaphore or a message, all further signaling results in `OSSignalXyz()` returning an error code, because the event is "full". The first event to be signaled will be processed, and subsequent ones will be lost. In the case of a counting semaphore, the value is simply incremented. In the case of a message queue, additional messages are enqueued until the queue is full. With these events, once the event is "full", subsequent signals will be lost.

What is more important to create first, an event or the task that waits it? Does the order of creation matter?

The order of creation doesn't matter. But when a task waits an event, the event must exist before the task runs.

What if I don't need one event anymore and want to use its slot for another event? Can I destroy event?

Absolutely! For example, you can destroy a binary semaphore and create a counting semaphore in its place by calling `OSCreateSem()` with the ecb you previously used for the binary semaphore. You should only do this if you know that there aren't any tasks waiting the binary semaphore.

Can I use messages or message queues to pass raw data between tasks?

Yes, with some restrictions. With messages, a null message pointer is treated as an empty message, and a task will wait an empty message forever. Therefore only non-zero raw data can be passed via messages. Message queues are different in that a task will wait a message queue indefinitely if there are no messages in it. Therefore null message pointers are allowed in message queues, and raw data of any value can be passed from one task to another using a message queue. In this case, the message queue acts like a FIFO buffer.

If you want to pass null-pointer messages to a task, use a message queue of size 1.

How can I test if there's room for additional messages in a message queue without signaling the message queue?

Use `OSMsgQEmpty()`. If the message queue is full – i.e. there is no room for an additional message in the message queue – `OSMsgQEmpty()` returns 0 (FALSE). If there is room, `OSMsgQEmpty()` returns the number of available slots in the message queue.

Interrupts

Why does Salvo disable all interrupts during a critical section of code?

It is common practice in an RTOS to disable interrupts during a critical section of code. To maintain system performance, interrupts should be disabled for the shortest times possible. However, it's imperative that while an RTOS performs certain critical functions, it must not be interrupted for fear of certain things in the RTOS being corrupted.

The major sources of corruption due to interference from an interrupt are access to a shared resource, and the operation of non-reentrant functions. Salvo must guarantee that while performing certain operations on its data structures (e.g. changing an event control block), no access (read or write) from any other part of the application is allowed. Salvo functions that access the data struc-

tures include `OSTimer()`, which is normally called from within a periodic interrupt, and `OSSignalMsg()`, which might be called from an entirely different interrupt.

Since Salvo services work without a general-purpose stack, certain steps must be taken to prevent data corruption from interrupts. Use the `OSCALL_Xyz()` configuration parameters if you want to be able to call a particular Salvo service (e.g. `OSSignalSem()`) from both main-line code and an ISR.

I'm concerned about interrupt latency. Can I modify Salvo to disable only certain interrupts during critical sections of code?

Yes, and it will require Salvo Pro. The approach to take is to redefine Salvo's `OSEi()` and `OSDi()` to only disable those interrupts that are associated with calls to Salvo services, and leave other interrupts alone. The implementation will differ from one target to another based on the target's interrupt control scheme, its interrupt vectors, its interrupt priorities, and whether Salvo controls interrupts via functions, macros, or through compiler extensions.

As an example, a Salvo customer on the PIC18 needed essentially zero jitter so that his interrupt-driven DSP algorithm ran at exactly 1280Hz. So, the Salvo solution for that particular chip (which has two interrupt priority levels) was to put the DSP stuff on the high-priority interrupt, and the rest on the low-priority interrupt, and configure Salvo to only disable low-priority interrupts in its critical sections. This, it turns out, was very easy for that particular target and compiler – just a small header file to build a custom library with the desired behavior. 5 minutes' work.

How big are the Salvo functions I might call from within an interrupt?

`OSTimer()` and `OSSignalXyz()` are the Salvo services you might call from an interrupt. They are all quite small and fast, and have no nested subroutines. While it varies among different target processors, these services will in many cases be faster than the actual interrupt save and restore.

Why did my interrupt service routine grow and become slower when I added a call to OSTimer()?

Some compilers assume the worst case with regard to register saves and restores when an external function is called from within an interrupt routine. As a result, the compiler may add a large amount of code to save and restore registers or temporary registers to preserve the program's context during an interrupt. Since it's always a good idea to have as fast an interrupt routine as possible, one solution is to include the necessary Salvo files[66] in your interrupt routine's source code instead of linking to the `OSTimer()` and related services as external functions (e.g. through the Salvo library). By including those Salvo files which completely define the necessary call chains for `OSTimer()` your compiler can "see" exactly which registers and temporary registers must be saved, instead of assuming the worst case and saving all of them.

Another option is to in-line OSTimer(). For more information, see the `OSUSE_INLINE_OSTIMER` configuration option.

My application can't afford the overhead of signaling from an ISR. How can I get around this problem?

Ideally you should signal from an ISR if the event that causes the signaling is an interrupt. If this is not possible, in your ISR you can set a simple flag (i.e. a bit) in a global variable, and then test-and-clear it[67] in your main loop. If the flag is set, you then call the appropriate signaling service prior to calling `OSSched()`, like this:

```
while (1) {
  disable_interrupts();
  localFlag = flag;
  flag = 0;
  enable_interrupts();
  if (localFlag) {
    OSSignalBinSem(binSemP);
  }
  OSSched();
}
```

This disadvantage of this approach is that it does not preserve the order in which events occur, whereas signaling from an ISR will preserve that order. This may affect the behavior of complex systems.

---

66 `timer.c`.

67 Interrupts should be disabled while you test and clear the flag.

Building Projects

What warning level should I use when building Salvo projects?

Use the compiler's default warning level. More pedantic warning levels may generate warnings that in some cases cannot be avoided, and thus cause unnecessary confusion.

What optimization level should I use when building Salvo projects?

Use the maximum optimization unless suggested otherwise.

Miscellaneous

Can Salvo run on a 12-bit PICmicro with only a 2-level call…return stack?

Yes. Certain compilers (e.g. HI-TECH PICC) circumvent this limitation by converting all function calls into long jumps through table lookup. Therefore function calls require some additional overhead and ROM, but call graphs of arbitrary depth are possible.

Will Salvo change my approach to embedded programming?

Maybe. Stranger things have happened … ☺

Chapter 7 • Reference

Run-Time Architecture

In order to run properly, every Salvo application must follow three basic rules. Failure to follow these rules may result in an application that compiles successfully, but does not run as expected. These rules are explained below.

Rule #1: Every Task Needs a Context Switch

Each Salvo task must have *at least one context switch.*

Tip In Salvo, context switches are denoted by a "`OS_`" prefix.

Functions with just an "`OS`" prefix (e.g. `OSSignalBinSem()`) are *not* context switches and may usually be called from anywhere in the Salvo application.

```
void HappyTask ( void )
{
  while (1) {
    …
    OS_Delay(10); // Return here in 10 ticks.
    …
  }
}
```

Listing 32: Task with a Proper Context Switch

In Listing 32 above, `HappyTask()` uses a single context switch (via `OS_Delay()`) to yield to the scheduler during its delay of 10 system ticks. During the delay period, the task is in the delayed state, and the application is free to run other, eligible tasks. Whenever the scheduler dispatches `HappyTask()`, `HappyTask()` will run the code inside its infinite loop, returning to the scheduler via `OS_Delay()`.

Note The requirement of having at least one context-switch per task is a general one for cooperative RTOSes and is not specific to Salvo.

Note The number of context switches a Salvo task can have is limited only by available program memory.

```
void ForlornTask ( void )
{
  putchar('!'); // Bad – untimely exit from task.
}

void StuckTask ( void )
{
  while (1) {
    MyFn(); // Bad – never returns to scheduler.
  }
}
```

Listing 33: Tasks that Fail to Context Switch

In Listing 33 above, `ForlornTask()` has no context switch. As a result, when the scheduler dispatches that task, it will call `putchar()` once *and then the application will continue with whatever code lies in program memory* after `ForlornTask()`.[68] `ForlornTask()` will *not* yield to the scheduler immediately after `MyFn()` is executed. Therefore the application's behavior is unpredictable.

Also in Listing 33 above, once the scheduler dispatches `StuckTask()`, it will call `MyFn()` indefinitely, and will never yield back to the scheduler. While this behavior is predictable, it is not desirable, as all multitasking will stop.

Rule #2: Context Switches May Only Occur in Tasks

The only valid location for a Salvo context switch is *within a task* (see In Listing 32, above).

68 It is likely to continue "into" `StuckTask()` if and only if the linker has placed `StuckTask()` immediately after `ForlornTask()` in memory.

```
void StuckTask ( void )
{
  while (1) {
    MyFn(); // Bad – where's the context switch?
  }
}

void MyFn ( void )
{
  DoThings();
  OS_Yield();  // Bad – not allowed inside a
               //  called function.
}
```

Listing 34: Incorrectly Context-Switching Outside of a Task

In Listing 34 above, the scheduler will dispatch `StuckTask()` and the task will, in turn, call `MyFn()`. After `MyFn()` calls `DoThings()`, it will attempt to yield to the scheduler via `OS_Yield()`. This will fail, as Salvo's context-switcher is not designed for yielding back to the scheduler at any call…return level other than the task's. The run-time behavior when violating this rule is unpredictable.

In C, the ability to context-switch outside of a task, at arbitrary call…return stack levels, requires considerable RAM for saving call…return addresses, function parameters and local (auto) variables. Salvo is designed expressly to minimize RAM requirements, and therefore does not support context-switching outside of tasks.

Note Context switches may not occur in mainline (background) code outside of tasks, nor in interrupt service routines (ISRs).

Rule #3: Persistent Local Variables Must be Declared as Static

Every local variable used in a Salvo task in a manner that requires persistence across context switches must be declared as `static`.

```
void TaskLowPrio ( void )
{
  static int i;


  while (1) {
    i = 20000;
    do {
      LED_PORT &= ~LED_PORT_MASK;
      LED_PORT |= ((i >> 8) & LED_PORT_MASK);
      OS_Delay(1);
```

```
    } while (--i);
  }
}
```

Listing 35: Task Using Persistent Local Variable

In Listing 35 above, `TaskLowPrio()`outputs the upper 8 bits of the loop counter `i` to eight LEDs every system tick while decrementing `i`. If `i` were not declared as static, `i` 's value would be unpredictable and so would be the output to the LED port.

Declaring local variables that require persistence as `static` is necessary because Salvo's context switcher performs a minimal context save that does not include local variables. Other tasks, functions and ISRs may use the memory allocated to the local variable for their own purposes when the task is not running, changing it in unpredictable ways.

With care, local variables can be used as auto variables in Salvo tasks. Whenever a local variable is initialized and fully used before the next context switch, it can be declared as a simple local (auto) variable instead of a `static` one.

```
void TaskCountElements( void )
{
  char i;
  element * p;


  while (1) {
    OS_WaitBinSem(BINSEM_COUNT_LIST);
    i = 0;
    p = headP;
    while (1) {
      if (p!=0) {
        i++;
        p = p->nextP;
      }
      else {
        break;
      }
    }
    LCDWrite("The list has %d elements.\n", i);
    …
    OS_Delay(delay);
    …
  }
}
```

Listing 36: Task Using Auto Local Variables

In Listing 36 above, `i` and `p` are used as local (auto) variables to traverse a linked list and count the number of objects therein. Afterwards the result is displayed on an LCD, and the task continues.

Note When in doubt, declare local variables as static.

User Services

This section describes the Salvo user services that you will use to build your multitasking application. Each user service description includes information on:

- • the service *type* (function or macro),
- • the service *prototype* (for a function) or *declaration* (for a macro),
- • where the service is *callable from* (the foreground, the background or within a task),
- • which Salvo C source or include files *contain* the source code for the service,
- • which configuration options (if any) *enable* the service,
- • which configuration options (if any) *affect* the service (i.e. alter its execution speed or code size),
- • a *description* of what the service does,
- • the *parameter(s)* (if any) expected by the service call,
- • the service's *return* value(s) (if any),
- • the service's *stack usage* (if any), in terms of levels of call…return stack used,[69]
- • *notes* particular to the service,
- • *related* services and
- • an *example* using the service.

Salvo functions comprise the majority of the user services you will call from C in your application. Salvo user services that do not result in a context switch are implemented as functions and are prefixed by just "`OS`".

Salvo uses macros wherever a context-switch is implicit in the action being performed (e.g. delaying for a number of ticks, via `OS_Delay()`). All of Salvo's services that result in a context-switch are implemented via macros and are prefixed by "`OS_`".

---

[69] For call…return stack depth calculations, `OSUSE_INSELIG_MACRO` is assumed to be the default value, `TRUE`. If `FALSE`, those services that cause a task to be placed in the eligible, delay and/or event queue(s) will consume an additional call…return stack level. Stack usage does not take into account any library functions invoked by the compiler.

Note Salvo context-switching services are implemented as macros and do not have return values.

It is important not to confuse a Salvo macro with its underlying function. For instance, the `OS_Delay()` macro will cause the current task to delay for the specified number of system ticks. On the other hand, using the `OSDelay()` function directly will have unpredictable results, and your application may crash as a result. These underlying functions are intended for use only within a Salvo macro, and are therefore not documented in this section. For the curious, they can be viewed in the Salvo source code.

Note Some services (e.g. `OSCreateXyz()` and `OSSignalXyz()`) can be either a macro that invokes a function, or a standalone function, depending on `OSCOMBINE_EVENT_SERVICES`. In all cases the argument list and return value and type are identical.

When compiling and linking Salvo into your application, the size and speed of many user services is dependent on the chosen configuration. By referring to the detailed descriptions of each user service below and inspecting the output of your compiler, you may be able to correlate changes in the size (in instructions) and/or speed (in cycles) of the Salvo services in your application against changes you've made to your compile-time configuration. Remember that each time you change the configuration options, you must recompile all of Salvo before linking it into your application.

Note The *foreground* is the interrupt level of your application. The *background* is the non-interrupt level, and includes `main()`, Salvo tasks and all other functions not called via interrupts.

This page is intentionally left blank.

OS_Delay(): Delay the Current Task and Context-switch

Type:	Macro (invokes `OSDelay()`)
Declaration:	`OS_Delay ( OStypeDelay delay);`
Callable from:	Task only
Contained in:	`salvompt.h`
Enabled by:	`OSBYTES_OF_DELAY`
Affected by:	`OSENABLE_STACK_CHECKING`, `OSLOGGING`
Description:	Delay the current task by the amount specified. Return to scheduler.
Parameters:	`delay`: an integer (>=0) specifying the desired delay in system ticks.
Returns:	–
Stack Usage:	2

Notes

A delay of 0 will stop the current task. A non-zero delay will delay[70] the current task by the number of ticks specified relative to the current value of the system timer.

Do not call `OS_Delay()` from within an ISR!

In order to use delays, Salvo's timer must be installed.

Long delays can be accomplished in a variety of ways – See "Timer and Timing" in *Chapter 6 • Frequently Asked Questions (FAQ)*.

In the example below (system tick rate = 40Hz, t = 25ms, Hitachi 44780 LCD controller), `OS_Delay()` is used to delay the LCD task `TaskDisp()` during startup while the LCD is being configured. By using `OS_Delay()` instead of an in-line delay, the other tasks may run while `TaskDisp()` is delayed and the LCD is initialized.

See Also

`OS_DelayTS()`, `OS_Stop()`, `OSTimer()`

[70] When delaying a task repetitively, remember that there is an additional, unpredictable delay between when the task's delay expires and when it actually runs. This may happen if there are other, higher-priority tasks eligible to run when the delayed task's delay expires. This can affect a task's "loop delay."

Example

```
#define LCD_CMD_REG    0    /* for commands    */
#define LCD_DATA_REG   1    /* for data        */
#define LCD_CMD_CLS    0x01 /* clear display   */
#define LCD_CMD_MODE   0x06 /* auto-inc address*/
#define LCD_CMD_ON_OFF 0x0C /* on, no cursor,  */
                            /*  no blink       */
#define LCD_CMD_FN_SET   0x3F
#define LCD_BITMASK_RS   0x01 /* reg select    */
#define LCD_BITMASK_RW   0x02 /* read/-write   */
#define LCD_BITMASK_E    0x04 /* E (strobe)    */

void TaskDisp ( void )
{
  static OStypeMsgP msgP;

  /* initialize the LCD Display                */
  char i;         /* doesn't need to be static */

  TRISD = 0x00; /* all LCD ports are outputs   */
  TRISE = 0x00; /*  "                          */
  PORTE = 0x00; /* RS=0, -WRITE, E=0           */

  /* we want to talk to the command register,  */
  /*  and we'll wait 50ms to ensure it's       */
  /*  listening.                               */
  LCDSelReg(LCD_CMD_REG);
  OS_Delay(2);

  /* Hitachi recommends 4 consecutive writes   */
  /*  to this register ...                     */
  for (i = 4; i--; )
    LCDWrData(LCD_CMD_FN_SET);

  /* configure LCD the "standard" way.         */
  LCDWrData(LCD_CMD_ON_OFF);
  LCDWrData(LCD_CMD_MODE);
  LCDWrData(LCD_CMD_CLS);

  /* wait another 50ms.                        */
  OS_Delay(2);

  /* now we're done initializing LCD display.  */
  …

  while (1) {
    OS_WaitMsg(MSG_UPDATE_DISP_P, &msgP,
      OSNO_TIMEOUT);
    …
  }
}
```

OS_DelayTS(): Delay the Current Task Relative to its Timestamp and Context-switch

Type:	Macro (invokes `OSDelay()`)
Declaration:	`OS_DelayTS ( OStypeDelay delay);`
Callable from:	Task only
Contained in:	`salvompt.h`
Enabled by:	`OSBYTES_OF_DELAY`, `OSBYTES_OF_TICKS`
Affected by:	`OSENABLE_STACK_CHECKING`, `OSLOGGING`
Description:	Delay the current task by the amount specified, relative to the task's timestamp. Return to scheduler.
Parameters:	`delay`: an integer (>=0) specifying the desired delay in system ticks.
Returns:	–
Stack Usage:	2

Notes

A delay of 0 will stop the current task. A non-zero delay will delay the current task by the number of ticks specified relative to the task's timestamp. The timestamp is automatically recorded by `OSInit()` and whenever a task's delay times out. In order to use delays with timestamps, Salvo's timer must be installed and the counting of system ticks must be enabled via `OSBYTES_OF_TICKS`.

If more than `delay` and less than 2 x `delay` system ticks occur between the task's delay expiring and the task running,[71] the task will attempt to resynchronize itself for the *following* delay period. The behavior for more than 2 x `delay` ticks is undefined.[72]

Do not call `OS_Delay()` from within an ISR!

In the example below, `TaskA()` will always run every fourth system tick because it is synchronized to the system timer. As long as the delay between the task's delay expiring and the task actually running[73] never exceeds 2 `delay` periods, the task will always run at t_0 + (number of iterations x `delay`).

---

71 I.e. the task is " very late".

72 In this situation you may need to chose a longer system tick period.

73 This might happen if, for instance, `TaskA()`'s priority is low, and there are other tasks eligible to run.

See Also

OS_Delay(), OSGetTS(), OSSetTS(), OS_Stop(), OSSyncTS(), OSTimer()

Example

```
void TaskA ( void )
{
  while (1) {
    OS_DelayTS(4);
    …
  }
}

int main ( void )
{
  …

  OSInit();

  OSCreateTask(TaskA, OSTCBP(1), 4);
  …

  enable_interrupts();

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

OS_Destroy(): Destroy the Current Task and Context-switch

Type:	Macro (invokes `OSDestroy()`)
Declaration:	`OS_Destroy ();`
Callable from:	Task only
Contained in:	`salvompt.h`
Enabled by:	–
Affected by:	`OSENABLE_STACK_CHECKING`
Description:	Destroy the current task. Return to scheduler.
Parameters:	–
Returns:	–
Stack Usage:	1

Notes

Once a task is destroyed, it cannot be restarted. However, a new task can be created in its place by using the same tcb.

Do not call `OS_Destroy()` from within an ISR!

In the example below, `TaskStartup()` creates and starts most of the other tasks in the application. `TaskDisp()` (see example for `OS_Delay()`) will run immediately after `TaskStartup()` begins its two-second delay. When the delay expires, `TaskStartup()` will resume, creating and starting `TaskMsg()`, `TaskRdKey()`, `TaskStatus()`, `TaskTx()` and `TaskRx()`. However, none of these tasks will run until `TaskStartup()` destroys itself and returns to the scheduler. Once `TaskRx()` runs it will create `TaskRcvRsp()` in place of `TaskStatus()`, thereby reusing the tcb for another task. `TaskStartup()` is not structured as an infinite loop – rather, it's simply a one-time sequence of events, which ends when `TaskStartup()` destroys itself and returns to the scheduler.

See Also

`OSCreateTask()`, `OSStop()`

Example

```
void TaskStartup ( void )
{
  /* create all the tasks we need early on.    */
  /*  Some of these tasks create other tasks   */
  /*  and resources! Start them up, too.       */

  /* TaskDisp() handles display updates. It    */
  /*  also creates MSG_DISP & SEM_UPDATE_DISP. */
  OSCreateTask(TaskDisp, TASK_DISP_P,
    TASK_DISP_PRIO);

  /* Leave startup screen showing for 2s.      */
  OS_Delay(TWO_SEC, TaskStartup1);

  /* TaskMsg() flashes messages. It also       */
  /*  creates MSG_FLASH_STRING.                */
  OSCreateTask(TaskMsg, TASK_MSG_P,
    TASK_MSG_PRIO);

  /* TaskRdKey() reads the keypad. It also     */
  /*  creates MSG_KEY_PRESSED and creates and  */
  /*  starts TaskRcvKeys().                    */
  OSCreateTask(TaskRdKey, TASK_RD_KEY_P,
    TASK_RD_KEY_PRIO);

  /* TaskStatus() monitors the PSR on Driver.  */
  /*  It also creates MSG_WAKE_STATUS and      */
  /*  MSG_LONG_OP_DONE.                        */
  OSCreateTask(TaskStatus, TASK_STATUS_P,
    TASK_STATUS_PRIO);

  /* TaskTx() send cmds out to the Driver. It  */
  /*  also creates MSG_WAKE_TX, MSG_RSP_RCVD   */
  /*  and MSG_TX_BUFF_EMPTY.                   */
  OSCreateTask(TaskTx, TASK_TX_P, TASK_TX_PRIO);

  /* TaskRx() receives responses back from the */
  /*  Driver. It also creates SEM_RX_RBUFF and */
  /*  creates and starts TaskRcvRsp().         */
  OSCreateTask(TaskRx, TASK_RX_P, TASK_RX_PRIO);

  /* we're finished starting up, so kill this  */
  /*  task permanently. TaskRcvKeys() will     */
  /*  "take over" its tcb – see                */
  /*  TaskRdKeys().                            */
  OS_Destroy();
}
```

OS_Replace(): Replace the Current Task and Context-switch

Type:	Macro (invokes `OSCreateTask()`)
Declaration:	`OS_Replace (tFP, prio);`
Callable from:	Task only
Contained in:	`salvompt.h`
Enabled by:	–
Affected by:	–
Description:	Replace the current task with the one specified. Return to scheduler.
Parameters:	`tFP`: a pointer to the task's start address. This is also the task's function prototype name. `prio`: the desired priority for the task. If OR'd with `OSDONT_START_TASK`, the task will not be started.
Returns:	–
Stack Usage:	3

Notes

The task that replaces the current task will use the same tcb. Once a task is replaced, it can be restarted only with a call to `OSCreateTask()`.

Do not call `OS_Replace()` from within an ISR!

`OS_Replace()` is useful in various situations. For instance, you could have a system initialization task that replaces itself with one of your run-time tasks when all initialization is complete. Or you could replace a large task containing a state machine with independent tasks for each state. `OS_Replace()` can be used wherever multiple tasks need never run at the same time, thus conserving tcb RAM.

In the example below, `TaskCountUp()` runs first. After 250 iterations, it replaces itself with `TaskCountDown()`. `TaskCountDown()` also runs for 250 iterations, but at a faster rate, and replaces itself with `TaskCountUp()` when done. The task priorities can be varied, as shown. This continues indefinitely. Only a single tcb is used.

See Also

`OSCreateTask()`, `OSDestroyTask()`, `OSStop()`

Example

```
void TaskCountUp ( void );
void TaskCountDown ( void );

void TaskCountUp ( void )
{
  static char i;

  for (i = 0; i <= 250; i++) {
    PORTB = i;

    OS_Delay(25);
  }

  OS_Replace(TaskCountDown, 5);
}


void TaskCountDown ( void )
{
  static char i;

  for (i = 250; i >= 0; i--) {
    PORTB = i;

    OS_Delay(5);
  }

  OS_Replace(TaskCountUp, 3);
}


int main ( void )
{
  …

  OSInit();

  OSCreateTask(TaskCountUp, OSTCBP(1), 4);

  …

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

OS_SetPrio(): Change the Current Task's Priority and Context-switch

Type:	Macro (invokes `OSSetPrio()`)
Declaration:	`OS_SetPrio ( OStypePrio prio);`
Callable from:	Task only
Contained in:	`salvompt.h`
Enabled by:	–
Affected by:	`OSENABLE_STACK_CHECKING`
Description:	Change the current task's priority. Return to scheduler.
Parameters:	`prio`: the desired (new) priority for the current task.
Returns:	–
Stack Usage:	1

Notes

0 (`OSHIGHEST_PRIO`) is the highest priority, 15 (`OSLOWEST_PRIO`) is the lowest.

Do not call `OS_SetPrio()` from within an ISR!

Tasks can share priorities. Eligible tasks with the same priority will round-robin schedule as long as they are the highest-priority eligible tasks.

The change in priority takes effect when the current task returns to the scheduler.

In the example below, `TaskStartupEtc()` is initially created with a high priority. The first time it runs, it will run at that priority. While running for the first time, it redefines its priority to be a lower one. Each subsequent time it runs, it will run at the lower priority. The task context-switches once at `OS_SetPrio()`, and subsequently at `OS_Yield()`.

See Also

`OSCreateTask()`, `OSGetPrio()`, `OSSetPrio()`, `OSDISABLE_TASK_PRIORITIES`

Example

```
#define MOST_IMPORTANT 0
#define LESS_IMPORTANT 5

int main ( void )
{
  …
  /* startup task gets highest priority.       */
  OSCreateTask(TaskStartupEtc,
    OSTCBP(1), MOST_IMPORTANT);
  …
}

/* while starting up this task runs at         */
/*  the highest priority, then it changes      */
/*  its priority to a lower one.               */
void TaskStartupEtc ( void )
{
  /* do initialization and other               */
  /*  startup code.                            */
  …

  /* MonitorSystem() will always be            */
  /*  called from this task while              */
  /*  running at a lower priority.             */
  OS_SetPrio(LESS_IMPORTANT);

  while (1) {
    MonitorSystem();

    OS_Yield();
  }
}
```

OS_Stop(): Stop the Current Task and Context-switch

Type:	Macro (invokes `OS_Delay()` or `OS_Stop()`)
Declaration:	`OS_Stop ();`
Callable from:	Task only
Contained in:	`salvompt.h`
Enabled by:	–
Affected by:	`OSBYTES_OF_DELAY`, `OSENABLE_STACK_CHECKING`, `OSLOGGING`
Description:	Stop the current task. Return to scheduler.
Parameters:	–
Returns:	–
Stack Usage:	1

Notes

A stopped task can only be restarted via `OSStartTask()`.

Do not call `OS_Stop()` from within an ISR!

If delays are enabled via `OSBYTES_OF_DELAYS`, `OS_Stop()` stops the current task via a call to `OSDelay(0)`. Otherwise it calls `OSStop()`. This is done to reduce the code size of your Salvo application.

In the example below, `TaskRunOnce()` is created and started, and will run as soon as it becomes the highest-priority eligible task. It will run only once. In order to make it run again, a call to `OSStartTask(TASK_RUN_ONCE)` is required. Note that `TaskRunOnce()` would also work without the infinite loop, but subsequent calls to `OSStartTask(TASK_RUN_ONCE)` would result in unpredictable behavior because task execution would resume outside of `TaskRunOnce()`.

See Also

`OSStartTask()`, `OSStopTask()`

Example

```
int main ( void )
{
  …
  OSCreateTask(TaskRunOnce, TASK_RUN_ONCE_P, 6);
  …
}

void TaskRunOnce ( void )
{
  while (1) {
    /* do one-time things ... */
    …
    OS_Stop();
  }
}
```

OS_WaitBinSem(): Context-switch and Wait the Current Task on a Binary Semaphore

Type:	Macro (invokes `OSWaitEvent()`)
Declaration:	`OS_WaitBinSem ( OStypeEcbP ecbP, OStypeDelay timeout);`
Callable from:	Task only
Contained in:	`salvompt.h`
Enabled by:	`OSENABLE_BINARY_SEMAPHORES`, `OSEVENTS`
Affected by:	`OSENABLE_STACK_CHECKING`, `OSENABLE_TIMEOUTS`, `OSLOGGING`
Description:	Wait the current task on a binary semaphore, with a timeout. If the semaphore is 0, return to the scheduler and continue waiting. If the semaphore is 1, reset it to 0 and continue. If the timeout expires before the semaphore becomes 1, continue execution of the task, with the timeout flag set.
Parameters:	`ecbP`: a pointer the binary semaphore's ecb. `timeout`: an integer (>=0) specifying the desired timeout in system ticks.
Returns:	–
Stack Usage:	2

Notes

Specify a timeout of `OSNO_TIMEOUT` if the task is to wait the binary semaphore indefinitely.

Do not call `OS_WaitBinSem()` from within an ISR!

After a timeout occurs the binary semaphore is undefined.

In the example below for a rocket launching system, a rocket is launched via a binary semaphore `BINSEM_LAUNCH_ROCKET` used as a flag. The semaphore is initialized to zero so that the rocket does not launch on system power-up.[74] Once the rocket is ready and the order has been given to launch (via `OSSignalBinSem()` elsewhere in the code), `TaskLaunchRocket()` starts the rocket on its journey. Since the rocket cannot be recalled, there is no need to continue running `TaskLaunchRocket()`, and it simply stops itself. There-

[74] That would be undesirable.

fore in order to launch a second rocket, the system must be re-started.

See Also

`OSCreateBinSem()`, `OSReadBinSem()`, `OSSignalBinSem()`, `OSTryBinSem()`

Example

```
#define BINSEM_LAUNCH_ROCKET_P OSECBP(2)

…

/* startup code: no clearance given to launch  */
/*  rocket.                                    */
OSCreateBinSem(BINSEM_LAUNCH_ROCKET_P, 0);

…

void TaskLaunchRocket ( void )
{
  /* wait here forever until the order is      */
  /* given to launch the rocket.               */
  OS_WaitBinSem(BINSEM_LAUNCH_ROCKET_P,
    OSNO_TIMEOUT);

  /* launch rocket.                            */
  IgniteRocketEngines();
  …

  /* rocket is on its way, therefore task is   */
  /*  no longer needed.                        */
  OS_Stop();
}
```

OS_WaitEFlag(): Context-switch and Wait the Current Task on an Event Flag

Type:	Macro (invokes `OSWaitEvent()`)
Declaration:	`OS_WaitEFlag ( OStypeEcbP ecbP, OStypeEFlag mask, OStypeOption options, OStypeDelay timeout);`
Callable from:	Task only
Contained in:	`salvompt.h`
Enabled by:	`OSENABLE_EVENT_FLAGS`, `OSEVENTS`
Affected by:	`OSBYTES_OF_EVENT_FLAGS`, `OSENABLE_STACK_CHECKING`, `OSENABLE_TIMEOUTS`, `OSLOGGING`
Description:	Wait the current task on an event flag, with a timeout. The bits in the event flag specified by the `mask` parameter are tested according to the condition specified by the `options` parameter. If the condition is not satisfied, return to the scheduler and continue waiting. If the condition is satisfied, continue without changing the event flag. If the timeout expires before the condition is satisfied, continue execution of the task, with the timeout flag set.
Parameters:	`ecbP`: a pointer the event flag's ecb. `mask`: a bitmask to apply to the event flag. `options`: `OSANY_BITS`, `OSALL_BITS` or `OSEXACT_BITS`. `timeout`: an integer (>=0) specifying the desired timeout in system ticks.
Returns:	–
Stack Usage:	2

Notes

Specify a timeout of `OSNO_TIMEOUT` if the task is to wait the event flag indefinitely.

Do not call `OS_WaitEFlag()` from within an ISR!

After a timeout occurs the event flag is undefined.

Salvo's event flag bits are "active high", i.e. an event is said to have occurred when its corresponding bit in the event flag is set to 1. The event has not occurred if the bit is cleared to 0.

When specifying OSANY_BITS, OS_WaitEFlag() checks if any of the corresponding mask parameter's bits in the event flag are set to 1, and if so, the task continues. With OSALL_BITS, all of the corresponding mask parameter's bits must be set to 1 for the task to continue. With OSEXACT_BITS, the event flag must match the mask parameter exactly for the task to continue.

In contrast to Salvo's other event services, successfully waiting an event flag *does not automatically reset the bits in the event flag* that resulted in the condition being satisfied. You must explicitly clear event flag bits via OSClrEFlag(). Failing to clear the appropriate event flag bits will cause unpredictable results – generally the task will fail to yield back to the scheduler.

In the example below for a secure access system with a power-assisted door, three separate interlocks must be deactivated before the door can be opened by TaskOpenDoor(). The three least significant bits of an eight-bit event flag are used to signify that the bottom, side and top interlocks have been deactivated by TaskReleaseBottomLock(), etc. Bits three and four in the event flag signify whether the door is fully open or fully closed and are maintained by TaskCheckDoor(). When the door is fully open, it's safe to re-activate (release) the door locks so that when it closes it's automatically locked shut.

The remaining three bits in the eight-bit event flag can be used for other purposes entirely independent of the interlock mechanism.

See Also

OSCreateEFlag(), OSClrEFlag(), OSReadEFlag(), OSSetEFlag()

Example

```
#define DOOR_EFLAG_P   OSECBP(1)
#define BOTTOM         0x01
#define SIDE           0x02
#define TOP            0x04
#define OPEN           0x08
#define CLOSED         0x10

void TaskReleaseBottomLock ( void )
{
  while (1) {
    /* wait for request to release bottom lock.*/
    …
    /* release bottom door lock.               */
    ReleaseBottomLock();
```

```
    /* tell TaskOpenDoor() about it.          */
    OSSetEFlag(DOOR_EFLAG_P, BOTTOM);

    /* verify that door is fully opened by    */
    /*  by waiting for the signal.            */
    OS_WaitEFlag(DOOR_EFLAG_P, OPEN, OSANY_BITS,
      OSNO_TIMEOUT);

    /* re-engage bottom door lock. When door  */
    /*  closes it will remain locked.         */
    OSClrEFlag(DOOR_EFLAG_P, BOTTOM);
    EngageBottomLock();

    /* remain inactive until the door closes. */
    OS_WaitEFlag(DOOR_EFLAG_P, CLOSED, OSANY_BITS,
      OSNO_TIMEOUT);
  }
}

void TaskReleaseSideLock ( void )
{
  while (1) {
    …
    ReleaseSideLock();
    OSSetEFlag(DOOR_EFLAG_P, SIDE);
    OS_WaitEFlag(DOOR_EFLAG_P, OPEN, OSANY_BITS,
      OSNO_TIMEOUT);
    OSClrEFlag(DOOR_EFLAG_P, SIDE);
    EngageSideLock();
    OS_WaitEFlag(DOOR_EFLAG_P, CLOSED, OSANY_BITS,
      OSNO_TIMEOUT);
  }
}

void TaskReleaseTopLock ( void )
{
  while (1) {
    …
    ReleaseTopLock();
    OSSetEFlag(DOOR_EFLAG_P, TOP);
    OS_WaitEFlag(DOOR_EFLAG_P, OPEN, OSANY_BITS,
      OSNO_TIMEOUT);
    OSClrEFlag(DOOR_EFLAG_P, TOP);
    EngageTopLock();
    OS_WaitEFlag(DOOR_EFLAG_P, CLOSED, OSANY_BITS,
      OSNO_TIMEOUT);
  }
}

void TaskOpenTheDoor ( void )
{
  /* door is initially closed.                */
  OSCreateEFlag(DOOR_EFLAG_P, CLOSED );

  while (1) {
    /* wait forever for all interlocks to be  */
    /*  released.                             */
```

```
    OS_WaitEFlag(DOOR_EFLAG_P,
      TOP | BOTTOM | SIDE, OSALL_BITS,
      OSNO_TIMEOUT);

    /* all locks are released – open door.     */
    OpenDoor();

    /* wait for the door to close again before */
    /*  repeating the cycle.                   */
    OS_WaitEFlag(DOOR_EFLAG_P, CLOSED, OSANY_BITS,
      OSNO_TIMEOUT);
  }
}

void TaskCheckDoor ( void )
{
  while (1) {
    /* check sensors every 1s.                 */
    OS_Delay(100);

    /* if open door has closed contact on its  */
    /*  sensor, then door must be open!        */
    if (DoorFullyOpen()) {
      OSSetEFlag(DOOR_EFLAG_P, OPEN);
    }
    else {
      OSClrEFlag(DOOR_EFLAG_P, OPEN);
    }

    /* similarly, if closed door has closed    */
    /*  contact on its sensor, then it must be */
    /*  closed!                                */
    if (DoorFullyClosed()) {
      OSSetEFlag(DOOR_EFLAG_P, CLOSED);
    }
    else {
      OSClrEFlag(DOOR_EFLAG_P, CLOSED);
    }
  }
}
```

OS_WaitMsg(): Context-switch and Wait the Current Task on a Message

Type:	Macro (invokes `OSWaitEvent()`)
Declaration:	`OS_WaitMsg ( OStypeEcbP ecbP, OStypeMsg msgP, OStypeDelay timeout);`
Callable from:	Task only
Contained in:	`salvompt.h`
Enabled by:	`OSENABLE_MESSAGES`, `OSEVENTS`
Affected by:	`OSENABLE_STACK_CHECKING`, `OSENABLE_TIMEOUTS`, `OSLOGGING`
Description:	Wait the current task on a message, with a timeout. If the message is available, make `msgP` point to it, and continue. If it's not available, return to the scheduler and continue waiting. If the timeout expires before the message becomes available, continue execution of the task, with the timeout flag set.
Parameters:	`ecbP`: a to the message's ecb. `msgP`: a pointer to a message `timeout`: an integer (>=0) specifying the desired timeout in system ticks.
Returns:	–
Stack Usage:	2

Notes

Specify a timeout of `OSNO_TIMEOUT` if the task is to wait the message indefinitely.

Do not call `OS_WaitMsg()` from within an ISR!

Should a timeout occur while waiting the message queue, the message pointer is *invalid*. A task may only extract the message's contents via the message pointer if it has successfully waited the message queue event without a timeout.

In the example below, `TaskRcvKeys()` waits forever for the message `MSG_KEY_PRESSED`. No processing power is allocated to `TaskRcvKeys()` while it is waiting. Once the message arrives, its contents (the key pressed) are copied to a local variable and appropriate action is taken. Note that correct casting and dereferencing of the pointer msgP are required in order to extract the contents of

the message correctly. After `TaskRcvKeys()` acts on the key pressed, it resumes waiting for the message.

See Also

`OSCreateMsg()`, `OSReadMsg()`, `OSSignalMsg()`, `OSTryMsg()`

Example

```
void TaskRcvKeys ( void )
{
  static char key;
  static OStypeMsgP msgP;

  while (1) {
    /* Wait forever for a new key.              */
    OS_WaitMsg(MSG_KEY_PRESSED_P,
      &msgP, OSNO_TIMEOUT);

    /* User pressed a key! – get it.            */
    key = *(char *) msgP;

    /* Act on key pressed.                      */
    switch (tolower(key)) {
      case KEY_MEM:
        …
    }
  }
}
```

OS_WaitMsgQ(): Context-switch and Wait the Current Task on a Message Queue

Type:	Macro (invokes `OSWaitEvent()`)

Declaration:	`OS_WaitMsgQ (`
	`  OStypeEcbP  ecbP,`
	`  OStypeMsg   msgP,`
	`  OStypeDelay timeout);`

Callable from:	Task only
Contained in:	`salvompt.h`
Enabled by:	`OSENABLE_MESSAGE_QUEUES`, `OSEVENTS`
Affected by:	`OSLOGGING`, `OSENABLE_STACK_CHECKING`
Description:	Wait the current task on a message queue, with a timeout. If the message queue contains a message, make `msgP` point to it, and continue. If it's empty, return to the scheduler and continue waiting. If the timeout expires before a message is added to the message queue, continue execution of the task, with the timeout flag set.
Parameters:	`ecbP`: a pointer to the message queue's ecb. `msgP`: a pointer to a message. `timeout`: an integer (>=0) specifying the desired timeout in system ticks.
Returns:	–
Stack Usage:	2

Notes

Specify a timeout of `OSNO_TIMEOUT` if the task is to wait the message queue indefinitely.

Do not call `OS_WaitMsgQ()` from within an ISR!

Should a timeout occur while waiting the message queue, the message pointer is *invalid*. A task may only extract the message's contents via the message pointer if it has successfully waited the message queue event without a timeout.

In the first example below, `TaskRcvInt()` forever waits a message queue containing messages to objects of type `int`. When a message arrives, the `TaskRcvInt()` extracts the message from the message queue and prints a message. The task continues printing messages until the message queue is empty, whereupon the task a context switch occurs.

Message queues can also be used to pass raw data. In the second example below, `TaskRcvRawData()` extracts `unsigned-char`-sized raw data instead of message pointers from the message queue.

Note `sizeof(raw data type)` must not exceed `sizeof(OStypeMsgP)`. E.g. on a target with 16-bit `void` pointers, raw data of up to 16 bits in size can be passed in each message.

See Also

`OSCreateMsgQ()`, `OSReadMsgQ()`, `OSSignalMsgQ()`, `OSTryMsgQ()`

Example #1

```
void TaskRcvInt ( void )
{
  static int myNum;
  static OStypeMsgP msgP;

  while (1) {
    /* Wait forever for a message.              */
    OS_WaitMsgQ(MSGQ1, &msgP, OSNO_TIMEOUT);

    /* A message has arrived – get it.          */
    myNum = *(int *) msgP;

    printf("The number was %d. \n", myNum);
  }
}
```

Example #2

```
/* send raw data in this message.               */
OSSignalMsgQ(MSGQ1_P, (OStypeMsgP) 'r');
…
void TaskRcvRawData( void )
{
  OStypeMsgP msgP;
  unsigned char rcvdChar;

  while (1) {
    /* wait forever for a message.              */
    OS_WaitMsgQ(MSGQ1_P, &msgP);

    /* cast (don't dereference) message         */
    /*  pointer since raw data was passed.      */
    localUC = (unsigned char) msgP;
    printf("received %c \n", rcvdChar);
  }
}
```

OS_WaitSem(): Context-switch and Wait the Current Task on a Semaphore

Type:	Macro (invokes `OSWaitEvent()`)
Declaration:	`OS_WaitSem ( OStypeEcbP ecbP, OStypeDelay timeout);`
Callable from:	Task only
Contained in:	`salvompt.h`
Enabled by:	`OSENABLE_SEMAPHORES`, `OSEVENTS`
Affected by:	`OSENABLE_STACK_CHECKING`, `OSENABLE_TIMEOUTS`, `OSLOGGING`
Description:	Wait the current task on a semaphore, with a timeout. If the semaphore is 0, return to the scheduler and continue waiting. If the semaphore is non-zero, decrement the semaphore and continue. If the timeout expires before the semaphore becomes non-zero, continue execution of the task, with the timeout flag set.
Parameters:	`ecbP`: a pointer to the semaphore's ecb. `timeout`: an integer (>=0) specifying the desired timeout in system ticks.
Returns:	–
Stack Usage:	2

Notes

Specify a timeout of `OSNO_TIMEOUT` if the task is to wait the semaphore indefinitely.

Do not call `OS_WaitSem()` from within an ISR!

After a timeout occurs the semaphore is undefined.

In the example below, `TaskRcvRsp()` removes incoming characters from a receive buffer one at a time and processes them. `SEM_RX_BUFF` always indicates how many characters are present in `rxBuff[]`, and is signaled by another task which puts the characters into `rxBuff[]` one-by-one. `TaskRcvRsp()` runs as long as there are characters present in `rxBuff[]` – when is empty, `TaskRcvRsp()` waits. By using a semaphore for inter-task communications there's no need to poll for the existence of characters in the buffer, and hence overall performance is improved.

See Also

`OSCreateSem()`, `OSReadSem()`, `OSSignalSem()`, `OSTrySem()`

Example

```
void TaskRcvRsp ( void )
{
  static char rcChar;

  while (1) {
    /* wait until there are response chars     */
    /*  waiting ... (TaskRx() signals us when  */
    /*  there are).                            */
    OS_WaitSem(SEM_RX_RBUFF_P, OSNO_TIMEOUT);

    /* then deal with them.                    */
    /* get the next char from the buffer       */
    rcChar = rxBuff[rxHead];
    rxHead++;
    if (rxHead >= SIZEOF_RX_BUFF) {
      rxHead = 0;
    }
    rxCount--;

    /* alphanumeric characters are the _only_  */
    /*  chars (other than reserved ones) we    */
    /*  expect to see in the incoming rcChar.  */
    if (isalnum(rcChar) || ( rcChar == '-' ))
    {
      …
    }
    else
    {
      …
    }
  }
}
```

OS_Yield(): Context-switch

Type:	Macro
Declaration:	`OS_Yield ();`
Callable from:	Task only
Contained in:	`salvompt.h`
Enabled by:	–
Affected by:	–
Description:	Return to scheduler.
Parameters:	–
Returns:	–
Stack Usage:	1 or 2, depending on compiler and target.

Notes

`OS_Yield()` causes an immediate, unconditional return to the scheduler.

Do not call `OS_Yield()` from within an ISR!

In the example below, `TaskUnimportant()` is assigned a low priority and runs only when no other higher-priority tasks are eligible to run. Each time it runs, it increments a counter by 1.

Example

```
unsigned long int unimportantCounter = 0;

int main ( void )
{

  OSCreateTask(TaskUnimportant,
    TASK_UNIMPORTANT_P, 14);

  …
}

void TaskUnimportant ( void )
{
  while (1) {
    unimportantCounter++;

    OS_Yield();
  }
}
```

OSClrEFlag(): Clear Event Flag Bit(s)

Type:	Function
Prototype:	`OStypeErr OSClrEFlag ( OStypeEcbP ecbP, OStypeEFlag mask );`
Callable from:	Anywhere
Contained in:	`salvoeflag.c`, `salvoevent.c`
Enabled by:	`OSENABLE_EVENT_FLAGS`, `OSEVENTS`
Affected by:	`OSCALL_OSSIGNALEVENT`, `OSENABLE_STACK_CHECKING`, `OSCOMBINE_EVENT_SERVICES`, `OSLOGGING`, `OSUSE_EVENT_TYPES`
Description:	Clear bits in an event flag. No task will be made eligible by this operation.
Parameters:	`ecbP`: a pointer to the event flag's ecb. `mask`: mask of bits to be cleared.
Returns:	`OSERR_BAD_P` if event flag pointer is incorrectly specified. `OSERR_EVENT_BAD_TYPE` if specified event is not an event flag. `OSERR_EVENT_CB_UNINIT` if event flag's control block is uninitialized. `OSERR_EVENT_FULL` if event flag doesn't change. `OSNOERR` if event flag bits are successfully cleared.
Stack Usage:	1

Notes

No tasks are made eligible by clearing bits in an event flag.

This service is typically used immediately after successfully waiting an event flag, since the bits in question are not automatically cleared by `OS_WaitEFlag()`.

In the example below, a task is configured to run only when two particular bits in an event flag are set. It then clears one of them and returns to the waiting state. It will run again when and only when both bits are set.

See Also

`OS_WaitEFlag()`, `OSCreateEFlag()`, `OSReadEFlag()`, `OSSetEFlag()`

Example

```
#define EFLAG1_P OSECBP(2)
…
void TaskC ( void )
{
  while (1) {
    /* wait forever for both bits to be set    */
    OS_WaitEFlag(EFLAG1_P, 0x0C, OSALL_BITS,
      OSNO_TIMEOUT);

    /* clear the upper bit, leave the lower    */
    /*  one alone.                             */
    OSClrEFlag(EFLAG1_P, 0x08);

    …
  }
}
```

OSCreateBinSem(): Create a Binary Semaphore

Type:	Function
Prototype:	`OStypeErr OSCreateBinSem ( OStypeEcbP ecbP, OStypeBinSem binSem );`
Callable from:	Anywhere
Contained in:	`salvobinsem.c`
Enabled by:	`OSENABLE_BINARY_SEMAPHORES`, `OSEVENTS`
Affected by:	`OSCALL_OSCREATEEVENT`, `OSENABLE_STACK_CHECKING`, `OSCOMBINE_EVENT_SERVICES`, `OSLOGGING`, `OSUSE_EVENT_TYPES`
Description:	Create a binary semaphore with the initial value specified.
Parameters:	`ecbP`: a pointer to the binary semaphore's ecb. `binSem`: the binary semaphore's initial value (0 or 1) .
Returns:	`OSNOERR`
Stack Usage:	1

Notes

Creating a binary semaphore assigns an event control block (ecb) to the semaphore.

A newly-created binary semaphore has no tasks waiting for it.

Signaling or waiting a binary semaphore before it has been created will result in an error if `OSUSE_EVENT_TYPES` is `TRUE`.

You can also implement binary semaphores via messages – see `OSCreateMsg()`.

In the example below, a binary semaphore is used to control access to a shared resource, an I/O port. The port is initially available for use, so the semaphore is initialized to 1.

See Also

`OS_WaitBinSem()`, `OSReadBinSem()`, `OSSignalBinSem()`, `OSTryBinSem()`

Example

```
/* PORTB is a general-purpose I/O port.        */
#define BINSEM_PORTB_P OSECBP(6)
…
/* PORTB is initially available to task that   */
```

```
/*  wants to use it.                                */
OSCreateBinSem(BINSEM_PORTB_P, 1);
…
```

OSCreateCycTmr(): Create a Cyclic Timer

Type:	Function
Prototype:	`OStypeErr OSCreateCycTmr ( OStypeTFP tFP, OStypeTcbP tcbP, OStypeDelay delay, OStypeDelay period, OStypeCTMode mode );`
Callable from:	Background only
Contained in:	`salvoyclic.c`
Enabled by:	`OSENABLE_CYLIC_TIMERS`
Affected by:	–
Description:	Create a cyclic timer with the initial delay and period specified.
Parameters:	`tFP`: a pointer to the cyclic timer's start address. This is also the cyclic timer's function prototype name. `tcbP`: a pointer to the cyclic timer's tcb. `delay`: the initial delay (> 0), in ticks before the cyclic timer is first called. `period`: the time, in ticks (> 0), between successive calls of the cyclic timer `mode`: `OSCT_ONE_SHOT` (the cyclic timer will run only once) or `OSCT_CONTINUOUS` (the cyclic timer will run indefinitely).
Returns:	`OSNOERR` if task is successfully created. `OSERR_BAD_P` if the specified tcb pointer is invalid (i.e. out-of-range). `OSERR_BAD_CT_MODE` if `mode` is unrecognized. `OSERR_BAD_CT_DELAY` if `delay` or `period` are 0.
Stack Usage:	3

Notes

Cyclic timers are structured like common functions (with a clear entry and exit), *not* like tasks. Cyclic timers take no arguments and return no values.

Creating a cyclic timer assigns a task control block (tcb) to the cyclic timer.

If you prefer to create the task now and explicitly start it later, OR `OSCreateCycTmr()`'s `mode` parameter with

OSDONT_START_CYCTMR. Then use OSStartCycTmr() to start the cyclic timer at a later time.

Cyclic timers require that timeouts be enabled. Setting OSENABLE_CYLIC_TIMERS to TRUE will automatically enable timeouts.

In the example below, cyclic timer CycTmr1() toggles bit 1 of an I/O port. CycTmr1() will begin running 23 system ticks after the scheduler is called, and will repeatedly toggle the port pin every 177 system ticks. CycTmr2() will set bit 2 of an I/O port 12 systems ticks after the scheduler is called, and will then stop.

See Also

OSCycTmrRunning(), OSDestroyCycTmr(), OSResetCycTmr(), OSSetCycTmrPeriod(), OSStartCycTmr(), OSStopCycTmr()

Example

```
/* Cyclic timer toggles I/O pin indefinitely.  */
void CycTmr1 ( void )
{
  PORT ^= 0x02;
}

/* Cyclic timer sets I/O pin once.             */
void CycTmr2 ( void )
{
  PORT |= 0x04;
}

…

/* Create the cyclic timers.                   */
OSCreateCycTmr(CycTmr1, OSTCBP(1), 23, 177,
  OSCT_CONTINUOUS);
OSCreateCycTmr(CycTmr2, OSTCBP(5), 12, 7,
  OSCT_ONE_SHOT);
```

OSCreateEFlag(): Create an Event Flag

Type:	Function
Prototype:	`OStypeErr OSCreateEFlag ( OStypeEcbP ecbP, OStypeEfcbP efcbP, OStypeEFlag eFlag );`
Callable from:	Anywhere
Contained in:	`salvoeflag.c`
Enabled by:	`OSENABLE_EVENT_FLAGS`, `OSEVENTS`
Affected by:	`OSCALL_OSCREATEEVENT`, `OSENABLE_STACK_CHECKING`, `OSCOMBINE_EVENT_SERVICES`, `OSLOGGING`, `OSUSE_EVENT_TYPES`
Description:	Create an event flag with the initial value specified.
Parameters:	`ecbP`: a pointer to the event flag's ecb. `efcbP`: a pointer to the event flag's efcb. `eFlag`: the event flag's initial value.
Returns:	`OSNOERR`
Stack Usage:	1

Notes

Creating an event flag assigns an event control block (ecb) and an event flag control block (efcb) to the event flag.

A newly-created event flag has no tasks waiting for it.

Signaling or waiting an event flag before it has been created will result in an error if `OSUSE_EVENT_TYPES` is `TRUE`.

Event flags can be 8, 16 or 32 bits, depending on `OSBYTES_OF_EVENT_FLAGS`. `OSCreateEFlag()` stores the value of the event flag in the event flag's pre-existing event flag control block (efcb) of type `OSgltypeEfcb`. The number of efcb's in your application is set by `OSEVENT_FLAGS`. The first efcb is accessed via `OSEFCBP(1)`, the second by `OSEFCBP(2)`, etc.

In the example below, an 8-bit event flag is used to signify the occurrence of keypresses from an 8-key machine control keypad. Each bit maps to a single key. The event flag is initialized to all `0`'s to indicate that no keypresses have occurred. `OSBYTES_OF_EVENT_FLAGS` is set to 1 in this example's `salvocfg.h`.

See Also

`OS_WaitEFlag()`, `OSReadEFlag()`, `OSSignalEFlag()`, `OSTryEFlag()`

Example

```
/* event flag is event #3, uses event flag     */
/*  control block #1.                           */
#define EFLAG_KEYS_P     OSECBP(3)
#define EFLAG_KEYS_CB_P  OSEFCBP(1)
…
/* Initially no keys have been pressed.        */
OSCreateEFlag(EFLAG_KEYS_P, EFLAG_KEYS_CB_P,
  0x00);
…
```

OSCreateMsg(): Create a Message

Type:	Function
Prototype:	`OStypeErr OSCreateMsg ( OStypeEcbP ecbP, OStypeMsgP msgP );`
Callable from:	Anywhere
Contained in:	`salvosg.c`
Enabled by:	`OSENABLE_MESSAGE`, `OSEVENTS`
Affected by:	`OSCALL_OSCREATEEVENT`, `OSENABLE_STACK_CHECKING`, `OSCOMBINE_EVENT_SERVICES`, `OSLOGGING`, `OSUSE_EVENT_TYPES`
Description:	Create a message with the initial value specified.
Parameters:	`ecbP`: a pointer to the message's ecb. `msgP`: a pointer to a message.
Returns:	`OSNOERR`
Stack Usage:	1

Notes

Creating a message assigns an event control block (ecb) to the message. A newly-created message has no tasks waiting for it. Messages are passed via pointer so that a message can point to anything.

Signaling or waiting a message before it has been created will result in an error if `OSUSE_EVENT_TYPES` is `TRUE`.

Binary semaphores and resource locking can be implemented via messages using the values `(OStypeMsgP) 0` and `(OStypeMsgP) 1` for the messages.

In the example below, a message is created to pass the key pressed (which is detected by the task `TaskReadKey()`) to the task `TaskHandleKey()`, which acts on the keypress. The message is initialized to zero because no keypress is initially detected. If, due to task priorities and timing, `TaskReadKey()` signals a new message before `TaskHandleKey()` reads the existing message, the new key will be lost.

See Also

`OS_WaitMsg()`, `OSReadMsg()`, `OSSignalMsg()`, `OSTryMsg()`

Example

```
/* pass key via a message. */
#define MSG_KEY_PRESSED_P OSECBP(4)
…
/* this task reads key presses from a keypad   */
/*  and sends them to TaskHandleKey via a      */
/*  message.                                   */
void TaskReadKey ( void )
{
  static char key;     /* holds key pressed    */

  /* initially no key has been pressed.        */
  OSCreateMsg(MSG_KEY_PRESSED_P, (OStypeMsgP) 0);

  while (1) {
    if (kbhit()) {
      key = getch();

      /* do debouncing, key-repeat, etc.       */

      /* send new key via message.             */
      OSSignalMsg(MSG_KEY_PRESSED_P,
        (OStypeMsgP) &key);
    }

    /* wait 10msec, then test for keypress     */
    /*  again.                                 */
    OS_Delay(TEN_MSEC);
  }
}

/* this task acts upon keypresses.             */
void TaskHandleKey ( void )
{
  static char key;         /* holds new key    */
  static OStypeMsgP msgP;  /* get msg via ptr  */

  while (1) {
    /* do nothing until a key is pressed.      */
    OS_WaitMsg(MSG_KEY_PRESSED_P, &msgP,
      OSNO_TIMEOUT);

    /* then get the new key and act on it.     */
    key = *(char *)msgP;
    switch (tolower(key)) {
      case KEY_UP:
        MoveUp();
        break;
      …
    }
  }
}
```

OSCreateMsgQ(): Create a Message Queue

Type:	Function
Prototype:	`OStypeErr OSCreateMsgQ ( OStypeEcbP ecbP, OStypeMqcbP mqcbP, OStypeMsgQPP msgPP, OStypeMsgQSize size );`
Callable from:	Anywhere
Contained in:	`salvomsgq.c`
Enabled by:	`OSENABLE_MESSAGE_QUEUES`, `OSEVENTS`
Affected by:	`OSCALL_OSCREATEEVENT`, `OSENABLE_STACK_CHECKING`, `OSCOMBINE_EVENT_SERVICES`, `OSLOGGING`, `OSUSE_EVENT_TYPES`
Description:	Create an empty message queue.
Parameters:	`ecbP`: a pointer to the message queue's ecb. `mqcbP`: a pointer to the message queue's message queue control block. `msgPP`: a pointer to the buffer that will hold the message queue's message pointers. `size`: the number of messages (`0` < `size` < `256`) that the message queue can hold.
Returns:	`OSNOERR`
Stack Usage:	1

Notes

Creating a message queue assigns an event control block (ecb) to the message.

Each message queue has a *message queue control block* (mqcb) associated with it. Salvo message queue services use mqcbs to manage the insertion and removal of messages into and out of each message queue. You must allocate memory for mqcbs using the `OSMESSAGE_QUEUES` configuration option. You must associate a unique mqcb with each message queue using a *message queue control block pointer*. These range from `OSMQCBP(1)` to `OSMQCBP(OSMESSAGE_QUEUES)`. A newly-created message queue contains no messages.

A message queue[75] holds its message pointers[76] within a circular buffer. You must declare this buffer in your source code as a simple array, and give `OSCreateMsgQ()` a handle to it via the `msgPP`

[75] Of type `OSgltypeMsgQP`.

[76] Of type `OStypeMsgP`.

parameter. The buffer must hold `size` message pointers. `OSCreateMsgQ()` does not have any effect on the contents of the buffer.

In the example below, a 7-element and a 16-element message queue are created with the buffers `MsgQBuff1[]` and `MsgQBuff2[]`, respectively. The message queue control block IDs are 1 and 2, since memory was allocated for two message queues via `OSMESSAGE_QUEUES` in `salvocfg.h`.

For this example `salvocfg.h` contains:

```
#define OSEVENTS         5
#define OSMESSAGE_QUEUES 2
```

In this example, all of the `OSLOC_XYZ` configuration options are at their default values. By using `OSLOC_MSGQ` and `OSLOC_MQCB` you can relocate the buffers and the mqcbs, respectively, into RAM banks other than the default banks.

See Also

`OS_WaitMsgQ()`, `OSReadMsgQ()`, `OSSignalMsgQ()`, `OSTryMsgQ()`, `OSLOC_MSGQ`, `OSLOC_MQCB`

Example

```
/* use #defines for legibility                */
#define SEM1_P       OSECBP(1)
#define SEM2_P       OSECBP(2)
#define BINSEM1_P    OSECBP(3)
#define MSGQ1_P      OSECBP(4)
#define MSGQ2_P      OSECBP(5)
#define MQCB1_P      OSMQCBP(1)
#define MQCB2_P      OSMQCBP(2)
#define SIZEOF_MSGQ1 7
#define SIZEOF_MSGQ2 16

/* allocate memory for buffers                */
OSgltypeMsgQP MsgQBuff1[SIZEOF_MSGQ1];
OSgltypeMsgQP MsgQBuff2[SIZEOF_MSGQ2];

/* create message queues from existing        */
/*  buffers and mqcbs.                        */
OSCreateMsgQ(MSGQ1_P, MQCBP1_P, MsgQBuff1,
  SIZEOF_MSGQ1);
OSCreateMsgQ(MSGQ2_P, MQCBP2_P, MsgQBuff2,
  SIZEOF_MSGQ2);
```

OSCreateSem(): Create a Semaphore

Type:	Function
Prototype:	`OStypeErr OSCreateSem ( OStypeEcbP ecbP, OStypeSem sem );`
Callable from:	Anywhere
Contained in:	`salvosem.c`
Enabled by:	`OSENABLE_SEMAPHORES`, `OSEVENTS`
Affected by:	`OSBIG_SEMAPHORES`, `OSCALL_OSCREATEEVENT`, `OSENABLE_STACK_CHECKING`, `OSCOMBINE_EVENT_SERVICES`, `OSLOGGING`, `OSUSE_EVENT_TYPES`
Description:	Create a counting semaphore with the initial value specified.
Parameters:	`ecbP`: a pointer to the semaphore's ecb. `sem`: the semaphore's initial value.
Returns:	`OSNOERR`
Stack Usage:	1

Notes

Creating a semaphore assigns an event control block (ecb) to the semaphore.

A newly-created semaphore has no tasks waiting for it.

Signaling or waiting a semaphore before it has been created will result in an error if `OSUSE_EVENT_TYPES` is `TRUE`.

In the example below, a counting semaphore is created to mark how much space is available in a transmit ring buffer. The buffer is initially empty, so the semaphore is initialized to the size of the buffer.

See Also

`OS_WaitSem()`, `OSReadSem()`, `OSSignalSem()`, `OSTrySem()`

Example

```
/* Ring buffer is used to receive characters.  */
#define SEM_TX_RBUFF_P OSECBP(3)

…

/* initialize semaphore (ring buffer is        */
/*  empty).                                    */
OSCreateSem(SEM_TX_RBUFF_P, 16);

…
```

OSCreateTask(): Create and Start a Task

Type:	Function
Prototype:	`OStypeErr OSCreateTask ( OStypeTFP tFP, OStypeTcbP tcbP, OStypePrio prio );`
Callable from:	Background only
Contained in:	`salvoinit2.c`
Enabled by:	–
Affected by:	`OSLOGGING`, `OSENABLE_STACK_CHECKING`
Description:	Create a task with the specified start address, tcb pointer and priority. Starts the task unless overridden by the user in the `prio` parameter.
Parameters:	`tFP`: a pointer to the task's start address. This is also the task's function prototype name. `tcbP`: a pointer to the task's tcb. `prio`: the desired priority for the task. If OR'd with `OSDONT_START_TASK`, the task will not be started.
Returns:	`OSNOERR` if task is successfully created. `OSERR_BAD_P` if the specified tcb pointer is invalid (i.e. out-of-range).
Stack Usage:	3

Notes

Creating a task assigns a task control block (tcb) to the task.

0 (`OSHIGHEST_PRIO`) is the highest priority, 15 (`OSLOWEST_PRIO`) is the lowest. If the specified task priority is out-of-range, the task will still be created, but with the lowest possible priority.

Tasks created via `OSCreateTask()` are automatically started, i.e. they are in the eligible state.

If you prefer to create the task now and explicitly start it later, OR `OSCreateTask()`'s `prio` parameter with `OSDONT_START_TASK`. Then use `OSStartTask()` to start the task at a later time.

If task priorities are disabled via `OSDISABLE_TASK_PRIORITIES`, `OSCreateTask()`'s third argument (`prio`) is used only with `OSDONT_START_TASK`, and the priority value is disregarded.

Caution `OSCreateTask()` overwrites the task control block specified via the `tcbP` parameter, i.e. it overwrites the tcb. When calling `OSCreateTask()` after task scheduling has started via `OSSched()`, extreme caution must be used to avoid overwriting an existing eligible, running, delayed, waiting or stopped task.

In the example below, a single task is created from the function `TaskDoNothing()` by assigning it a tcb pointer of `TASK1_P`, and a priority of 7.

See Also

`OSStartTask()`, `OSStopTask()`

Example

```
#define TASK1_P OSTCBP(1)/* taskIDs start at 0 */

/* this task does nothing but run, context-    */
/*  switch, run, context-switch, etc.          */
void TaskDoNothing ( void )
{
  while (1) {
    OS_Yield();
  }
}

/* create a single task and run it (over and   */
/*  over).                                     */
int main ( void )
{
  …
  /* initialize Salvo. */
  OSInit();

  /* create a task to do nothing but context-  */
  /*  switch. Tcb pointer is 0, priority is 7  */
  /*  (middle). A call to OSSTartTask() is not */
  /*  required …                               */
  OSCreateTask(TaskDoNothing, TASK1_P, 7);

  …
  /* start multitasking.                       */
  while (1) {
    OSSched();
  }
}
```

OSDestroyCycTmr(): Destroy a Cyclic Timer

Type:	Function
Prototype:	`OStypeErr OSDestroyCycTmr ( OStypeTcbP tcbP );`
Callable from:	Background only
Contained in:	`salvoyclic4.c`
Enabled by:	`OSENABLE_CYCLIC_TIMERS`
Affected by:	–
Description:	Destroy the specified cyclic timer.
Parameters:	`tcbP`: a pointer to the cyclic timer's tcb.
Returns:	`OSNOERR` if cyclic timer is destroyed. `OSERR_BAD_CT` if the tcb in question does not belong to a cyclic timer.
Stack Usage:	3

Notes

`OSDestroyCycTmr()` destroys both running and stopped cyclic timers.

In the example below, `CycTmr3()` is created and then destroyed from within a task after being allowed to run for 200 system ticks. The task then continues, creating another task – `Task4()`– which uses the same tcb.

See Also

`OSCreateCycTmr()`, `OSCycTmrRunning()`, `OSResetCycTmr()`, `OSSetCycTmrPeriod()`, `OSStartCycTmr()`, `OSStopCycTmr()`

Example

```
…
OSCreateCycTmr(CycTmr3, OSTCBP(7), 1, 2,
  OSCT_CONTINUOUS);
OS_Delay(200);
OSDestroyCycTmr(OSTCBP(7));
OSCreateTask(Task4, OSTCBP(7), 12);
```

OSDestroyTask(): Destroy a Task

Type:	Function
Prototype:	`OStypeErr OSDestroyTask ( OStypeTcbP tcbP, OStypeID events );`
Callable from:	Task or Background
Contained in:	`salvotask3.c`
Enabled by:	–
Affected by:	`OSENABLE_STACK_CHECKING`
Description:	Destroy the specified task.
Parameters:	`tcbP`: a pointer to the task's tcb. `events: OSEVENTS`.
Returns:	`OSNOERR` if specified task was successfully destroyed. `OSERR` if unable to destroy the specified task.
Stack Usage:	3

Notes

`OSDestroyTask()` can destroy any task that is not already destroyed or waiting an event.

The destroyed task's tcb is re-initialized.

The second parameter of `OSEVENTS` is required for all configurations where events are enabled. If events are not enabled, then `OSDestroyTask()` takes only a single parameter.

In the example below, `TaskMain()` has a relatively high priority of 3. When it runs, it creates another, lower-priorty task, `TaskWarmUp()`. During the next thirty seconds, `TaskWarmUp()` runs whenever it is the highest-priority eligible task. Then `TaskMain()` destroys `TaskWarmUp()`. Thereafter, `OSCreateTask()` can be used to create another task in `TaskWarmUp()`'s place, using the same tcb pointer.

See Also

`OSCreateTask()`, `OS_Destroy()`

Example

```
OSCreateTask(TaskMain, TASKMAIN, 3);
…
void TaskMain ( void )
{
  OSCreateTask(TaskWarmUp, TASKWARMUP_P, 7);

  while (1) {
    OS_Delay(THIRTY_SEC);
    OSDestroyTask(TASKWARMUP_P, OSEVENTS);
    …
  }
}
```

OSGetPrio(): Return the Current Task's Priority

Type:	Macro (invokes `OSGetPrioTask()`)
Prototype:	`OStypePrio OSGetPrio ( );`
Callable from:	Task only
Contained in:	`salvoprio2.c`
Enabled by:	–
Affected by:	`OSENABLE_STACK_CHECKING`
Description:	Return the priority of the current (running) task.
Parameters:	–
Returns:	–
Stack Usage:	1

Notes

0 (`OSHIGHEST_PRIO`) is the highest priority, 15 (`OSLOWEST_PRIO`) is the lowest.

In the example below, `TaskB()` lowers its priority each time it runs, until it reaches the lowest allowed priority and remains there.

See Also

`OS_SetPrio()`, `OSGetPrioTask()`, `OSSetPrio()`, `OSSetPrioTask()`, `OSDISABLE_TASK_PRIORITIES`

Example

```
void TaskB ( void )
{
  OStypePrio prio;


  while (1) {
    …
    prio-- = OSGetPrio();
    OS_SetPrio(prio);
  }
}
```

OSGetPrioTask(): Return the Specified Task's Priority

Type:	Function
Prototype:	`OStypePrio OSGetPrioTask ( OStypeTcbP tcbP );`
Callable from:	Task or Background
Contained in:	`salvoprio2.c`
Enabled by:	–
Affected by:	`OSENABLE_STACK_CHECKING`
Description:	Return the priority of the specified task.
Parameters:	`tcbP`: a pointer to the task's tcb.
Returns:	–
Stack Usage:	1

Notes

0 (`OSHIGHEST_PRIO`) is the highest priority, 15 (`OSLOWEST_PRIO`) is the lowest.

In the example below, `DispTaskPrio()` displays the priority of the specified task.

See Also

`OS_SetPrio()`, `OSGetPrio()`, `OSSetPrio()`, `OSSetPrioTask()`, `OSDISABLE_TASK_PRIORITIES`

Example

```
#define TASKE_P  OSTCBP(5)
…
void DispTaskPrio ( OStypeTcbP tcbP )
{
  printf("Task %d has priority %d.\n",
   OStID(tcbP, OSTASKS), OSGetPrioTask(tcbP));
}
```

OSGetState(): Return the Current Task's State

Type:	Macro (invokes `OSGetStateTask()`)
Prototype:	`OStypeState OSGetState ( );`
Callable from:	Task only
Contained in:	`salvompt.h`
Enabled by:	–
Affected by:	`OSENABLE_STACK_CHECKING`
Description:	Return the state of the current (running) task.
Parameters:	–
Returns:	Task state.
Stack Usage:	1

Notes

The current task's state is always `OSTCB_TASK_RUNNING`. This service is included for completeness.

In the example below, `TaskG()` verifies that it is in fact running.

See Also

`OSGetStateTask()`

Example

```
void TaskC ( void )
{
  while (1) {
    if (OSGetState() != OSTCB_TASK_RUNNING)
      printf("Houston, we have a problem.\n");
  }
}
```

OSGetStateTask(): Return the Specified Task's State

Type:	Function
Prototype:	`OStypeState OSGetState ( OStypeTcbP tcbP );`
Callable from:	Task or Background
Contained in:	`salvotask5.c`
Enabled by:	–
Affected by:	`OSENABLE_STACK_CHECKING`
Description:	Return the state of the specified task.
Parameters:	–
Returns:	Task state.
Stack Usage:	1

Notes

A task may be in one of the following states:

`OSTCB_DESTROYED`	destroyed / uninitialized
`OSTCB_TASK_STOPPED`	stopped
`OSTCB_TASK_DELAYED`	delayed
`OSTCB_TASK_WAITING`	waiting on an event
`OSTCB_TASK_WAITING_TO`	waiting on an event, with a timeout if in an event queue. Waited for an event and timed out if in the eligible queue
`OSTCB_TASK_ELIGIBLE`	eligible to run
`OSTCB_TASK_SIGNALED`	in the eligible queue, having waited an event that was signaled
`OSTCB_TASK_RUNNING`	running

In the example below, mainline code verifies that a particular task has indeed been stopped.

See Also

`OSGetState()`

Example

```
#define TASKC_P OSTCBP(3)
…
if (OSGetStateTask(TASKC_P) != OSTCB_TASK_STOPPED)
  /* something's wrong with TaskC().           */
…
```

OSGetTicks(): Return the System Timer

Type:	Function
Prototype:	`OStypeTick OSGetTicks ( void );`
Callable from:	Anywhere
Contained in:	`salvotick.c`
Enabled by:	`OSBYTES_OF_TICKS`
Affected by:	`OSENABLE_STACK_CHECKING`
Description:	Obtain the current value of the system timer (in ticks).
Parameters:	–
Returns:	Current system timer in ticks.
Stack Usage:	1

Notes

The system timer is initialized to 0 via `OSInit()`.

In the example below, the current value of the system timer is stored in a variable.

See Also

`OSSetTicks()`

Example

```
…

OStypeTick ticksNow;

…

/* obtain current value of system ticks.       */
ticksNow = OSGetTicks();
…
```

On certain targets it may be advantageous to read the current system ticks (`OStimerTicks`) directly instead of through `OSGetTicks()`. Possible scenarios include substantial function call overhead and/or no need to manage interrupts.[77] In the example below, the current value of the system timer is stored in a variable by accessing `OStimerTicks` directly.

```
…

OStypeTick ticksNow;

…

/* obtain current value of system ticks.       */
disable_interrupts();
ticksNow = OStimerTicks;
enable_interrupts();
…
```

[77] Both of these conditions occur on the baseline PICmicro devices, e.g. PIC12C509.

OSGetTS(): Return the Current Task's Timestamp

Type:	Macro (invokes `OSGetTSTask()`)
Prototype:	`OStypeTS OSGetTS (void);`
Callable from:	Task only
Contained in:	`salvodelay3.c`
Enabled by:	`OSBYTES_OF_TICKS`
Affected by:	`OSENABLE_STACK_CHECKING`
Description:	Obtain the value of the current task's timestamp (in ticks).
Parameters:	–
Returns:	Current task's timestamp in ticks.
Stack Usage:	1

Notes

When a task is created, its timestamp is initialized to an `OStypeTS`-sized version of the system timer ticks, i.e. `(OStypeTS) OStimerTicks`.

In the example below, the current task's timestamp is displayed whenever it times out.

See `OS_DelayTS()` for more information on timestamps.

See Also

`OS_DelayTS()`, `OSSetTS()`, `OSSyncTS()`

Example

```
void Task ( void )
{
  while (1) {
    OS_Delay(7);[78]

    printf("Task %d timed out at %d\n",
      OStID(OScTcbP, OSTASKS), OSGetTS());

    …
  }
}
```

[78] The timestamp is redefined whenever a delay expires, whether through `OS_Delay()` or `OS_DelayTS()`.

OSInit(): Prepare for Multitasking

Type:	Function
Prototype:	`void OSInit ( void );`
Callable from:	Background only
Contained in:	`salvoinit.c`
Enabled by:	–
Affected by:	`OSBYTES_OF_DELAYS`, `OSCLEAR_GLOBALS`, `OSENABLE_STACK_CHECKING`, `OSEVENTS`, `OSLOGGING`, `OSTASKS`
Description:	Initialize Salvo's pointers, counters, etc.
Parameters:	–
Returns:	–
Stack Usage:	2

Notes

`OSInit()` must be called first, before any other Salvo functions.

The executable code size of `OSInit()` can be minimized by setting `OSCLEAR_GLOBALS` to `FALSE`. Do this only if you are certain that your compiler initializes all global variables to 0 at runtime, and you do not call `OSInit()` more than once in your application.

`OSInit()` does not initialize tcbs or ecbs – this is done on a per-tcb and per-ecb basis when tasks and events are created, respectively.

In the example below, `OSInit()` is called before any other Salvo calls.

Example

```
int main ( void )
{
  …
  /* initialize Salvo.                          */
  OSInit();
  …
  /* start multitasking.                        */
  while (1) {
    OSSched();
  }
}
```

OSMsgQCount(): Return Number of Messages in Message Queue

Type:	Function
Prototype:	`OStypeMsgQSize OSMsgQCount ( OStypeTcbP ecbP );`
Callable from:	Anywhere
Contained in:	`salvomsgq4.c`
Enabled by:	`OSENABLE_MESSAGE_QUEUES`
Affected by:	`OSCALL_OSMSGQCOUNT`
Description:	Check whether the specified message queue has room for additional message(s).
Parameters:	`ecbP`: a pointer to the message queue's ecb.
Returns:	Number of messages in message queue, i.e. returns 0 if message queue is empty.
Stack Usage:	1

Notes

`OSMsgQCount()` can be used to obtain the current status of the message queue. `OSMsgQCount()` returns the `count` record in the message queue's message queue control block (mqcb) – therefore it's very fast.

No error checking is performed on the `ecbP` parameter. Calling `OSMsgQCount()` with an invalid `ecbP`, or an `ecbP` belonging to an event other than a message queue, will return an erroneous result.

In the example below, `OSMsgQCount()` is used to obtain the number of messages in a message queue, and the space available for new messages. When using `OSMsgQCount()` to calculate available space in a message queue, it must be subtracted from the size parameter originally used to create the message queue.

See Also

`OS_WaitMsgQ()`, `OSCreateMsgQ()`, `OSMsgQEmpty()`, `OSReadMsgQ()`, `OSSignalMsgQ()`, `OSTryMsgQ()`

Example

```
#define MSGQ1_P OSECBP(1)

printf("msgQ contains %d messages\n",
  OSMsgQCount(MSGQ1_P));
printf("msgQ has room for %d messages\n",
  SIZEOF_MSGQ1 - OSMsgQCount(MSGQ1_P));
```

OSMsgQEmpty(): Check for Available Space in Message Queue

Type:	Function
Prototype:	`OStypeMsgQSize OSMsgQEmpty ( OStypeTcbP ecbP );`
Callable from:	Anywhere
Contained in:	`salvomsgq3.c`
Enabled by:	`OSENABLE_MESSAGE_QUEUES`
Affected by:	`OSCALL_OSMSGQEMPTY`
Description:	Check whether the specified message queue has room for additional message(s).
Parameters:	`ecbP`: a pointer to the message queue's ecb.
Returns:	Number of available (empty) spots in message queue, i.e. returns 0 (FALSE) if message queue is full.
Stack Usage:	1

Notes

Each message queue can contain up to a maximum number of messages. If messages are added to the message queue (via `OSSignalMsgQ()`) faster than they are removed (via `OS_WaitMsgQ()`), the queue will eventually fill up. `OSMsgQEmpty()` can be used to obtain the current status of the message queue without signaling the message queue.

No error checking is performed on the `ecbP` parameter. Calling `OSMsgQEmpty()` with an invalid `ecbP`, or an `ecbP` belonging to an event other than a message queue, will return an erroneous result.

Note `OSMsgQEmpty()` performs pointer subtraction when computing the available room in the specified message queue. On some[79] targets, this may result in very slow execution. Since interrupts are disabled during `OSMsgQEmpty()`, this is not desirable. `OSMsgQCount()` always executes very quickly, and is preferred in these cases.

In the first example below, mainline code signals a message queue with a message from the user's `msg` array only if space is available. If not, an error counter is incremented. This example will give erroneous results if messages are also signaled to the same message queue from within an interrupt handler. That's because interrupts

[79] For example, on an 8-bit target where data pointers are 16 bits.

are enabled between the call to `OSMsgQEmpty()` and the call to `OSSignalMsgQ()`. In that case, `OSSignalMsgQ()`'s return code of `OSERR_EVENT_FULL` can be used to detect the inability to enqueue a message into a message queue.

In the second example below, the message queue is filled to capacity with new message pointers of ascending value, starting at 0.

See Also

`OS_WaitMsgQ()`, `OSCreateMsgQ()`, `OSMsgQCount()`, `OSReadMsgQ()`, `OSSignalMsgQ()`, `OSTryMsgQ()`

Example #1

```
#define MSGQ3_P OSECBP(4)

unsigned int counter;

if (OSMsgQEmpty(MSGQ3_P)) {
  OSSignalMsgQ(MSGQ3_P, (OStypeMsgP) &msg[i]);
}
else {
  counter++;
}
```

Example #2

```
OStypeMsgQSize roomLeft;

roomLeft = OSMsgQEmpty(MSGQ1_P);
for (i = 0; i < roomLeft; i++) {
      OSSignalMsgQ(MSGQ1_P, (OStypeMsgP) i);
}
```

OSReadBinSem(): Obtain a Binary Semaphore Unconditionally

Type:	Function
Prototype:	`OStypeBinSem OSReadBinSem ( OStypeEcbP ecbP );`
Callable from:	Anywhere
Contained in:	`salvobinsem.c`
Enabled by:	`OSENABLE_BINARY_SEMAPHORES`, `OSENABLE_EVENT_READING`, `OSEVENTS`
Affected by:	`OSCALL_OSRETURNEVENT`
Description:	Returns the binary semaphore specified by `ecbP`.
Parameters:	`ecbP`: a pointer to the binary semaphore's ecb.
Returns:	Binary semaphore (0 or 1).
Stack Usage:	1

Notes

`OSReadBinSem()` has no effect on the specified binary semaphore. Therefore it can be used to obtain the binary semaphore's value without affecting the state(s) of any task(s).

No error checking is performed on the `ecbP` parameter. Calling `OSReadBinSem()` with an invalid `ecbP`, or an `ecbP` belonging to an event other than a binary semaphore, will return an erroneous result.

In the example below, a binary semaphore employed as a resource is tested before making a decision to delay a task.

See Also

`OS_WaitBinSem()`, `OSCreateBinSem()`, `OSTryBinSem()`, `OSSignalBinSem()`

Example

```
…
/* initially, resource #2 is available.        */
OSCreateBinSem(BINSEM_RSRC2_P, 1);

void TaskD ( void )
{
  while (1) {
    …
    if (OSReadBinSem(BINSEM_RSRC2_P)) {
      MyFn();
    }
    else {
      OS_Delay(100);
    }
  }
}
```

OSReadEFlag(): Obtain an Event Flag Unconditionally

Type:	Function
Prototype:	`OStypeEFlag OSReadEFlag ( OStypeEcbP ecbP );`
Callable from:	Anywhere
Contained in:	`salvoeflag2.c`
Enabled by:	`OSENABLE_EVENT_FLAGS`, `OSENABLE_EVENT_READING`, `OSEVENTS`
Affected by:	`OSCALL_OSRETURNEVENT`
Description:	Returns the event flag specified by `ecbP`.
Parameters:	`ecbP`: a pointer to the event flag's ecb.
Returns:	Event flag.
Stack Usage:	1

Notes

`OSReadEFlag()` has no effect on the specified event flag. Therefore it can be used to obtain the event flag's value without affecting the state(s) of any task(s).

No error checking is performed on the `ecbP` parameter. Calling `OSReadEFlag()` with an invalid `ecbP`, or an `ecbP` belonging to an event other than an event flag, will return an erroneous result.

In the example below, `TaskF()` waits on one of two bits to be set in an event flag pointed to by `EFLAG_P`. `OSReadEFlag()` is then used to determine which of the two bits was set.

See Also

`OS_WaitEFlag()`,`OSClrEFlag()`, `OSCreateEFlag()`, `OSSetEFlag()`

Example

```
void TaskF ( void )
{
  OStypeEFlag eFlag;


  while (1) {
    OS_WaitEFlag(EFLAG_P, 0xC0, OSANY_BITS,
      OSNO_TIMEOUT);

    eFlag = OSReadEFlag(EFLAG_P);

    if (eFlag & 0x80) {
      /* topmost bit was set …                */
      …
    }
    else {
      /* other bit was set …                  */
      …
    }
  }
}
```

OSReadMsg():Obtain a Message's Message Pointer Unconditionally

Type:	Function
Prototype:	`OStypeMsgP OSReadMsg ( OStypeEcbP ecbP );`
Callable from:	Anywhere
Contained in:	`salvomsg.c`
Enabled by:	`OSENABLE_MESSAGES`, `OSENABLE_EVENT_READING`, `OSEVENTS`
Affected by:	`OSCALL_OSRETURNEVENT`
Description:	Returns a pointer to the message specified in `ecbP`.
Parameters:	`ecbP`: a pointer to the message's ecb.
Returns:	Message pointer.
Stack Usage:	1

Notes

`OSReadMsg()` has no effect on the specified message. Therefore it can be used to obtain the message's message pointer without affecting the state(s) of any task(s).

No error checking is performed on the `ecbP` parameter. Calling `OSReadMsg()` with an invalid `ecbP`, or an `ecbP` belonging to an event other than a message, will return an erroneous result.

In the example below, a task checks to see if a message is non-empty before signaling the message.[80] Thus it avoids losing the message.

See Also

`OS_WaitMsg()`, `OSCreateMsg()`, `OSSignalMsg()`, `OSTryMsg()`

[80] If the application allowed signaling the message from an interrupt, additional interrupt control would be required in `TaskC()` in order to guarantee that the message is empty before signaling it.

Example

```
/* send this when there's a problem.          */
const char strImpMsg[] = "Important Message!\n";

void TaskC ( void )
{
  while (1) {
    …
    /* delay one system tick as long as MSG    */
    /*  has a message in it.                   */
    while (OSReadMsg(MSG_P)) {
      OS_Delay(1);
    }

    /* now that MSG is empty, we can send our  */
    /*  important message.                     */
    OSSignalMsg (MSG_P, (OStypeMsgP) &strImpMsg);
  }
}
```

OSReadMsgQ(): Obtain a Message Queue's Message Pointer Unconditionally

Type:	Function
Prototype:	`OStypeMsgP OSReadMsgQ ( OStypeEcbP ecbP );`
Callable from:	Anywhere
Contained in:	`salvomsgq.c`
Enabled by:	`OSENABLE_EVENT_READING`, `OSENABLE_MESSAGE_QUEUES`, `OSEVENTS`
Affected by:	`OSCALL_OSRETURNEVENT`
Description:	Returns a pointer to the next message in the message queue specified in `ecbP`.
Parameters:	`ecbP`: a pointer to the message's ecb.
Returns:	Message pointer.
Stack Usage:	1

Notes

`OSReadMsgQ()` has no effect on the specified message queue. Therefore it can be used to obtain the message queue's message pointer without affecting the state(s) of any task(s).

No error checking is performed on the `ecbP` parameter. Calling `OSReadMsgQ()` with an invalid `ecbP`, or an `ecbP` belonging to an event other than a message queue, will return an erroneous result.

In the example below, message queue #2 is slowly filled with a new character message every few seconds. `TaskB()` monitors the message queue every second. Whenever there are one or more valid messages in the message queue, `TaskB()` displays the first message's contents.[81] As the waiting task (not shown) waits the message queue and obtains the messages, `TaskB()`'s output will change as well.

See Also

`OS_WaitMsgQ()`, `OSCreateMsgQ()`, `OSSignalMsgQ()`, `OSTryMsgQ()`

Example

```
/* message queue #2 contains single chars.     */
#define MSGQ2_P OSECBP(6)
```

[81] Note that `TaskB()`, as written, cannot distinguish between successive, identical messages. Therefore it will report on a stream of messages 'h','e','l','l','o' as 'h','e','l','o'. However, the waiting task will receive all five characters in the string.

```
void TaskB ( void )
{
  static char oldchar;
  char newchar;
  OStypeMsgP msgP;

  while (1) {
    OS_Delay(ONE_SEC);
    …

    /* test message queue #2                    */
    msgP = OSReadMsgQ(MSGQ2_P);

    /* get the message if there is one.         */
    if (msgP) {
      newchar = *(char *) msgP;
      if ( newchar != oldchar ) {
        oldchar = newchar;
        printf("The new message is: %c\n.",
          newchar);
      }
    }
    …
  }
}
```

OSReadSem(): Obtain a Semaphore Unconditionally

Type:	Function
Prototype:	`OStypeSem OSReadSem ( OStypeEcbP ecbP );`
Callable from:	Anywhere
Contained in:	`salvosem.c`
Enabled by:	`OSENABLE_EVENT_READING`, `OSENABLE_SEMAPHORES`, `OSEVENTS`
Affected by:	`OSCALL_OSRETURNEVENT`
Description:	Returns the current value of the semaphore specified in `ecbP`.
Parameters:	`ecbP`: a pointer to the semaphore's ecb.
Returns:	Semaphore.
Stack Usage:	1

Notes

`OSReadSem()` has no effect on the specified semaphore. Therefore it can be used to obtain the semaphore's value without affecting the state(s) of any task(s).

No error checking is performed on the `ecbP` parameter. Calling `OSReadSem()` with an invalid `ecbP`, or an `ecbP` belonging to an event other than a semaphore, will return an erroneous result.

In the example below, a binary semaphore is used to manage a 15-character ring buffer. In case of an error, the program displays a descriptive message[82] before re-initializing the buffer.

See Also

`OS_WaitSem()`, `OSCreateSem()`, `OSSignalSem()`, `OSTrySem()`

[82] `printf()` does not use the system's `Tx` facilities.

Example

```
/* initially, Tx buffer has room for 15 chars. */
#define SIZEOF_TXBUFF 15
…
/* manage the Tx buffer as a resource.         */
OSCreateSem(SEM_TXBUFF_P, SIZEOF_TXBUFF);


…
/* if there's a Tx error, flush and recreate   */
/*  the buffer after displaying a message.     */
if (TxErr)
{
  DisableTxInts();
  printf("Error: %d chars stuck in Tx buffer.\n",
    SIZEOF_TXBUFF - OSReadSem(SEM_TXBUFF_P));
  FlushTxBuff();
  OSCreateSem(SEM_TXBUFF_P, SIZEOF_TXBUFF);
  EnableTxInts();
}
…
```

OSResetCycTmr(): Reset a Cyclic Timer

Type:	Function
Prototype:	`OStypeErr OSResetCycTmr ( OStypeTcbP tcbP );`
Callable from:	Background only
Contained in:	`salvocyclic6.c`
Enabled by:	`OSENABLE_CYCLIC_TIMERS`
Affected by:	–
Description:	(Re-)set the specified cyclic timer.
Parameters:	`tcbP`: a pointer to the cyclic timer's tcb.
Returns:	`OSNOERR` if cyclic timer is successfully reset. `OSERR_BAD_CT` if the tcb in question does not belong to a cyclic timer.
Stack Usage:	3

Notes

`OSResetCycTmr()` restarts the cyclic timer with its period regardless of whether the cyclic timer is running or not.

A cyclic timer can be re-synchronized with `OSResetCycTmr()`.

In the example below, a task waits for a signal to restart a cyclic timer. When that signal is received, the cyclic timer is stopped and restarted. Regardless of how close it was previously to timing out, it will now time out in its normal period.

See Also

`OSCreateCycTmr()`, `OSCycTmrPeriod()`, `OSCycTmrRunning()`, `OSDestroyCycTmr()`, `OSStartCycTmr()`, `OSStopCycTmr()`

Example

```
…
OS_WaitBinSem(BINSEM_RESTART_CYCTMR3,
  OSNO_TIMEOUT);
OSResetCycTmr(OSTCBP(6));
```

OSRpt(): Display the Status of all Tasks, Events, Queues and Counters

Type:	Function
Prototype:	`void OSRpt ( OStypeID tasks, OStypeID events );`
Callable from:	Task or Background
Contained in:	`salvorpt.c`
Enabled by:	–
Affected by:	`OSBYTES_OF_COUNTS`, `OSBYTES_OF_DELAYS`, `OSENABLE_STACK_CHECKING`, `OSENABLE_STATISTICS`, `OSMON_HIDE_INVALID_PTRS`, `OSMON_SHOW_ONLY_ACTIVE`, `OSMON_SHOW_TOTAL_DELAY`, `OSUSE_EVENT_TYPES`
Description:	Display the current status of all Salvo tasks, events and counters in tabular form.
Parameters:	`tasks`: `OSTASKS`. `events`: `OSEVENTS`.
Returns:	–
Stack Usage:	3 + `printf()`'s stack usage

Notes

`OSRpt()` requires a working `printf()` function in the target application.[83] `OSRpt()` is quite large and is intended for use only in those systems that have sufficient code space (e.g. x86-based systems) to include it in the target application.

`OSRpt()` displays the current task, the members of the eligible and delayed queues (shown in their priority order), and the fields of each task control block (tcb) and event control block (ecb). If so configured, it also displays error, warning and timeout counter values, the maximum call ... return depth, and the total delay of the tasks in the delay queue.

`OSRpt()` reads and displays Salvo's data structures on-the-fly, i.e. no local copy is made. Depending on the speed at which the `printf()` function is able to output characters, `OSRpt()` may take quite a while to complete. This may result in a display of informa-

[83] Some libraries (e.g. Hi-Tech PICC) contain a dummy `putch()` function called by `printf()`. You must supply your own, working `putch()` for `printf()` output to occur.

tion that appears to be contradictory (e.g. a task is shown in the eligible queue and simultaneously waiting for an event). In order to avoid this, your application must control or disable interrupts while `OSRpt()` is executing.

See Also

Chapter 5 • Configuration

Example

```
…
/* display the current status of all tasks     */
/*  and events (and counters, if so enabled)   */
/*  to the system's terminal screen.           */
OSRpt(OSTASKS, OSEVENTS);
…
```

A call to `OSRpt()` resulted in the following display on a simple terminal program connected via RS-232 to a Salvo system[84] with a working `printf()`:

```
Salvo v2.2.beta7  Max call...rtn stack depth: 3
CtxSws, total=idle+eligible: 1000358326 = 922445444 + 77912882
Errors: 0  Warnings: 0  Timeouts: 255  Ticks: 33163186
EligQ: t6,t3,t8
DelayQ: t7,t1,t2,t5,t4  Total delay: 60 ticks
task stat prio    addr   t->  e->  d-> delay
  1  wait   2      6F8h    .  e 1  t 2   22
  2  wait   3      6F8h    .  e 2  t 5   10
  3  elig   4      6FBh  t 8  n/a
  4  wait   5      6F8h    .  e 4    .   13
  5  wait   5      6F8h  t 4  e 4  t 4   13
  6  elig   1      6F5h  t 3  n/a
  7  dlyd   0      70Ah    .       t 1    2
  8  elig  15      70Dh    .  n/a

evnt type t->    value
  1   Sem t 1        0
  2   Sem t 2        0
  3   Sem   .        0
  4   Sem t 5        0
  5   Sem   .      255
```

Figure 31: OSRpt() Output to Terminal Screen

In Figure 31 we can see that when `OSRpt()` was called, three tasks were eligible, five were waiting and/or delayed, and over one billion context switches had occurred over a nearly four-day-long period.[85]

---

84 This output is from the program in `\salvo\demo\d1\sysa`, running on a PIC16C77 with a 4MHz crystal.

85 System tick rate of 100Hz.

OSSched(): Run the Highest-Priority Eligible Task

Type:	Function
Prototype:	`void OSSched ( void );`
Callable from:	`main()`
Contained in:	`salvosched.c`
Enabled by:	–
Affected by:	`OSCLEAR_UNUSED_POINTERS`, `OSCLEAR_WATCHDOG_TIMER`, `OSENABLE_STACK_CHECKING`, `OSENABLE_STATISTICS`, `OSLOGGING`, `OSOPTIMIZE_FOR_SPEED`,
Description:	Dispatch Salvo's tasks via a cooperative multitasking priority-based scheme.
Parameters:	–
Returns:	–
Stack Usage:	2 if `OSUSE_INLINE_OSSCHED` is `FALSE`. Tasks will run 2 levels below scheduler. 1 if `OSUSE_INLINE_OSSCHED` is `TRUE`. Tasks will run 1 level below scheduler.

Notes

`OSSched()` causes the highest-priority task currently in the eligible queue to execute.

Your application must call `OSInit()` before calling `OSSched()`.

Your application must repeatedly call `OSSched()` in order for multitasking to continue.

In the example below, `OSSched()` is called from within an infinite loop.

See Also

`OSCreateTask()`, `OSInit()`, `OSStartTask()`

Example

```
int main ( void )
{
  /* OS must be initialized.                    */
  OSInit();
  …
  /* create and start several tasks ...         */
  OSCreateTask(Task0, OSTCBP(1), TASK0_PRIORITY);
  OSCreateTask(Task1, OSTCBP(2), TASK1_PRIORITY);
  …
  /* tasks are ready to run – begin multi-      */
  /*  tasking.                                  */
  while (1) {
    /* OSSched() is usually the only function  */
    /*  called inside this never-ending loop.  */
    OSSched();
  }
}
```

OSSetCycTmrPeriod(): Set a Cyclic Timer's Period

Type:	Function
Prototype:	`OStypeErr OSSetCycTmrPeriod ( OStypeTcbP tcbP, OstypeDelay period );`
Callable from:	Background only
Contained in:	`salvocyclic5.c`
Enabled by:	`OSENABLE_CYCLIC_TIMERS`
Affected by:	–
Description:	(Re-)set the specified cyclic timer's period.
Parameters:	`tcbP`: a pointer to the cyclic timer's tcb. `period`: the new period.
Returns:	`OSNOERR` if cyclic timer's period is successfully redefined. `OSERR_BAD_CT` if the tcb in question does not belong to a cyclic timer.
Stack Usage:	3

Notes

`OSSetCycTmrPeriod()` (re-)sets the cyclic timer's period regardless of whether the cyclic timer is running or not.

A cyclic timer's period can be changed on-the-fly with `OSSetCycTmrPeriod()`.

In the example below, the cyclic timer's period is changed from its previous value to 200 system ticks. If it is already running, it will begin running once every 200 system ticks as soon as its current period timer times out.

See Also

`OSCreateCycTmr()`, `OSCycTmrRunning()`, `OSDestroyCycTmr()`, `OSResetCycTmr()`, `OSStartCycTmr()`, `OSStopCycTmr()`

Example

```
…
OSSetCycTmrPeriod(OSTCBP(11), 200);
```

OSSetEFlag(): Set Event Flag Bit(s)

Type:	Macro or Function
Prototype:	`OStypeErr OSSetEFlag ( OStypeEcbP ecbP, OStypeEFlag mask );`
Callable from:	Anywhere
Contained in:	`salvoeflag.c`, `salvoevent.c`
Enabled by:	`OSENABLE_EVENT_FLAGS`, `OSEVENTS`
Affected by:	`OSLOGGING`, `OSENABLE_STACK_CHECKING`, `OSCOMBINE_EVENT_SERVICES`, `OSUSE_EVENT_TYPES`
Description:	Set bits in an event flag. If any bits change, every task waiting it is made eligible.
Parameters:	`ecbP`: a pointer to the event flag's ecb. `mask`: mask of bits to be set.
Returns:	`OSERR_BAD_P` if event flag pointer is incorrectly specified. `OSERR_EVENT_BAD_TYPE` if specified event is not an event flag. `OSERR_EVENT_CB_UNINIT` if event flag's control block is uninitialized. `OSERR_EVENT_FULL` if event flag doesn't change. `OSNOERR` if event flag bits are successfully set.
Stack Usage:	1

Notes

All tasks[86] waiting an event flag are made eligible by forcing any zeroed bits to one in the event flag via `OSSetEFlag()`. Upon running, each such task will either continue running or will return to the waiting state, depending on the outcome of its call to `OS_WaitEFlag()`. Thus, multiple tasks waiting a single event flag can be made eligible simultaneously.

In the example below, two tasks are each waiting different bits of an event flag. When those bits are set via `OSSetEFlag()`, both tasks are made eligible. Each task will run when it becomes the highest-priority eligible task.

[86] Not just the highest-priority waiting task.

See Also

OS_WaitEFlag(), OSClrEFlag(), OSCreateEFlag(), OSReadEFlag()

Example

```
#define EFLAG2_P OSECBP(4)
…
/* force TaskA() and TaskB() to wake up.       */
OSSetEFlag(EFLAG2_P, 0x03);
…
void TaskA ( void )
{
  while (1) {
    /* wait forever for bit 0 to be set        */
    OS_WaitEFlag(EFLAG2_P, 0x01, OSALL_BITS,
      OSNO_TIMEOUT);

    /* clear it and continue                   */
    OSClrEFlag(EFLAG2_P, 0x01);

    …
  }
}

void TaskB ( void )
{
  while (1) {
    OS_WaitEFlag(EFLAG2_P, 0x02, OSALL_BITS,
      OSNO_TIMEOUT);
    OSClrEFlag(EFLAG2_P, 0x02);
    …
  }
}
```

OSSetPrio(): Change the Current Task's Priority

Type:	Function
Prototype:	`void OSSetPrio ( OStypePrio prio );`
Callable from:	Task only
Contained in:	`salvoprio.c`
Enabled by:	–
Affected by:	`OSENABLE_STACK_CHECKING`
Description:	Change the priority of the current (running) task.
Parameters:	`priority`: the desired (new) priority for the current task.
Returns:	–
Stack Usage:	1

Notes

0 (`OSHIGHEST_PRIO`) is the highest priority, 15 (`OSLOWEST_PRIO`) is the lowest.

Tasks can share priorities. Eligible tasks with the same priority will round-robin schedule as long as they are the highest-priority eligible tasks.

The new priority will take effect immediately after the next context switch.

In the example below, `TaskStatusLED()` is dedicated to flashing an LED at one of two rates – 1Hz for a simple heartbeat indication, and 25Hz for an alert indication. The system timer ticks every 10ms. When an alert is not present, it's sensible to run `TaskStatusLED()` at a low priority, so that other more important tasks can run. However, when an alert condition occurs, it's imperative that the user see the LED flash at 25Hz, so `TaskStatusLED()` elevates itself to a higher priority to ensure that it runs often enough to flash the LED at 25Hz. This example assumes that all other tasks are either delayed or waiting at any particular time. Note that in this example `TaskStatusLED()` will fail to flash the LED at 25Hz if it is blocked (i.e. if there are always higher-priority tasks running) at priority 14 when alert is `TRUE`.

See Also

`OS_SetPrio()`, `OSGetPrio()`, `OSGetPrioTask()`, `OSSetPrioTask()`, `OSDISABLE_TASK_PRIORITIES`

Example

```
char alert = FALSE;  /* global, set & reset    */
                     /* elsewhere in code      */

void TaskStatusLED ( void )
{
  while (1) {
    /* toggle alert LED                        */
    PORT_LED ^= 0x01;

    /* if there's an alert, elevate the task's */
    /*  priority (to ensure that we see the LED*/
    /*  flash) and change the flash rate to    */
    /*  25Hz to be sure to catch the user's    */
    /*  attention.                             */
    if ( alert )
    {
      OSSetPrio(5);
      OS_Delay(2);
    }

    /* otherwise  lower the task's priority to */
    /*  rock-bottom and toggle the LED at 1Hz. */
    else
    {
      OSSetPrio(OSLOWEST_PRIO);
      OS_Delay(50);
    }
  }
}
```

OSSetPrioTask(): Change a Task's Priority

Type:	Function
Prototype:	`OStypeErr OSSetPrioTask ( OStypeTcbP tcbP, OStypePrio prio );`
Callable from:	Task or Background
Contained in:	`salvotask6.c`
Enabled by:	–
Affected by:	`OSENABLE_STACK_CHECKING`
Description:	Change the priority of the specified task.
Parameters:	`tcbP`: a pointer to the task's tcb. `prio`: the desired (new) priority for the specified task.
Returns:	`OSNOERR` if specified task's priority was changed successfully `OSERR` if `OSSetPrioTask()` was unable to change the specified task's priority.
Stack Usage:	3

Notes

`OSSetPrioTask()`can change the priority of any task that is not already destroyed or waiting an event.

0 (`OSHIGHEST_PRIO`) is the highest priority, 15 (`OSLOWEST_PRIO`) is the lowest.

Tasks can share priorities. Eligible tasks with the same priority will round-robin schedule as long as they are the highest-priority eligible tasks.

The new priority will take effect immediately.

In the example below, every ten minutes `TaskE()` elevates the priority of `TaskC()`for one minute, then reduces `TaskC()`'s priority back to its original priority.

See Also

`OSGetPrioTask()`, `OSDISABLE_TASK_PRIORITIES`

Example

```
/* initially, run TaskD() at priority 7.       */
OSCreateTask(TaskD, TASKD_P, 7);
OSCreateTask(TaskE, TASKE_P, 3);

void TaskE ( void )
{
  while (1) {
    /* delay ten minutes.                       */
    OS_Delay(TEN_MINUTES);

    /* elevate TaskD()'s priority.              */
    OSSetPrioTask(TASKD_P, 5);

    /* delay another minute.                    */
    OS_Delay(ONE_MINUTE);

    /* restore TaskD()'s priority.              */
    OSSetPrioTask(TASKD_P, 7);
  }
}
```

OSSetTicks(): Initialize the System Timer

Type:	Function
Prototype:	`void OSSetTicks ( OStypeTick tick );`
Callable from:	Anywhere
Contained in:	`salvoticks.c`
Enabled by:	`OSBYTES_OF_TICKS`
Affected by:	`OSENABLE_STACK_CHECKING`
Description:	(Re-)define the current value of the system timer (in ticks).
Parameters:	`tick`: an integer (>=0) value for the system timer.
Returns:	–
Stack Usage:	1

Notes

The system timer is initialized to 0 via `OSInit()`.

In the example below, the current value of the system timer is reset to zero during runtime.

See Also

`OSGetTicks()`

Example

```
…

/* reset system ticks to 0.                          */
OSSetTicks(0);

…
```

On certain targets it may be advantageous to write the current system ticks (`OStimerTicks`) directly instead of through `OSSetTicks()`. Possible scenarios include substantial function call overhead and/or no need to manage interrupts. In the example below, the current value of the system timer is reset to zero during runtime.

```
…

/* reset system ticks to 0.                          */
disable_interrupts();
OStimerTicks = 0;
enable_interrupts();

…
```

OSSetTS(): Initialize the Current Task's Timestamp

Type:	Macro (invokes `OSSetTSTask()`)
Prototype:	`void OSSetTS ( OStypeTS timestamp );`
Callable from:	Task only
Contained in:	`salvodelay3.c`
Enabled by:	`OSBYTES_OF_TICKS`
Affected by:	`OSENABLE_STACK_CHECKING`
Description:	(Re-)define the current task's timestamp (in ticks).
Parameters:	`timestamp`: an integer (>=0) value for the timestamp.
Returns:	–
Stack Usage:	1

Notes

When a task is created, its timestamp is initialized to an `OStypeTS`-sized version of the system timer ticks, i.e. `(OStypeTS) OStimerTicks`.

In the example below, the task resets its timestamp upon starting. It then preserves its timestamp prior to invoking `OS_Delay()` as part of a hardware initialization sequence. Thereafter, it will time out every 6 ticks relative to when it started. If `OS_Delay()` had been used, it would time out every six ticks relative to when `OS_Delay()` was called.

See `OS_DelayTS()` for more information on timestamps.

See Also

`OS_DelayTS()`, `OSGetTS()`, `OSSyncTS()`

Example

```
void Task ( void )
{
  OStypeTS timestamp;


  /* synchronize delays with the start of this */
  /*  task, i.e. timestamp = now.              */
  OSSetTS((OStypeTS) OSGetTicks());

  /* do various things here.                   */
  …
  OS_Yield();
  …

  /* initialize some peripheral that requires  */
  /*  a short delay. Must preserve timestamp   */
  /*  when calling OS_Delay().                 */
  …
  timestamp = OSGetTS();
  OS_Delay(1);
  OSSetTS(timestamp);
  /* continue initializing said peripheral.    */
  …

  while (1)
  {
    /* as long as no more than 5 ticks have    */[87]
    /*  passed since this task was started,    */
    /*  the task will timeout at timestamp + 6 */
    /*  ticks, and then timestamp + 12, + 18,  */
    /*  etc.                                   */
    OS_DelayTS(6);
    …
  }
}
```

[87] 5 ticks because of the system timer's inherent +/- 1 tick accuracy.

OSSignalBinSem(): Signal a Binary Semaphore

Type:	Macro or Function
Prototype:	`OStypeErr OSSignalBinSem ( OStypeEcbP ecbP );`
Callable from:	Anywhere
Contained in:	`salvobinsem.c`
Enabled by:	`OSENABLE_BINARY_SEMAPHORES`, `OSEVENTS`
Affected by:	`OSCALL_OSSIGNALEVENT`, `OSENABLE_STACK_CHECKING`, `OSCOMBINE_EVENT_SERVICES`, `OSLOGGING`, `OSUSE_EVENT_TYPES`
Description:	Signal a binary semaphore. If one or more tasks are waiting for the semaphore, the highest-priority task is made eligible.
Parameters:	`ecbP`: a pointer to the semaphore's ecb.
Returns:	`OSERR_BAD_P` if binary semaphore pointer is incorrectly specified. `OSERR_EVENT_BAD_TYPE` if specified event is not a binary semaphore. `OSERR_EVENT_FULL` if binary semaphore is already 1. `OSNOERR` on success.
Stack Usage:	1

Notes

No more than one task can be made eligible by signaling a binary semaphore.

In the example below, a binary semaphore is used to signal a waiting task. `TaskWaveformGenerator()` outputs an 8-bit waveform to a DAC whenever it receives a signal to do so. The binary semaphore is initialized to 0, so `TaskWaveformGenerator()` remains in the waiting state until the `BINSEM_GEN_WAVEFORM` is signaled elsewhere in the program, whereupon it outputs an array of 8-bit values to a port. It then resumes waiting until `BINSEM_GEN_WAVEFORM` is signaled again.

See Also

`OS_WaitBinSem()`, `OSCreateBinSem()`, `OSReadBinSem()`, `OSTryBinSem()`

Example

```
…

#define BINSEM_GEN_WAVEFORM_P OSECBP(5)

…

OSCreateBinSem(BINSEM_GEN_WAVEFORM_P, 0);

…

/* tell waveform-generating task to create a   */
/*  single waveform.                           */
OSSignalBinSem(BINSEM_GEN_WAVEFORM_P);

…

void TaskWaveformGenerator ( void )
{
  char i;

  while (1) {
    /* wait forever for signal to generate     */
    /*  waveform.                              */
    OS_WaitBinSem(BINSEM_GEN_WAVEFORM_P,
      OSNO_TIMEOUT);

    /* output waveform to DAC.                 */
    for (i = 0; i < 256; i++) {
      DACPORT = WAVEFORM_TABLE[i];
    }
  }
}
```

OSSignalMsg(): Send a Message

Type:	Macro or Function
Prototype:	`OStypeErr OSSignalMsg ( OStypeEcbP ecbP, OStypeMsgP msgP );`
Callable from:	Anywhere
Contained in:	`salvomsg.c`
Enabled by:	`OSENABLE_MESSAGES`, `OSEVENTS`
Affected by:	`OSCALL_OSSIGNALEVENT`, `OSENABLE_STACK_CHECKING`, `OSCOMBINE_EVENT_SERVICES`, `OSLOGGING`, `OSUSE_EVENT_TYPES`
Description:	Signal a message with the value specified. If one or more tasks are waiting for the message, the highest-priority task is made eligible.
Parameters:	`ecbP`: a pointer to the message's ecb. `msgP`: a pointer to a message.
Returns:	`OSERR_BAD_P` if message pointer is incorrectly specified. `OSERR_EVENT_BAD_TYPE` if specified event is not a message. `OSERR_EVENT_FULL` if message is already defined. `OSNOERR` on success.
Stack Usage:	1

Notes

No more than one task can be made eligible by signaling a message.

In the example below, a message is used (in place of a binary semaphore) to control access to a shared resource, an LCD. When either `TaskDisplay()` or `TaskFlashWarning()` needs to write to the display, it must first acquire the display by successfully waiting on the message `MSG_LCD_RSRC`. Once obtained, the task can write to the LCD. When finished, it must release the resource by signaling the message.

`TaskFlashWarning()` displays a warning message for five seconds by writing to the display and then delaying itself for five seconds before releasing the resource. The use of a message to control access to the LCD prevents `TaskDisplay()` from overwriting the LCD while the warning message is displayed.

See Also

OS_WaitMsg(), OSCreateMsg(), OSReadMsg(), OSTryMsg()

Example

```
#define MSG_DISP_UPDATE_P OSECBP(2)    /* flag */
#define MSG_LCD_RSRC_P    OSECBP(3)    /* rsrc */
#define MSG_WARNING_P     OSECBP(4)    /* flag */
char strLCD[LCD_LENGTH+1]; /* 1 row chars + \0 */

void TaskDisplay ( voi d)
{
  static OStypeMsgP msgP;

  /* display is initially available to all.    */
  OSCreateMsg(MSG_LCD_RSRC_P, (OStypeMsgP) 1);

  while (1) {
    /* wait until display update is required   */
    OS_WaitMsg(MSG_DISP_UPDATE_P, &msgP,
      OSNO_TIMEOUT);

    /* wait if we can't acquire the resource.  */
    OS_WaitMsg(MSG_LCD_RSRC_P, &msgP,
      OSNO_TIMEOUT);

    /* write global string to display.         */
    WriteLCD(strLCD);

    /* free display for others to use.         */
    OSSignalMsg(MSG_LCD_RSRC_P, (OStypeMsgP) 1);
  }
}

void TaskFlashWarning ( void )
{
  static OStypeMsgP msgP, msgP2;

  while (1) {
    /* wait for the warning ...                */
    OS_WaitMsg(MSG_WARNING_P, &msgP,
      OSNO_TIMEOUT);

    /* grab the LCD, locking others out.       */
    OS_WaitMsg(MSG_LCD_RSRC_P, &msgP2,
      OSNO_TIMEOUT);

    /* Flash warning on LCD for 5 seconds.     */
    WriteLCD((char *)msgP);
    OS_Delay(FIVE_SEC);

    /* refresh / restore LCD, and free it.     */
    WriteLCD(strLCD);
    OSSignalMsg(MSG_LCD_RSRC_P, (OStypeMsgP) 1);
  }
}
```

OSSignalMsgQ(): Send a Message via a Message Queue

Type:	Macro or Function
Prototype:	`OStypeErr OSSignalMsgQ ( OStypeEcbP ecbP, OStypeMsgP msgP );`
Callable from:	Anywhere
Contained in:	`salvomsgq.c`
Enabled by:	`OSENABLE_MESSAGE_QUEUES`, `OSEVENTS`
Affected by:	`OSCALL_OSSIGNALEVENT`, `OSENABLE_STACK_CHECKING`, `OSCOMBINE_EVENT_SERVICES`, `OSLOGGING`, `OSUSE_EVENT_TYPES`
Description:	Send a message to a task via the message queue specified with ecbP. If one or more tasks are waiting the message queue, the highest-priority task is made eligible.
Parameters:	`ecbP`: a pointer to the message queue's ecb. `msgP`: a pointer to a message.
Returns:	`OSERR_BAD_P` if message queue pointer is incorrectly specified. `OSERR_EVENT_BAD_TYPE` if specified event is not a message queue. `OSERR_EVENT_CB_UNINIT` if the message queue's control block is uninitialized. `OSERR_EVENT_FULL` if message queue is full. `OSNOERR` on success.
Stack Usage:	1

Notes

No more than one task can be made eligible by signaling a message.

In the example below, `Commands[]` is a constant array of one-character commands. A message queue is used to send multiple commands to a waiting task. The two successive calls to `OSSignalMsg()` will place the HALT ('h') and EXIT ('x') commands into the message queue, but only if room is available. Upon arrival of the messages, the receiving task will act accordingly.

See Also

`OS_WaitMsgQ()`, `OSCreateMsgQ()`, `OSReadMsgQ()`, `OSTryMsgQ()`

Example

```
const char Commands[4] = { 'a', 'g', 'h', 'x' };
…

OSSignalMsgQ(MSGQ5_P, (OStypeMsgP) &Commands[2]);
OSSignalMsgQ(MSGQ5_P, (OStypeMsgP) &Commands[3]);
…
```

OSSignalSem(): Signal a Semaphore

Type:	Macro or Function
Prototype:	`OStypeErr OSSignalSem ( OStypeEcbP ecbP );`
Callable from:	Anywhere
Contained in:	`salvosem.c`
Enabled by:	`OSENABLE_SEMAPHORES`, `OSEVENTS`
Affected by:	`OSBIG_SEMAPHORES`, `OSCALL_OSSIGNALEVENT`, `OSENABLE_STACK_CHECKING`, `OSCOMBINE_EVENT_SERVICES`, `OSLOGGING`, `OSUSE_EVENT_TYPES`
Description:	Increment a counting semaphore. If one or more tasks are waiting for the semaphore, the highest-priority task is made eligible.
Parameters:	`ecbP`: a pointer to the semaphore's ecb.
Returns:	`OSERR_BAD_P` if semaphore pointer is incorrectly specified. `OSERR_EVENT_BAD_TYPE` if specified event is not a semaphore. `OSERR_EVENT_FULL` if semaphore is already at its maximum allowed value. `OSNOERR` on success.
Stack Usage:	1

Notes

No more than one task can be made eligible by signaling a semaphore.

8- or 16-bit semaphores can be selected via the `OSBIG_SEMAPHORES` configuration option.

In the example below, a counting semaphore is used to keep track of how many characters are waiting in the receive buffer `rxBuff`. Another task that waits on `SEM_RX_BUFF` will remove and process them, one at a time, from the buffer. By communicating between the tasks with a semaphore, the tasks can run at different priorities – `TaskRx()` can run at a high priority to ensure that the UART's receive buffer is not overrun, and the processing task (which waits on `SEM_RX_BUFF`) can run at a lower priority while parsing incoming command strings.

See Also

`OS_WaitSem()`, `OSCreateSem()`, `OSReadSem()`, `OSTrySem()`

Example

```
void TaskRx ( voi d)
{
  /* initially there are no Rx chars for       */
  /* TaskRcvRsp() to process.                  */
  OSCreateSem(SEM_RX_RBUFF_P, 0);

  /* The task to interpret responses is driven */
  /*  solely by TaskRx()'s collecting incoming */
  /*  incoming chars for it, so we'll launch   */
  /*  it from here.                            */
  OSCreateTask(TaskRcvRsp, TASK_RCV_RSP_P,
    TASK_RCV_RSP_PRIO);

  /* deal with Rx chars. */
  while (1) {
    /* if there are any Rx chars waiting,      */
    /*  signal the command interpreter.        */
    while (SioRxQue(Port) > 0)
    {
      /* put new Rx char into local buffer     */
      rxBuff[rxTail] = (char) SioGetc(Port, 10);

      /* massage buffer pointers               */
      rxTail++;
      rxCount++;
      if (rxTail >= SIZEOF_RX_BUFF)
        rxTail = 0;

      /* signal the command interpreter that   */
      /*  there's work to be done. In this     */
      /*  implementation we signal once for    */
      /*  every new character received.        */
      OSSignalSem(SEM_RX_RBUFF_P);

    }

  /* wait a while and poll again.              */
  OS_Delay(1);
  }
}
```

OSStartCycTmr(): Start a Cyclic Timer

Type:	Function
Prototype:	`OStypeErr OSStartCycTmr ( OStypeTcbP tcbP );`
Callable from:	Background only
Contained in:	`salvocyclic2.c`
Enabled by:	`OSENABLE_CYCLIC_TIMERS`
Affected by:	–
Description:	Start the specified cyclic timer.
Parameters:	`tcbP`: a pointer to the cyclic timer's tcb.
Returns:	`OSNOERR` if cyclic timer is successfully started. `OSERR_BAD_CT` if the tcb in question does not belong to a cyclic timer. `OSERR_BAD_P` if the specified tcb pointer is invalid (i.e. out-of-range). `OSERR_CT_RUNNING` if the cyclic timer is already running.
Stack Usage:	3

Notes

`OSStartCycTmr()` can only start a cyclic timer that is stopped.

If `OSStartCycTmr()` operates on a cyclic timer that has not yet started (e.g. it was created with `OSDONT_START_CYCTMR`), then it will begin with its delay period, followed by its normal period. If, on the other hand, the cyclic timer was already started and then stopped, invoking `OSStartCycTmr()` will cause it to restart after its normal period.

In the example below, `Task3()` allows the cyclic timer to run for 400ms[88] while bit 3 of the port is high, and stops the cyclic timer from running when bit 3 is low. This is repeated indefinitely, and requires that the cyclic timer be in continuous mode.

See Also

`OSCreateCycTmr()`, `OSCycTmrRunning()`, `OSDestroyCycTmr()`, `OSResetCycTmr()`, `OSSetCycTmrPeriod()`, `OSStopCycTmr()`

[88] Assumes 10ms system tick period.

Example

```
void Task3( void )
{
  while (1) {
    OS_Delay(40);

    PORT ^= 0x08;

    if (PORT & 0x08) {
      OSStartCycTmr(OSTCBP(1));
    }
    else {
      OSStopCycTmr(OSTCBP(1));
    }
  }
}
```

OSStartTask(): Make a Task Eligible To Run

Type:	Function
Prototype:	`OStypeErr OSStartTask ( OStypeTcbP tcbP );`
Callable from:	Anywhere
Contained in:	`salvotask.c`
Enabled by:	–
Affected by:	`OSLOGGING`, `OSENABLE_STACK_CHECKING`
Description:	Start the specified task.
Parameters:	`tcbP`: a pointer to the task's tcb.
Returns:	`OSNOERR` if task is successfully started. `OSERR` if either the specified tcb pointer is invalid (i.e. out-of-range), or if the specified task's state is not `OSTCB_TASK_STOPPED`.
Stack Usage:	3

Notes

`OSStartTask()` can only start a task that is in the stopped (`OSTCB_TASK_STOPPED`) state.

Starting a task simply places it into the eligible queue. It will not run until it becomes the highest-priority eligible task.

A task that has been started is in the eligible state.

A task must be created via `OSCreateTask()` before it can be started via `OSStartTask()`.

In the example below, `TaskToggleLED()` is created but is only made eligible to run via the call to `OSStartTask()`. Without the call to `OSStartTask()`, the task would remain stopped indefinitely.

See Also

`OSCreateTask()`, `OSInit()`

Example

```
…

/* this task toggles an LED each time it       */
/*  runs, i.e. whenever it's the highest-      */
/*  priority eligible task.                    */
void TaskToggleLED ( void )
{
  while (1) {
    /* toggle LED on pin 0 of PORT B */
    PORTB ^= 0x01;

    OS_Yield();
  }
}

int main ( void )
{
  …

  /* create and start TaskToggleLED0() with    */
  /*  the lowest priority. We'll observe the   */
  /*  LED toggling when no other tasks are     */
  /*  eligible to run.                         */
  OSCreateTask(TaskToggleLED, OSTCBP(5),
    OSDONT_START_TASK | OSLOWEST_PRIO);

  …

  OSStartTask(OSTCBP(5));

  …

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

OSStopCycTmr(): Stop a Cyclic Timer

Type:	Function
Prototype:	`OStypeErr OSStopCycTmr ( OStypeTcbP tcbP );`
Callable from:	Background only
Contained in:	`salvocyclic3.c`
Enabled by:	`OSENABLE_CYCLIC_TIMERS`
Affected by:	–
Description:	Stop the specified cyclic timer.
Parameters:	`tcbP`: a pointer to the cyclic timer's tcb.
Returns:	`OSNOERR` if cyclic timer is already stopped or is successfully stopped. `OSERR_BAD_CT` if the tcb in question does not belong to a cyclic timer.
Stack Usage:	3

Notes

`OSStopCycTmr()` takes no action when the cyclic timer is already stopped.

In the example below, the cyclic timer occupying the fifth task control block is stopped.

See Also

`OSCreateCycTmr()`, `OSCycTmrRunning()`, `OSDestroyCycTmr()`, `OSResetCycTmr()`, `OSSetCycTmrPeriod()`, `OSStartCycTmr()`

Example

```
…
OSStopCycTmr(OSTCBP(5));
```

OSStopTask(): Stop a Task

Type:	Function
Prototype:	`OStypeErr OSStopTask ( OStypeTcbP tcbP );`
Callable from:	Task or Background
Contained in:	`salvotask2.c`
Enabled by:	–
Affected by:	`OSENABLE_STACK_CHECKING`
Description:	Stop the specified task.
Parameters:	`tcbP`: a pointer to the task's tcb.
Returns:	`OSNOERR` if specified task was successfully stopped. `OSERR` if `OSStopTask()` was unable to stop the specified task.
Stack Usage:	3

Notes

`OSStopTask()` can stop any task that is not already destroyed or waiting an event.

A stopped task can be restarted with `OSStartTask()`.

In the example below, `TaskStopBeep()` exists only to stop another task, `TaskBeep()`. `TaskStopBeep()` waits forever for the binary semaphore `BINSEM_STOP_BEEP` to be signaled. When this occurs, it calls `OSStopTask()`, which stops `TaskBeep()`. `TaskStopBeep()` then begins waiting the binary semaphore again. By setting `TaskStopBeep()`'s priority to be higher than `TaskBeep()`'s, `TaskStopBeep()` is able to stop `TaskBeep()` at the earliest opportunity.

This example also illustrates how program control can pass from an interrupt through a task and affect another task, even if `OSStopTask()` is not called from an interrupt. By calling `OSSignalBinSem(BINSEM_STOP_BEEP)` from an ISR, `TaskBeep()` will be stopped by `TaskStopBeep()` before its earliest opportunity to run again.

See Also

`OSStartTask()`, `OS_Stop()`

Example

```
OSCreateTask(TaskBeep,     TASK_BEEP_P,     7);
OSCreateTask(TaskStopBeep, TASK_STOPBEEP_P, 6);
OSCreateSem(BINSEM_STOP_BEEP_P, 0);
…
void TaskStopBeep ( void )
{
  while (1) {
    OS_WaitBinSem(BINSEM_STOP_BEEP_P,
      OSNO_TIMEOUT);
    OSStopTask();
  }
}
```

OSSyncTS(): Synchronize the Current Task's Timestamp

Type:	Macro (invokes `OSSyncTSTask()`)
Prototype:	`void OSSyncTS ( OStypeInterval interval );`
Callable from:	Task only
Contained in:	`salvodelay2.c`
Enabled by:	–
Affected by:	`OSENABLE_DELAYS`, `OSENABLE_TICKS`
Description:	Synchronize the current task's timestamp against the current timer ticks.
Parameters:	`interval`: a signed offset relative to the current timer ticks.
Returns:	–
Stack Usage:	2

Notes

`OSSyncTS()` is used in conjunction with `OS_DelayTS()` to synchronize the current task's delays against an absolute value of the system's timer ticks. With `OSSyncTS()`, you can increment or decrement the value of current task's timestamp.[89]

In the example below, `TaskPeriodic()` begins by running every 16 system ticks. If the global variable `shiftTicks` is found to be non-zero, it is copied to a local variable `offset`, cleared, and then used to phase-shift `TaskPeriodic()` with a resolution of 1 system tick.

See Also

`OS_DelayTS()`, `OSGetTS()`, `OSSetTS()`

[89] Use `OSSetTS()` to change the absolute value of the current task's timestamp.

Example

```
OStypeInterval  shiftTicks;   /* -15 to +15    */
…
void TaskPeriodic ( void )
{
  OStypeInterval offset;

  while (1) {
    OS_DelayTS(16);
    …
    if (shift) {
      disable_interrupts();
      offset = shiftTicks;
      shiftTicks = 0;
      enable_interrupts();
      OSSyncTS(offset);
    }
  }
}
```

OSTimer(): Run the Timer

Type:	Function
Prototype:	`void OSTimer ( void );`
Callable from:	Foreground (preferred) or background.
Contained in:	`salvotimer.c`
Enabled by:	`OSBYTES_OF_DELAYS`, `OSBYTES_OF_TICKS`
Affected by:	`OSDISABLE_ERROR_CHECKING`, `OSENABLE_DELAYS`, `OSENABLE_STACK_CHECKING`, `OSENABLE_TICKS`, `OSTIMER_PRESCALAR`
Description:	Perform Salvo's timer-based services.
Parameters:	–
Returns:	–
Stack Usage:	2 if `OSUSE_INLINE_OSTIMER` is `FALSE`. 1 if `OSUSE_INLINE_OSTIMER` is `TRUE`.

Notes

If delay, elapsed time and/or timeout services are desired, `OSTimer()` must be called at the desired system tick rate. Context switching and event services do not require `OSTimer()` to be installed.

The rate at which `OSTimer()` is called by your application (typically every 5-100ms) must allow sufficient time for `OSTimer()` to complete its actions.

In the example below, the timer is called from within an interrupt service routine (ISR) as a periodic event. Each time `OSTimer()` is called it checks to see if any delayed or waiting tasks have timed out, and if so, re-enters them into the eligible queue.

`OSTimer()` is very small and is easily incorporated into an ISR without major deleterious effects.

Example

```
void interrupt ISR ( void )
{
  /* OSTimer() is called on every timer0       */
  /*  interrupt.                               */

  if (TOIF) {
    /* must clear timer0 interrupt flag.       */
    TOIF = 0;

    /* let Salvo handle delays, ticks          */
    /*  and timeouts.                          */
    OSTimer();
  }

  /* handle other interrupt sources.           */
  …
}
```

OSTryBinSem(): Obtain a Binary Semaphore if Available

Type:	Function
Prototype:	`OStypeBinSem OSTryBinSem ( OStypeEcbP ecbP );`
Callable from:	Anywhere
Contained in:	`salvobinsem2.c`
Enabled by:	`OSENABLE_BINARY_SEMAPHORES`, `OSENABLE_EVENT_READING`, `OSEVENTS`
Affected by:	`OSCALL_OSRETURNEVENT`
Description:	Returns the binary semaphore specified by `ecbP`. If the semaphore is 1, reset it to 0.
Parameters:	`ecbP`: a pointer to the binary semaphore's ecb.
Returns:	Binary semaphore (0 or 1).
Stack Usage:	1

Notes

`OSTryBinSem()` is like `OS_WaitBinSem()`, but it does not context-switch the current task if the binary semaphore is not available (i.e. has a value of 0). Therefore `OSTryBinSem()` can be used outside of the current task to obtain the binary semaphore, e.g. in an ISR.

No error checking is performed on the `ecbP` parameter. Calling `OSTryBinSem()` with an invalid `ecbP`, or an `ecbP` belonging to an event other than a binary semaphore, will return an erroneous result.

In the example below, `TaskC()`has a higher priority than `TaskD()` and obtains the binary semaphore whenever it is set to 1. Signaling the binary semaphore does not change the state of `TaskC()`. As long as `TaskC()` is running, `TaskD()` will wait forever for the binary semaphore.[90]

See Also

`OS_WaitBinSem()`, `OSCreateBinSem()`, `OSReadBinSem()`, `OSSignalBinSem()`

[90] This assumes that `TaskD()` unsuccessfully waited the binary semaphore before `TaskC()` started running.

Example

```
/* priority of 3                                    */
void TaskC ( void )
{
  while (1) {
    if (OSTryBinSem(BINSEM2_P)) {
      printf("binSem #2 was 1, now 0.\n");
    }
    else {
      printf("binSem #2 is 0.\n");
    }

    OS_Yield();
    …
  }
}

/* priority of 9 (lower)                            */
void TaskD ( void )
{
  while (1) {
    OS_WaitBinSem(BINSEM2_P,
      OSNO_TIMEOUT);
    …
  }
}
```

OSTryMsg(): Obtain a Message if Available

Type:	Function
Prototype:	`OStypeMsg OSTryMsg ( OStypeEcbP ecbP );`
Callable from:	Anywhere
Contained in:	`salvomsg2.c`
Enabled by:	`OSENABLE_MESSAGES`, `OSENABLE_EVENT_READING`, `OSEVENTS`
Affected by:	`OSCALL_OSRETURNEVENT`
Description:	Returns a pointer to the message specified by `ecbP`. If the message exists, the message's own pointer is cleared.
Parameters:	`ecbP`: a pointer to the message's ecb.
Returns:	Message pointer.
Stack Usage:	1

Notes

`OSTryMsg()` is like `OS_WaitMsg()`, but it does not context-switch the current task if the message is not available (i.e. the message pointer has a value of 0). Therefore `OSTryMsg()` can be used outside of the current task to obtain the message, e.g. in an ISR.

No error checking is performed on the `ecbP` parameter. Calling `OSTryMsg()` with an invalid `ecbP`, or an `ecbP` belonging to an event other than a binary semaphore, will return an erroneous result.

Waiting on a message (i.e. via `OS_WaitMsg()`) is not permitted within an interrupt service routine. In the example below, `OSTryMsg()` is used within the ISR in order to obtain a message without waiting. Regardless of whether or not a message was available, the message will be empty at the end of the ISR.

See Also

`OS_WaitMsg()`, `OSCreateMsg()`, `OSReadMsg()`, `OSSignalMsg()`

Example

```
void interrupt myISR ( void )
{
  OStypeMsgP msgP;

  /* get message pointer (may be 0).          */
  msgP = OSTryMsg(MSG3_P);

  while (1) {
    /* do something with the message.         */
    …
  }
  else
  {
    /* message wasn't available.              */
    …
  }
  …
}
```

OSTryMsgQ(): Obtain a Message from a Message Queue if Available

Type:	Function
Prototype:	`OStypeMsgQ OSTryMsgQ ( OStypeEcbP ecbP );`
Callable from:	Anywhere
Contained in:	`salvomsgq2.c`
Enabled by:	`OSENABLE_MESSAGE_QUEUES`, `OSENABLE_EVENT_READING`, `OSEVENTS`
Affected by:	`OSCALL_OSRETURNEVENT`
Description:	Returns a pointer to the first available message in the message queue specified by `ecbP`. If the message queue contains any messages, remove the message from the queue.
Parameters:	`ecbP`: a pointer to the message queue's ecb.
Returns:	Message pointer.
Stack Usage:	1

Notes

`OSTryMsgQ()` is like `OS_WaitMsgQ()`, but it does not context-switch the current task if the message queue is empty. Therefore `OSTryMsgQ()` can be used outside of the current task to obtain the message in the message queue, e.g. in an ISR.

No error checking is performed on the `ecbP` parameter. Calling `OSTryMsgQ()` with an invalid `ecbP`, or an `ecbP` belonging to an event other than a binary semaphore, will return an erroneous result.

In the example below, after each call to the scheduler, a `char` message is removed from a message queue and then re-inserted. As long as no services involving this message queue are called from within an interrupt, this will rotate the order of the messages in the message queue indefinitely. For example, a message queue containing the four single-character messages 's', 't', 'o' and 'p' becomes 't', 'o', 'p' and 's'.

See Also

`OS_WaitMsgQ()`, `OSCreateMsgQ()`, `OSReadMsgQ()`, `OSSignalMsgQ()`

Example

```
OStypeMsgP msgP;
…
while (1) {
  OSSched();

  msgP = OSTryMsgQ(MSGQ3_P);

  if (msgP) {
    printf("removed message %c from msgQ.\n",
      *(char *) msgP);

    OSSignalMsgQ(MSGQ3_P, msgP);

    printf("re-inserted message into msgQ.\n");
  }
}
```

OSTrySem(): Obtain a Semaphore if Available

Type:	Function
Prototype:	`OStypeSem OSTrySem ( OStypeEcbP ecbP );`
Callable from:	Anywhere
Contained in:	`ssalvosem2.c`
Enabled by:	`OSENABLE_SEMAPHORES`, `OSENABLE_EVENT_READING`, `OSEVENTS`
Affected by:	`OSCALL_OSRETURNEVENT`
Description:	Returns the semaphore specified by `ecbP`. If the semaphore is non-zero, decrement it.
Parameters:	`ecbP`: a pointer to the semaphore's ecb.
Returns:	Semaphore.
Stack Usage:	1

Notes

`OSTrySem()` is like `OS_WaitSem()`, but it does not context-switch the current task if the semaphore is not available (i.e. has a value of 0). Therefore `OSTrySem()` can be used outside of the current task to obtain the semaphore, e.g. in an ISR.

No error checking is performed on the `ecbP` parameter. Calling `OSTrySem()` with an invalid `ecbP`, or an `ecbP` belonging to an event other than a binary semaphore, will return an erroneous result.

In the example below, `OSTrySem()` is used by `FlushBuffer()`[91] to flush a buffer that is managed through a counting semaphore. Afterwards, `i` holds the count of the items that were in the the buffer before it was flushed.

See Also

`OS_WaitSem()`, `OSCreateSem()`, `OSReadSem()`, `OSSignalSem()`

[91] Note that `FlushBuffer()` is a simple function, and not a task. The flushing operation could also be performed in a task.

Example

```
/* buffer is initially empty.                    */
OSCreateSem(SEM2_P, 0);
…
void FlushBuffer ( void )
{
  char i;

  /* count and remove the buffer's contents.    */
  i = 0;
  while (OSTrySem(SEM2_P)) {
    i++;
  }
}
```

Additional User Services

OSAnyEligibleTasks (): Check for Eligible Tasks

Type:	Macro
Declaration:	`OSAnyEligibleTasks()`
Callable from:	Outside `OSSched()` (background) or inside a task or its subroutines.
Contained in:	`salvomac.h`
Enabled by:	–
Affected by:	–
Description:	Detect if any tasks are currently eligible to run.
Parameters:	–
Returns:	`TRUE` if one or more tasks are eligible, `FALSE` otherwise.
Stack Usage:	0

Notes

`OSAnyEligibleTasks()` cannot predict when waiting and/or delayed tasks will become eligible. This must be considered when using `OSAnyEligibleTasks().`

`OSAnyEligibleTasks()` returns `FALSE` if a task is running and no tasks are eligible.

In the first example below, a Salvo application's main loop has been modified to run an alternative process (e.g. some legacy code written in assembler) in addition to the scheduler. This alternative process must terminate within a short time in order to avoid problems scheduling tasks. By invoking the alternative process only when no tasks are eligible, it can "steal cycles" that the scheduler does not currently need.

In the second example, a user *function* (not a task) is called only when the system is idling, i.e. when tasks are eligible to run. This idling function must execute quickly so as not to affect task execution.

Note that in both examples, Salvo's idling hook could be used in place of `OSAnyEligibleTasks()` if it were not already in use.

Example #1

```
int main ( void )
{
  …
  while (1) {
    OSSched();

    if (!OSAnyEligibleTasks()) {
      /* do alternative background process */
      #asm
      #include "mystuff.asm"
      #endasm
    }
  }
}
```

Example #2

```
int main ( void )
{
  …
  while (1) {
    OSSched();

    if (!OSAnyEligibleTasks())
      DoWhileIdling();
  }
}
```

OScTcbExt0|1|2|3|4|5, OStcbExt0|1|2|3|4|5(): Return a Tcb Extension

Type:	Macro
Declaration:	`OScTcbExt0|1|2|3|4|5, OStcbExt0|1|2|3|4|5(tcbP)`
Callable from:	`OScTcbExt0|1|2|3|4|5` should only be called from the task level. `OStcbExt0|1|2|3|4|5()` can be called from anywhere.
Contained in:	`salvomac.h`
Enabled by:	`OSENABLE_TCBEXT0|1|2|3|4|5`
Affected by:	–
Description:	`OScTcbExt0|1|2|3|4|5` returns the specified tcb extension of the current task. `OStcbExt0|1|2|3|4|5` returns the specified tcb extension of the specified task.
Parameters:	–
Returns:	Tcb extension.
Stack Usage:	0.

Notes

These macros are used to obtain the desired tcb extension from the task's tcb.

See Also

`OSENABLE_TCBEXT0|1|2|3|4|5`, `OSTYPE_TCBEXT0|1|2|3|4|5`

Example

```
void CommTask ( void )
{
  /* ascertain mode at startup */
  switch (OScTcbExt3) {

    case SW_HANDSHAKING:
      while (1) {
        /* do comms w/ XON/XOFF */
        OpenSWUART();
        …
        OS_Yield();
      }
      break;

    case HW_HANDSHAKING:
      while (1) {
        /* do comms w/ DTR & CTS */
        OpenHWUART();
        …
        OS_Yield();
      }
      break;

    default:
      break;
  }
}

int main ( void )
{
  …
  /* we want hardware handshaking … */
  OSCreateTask(CommTask, OSTCBP(7), 5);
  OStcbExt3(OSTCBP(7)) = HW_HANDSHAKING;
  …
  while (1) {
    OSSched();
  }
}
```

OSCycTmrRunning(): Check Cyclic Timer for Running

Type:	Function
Prototype:	`OStypeErr OSCycTmrRunning ( OStypeTcbP tcbP );`
Callable from:	Background only
Contained in:	`salvocyclic7.c`
Enabled by:	`OSENABLE_CYCLIC_TIMERS`
Affected by:	–
Description:	Detect if cyclic timer is running or not.
Parameters:	`tcbP`: a pointer to the cyclic timer's tcb.
Returns:	`FALSE` if cyclic timer is stopped, or if the tcb in question does not belong to a cyclic timer. `TRUE` if cyclic timer is running.
Stack Usage:	1

Notes

`OSCycTmrRunning()` indicates whether or not a cyclic timer is running.

In the example below, a task waits for a signal to restart a cyclic timer. When that signal is received, the cyclic timer is stopped and restarted. Regardless of how close it was previously to timing out, it will now time out in its normal period.

See Also

`OSCreateCycTmr()`, `OSCycTmrPeriod()`, `OSDestroyCycTmr()`, `OSResetCycTmr()`, `OSStartCycTmr()`, `OSStopCycTmr()`

Example

```
…
if (OSCycTmrRunning(OSTCBP(3)))
{
  /* do something if cyclic timer is running.  */
}
```

OSProtect(), OSUnprotect(): Protect Services Against Corruption by ISR

Type:	Macro
Declaration:	`OSProtect(), OSUnprotect()`
Callable from:	Background
Contained in:	`salvoportXyz.h`
Enabled by:	–
Affected by:	–
Description:	Disable or enable interrupts, respectively, if such control is required on given target.
Parameters:	–
Returns:	n/a
Stack Usage:	0, unless defined otherwise.

Notes

When compiling for a target that does not have a software stack, certain steps must be taken to protect servicse with multiple callgraphs. By calling `OSProtect()` immediately before each such service, and `OSUnprotect()` immediately thereafter, the service is protected against any corruption that might occur if an interrupt that calls the service were to occur simultaneously.

These macros are empty for all targets whose compilers pass parameters on a stack. To ensure cross-platform compatibility, *all* Salvo applications should use `OSProtect()` and `OSUnprotect()` as specified, even if these macros are empty for a particular compiler.

Warning Because a stackless compiler may overlay the local / parameter areas of one or more services with multiple callgraphs, `OSProtect()` and `OSUnprotect()` should be used around *every* service whose `OSCALL_XYZ` is set to `OSFROM_ANYWHERE`.

In the example below, `OSSignalBinSem()` is called from mainline code and from within an ISR. Therefore `OSProtect()` and `OSUnprotect()` are required in the mainline code.

See Also

`OSCALL_OSXYZ`, `OSFROM_ANYWHERE`, `OSDi()`, `OSEi()`, *Salvo Compiler Reference Manuals*

Example

```
void TestCode ( void )
{
    …
    if (PutTx1Buff(data)) {
        OSProtect();
        OSSignalBinSem(BINSEM_TXBUFF_P);
        OSUnprotect();
    }
    …
}

void interrupt ISR ( void )
{
    …
    if (txState == TXSTATE_DONE) {
        txState = TXSTATE_IDLE;
        OSSignalBinSem(BINSEM_TXDONE_P);
    }
    …
}
```

OSTaskStopped(): Check whether Task has Stopped

Type:	Macro
Declaration:	`OSTaskStopped ( OStypeTcbP tcbP );`
Callable from:	anywhere
Contained in:	`salvomac.h`
Enabled by:	–
Affected by:	–
Description:	Detect if the current task is stopped.
Parameters:	–
Returns:	`TRUE` if task is stopped, `FALSE` otherwise.
Stack Usage:	0

Notes

`OSTaskStopped()` does not check the validity of the task handle passed to it.

In the example below, the task pointed to by `TASK_FREQ_P` is (re-) started if already stopped. Otherwise it is stopped.

See Also

–

Example

```
…
OS_WaitSem(SEM_CMD_CHAR_P, OSNO_TIMEOUT);

if (cmd = getchar1()) {

  switch (tolower((char) cmd)) {

    …

    case 'f':
      if (OSTaskStopped(TASK_FREQ_P)) {
        OSStartTask(TASK_FREQ_P);
        user_msg(STR_TASK_CMD STR_TABS "f:" \
                 " Started task_freq().");
      }
      else {
        OSStopTask(TASK_FREQ_P);
        user_msg(STR_TASK_CMD STR_TABS "f:" \
                 " Stopped task_freq().");
      }
      break;
    …
  }
}
```

OSTimedOut(): Check for Timeout

Type:	Macro
Declaration:	`OSTimedOut()`
Callable from:	Task only
Contained in:	`salvomac.h`
Enabled by:	`OSENABLE_TIMEOUTS`
Affected by:	–
Description:	Detect if the current task timed out waiting for an event.
Parameters:	–
Returns:	`TRUE` if a timeout occurred, `FALSE` otherwise.
Stack Usage:	0

Notes

By specifying a non-zero timeout in `OS_WaitBinSem()`, `OS_WaitMsg()`, `OS_WaitMsgQ()` or `OS_WaitSem()`, you can control program execution in the case where an event does not occur within a specified number of system ticks. This is very useful in handling errors that may result from expected events failing to occur.

Once a timeout occurs, the task is no longer waiting the event. The fact that a timeout occurred only indicates that the task did not successfully wait the event in the allotted time … it does not in any way reflect on the current status of the event, or on other tasks waiting the event.

In the example below, a bidirectional communications channel is used to send commands and receives a response (acknowledgments) for each command sent. A new command can be sent only after the acknowledgment for the previous command has been received. By specifying a response timeout (`RSP_TIMEOUT`) that's larger than the expected time for the receiver to respond to a command, `TaskTx()` can conditionally wait for the response instead of waiting indefinitely if the acknowledgment never arrives.

When a timeout occurs, a task's execution resumes where it was originally waiting for the event, and the Salvo function `OSTimedOut()` returns `TRUE` until the task context-switches back to the scheduler. `TaskTx()` checks to see if a timeout occurred after it acquires the message.

See Also

OS_WaitBinSem(), OS_WaitMsg(), OS_WaitMsgQ(), OS_WaitSem()

Example

```
void TaskTx ( void )
{
  static OStypeMsgP msgP;

  /* No cmds have been sent yet, so no         */
  /*  responses have been received.            */
  OSCreateMsg(MSG_RSP_RCVD_P, (OStypeMsgP) 0);

  while (1) {
    /* send command to receiver.               */
    …

    /* wait here until response has been       */
    /*  received for the command we sent.      */
    /* if we timed out, reset the expected     */
    /*  response, STOP, clear the buffer and   */
    /*  tell the user.                         */
    OS_WaitMsg(MSG_RSP_RCVD_P, &msgP,
      RSP_TIMEOUT);

    if (OSTimedOut()) {
      FlushCmdInterpreter();
      setSTOP();
      txBuff[0] = 0;
      FlashMsg(&msgBadComms);
    }

    /* continue processing outgoing commands.  */
    …
  }
}
```

OSVersion(), OSVERSION: Return Version as Integer

Type:	Macro
Declaration:	`OSVersion(), OSVERSION`
Callable from:	Anywhere
Contained in:	`salvover.h`
Enabled by:	–
Affected by:	–
Description:	Returns the version number.
Parameters:	–
Returns:	Returns the version number as an unsigned integer.
Stack Usage:	0

Notes

Salvo uses three version number fields: `OSVER_MAJOR`, `OSVER_MINOR` and `OSVER_SUBMINOR`. Each field is a numeric integer constant. They are combined into a single symbol, `OSVERSION`, in the following manner:

```
OSVERSION = OSVER_MAJOR   * 100
          + OSVER_MINOR   *  10
          + OSVER_SUBMINOR
```

Therefore in v3.0.0, `OSVERSION` equals 300.

`OSVersion()` is identical to `OSVERSION`.

Example

```
printf("Salvo version: %d (v%c.%c.%c)\n",
  '0' + OSVER_MAJOR,
  '0' + OSVER_MINOR,
  '0' + OSVER_SUBMINOR,
  OSVersion());
```

User Macros

This section describes the Salvo user macros that you will use to build your multitasking application.

The macros are described below.

OSECBP(), OSEFCBP(),OSMQCBP(), OSTCBP(): Return a Control Block Pointer

Type:	Macro

Declaration:	`OSECBP( index )`
	`OSEFCBP( index )`
	`OSMQCBP( index )`
	`OSTCBP( index )`

Callable from:	n/a
Contained in:	`salvo.h`
Enabled by:	–
Affected by:	–
Description:	Shorthand for pointer to specified control block.
Parameters:	`index`: an index from `1` to `OSEVENTS`, `1` to `OSEVENT_FLAGS`, `1` to `OSMESSAGE_QUEUES` or `1` to `OSTASKS`, respectively.
Returns:	pointer to (i.e. address of) desired event, message queue or task control block, respectively.
Stack Usage:	n/a

Notes

RAM memory for control blocks is allocated at compile time using the `OSEVENTS`, `OSEVENT_FLAGS`, `OSMESSAGE_QUEUES` and `OSTASKS` configuration options. Instead of obtaining the compile-time address of a particular event, event flag, message queue or task control block by using

```
&OSecbArea[i-1]
&OsefcbArea[i-1]
&OSmqcbArea[i-1]
&OStcbArea[i-1]
```

you can and *should* use these macros.

Example

```
#define TASK1_P      OSTCBP(1)
#define TASK2_P      OSTCBP(2)
#define SEM1_P       OSECBP(1)

…
OSCreateTask(Task1, TASK1_P, 7);
…
OSCreateSem(SEM1_P, 14);
…
```

User-Defined Services

OSDisableIntsHook(), OSEnableIntsHook(): Interrupt-control Hooks

Type:	Function
Declaration:	`void OSDisableIntsHook( void )` `void OSEnableIntsHook( void )`
Called from:	`OSDi()` and `OSEi()`
Contained in:	`salvo.h` if left undefined, otherwise in user source code.
Enabled by:	`OSENABLE_INTERRUPT_HOOKS`
Affected by:	–
Description:	User-defined.
Parameters:	–
Returns:	–
Stack Usage:	Dependent on user definition.

Notes

You may find it useful or necessary to perform certain operations coincident with Salvo's disabling and (re-)enabling of interrupts during critical sections of code.

If these functions are enabled via `OSENABLE_INTERRUPT_HOOKS`, `OSDisableIntsHook()` is called *immediately after disabling interrupts*, and `OSEnableIntsHook()` is called *immediately before (re-)enabling interrupts*. Therefore each function is called with interrupts disabled.

By default, these functions are undefined.

In the example below, two separate counters, `diCounter` and `eiCounter`, are used to count the number of times that Salvo disables and (re-)enables interrupts, respectively.

See Also

`OSDi()`, `OSEi()`

Example

```
unsigned long int diCounter, eiCounter;
…
void OSDisableIntsHook( void )
{
  diCounter++;
}

void OSEnableIntsHook( void )
{
  eiCounter++;
}
```

OSIdlingHook(): Idle Function Hook

Type:	Function
Declaration:	`void OSIdlingHook( void )`
Called from:	`OSSched()`
Contained in:	User source code, called from `sched.c`.
Enabled by:	`OSENABLE_IDLING_HOOK`
Affected by:	–
Description:	User-defined.
Parameters:	–
Returns:	–
Stack Usage:	Dependent on user definition.

Notes

Salvo's scheduler normally runs in a tight loop when no tasks are eligible to run, i.e. when it is idling. By defining an idle function and setting `OSENABLE_IDLING_HOOK` to `TRUE`, you can do something useful while the system is idling. Your idle function should be short and fast, as time spent in it delays the operation of the scheduler.

By default, `OSIdlingHook()` is undefined. However, Salvo libraries configured for the idling hook contain a dummy `OSIdlingHook()` function to avoid linker errors when the user fails to define a `OSIdlingHook()`.

In the example below, the least significant bit on an output port is toggled whenever there are no eligible or running tasks.

Example

```
void OSIdlingHook( void )
{
  PORTB ^= 0x01;
}
```

OSSchedDispatchHook(), OSSchedEntryHook(), OSSchedReturnHook(): Scheduler Hooks

Type:	Function

Declaration:	`void OSSchedDispatchHook( void )`
	`void OSSchedEntryHook( void )`
	`void OSSchedReturnHook( void )`

Called from:	`OSSched()`
Contained in:	User source code, called from `sched.c`.
Enabled by:	`OSENABLE_OSSCHED_DISPATCH_HOOK`, `OSENABLE_OSSCHED_ENTRY_HOOK`, and `OSENABLE_OSSCHED_RETURN_HOOK`, respectively
Affected by:	–
Description:	User-defined.
Parameters:	–
Returns:	–
Stack Usage:	Dependent on user definition.

Notes

It may be useful when debugging a Salvo application to have run-time information on the scheduler's behavior. These hooks are provided so that user-defined functions can be invoked at strategic times within `OSSched()`'s execution.

`OSSchedEntryHook()` is called immediately upon entry into the scheduler. `OSSchedDispatchHook()` is called immediately prior to dispatching the current eligible task, with interrupts enabled and `OScTcbP` pointing to the current task's control block. `OSSchedReturnHook()` is called immediately after the current task returns (yields) to the scheduler ... the current task can be in any state, interrupts are enabled, and `OScTcbP` still points to the current task's control block.

When the system is idling (i.e. there are no eligible tasks), neither `OSSchedDispatchHook()` nor `OSSchedReturnHook()` will be called.

By default, `OSSchedDispatchHook()`, `OSSchedEntryHook()` and `OSSchedReturnHook()` are all undefined.

In the example below, `PORTB[5]` is set just prior to dispatching the current task, and is cleared after the current task yields back to the scheduler. The time that `PORTB[5]` is high represents the dispatch overhead in `OSSched()`, plus the task's execution time. The time

between successive rising edges of `PORTB[5]` represents the instantaneous context-switching speed of the application.

Example

```
void OSSchedDispatchHook ( void )
{
  PORTB |= 0x20;
}

void OSSchedReturnHook ( void )
{
  PORTB &= ~0x20;
}
```

Return Codes

Many Salvo user services have return codes to indicate whether or not they were called successfully. Some are listed below. See the individual user service descriptions for more information on return codes.

`OSNOERR`:	No error.
`OSERR`:	An error was encountered while executing the user service.
`OSERR_TASK_BAD_P`:	An invalid pointer was passed to the user service.
`OSERR_EVENT_NA`:	The specified event was not available
`OSERR_EVENT_FULL`:	The specified event (e.g. message) is already full.
`OERR_EVENT_CB-_UNINIT`:	The specified control block (e.g. for message queues or event flags) has not yet been initialized.
`OSERR_TIMEOUT`:	The current task has timed out while waiting for an event.

Table 5: Return Codes

Salvo Defined Types

The following types are defined for use with Salvo user services. Because the types are affected by configuration options, *when interfacing to Salvo user services you should always declare variables with these defined types*. Failing to do so is likely to result in unpredictable behavior.

Salvo has two classes of predefined types: those where the memory (RAM) location of the object is not specified (*normal*, `OStypeXyz`), and those where the location is explicitly specified (*qualified*, `OSgltypeXyz`). The need for both types arises on those processors with banked RAM. If your target processor has a single linear RAM space, the two types are identical. When in doubt, use the qualified type if one exists.

The normal types are used in the Salvo source code when declaring auto variables, parameters and function return values. You can also use the normal types when declaring your own local variables (e.g.

message pointers of type `OStypeMsgP`), and when typecasting (e.g. `OSSignalMsg(MSGP, (OStypeMsgP) &array[2])`);

The qualified types are used to declare Salvo's global variables, and are also provided so that you can properly declare your own global variables for Salvo, e.g. message queues – `OSgltypeMsgQP MsgQBuff[SIZEOF_MSGQ]`.

Tip Refer to the Salvo source code for examples of when to use normal or qualified Salvo types.

The normal types are:

`OStypeBinSem`:	binary semaphore: `OStypeBoolean`
`OStypeBitField`:	size of bit fields in structures: `int` or `char`, depending on `OSUSE_CHAR_SIZED_BITFIELDS`
`OStypeBoolean`:	Boolean: `FALSE` (0) or `TRUE` (non-zero)
`OStypeCount`:	counter: `OStypeInt8u/16u/32u`, depending on `OSBYTES_OF_COUNTS`
`OStypeDelay`:	delay: `OStypeInt8u/16u/32u`, depending on `OSBYTES_OF_DELAYS`
`OStypeDepth`:	stack depth counter: `OStypeInt8u`
`OStypeEcb:`	event control block: structure
`OStypeEfcb:`	event flag control block: structure
`OStypeEFlag`:	event flag: `OStypeInt8u/16u/32u`, depending on configuration
`OStypeErr`:	function return code or error / warning / timeout counter: `OStypeInt8u`
`OStypeEType`:	event type: `OStypeInt8u`
`OStypeID:`	object ID: `OStypeInt8u`
`OStypeInt8u`:	integer: 8-bit, unsigned
`OStypeInt16u`:	integer: 16-bit, unsigned
`OStypeInt32u`:	integer: 32-bit, unsigned
`OStypeInterval`:	interval: `OStypeInt8/16/32`, depending on `OSBYTES_OF_DELAYS`
`OStypeMqcb:`	message queue control block: structure
`OStypeMsg`:	message: `void` or `const`, depending on `OSMESSAGE_TYPE`
`OStypeMsgQSize`:	number of messages in a message queue: `OStypeInt8u`
`OStypeOption:`	generic option: `OStypeInt8u`

OStypePrio:	task priority: OStypeInt8u, values from 0 to 15 are defined
OStypePS:	timer prescalar: OStypeInt8u/16u/32u, depending on configuration
OStypeSem:	semaphore: OStypeInt8u or OStypeInt16u, depending on configuration
OStypeState:	task state: OStypeInt8u, values from 0 to 7 are defined
OStypeStatus:	task status: bitfields of type OStypeInt8u for a task's running bit, state and priority
OStypeTcb:	task control block: structure
OStypeTcbExt:	tcb extension: void *, user-(re-)definable
OStypeTick:	timer ticks: OStypeInt8u/16u/32u, depending on configuration
OStypeTS:	timestamp: OStypeInt8u/16u/32u, depending on configuration of OSBYTES_OF_DELAYS

Table 6: Normal Types

The normal pointer types are:

OStypeCharEcbP:	pointer to banked (OSLOC_ECB) char
OStypeCharTcbP:	pointer to banked (OSLOC_TCB) char
OStypeEcbP:	pointer to banked (OSLOC_ECB) event control block
OStypeEfcbP:	pointer to banked (OSLOC_EFCB) event flag control block
OStypeMqcbP:	pointer to banked (OSLOC_MQCB) message queue control block
OStypeMsgP:	pointer to message
OStypeMsgPP:	pointer to pointer to message
OStypeMsgQPP:	pointer to banked (OSLOC_MSGQ) pointer to message
OStypeTcbP:	pointer to banked (OSLOC_TCB) task control block
OStypeTcbPP:	pointer to banked (OSLOC_ECB) pointer to banked (OSLOC_TCB) task control block
OStypeTFP:	pointer to (task) function

Table 7: Normal Pointer Types

The qualified types are:

OSgltypeCount:	qualified OStypeCount: banked (OSLOC_COUNT) counter
OSgltypeDepth:	qualified OStypeDepth: banked (OSLOC_DEPTH) stack depth counter
OSgltypeEcb:	qualified OStypeEcb: banked (OSLOC_ECB) event control block
OSgltypeEfcb:	qualified OStypeEfcb: banked (OSLOC_EFCB) event flag control block
OSgltypeErr:	qualified OStypeErr: banked (OSLOC_ERR) error counter
OSgltypeGlStat:	qualified OStypeGlStat: banked (OSLOC_GLSTAT) global status bits
OSgltypeLogMsg:	qualified char: banked (OSLOC_LOGMSG) log message character or string
OSgltypeMqcb:	qualified OStypeMqcb: banked (OSLOC_MQCB) message queue control block
OSgltypePS:	qualified OStypePS: banked (OSLOC_PS) timer prescalar
OSgltypeTcb:	qualified OStypeTcb: banked (OSLOC_TCB) task control block
OSgltypeTick:	qualified OStypeTick: banked (OSLOC_TICK) system ticks

Table 8: Qualified Types

The qualified pointer types are:

OSgltypeCTcbP:	qualified OStypeTcbP: banked (OSLOC_CTCB) pointer to banked task control block
OSgltypeEcbP:	qualified OStypeEcbP: banked (OSLOC_ECB) pointer to banked event control block
OSgltypeMsgQP:	qualified OStypeMsgP: banked (OSLOC_MSGQ) pointer to message
OSgltypeSigQP:	qualified OStypeTcbP: banked (OSLOC_SIGQ) pointer to banked task control block
OSgltypeTcbP:	qualified OStypeTcbP: banked (OSLOC_ECB) pointer to banked task control block

Table 9: Qualified Pointer Types

Note When declaring *pointers* using predefined Salvo pointer types on targets that have banked RAM, always declare each pointer on its own, like this:

```
OStypeMsgP msgP1;
OStypeMsgP msgP2;
```

Failing to do so (i.e. declaring multiple pointers by comma-delimiting them on one line) will result in an improper declaration.

Salvo Variables

Salvo's global variables (declared in `salvomem.c`) are listed below. The variable, the qualified type corresponding to the variable and a description of the variable are listed for each one. Advanced programmers may find it useful to read these variables during runtime or while debugging. In some development environments (e.g. Microchip MPLAB), these variable names will be available for symbolic debugging.

Warning Do not modify any of these variables during runtime – unpredictable results may occur.

`OScTcbP`	`OSgltypeCTcbP`	pointer to current task's task control block
`OSctxSws`	`OSgltypeCount`	context switch counter
`OSdelayQP`	`OSgltypeDelayQP`	pointer to delay queue
`OSecbArea[]`	`OSgltypeEcb`	event control block storage
`OSefcbArea[]`	`OSgltypeEfcb`	event flag control block storage
`OSeligQP`	`OSgltypeEligQP`	pointer to eligible queue
`OSerrs`	`OSgltypeErr`	runtime error counter
`OSframeP`	`OsgltypeFrameP`	frame pointer[92]
`OSglStat`	`OSgltypeGlStat`	global status bits

[92] Used in some Salvo context switcher to assist in stack frame operations.

OSidleCtxSws	OSgltypeCount	idle function calls counter
OSlogMsg[]	OSgltypeLogMsg	log (debug) message string
OSlostTicks	OSgltypeLostTick	accumulated timer ticks
OSmaxStkDepth	OSgltypeDepth	maximum stack depth achieved by Salvo functions
OSmqcbArea[]	OSgltypeMqcb	message queue control block storage
OSrtnAddr	OSgltypeTFP	task's return / resume address
OSsigQinP, OSsigQoutP	OSgltypeSigQP	signaled event queue insert and removal pointers
OSstkDepth	OSgltypeDepth	current stack depth of Salvo function
OStcbArea[]	OSgltypeTcb	task control block storage
OStimerTicks	OSgltypeTick	system timer ticks counter
OStimerPS	OSgltypePS	runtime timer prescalar
OStimeouts	OSgltypeErr	runtime timeout counter
OSwarns	OSgltypeErr	runtime warning counter

Table 10: Salvo Variables

Salvo Source Code

The Salvo source code is organized into files that handle tasks, resources, queues, data structures, utility functions, the monitor, and the many `#defines` that are used to configure Salvo for a variety of applications.

You can always review the source code if the manual is unable to answer your question(s). Modifying the source code is not recommended, as your application may not run properly when compiled with a later release of Salvo. Where applicable, user `#defines` and hooks for user functions are provided so that you can use Salvo in

conjunction with features that are not yet supported in the current release.

Salvo's source (`*.h` and `*.c`) files are listed below.

```
Pumpkin\Salvo\Inc\salvo.h
Pumpkin\Salvo\Inc\salvoadc.h
Pumpkin\Salvo\Inc\salvocri.h
Pumpkin\Salvo\Inc\salvoctx.h
Pumpkin\Salvo\Inc\salvodef.h
Pumpkin\Salvo\Inc\salvofpt.h
Pumpkin\Salvo\Inc\salvolbo.h
Pumpkin\Salvo\Inc\salvolib.h
Pumpkin\Salvo\Inc\salvoloc.h
Pumpkin\Salvo\Inc\salvolvl.h
Pumpkin\Salvo\Inc\salvomac.h
Pumpkin\Salvo\Inc\salvomcg.h
Pumpkin\Salvo\Inc\salvomem.h
Pumpkin\Salvo\Inc\salvompt.h
Pumpkin\Salvo\Inc\salvoocp.h
Pumpkin\Salvo\Inc\salvoprg.h
Pumpkin\Salvo\Inc\salvopsh.h
Pumpkin\Salvo\Inc\salvoscb.h
Pumpkin\Salvo\Inc\salvoscg.h
Pumpkin\Salvo\Inc\salvostr.h
Pumpkin\Salvo\Inc\salvotyp.h
Pumpkin\Salvo\Inc\salvover.h
Pumpkin\Salvo\Inc\salvowar.h

Pumpkin\Salvo\Src\salvobinsem.c
Pumpkin\Salvo\Src\salvobinsem2.c
Pumpkin\Salvo\Src\salvochk.c
Pumpkin\Salvo\Src\salvocyclic.c
Pumpkin\Salvo\Src\salvocyclic2.c
Pumpkin\Salvo\Src\salvocyclic3.c
Pumpkin\Salvo\Src\salvocyclic4.c
Pumpkin\Salvo\Src\salvocyclic5.c
Pumpkin\Salvo\Src\salvocyclic6.c
Pumpkin\Salvo\Src\salvocyclic7.c
Pumpkin\Salvo\Src\salvodebug.c
Pumpkin\Salvo\Src\salvodelay.c
Pumpkin\Salvo\Src\salvodelay2.c
Pumpkin\Salvo\Src\salvodelay3.c
Pumpkin\Salvo\Src\salvodestroy.c
Pumpkin\Salvo\Src\salvoeflag.c
Pumpkin\Salvo\Src\salvoeflag2.c
Pumpkin\Salvo\Src\salvoeid.c
Pumpkin\Salvo\Src\salvoevent.c
Pumpkin\Salvo\Src\salvohook_idle.c
Pumpkin\Salvo\Src\salvohook_interrupt.c
Pumpkin\Salvo\Src\salvohook_wdt.c
Pumpkin\Salvo\Src\salvoidle.c
Pumpkin\Salvo\Src\salvoinit.c
Pumpkin\Salvo\Src\salvoinit2.c
Pumpkin\Salvo\Src\salvoinit3.c
Pumpkin\Salvo\Src\salvoinit4.c
```

```
Pumpkin\Salvo\Src\salvointvl.c
Pumpkin\Salvo\Src\salvolicense.c
Pumpkin\Salvo\Src\salvomem.c
Pumpkin\Salvo\Src\salvomsg.c
Pumpkin\Salvo\Src\salvomsg2.c
Pumpkin\Salvo\Src\salvomsgq.c
Pumpkin\Salvo\Src\salvomsgq2.c
Pumpkin\Salvo\Src\salvomsgq3.c
Pumpkin\Salvo\Src\salvomsgq4.c
Pumpkin\Salvo\Src\salvoprio.c
Pumpkin\Salvo\Src\salvoprio2.c
Pumpkin\Salvo\Src\salvoqdel.c
Pumpkin\Salvo\Src\salvoqins.c
Pumpkin\Salvo\Src\salvorpt.c
Pumpkin\Salvo\Src\salvosched.c
Pumpkin\Salvo\Src\salvosem.c
Pumpkin\Salvo\Src\salvosem2.c
Pumpkin\Salvo\Src\salvostop.c
Pumpkin\Salvo\Src\salvotask.c
Pumpkin\Salvo\Src\salvotask2.c
Pumpkin\Salvo\Src\salvotask3.c
Pumpkin\Salvo\Src\salvotask4.c
Pumpkin\Salvo\Src\salvotask5.c
Pumpkin\Salvo\Src\salvotask6.c
Pumpkin\Salvo\Src\salvotask7.c
Pumpkin\Salvo\Src\salvotask8.c
Pumpkin\Salvo\Src\salvotick.c
Pumpkin\Salvo\Src\salvotid.c
Pumpkin\Salvo\Src\salvotimer.c
Pumpkin\Salvo\Src\salvoutil.c
Pumpkin\Salvo\Src\salvover.c
```

Listing 37: Source Code Files

Compiler-specific header and source files are listed in each compiler's *Salvo Compiler Reference Manual*.

Note Salvo source code uses tab settings of 2, i.e. tabs are equivalent to 2 spaces.

Locations of Salvo Functions

Below is a list of each Salvo function (including user services and certain internal functions called by user services, shown in *italics*) and the source file in which it resides. This list is provided to assist source code users in resolving compile-time link errors due to the failure to include a particular Salvo source code file in their project.

Note Under certain configurations, those functions marked with an '*' may be macros or in-lined code instead of functions.

OSClrEFlag()*	salvoeflag.c
OSCreateBinSem()*	salvobinsem.c
OSCreateEFlag()*	salvoeflag.c
OSCreateEvent()	salvoevent.c
OSCreateMsg()*	salvomsg.c
OSCreateMsgQ()*	salvomsgq.c
OSCreateSem()*	salvosem.c
OSCreateTask()	salvoinit2.c
OSCtxSw()*	salvoportxyz.h
OSDelay()	salvodelay.c
OSDelDelayQ()	salvoqdel.c
OSDelPrioQ()	salvoqdel.c
OSDelTaskQ()	salvotask7.c
OSDestroy()	salvodestroy.c
OSDestroyTask()	salvotask3.c
OSDispTcbP()	salvorpt.c
OSeID()	salvoeid.c
OSGetPrio()*	salvoprio2.c
OSGetPrioTask()	salvoprio2.c
OSGetTicks()	salvoticks.c
OSGetState()	salvotask.c
OSGetStateTask()	salvotask5.c
OSGetTS()	salvodelay2.c
OSInit()	salvoinit.c
OSInitEcb()	salvoinit4.c
OSInitPrioTask()	salvoinit2.c
OSInitTcb()	salvoinit3.c
OSInsDelayQ()	salvoqins.c
*OSInsElig()**	salvoqins.c
OSInsPrioQ()	salvoqins.c
OSInsTaskQ()	salvotask8.c
*OSLogErr()**	salvodebug.c
*OSLogMsg()**	salvodebug.c
*OSLogWarn()**	salvodebug.c
OSMakeStr()	salvodebug.c
OSMsgQEmpty()	salvomsgq3.c
OSPrintEcb()	salvorpt.c
OSPrintEcbP()	salvorpt.c
OSPrintTcb()	salvorpt.c
OSPrintTcbP()	salvorpt.c
OSReturnBinSem()	salvobinsem2.c
OSReturnEFlag()	salvoeflag2.c
OSReturnMsg()	salvomsg2.c
OSReturnMsgQ()	salvomsgq2.c
OSReturnSem()	salvosem2.c
OSRpt()	salvorpt.c
OSSaveRtnAddr()	salvoutil.c
OSSched()*	salvosched.c
OSSchedEntryHook()	salvosched.c
OSSchedDispatchHook()	salvosched.c
OSSchedReturnHook()	salvosched.c
OSSetEFlag()*	salvoeflag.c
OSSetPrio()	salvoprio.c

```
OSSetPrioTask()           salvotask6.c
OSSetTicks()              salvoticks.c
OSSetTS()                 salvodelay2.c
OSSignalBinSem()*         salvobinsem.c
OSSignalEvent()           salvoevent.c
OSSignalMsg()*            salvomsg.c
OSSignalMsgQ()*           salvomsgq.c
OSSignalSem()*            salvosem.c
OSStartTask()             salvotask.c
OSStop()                  salvostop.c
OSStopTask()              salvotask2.c
OSSyncTS()                salvodelay3.c
OSTaskUsed()              salvotask7.c
OSTaskRunning()           salvotask4.c
OStID()                   salvotid.c
OSTimer()*                salvotimer.c
OSWaitEvent()             salvoevent.c
```

Listing 38: Location of Functions in Source Code

Abbreviations Used by Salvo

The following abbreviations are used throughout the Salvo source code:

```
address                   addr
array                     A
binary                    bin
change                    change, chg
check                     chk
circular                  circ
clear                     clr
create                    create
configuration             config
context                   ctx
current                   curr, c
cyclic timer              cycTmr
delay                     delay
delete                    del
depth                     depth
destroy                   destroy
disable                   dis
disable interrupt(s)      di
ecb pointer               ecbP
eligible                  elig
enable                    en
enable interrupt(s)       ei
enter                     enter
event                     event, e
event control block       ecb
event flag                eFlag
event flag control block  efcb
event type                eType
error                     err
```

from	fm
global	gl
global type	gltype
identifier	ID
include guard	IG
initialize	init
insert	ins
length	len
local	l
location	loc
maximum	max
message	msg
message queue	msgQ
message queue control block	mqcb
minimum	min
not available	NA
number	num
operating system	OS
pointer	ptr, p
pointer to a pointer	pp
prescalar	PS
previous	prev
priority	prio
queue	Q
report	rpt
reset	rst
restore	rstr
return	rtn
save	save
scheduler	sched
semaphore	sem
set	set
signal	signal
stack	stk
status	stat
statistics	stats
string	str
switch	sw
synchronize	sync
task	task, t
task control block	tcb
task function pointer	tFP
tcb extension	tcbExt
tcb pointer	tcbP
tick	tick
timeout	timeout
timer	timer
timestamp	TS
toggle	tgl
utility	util
value	val
version	ver
wait(ing) (for)	wait, w
warning	warn

Listing 39: List of Abbreviations

Chapter 8 • Libraries

Note This chapter provides an overview of using and (re-)building Salvo libraries. Only general issues that affect all of Salvo's libraries are covered here.

For library particulars, please refer to your compiler's *Salvo Compiler Reference Manual*.

Library Types

Salvo ships with two types of precompiled libraries – *standard libraries* and *freeware libraries*. The standard libraries contain all of Salvo's basic functionality, configured for each supported compiler and target processor. The standard libraries are included in their respective Salvo standard distributions. The freeware libraries are identical to the corresponding standard libraries except for the relatively limited numbers of supported tasks and events, and are included in the Salvo Lite distributions.

Salvo Pro users can create applications using the Salvo source files, the standard libraries, or a combination thereof. All other Salvo users must use libraries when creating their applications. For functionality and flexibility greater than that provided by the libraries, you'll need to purchase Salvo for full access to the Salvo source code, and all the configuration options.

Libraries for Different Environments

The various Salvo distributions contain libraries for two different kinds of compilers – *native* and *non-native* compilers.

Native Compilers

By native compilers we mean compilers that generate output (usually in `.hex` format) for a specific embedded target. You would use a native compiler to create a Salvo application for a real product. Native compilers are usually *cross-compilers*, i.e. they run on one machine architecture (usually x86-based PCs) and generate code for another (e.g. TI MSP430).

Non-native Compilers

By non-native compilers we mean compilers that generate code for another target altogether (usually an x86-based PC). Salvo's support for these "pure" compilers[93] is intended to facilitate cross-platform development of Salvo applications for embedded targets. Users can build C console applications and test, run, and debug them on their main development machine (e.g. a PC) before building the same application for the intended embedded target (e.g. a PICmicro MCU). The editing and debugging features available on PCs are powerful tools that can aid in project management, testing and debugging.

If you wish to develop your embedded application on the PC and then recompile your Salvo application for your embedded target, keep in mind that the non-native compilers generally lack any support for non-console-oriented subsystems that may exist on your embedded target. Therefore you will need to simulate things like serial I/O, A/D, D/A, interrupts, etc.

This "build on two, run on one" technique can be quite useful. For example, you could write, test and debug a Salvo application that passes floating-point data between two tasks via a message queue. The PC's enormous[94] resources (stdout buffers, memory, etc.), coupled with a good IDE, present an ideal environment for developing this sort of application. You could debug your application using `printf()` or the IDE's debugger. Once your application works on the PC – and as long as you've used C library functions that are also included in your target compiler's libraries – then building a Salvo application for the embedded target should be a snap!

Using the Libraries

In order to use a Salvo library, place the `OSUSE_LIBRARY` and `OSLIBRARY_XYZ` configuration options particular to your compiler into your `salvocfg.h`. These configuration options ensure that the same configuration options used to generate the chosen library will also be used in your source code.

For example, to use the full-featured standard library for HI-TECH PICC and the PIC16F877A, your `salvocfg.h` file would contain only:

93 As opposed to cross-compilers.

94 When compared to an embedded microcontroller.

```
#define OSUSE_LIBRARY            TRUE
#define OSLIBRARY_TYPE           OSL
#define OSLIBRARY_CONFIG         OSA
#define OSLIBRARY_VARIANT        OSB
```

Listing 40: Example salvocfg.h for Use with Standard Library

and your project would link to the standard library `slp42Cab.lib`.

Please see *Chapter 5 • Configuration* for more information on these configuration options. *Figure 25: Salvo Library Build Overview* illustrates the process of building a Salvo application from a Salvo library.

Note `OSCOMPILER` and `OSTARGET` are not included in the `salvocfg.h` file listed above. That's because in most cases Salvo can automatically detect the compiler in use and then set the target processor accordingly. This is done in the preprocessor via predefined symbols supplied by the compiler.

Overriding Default RAM Settings

Each library is compiled with default values for the number of objects (tasks, events, etc.). By setting configuration parameters in `salvocfg.h` it's possible to increase or decrease the RAM allocated to Salvo, and hence the number of objects in your application.

If the number of objects in your application is smaller than what the library is compiled for, or your application doesn't use certain objects (e.g. message queues) that have their own, dedicated control blocks, you can reduce Salvo's RAM usage. Just add the appropriate configuration options to `salvocfg.h` and rebuild your project.

For example, to set the amount of RAM allocated to tasks in the above example to just two, your `salvocfg.h` file would contain:

```
#define OSUSE_LIBRARY            TRUE
#define OSLIBRARY_TYPE           OSL
#define OSLIBRARY_CONFIG         OSA
#define OSLIBRARY_VARIANT        OSB
#define OSTASKS                  2
```

Listing 41: Example salvocfg.h for Use with Standard Library and Reduced Number of Tasks

and you would link these three files:

```
main.obj, salvomem.obj, slp42Cab.lib
```

to build your application. By adding the following two lines to your `salvocfg.h`:

```
#define OSEVENT_FLAGS            0
#define OSMESSAGE_QUEUES         0
```

Listing 42: Additional Lines in salvocfg.h for Reducing Memory Usage with Salvo Libraries

you can prevent any RAM from being allocated to event flag and message queue control blocks, respectively.

Caution This technique frees RAM for other uses in your application, and must be used with caution. If you reduce `OSTASKS` or `OSEVENTS` from their default values, you must ensure that you do not perform any Salvo services on tasks or events that are now "out of range." E.g. for libraries that support three tasks, if you reduce `OSTASKS` to 2 as outlined above, you must not call `OSCreateTask(TaskName, OSTCBP(3), prio)`. If any of your own variables are located in RAM immediately after the tcbs, they will be overwritten with the call to `OSCreateTask()`.

Setting the number of objects in an application above the library defaults is only possible with the standard libraries – the preset limits in the freeware libraries cannot be overridden.

Note Illegal or incorrect values for the number of objects in an application that uses a library will usually be flagged by the compiler as an error.

Library Functionality

By linking your application to the appropriate library, you can use as few or as many of Salvo's user services as you like. Each library supports up to some number of tasks and events.

Note Because of the enormous number of possible configurations, the standard and freeware libraries support most, but not all, of Salvo's functionality. Each library is compiled with a particular set of configuration options. See the library-specific details (below) or `Pumpkin\Salvo\Inc\salvolib.h` for more information.

Warning Do not edit `Pumpkin\Salvo\Inc\salvolib.h`. Doing so may cause problems when compiling and/or linking your application to the freeware libraries.

Types

The library *type* is specified using the `OSLIBRARY_TYPE` configuration option in `salvocfg.h`.

The library types, shown in Table 11, are self-explanatory.

type code	description
`f / OSF:`	*F*reeware library. Number of tasks, events, etc. is restricted.[95]
`l / OSL:`	Standard *l*ibrary. Number of tasks, events, etc. is limited only by available RAM.

Table 11: Type Codes for Salvo Libraries

Note The standard libraries are slightly smaller than the corresponding freeware libraries.

Memory Models

Where applicable, Salvo libraries are compiled for different memory models. There is no configuration option for specifying the memory model.

Options

Where applicable, Salvo libraries are compiled with different options. There is generally no configuration option for specifying the option.

Global Variables

Salvo uses a variety of objects for internal housekeeping. Where applicable, the `OSLIBRARY_GLOBALS` configuration option in `salvocfg.h` is used to specify the storage type for these global variables. The configuration codes vary by compiler.

95 Most freeware libraries are compiled with `OSSET_LIMITS` set to `TRUE`.

Configurations

The library *configuration* is specified using the `OSLIBRARY_CONFIG` configuration option in `salvocfg.h`.

The library configurations, shown in Table 12, indicate which services are included in the library specified. Use the library that includes the minimum functionality that your application requires. For example, don't use an `a`-series library unless your application requires both delay (e.g. `OS_Delay()`) and event (e.g. `OSSignalSem()`) services.

configuration code	description
`a / OSA:`	Library supports multitasking with delay and event services – *a*ll default functionality is included.
`d / OSD:`	Library supports multitasking with *d*elay services only – event services are not supported.
`e / OSE:`	Library supports multitasking with *e*vent services only – delay services are not supported.
`m / OSM:`	Library supports *m*ultitasking only – delay and event services are not supported.
`s / OSS:`	Library supports only Salvo SE features.
`t / OST:`	Library supports multitasking with delay and event services. Tasks can wait on events with a *t*imeout.
`y / OSY:`	Library supports only Salvo tiny features.

Table 12: Configuration Codes for Salvo Libraries

Note Using a library that's been created with support for services you don't use will have an impact on your application's ROM and RAM requirements.

Table 13 shows the essential differences among the library configurations.

configuration	a	d	e	m	s	t	y
Delay services:	+	+	-	-	+	+	+
Event services:	+	-	+	-	+[96]	+	+[97]
Idling function:	+	+	+	-	+	+	+
Task priorities:	+	+	+	-	+	+	-
Timeouts:	-	-	-	-	-	+	-

Table 13: Features Common to all Salvo Library Configurations

+: enabled
-: disabled

Variants

The library *variant* is specified using the `OSLIBRARY_VARIANT` configuration option in `salvocfg.h`.

A variety of different compilers are certified for use with Salvo. Some compilers use the target processor's stack or registers to pass parameters and store auto variables – this is true for all compilers for x86 targets. *There are no library variants for these conventional compilers.*

Other compilers certified for use with Salvo maintain parameters and auto variables as static objects in dedicated RAM – this is the case for targets that do not have or use general-purpose stacks for parameter and auto variable storage. *The libraries for these compilers have variants.* The remainder of this section applies to the libraries for these compilers.

Some of Salvo's services can be called from within interrupts. Those services include:

96 Binary semaphores, semaphores and messages.
97 Binary semaphores and semaphores.

- • `OSGetPrioTask()`
- • `OSGetStateTask()`
- • `OSReadBinSem()`
- • `OSReadEFlag()`
- • `OSReadMsg()`
- • `OSReadMsgQ()`
- • `OSReadSem()`
- • `OSMsgQEmpty()`
- • `OSSignalBinSem()`
- • `OSSignalMsg()`
- • `OSSignalMsgQ()`
- • `OSSignalSem()`
- • `OSStartTask()`

Listing 43: Partial Listing of Services than can be called from Interrupts

If the target processor does not have a general-purpose stack, the Salvo source code must be properly configured via the appropriate configuration parameters. The library variants, shown in Table 14, are provided for those applications that call these services from within interrupts.

If your application does not call any of the services above from within interrupts, use the *b* variant. If you wish to these services exclusively from within interrupts, use the *f* variant. If you wish to do this from both inside and outside of interrupts, use the *a* variant. In each case, you must call the services that you use from the correct place in your application, or either the linker will generate an error or your application will fail during runtime.

variant code	description
`a / OSA:`	Applicable services can be called from *a*nywhere, i.e. from the foreground and the background, simultaneously.
`b / OSB:`	Applicable services may only be called from the *b*ackground (default).
`e / OSE:`	Applicable services may only be called from *e*ither the foreground or the background, but not both.
`f / OSF:`	Applicable services may only be called from the *f*oreground.
`- / OSNONE:`	Library has no variants.[98]

Table 14: Variant Codes for Salvo Libraries

See the `OSCALL_OSXYZ` configuration parameters for more information on calling Salvo services from interrupts.

Library Reference

Refer to your compiler's *Salvo Compiler Reference Manual* for details on the associated Salvo libraries.

Rebuilding the Libraries

One common reason to rebuild the Salvo libraries occurs when the compiler you are using has been upgraded (new versions, enhancements, bug fixes, etc.) and pre-compiled Salvo libraries built with the new compiler have not yet been released. In a situation like this, you must rebuild the Salvo libraries in order to build your library-build Salvo projects.

Doing source-code builds is generally an easier way to set configuration options for a Salvo project. In multi-user environments, however, it may be wiser to force all Salvo users working on a single application to link to a single, custom library so as to ensure that they are all configured identically.

Note Libraries can only be rebuilt by Salvo Pro users, as the Salvo source code is required.

[98] A library may have no variants if the target processor does not support interrupts, or if the target processor has a conventional stack and the ability to save and restore the state of interrupts.

GNU Make and the bash Shell

The Salvo libraries are generated with GNU `make` in the `bash` shell.[99] If you have Salvo Pro you can rebuild the libraries using the makefiles in the `Pumpkin\Salvo\Src` directory.

Note The Salvo library makefiles are designed to run from the `Pumpkin\Salvo\Src` directory.

In addition to the `make` utility, other utilities commonly used in the `bash` shell are also required for a successful make, including `expr(.exe)`. Refer to your `bash` shell documentation for information on installing the various utilities.

Salvo's makefile system is relatively complex and uses make recursively. Normally, users need not edit the makefiles. However, if you have installed your compiler(s) in places that differ from those specified in the Salvo makefiles, you may need to edit the appropriate makefile for a successful compile.

Rebuilding Salvo Libraries

Linux/Unix Environment

To rebuild a particular library in the `bash` shell, simply specify it as `make`'s target, e.g.

- $: cd /Pumpkin/Salvo/Src
- $: make –f Makefile libsalvolmcc30it.a

Listing 44: Making a Single Salvo Library

The Salvo makefiles also allow for groups of libraries to be made, e.g.

- $: cd /Pumpkin/Salvo/Src
- $: make –f Makefile ra430

Listing 45: Making all Salvo Libraries for a Particular Compiler

to generate all of the Salvo libraries for the Rowley Associates CorssWorks for MSP430 toolset (Salvo code `RA430`), and

[99] Bourne-again shell, a Unix command language interpreter.

- $: cd /Pumpkin/Salvo/Src
- $: make –f Makefile msp430

Listing 46: Making all Salvo Libraries for a Particular Target

to generate all of the Salvo libraries for MSP430 targets. Naturally, you will need all of the compiler(s) associated with the Salvo libraries you're rebuilding.

A list of target groups can be obtained by issuing the commands:

- $: cd /Pumpkin/Salvo/Src
- $: make –f Makefile

Listing 47: Obtaining a List of Library Targets in the Makefile

Multiple Compiler Versions

Some of Salvo's supported compilers are in use at different version levels. For these compilers, the make command-line argument `CVER` must also be specified, e.g.

- $: cd /Pumpkin/Salvo/Src
- $: make –f Makefile iar430 CVER=2

Listing 48: Making Salvo Libraries for IAR's MSP430 C Compiler v2.x

will result in Salvo libraries being built and placed in `\Pumpkin\Salvo\Lib\IAR430-v2`. `CVER` details are compiler-dependent – see the Salvo makefiles for more information.

Note `CVER` can be combined with `CLC` when building custom libraries (see below).

Win32 Environment

To rebuild Salvo libraries in a Win32 environment, you will need a `bash` shell along with GNU `make`. One free source for both is the Cygwin `bash` shell. Another is the MinGW project, along with associated utilities.[100]

[100] A MinGW installation is reported to require only MinGW (e.g. `Mingw-2.0.0-3.exe`) and Msys (e.g. `Msys-1.0.8.exe`), available on http://www.SourceForge.net. MinGW should be installed before Msys.

Currently, all libraries included in Salvo distributions are built in the Cygwin `bash` shell using `make` recursively, as outlined above.[101] Therefore you are strongly encouraged to set up a working Cygwin `bash` shell from the latest Cygwin releases for generating Salvo libraries.

Customizing the Libraries

You can rebuild the Salvo libraries to a configuration that differs from the standard build.[102] This is useful in situations where you prefer to do library builds, and the standard libraries differ somewhat from the configuration that you require.

Using custom libraries is a three-step process, involving:

- • creating a custom library configuration file,
- • building the custom library and
- • using the custom library in a library build

Creating a Custom Library Configuration File

Salvo provides for 20 different user-definable custom library configuration files, `salvoclc1.h` through `salvoclc20.h`.[103] When a custom library is in use, one of these files will be included in the salvo configuration file `Pumpkin\Salvo\Inc\salvolib.h` via the C preprocessor's `#include "filename"` directive.

Note Because of the use of `""` in the `#include` directive, the custom library configuration file must be located in the preprocessor's user search path. It is up to the user to ensure that the preprocessor can find the selected custom library configuration file. A safe location for such files is the `Pumpkin\Salvo\Inc` directory, or the project directory.

Each custom library configuration file includes overrides of Salvo configuration option settings used to generate the library. For each configuration option to be overridden, the Salvo symbol should

101 PCs with large (e.g. 1GB) amounts of RAM are used to avoid the recursive make problems that have plagued Cygwin.

102 Note that Pumpkin cannot provide support for libraries that differ from those provided in the Salvo distributions.

103 Salvo installers do not install any `salvoclcN.h` files. The installers will not replace, overwrite or delete any such user files.

first be `#undef`'d, then `#define`'d, so as to avoid any preprocessor warnings.

Building the Custom Library

Once your custom library configuration file is ready, you rebuild the Salvo library or libraries using the Salvo makefiles and an additional `make` command-line option, `CLC=N`, where `N` is the number of the custom library configuration file you are using.

Note Most users of custom Salvo libraries will only need to override a few of the configuration options for the standard libraries. The library or libraries you choose to rebuild should have a default configuration that is as close as possible to what you are trying to achieve with your custom library.

Using the Custom Library in a Library Build

After you have built your custom library, you must set the `OSCUSTOM_LIBRARY_CONFIG` configuration option in your project's `salvocfg.h` configuration file to the number of your custom library configuration file. And of course you must link to the custom library instead of a standard library.

Example – Custom Library with 16-bit Delays and Non-Zero Prescalar

To build a Salvo library for the Archelon / Quadravox AQ430 Development Tools that has all of the features of an "`ia`" library, but also has 16-bit delays and a timer prescalar of 5, one would start with `slaq430ia.lib`. Assuming this will be custom library configuration 4, create a `Pumpkin\Salvo\Inc\salvoclc4.h` with the following entries:

```
#undef  OSBYTES_OF_DELAYS
#define OSBYTES_OF_DELAYS 2

#undef  OSTIMER_PRESCALAR
#define OSTIMER_PRESCALAR 5
```

Listing 49: Example Custom Library Configuration File salvoclc4.h

and then build the new library:

- $: cd /Pumpkin/Salvo/Src
- $: make –f Makefile libsalvolmcc32l-t.a CLC=4

Listing 50: Making a Custom Salvo Library with Custom Library Configuration 4

Note The `CLC=` command-line argument to `make` is case-sensitive.

Making the custom library as above will result in a new library, `\Pumpkin\Salvo\Lib\MCC32\libsalvolmcc32l-t-clc4.a`.

To use the new library, add `OSCUSTOM_LIBRARY_CONFIG` to your project's `salvocfg.h`, e.g.:

```
#define OSUSE_LIBRARY             TRUE
#define OSLIBRARY_TYPE            OSL
#define OSLIBRARY_CONFIG          OSA
#define OSCUSTOM_LIBRARY_CONFIG   4
```

Listing 51: Example salvocfg.h for Library Build Using Custom Library Configuration 4 and Archelon / Quadravox AQ430 Development Tools

and link your project to your new custom library `\Pumpkin\Salvo\Lib\MCC32\libsalvolmcc32l-t-clc4.a`.

Note In this example, we've only altered the standard library slightly. In general, you should pick a standard library that is as close as possible to the configuration you want in your custom library. Deviating substantially from the standard library's configuration may cause problems when building the library because of conflicts between configuration options. Also, it may result in an unnecessarily large library. Advanced users may want to review `Pumpkin\Salvo\Inc\salvolib.h` to solve such problems using the defined symbols contained therein.

To build a custom library for a particular library and a particular version of the associated compiler, combine the CLC and CVER arguments to the makefile:

- $: cd /Pumpkin/Salvo/Src
- $: make –f Makefile libsalvolra430-t.hza CLC=2 CVER=1

Listing 52: Making a Custom Salvo Library with Custom Library Configuration 4

Making the custom library as above will result in a new library, `\Pumpkin\Salvo\Lib\RA430-v1\libsalvolra430-t-clc2.hza`.

Note To avoid problems associated with different compilers and/or targets, each custom library configuration file `salvoclcN.h` should only be used with a single compiler and target combination.

Preserving a User's salvoclcN.h Files

The Salvo installers will not touch or delete any existing `salvoclcN.h` files. Therefore custom library configuration files can be left in place when Salvo is upgraded.

Restoring the Standard Libraries

The standard Salvo libraries can be restored by either re-installing them from the Salvo installer, or by rebuilding the libraries without any `CLC=` command-line options to `make`. Since the Salvo library makefile system automatically assigns unique, descriptive names to custom libraries, there is no good reason to alter or move the standard libraries.

Custom Libraries for non-Salvo Pro Users

Occasionally, potential Salvo users will request a custom library for evaluation. This will invariably be a custom Salvo Lite (freeware) library. Using a custom Salvo freeware library is no different from using a custom Salvo standard library – just follow the steps outlined above.

Makefile Descriptions

Pumpkin\Salvo\Src\Makefile

This makefile uses a regular expression to parse the name of the desired library or libraries. It then calls make recursively using `Makefile2` to generate one or more libraries.

Pumpkin\Salvo\Src\Makefile2

This makefile references the compiler- and target-specific Makefile in the CODE subdirectory.

Pumpkin\Salvo\Src\CODE\Makefile

This makefile file contains drives the compiler(s) and assembler(s) required to generate the libraries. Compiler-specific paths are located in this file.

Pumpkin\Salvo\Src\CODE\targets.mk

This include file contains the names of all valid Salvo libraries for the selected compiler and target.

Chapter 9 • Performance

Introduction

In this chapter we'll address the runtime aspects of Salvo which affect performance. A good understanding is essential if you wish to extract the maximum possible performance from your target processor.

Interrupts

Salvo controls interrupts in two distinct regions of its code – in the *context switcher*, and in *critical sections*. These two regions of the Salvo code are target- and sometimes compiler-specific, unlike the main body of Salvo code, which is target-independent. These code regions and their impact on your application are discussed below.

Context Switcher

The Salvo context switcher for each compiler and target family is unique. In general terms, the context switcher handles:

- Vectoring from the scheduler to the task
- Generating a local stack frame for the task
- Storing the task's updated resume address in the task's task control block (tcb)
- Any required register save and restores
- Returning from the task to the scheduler

Note All Salvo tasks execute with interrupts enabled. Therefore interrupts are enabled when entering and exiting the Salvo context switcher.

For most Salvo context switchers, the operations listed above involve changes to the stack and stack pointer (SP). Wherever possible, interrupts are not disabled during the operation of the context

switcher. This is possible[104] in most Salvo context switchers, and depends on the target architecture.

Note Most Salvo context switchers are implemented in assembly language and are unaffected by any project optimizations.

Tip Each Salvo Compiler Reference Manual clearly states the interrupt-disabling behavior of the particular context switcher.

In the rare cases where it is not possible to context switch without disabling interrupts, every effort has been made to minimize the number of cycles during which interrupts are disabled. Therefore, for Salvo distributions whose context switcher have non-zero interrupt latencies, the latency represents the maximum interrupt latency due to the Salvo context switcher. Even in these cases, the latency is usually less than 20 instruction cycles.

Note The latency of the Salvo context switcher is constant and is independent of all other aspects of a Salvo application.

Summary

Most Salvo context switcher do not disable interrupts and therefore introduce no interrupt latency into a Salvo application.

Those Salvo context switchers that do disable interrupts do so for the minimum time possible.

Critical Sections

Critical sections of code are sections of code that must not be preempted. In a single-threaded application, preemption occurs through interrupts. If a critical section of code is preempted, then there is a real possibility of corruption of global variables. Since the vast majority of microcontrollers do not have protected memory features, it is imperative that Salvo take steps to prevent preemption during critical sections.

Note Most callable Salvo services include critical sections.

[104] If the Stack Pointer on the target architecture can be changed atomically, then this usually means that interrupts need not be disabled during a Salvo context switch.

Salvo has two user-definable hooks (i.e. functions) that are used to prevent preemption (and therefore corruption of Salvo's own global variables).[105] They are `OSDisableHook()` and `OSEnableHook()`. `OSDisableHook()` is called inside a Salvo service at the beginning of a critical section, and `OSEnableHook()` is called inside a Salvo service at the end of a critical section.

Note Interrupt hooks are contained in every Salvo library. Refer to the appropriate Salvo Compiler Reference Manual for the functionality of the hooks. All Salvo hooks can be overridden by the user, in both source-code builds and library builds.

Inside the Salvo source code, the interrupt hooks are used like this:

```
… // Non-critical section of Salvo code
OSDisableHook();
… // Critical section of Salvo code
OSEnableHook();
… // Non-critical section of Salvo code
OSDisableHook();
… // Critical section of Salvo code
OSEnableHook();
… // etc.
```

Listing 53: Use of interrupt hooks in Salvo source code.

Note Non-dummy (i.e. non-empty) interrupt hooks are target- and sometimes even compiler-specific.[106]

Warning Salvo users cannot change how or when these hooks are called. Their positions in the Salvo code have been chosen to disable interrupts only while required for critical sections. Salvo's critical sections have been coded to be as short as possible.

Effect on Runtime Performance

The runtime length of a Salvo service – and hence the runtime length of a critical section[107] in Salvo's code – can only be obtained

105 An example of one of Salvo's global variables is the pointer to the head of the queue of delayed tasks. If a mainline Salvo service is in the process of making changes to the head of this queue and an interrupt occurs which calls a Salvo service that changes the head of this queue, the result will be unpredictable and will lead to a malfunction of the application. Therefore *all* interrupt-level calls to Salvo services must be suppressed while *any* Salvo service is making any changes to a Salvo global variable.

106 A compiler-specific hook might include the `weak` keyword when the compiler supports this feature.

through measurement in an actual application.[108] Some Salvo services have very short critical sections. Some even have no critical sections. Yet others can potentially have very long critical sections (e.g. when a low-priority task must be enqueued into the eligible queue where several higher-priority tasks are already eligible). To the Salvo user, the main area of concern here is *"How long does Salvo disable my interrupts?",* as this can adversely affect on-board peripherals that are used in an interrupt-driven manner.[109] As you will see below, Salvo *can be configured for zero interrupt latency for any desired interrupt source*.

We will now examine various scenarios for the coding of the interrupt-disabling hooks

Controlling Interrupts Globally

The most general and safest configuration for the user interrupt hooks is for the hooks to disable interrupts globally during a critical section. This is the default for the hooks contained in all Salvo library builds where the target architecture has a single, consistent method of disabling and enabling global interrupts.

```
void OSDisableHook ( void )
{
  __disable_interrupt();
}

void OSEnableHook ( void )
{
  __enable_interrupt();
}
```

Listing 54: Most general configuration for Salvo's interrupt hooks.

The advantage of this approach is that it is safe for *all* application. With the hooks defined as shown in Listing 54, *any* Salvo service can be called from *any* interrupt without fear of corrupting Salvo's global variables. That's why this is the default for all Salvo libraries.

---

107 For a given Salvo service, the runtime length of the critical section contained therein cannot exceed the runtime length of the service itself.

108 This is due in no small part to the wide range of Salvo configuration options and their effect on the runtime performance of the Salvo code. Its is also due to the priority-queue-based priority-resolution algorithms used in Salvo.

109 For example, an interrupt-driven single-byte-buffer asynchronous serial receiver operating at 115200,N,8,1 cannot tolerate its interrupts being disabled for longer than 87µs or it risks losing incoming characters.

The disadvantage of this approach is that *all* interrupt sources are disabled while Salvo is in a critical section, even if said interrupts do not call Salvo services. Clearly, this non-targeted approach to controlling interrupts is not well-suited to high-performance, interrupt-driven Salvo applications, due to the substantially non-zero interrupt latencies imposed on the application.

Controlling Interrupts Individually

For better performance from interrupt-driven peripherals, individual control of interrupts during Salvo's critical sections is recommended. With this approach, *only those interrupt sources which themselves call Salvo services need to be disabled during critical sections*. Since this approach is target-specific, it is best illustrated by example.

```
void OSDisableHook ( void )
{
  IE2     &= ~URXIE1;

  TBCCTL6 &=   ~CCIE;
}


void OSEnableHook ( void )
{
  IE2     |=  URXIE1;

  TBCCTL6 |=    CCIE;
}
```

```
#pragma vector=USART1RX_VECTOR
__interrupt void ISRRx1 (void)
{
  USART_UART1_inchar();
  OSSignalSem(SEM_CMD_CHAR_P);
  __low_power_mode_off_on_exit();
}

#pragma vector=TIMERB1_VECTOR
__interrupt void ISRTimerB1 (void)
{
  switch(__even_in_range(TBIV,14))
  {
    case 0x0C:
      TBCCR6 += TIMER_TICKS_RELOAD;
      OSTimer();
      __low_power_mode_off_on_exit();
      break;

    default:
      fatal(FATAL_ERROR_UNUSED_ISR);
      break;
  }
}
```

Listing 55: Application-specific configuration for Salvo's interrupt hooks. Relevant ISRs also shown. Target is TI's MSP430FG4619.

In the Salvo application associated with the interrupt hooks of Listing 55, two ISRs call Salvo services: `ISRRx1()` calls `OSSignalSem()` when a valid incoming character has been received via USART1 and put into a buffer, and `ISRTimerB1()`[110] calls `OSTimer()` at a period rate. Since these are the only interrupts that calls Salvo services, these are the only interrupt sources that must be disabled during Salvo's critical sections. Therefore we see that `OSDisableHook()` disables USART1 Rx interrupt generation and TimerB6 interrupt generation, and `OSEnableHook()` re-enables the same.

Note In this example it's assumed that interrupts are globally enabled at all times, and are not controlled by Salvo.

The net effect of the hooks in this example is that other interrupt sources *operate with zero interrupt latency* because Salvo does not disable global interrupts or the individual interrupt sources, as there is no need to. Thus, performance is maximized with these other interrupt-driven peripherals.

110 On the MSP430FG4619, the TimerB1 ISR handles interrupts for Timers B1 through B6, based on the Timer B Interrupt Vector (TBIV).

Warning Failure in the interrupt hooks to disable an interrupt source that calls a Salvo service will inevitably lead to runtime problems in a Salvo application due to the unavoidable corruption of global variables. Therefore it's important to keep track of which Salvo services are called from ISRs, and configure the interrupt hooks accordingly.

Tip There is no limit to how many different interrupt sources can be controlled by the interrupt hooks. Just write `OSDisableHook()` and `OSEnableHook()` accordingly.

Avoiding Interrupt Control Altogether

Strange as it may seem, there are Salvo applications that do not require any control of interrupts. They include:

- Salvo applications built on microcontrollers that do not have interrupts (e.g. Microchip PIC12F509).
- Salvo applications that do not use services that are traditionally called from an ISR (like Salvo's timer).
- Salvo applications that cannot tolerate any interrupt latency yet, wish to call one or more Salvo services from an ISR.

In the first two cases above the interrupt hooks need only be redefined as shown in Listing 56.

```
void OSDisableHook ( void )
{
  ;
}

void OSEnableHook ( void )
{
  ;
}
```

Listing 56: Interrupt hooks for applications that do not call Salvo services from any interrupts.

Here, Salvo's critical sections do not involve any change to the interrupt status of the target microcontroller. If the target's Salvo context switcher (see *Context Switcher*, above) has zero interrupt

latency as well, then Salvo's total contribution to overall interrupt latency is zero for all interrupt sources.

In the case where a user wishes to call a Salvo service from an interrupt, yet cannot tolerate any interrupt latency on that interrupt source due to Salvo, then a slightly indirect approach is required.

Tip This situation can arise for example in targets that do not have vectored interrupts, or in targets where a single interrupt vector services several interrupt sources.

In this situation, Salvo's interrupt hooks *do not* disable the source of interrupt that would normally call the Salvo service. Instead, the user must create a semaphore that is used to pass information up from the ISR to the main loop of the Salvo application:

```
int main ( void )
{
  …
  while (1) {
    if (HighPrioISRDataReady == 1) {
      GIEH = 0;
      HighPrioISRDataReady = 0;
      GIEH = 1;
      OSSignalBinSem(HIGH_PRIO_ISR_DATA_READY_P);
    }
    OSSched();
  }
}
```

Listing 57: Passing interrupt activity up from an ISR to call a Salvo service without a corresponding interrupt hook. Target is Microchip PIC18F452.

In Listing 57, a Salvo application built for the Microchip PIC18F452 passes information up from a high-priority ISR[111] to ultimately cause a Salvo binary semaphore to be signaled. It does this simply by setting a semaphore (`HighPrioISRDataReady` in this example) inside the high-priority ISR when event signaling is required. In the application's `main()` loop, this semaphore is tested prior to calling the scheduler and if set, is reset with high-priority interrupts disabled,[112] and finally `OSSignalBinSem()` is called.

111 The PIC18 architecture has just two interrupt vectors – the low-priority interrupt vector and the high-priority interrupt vector. Each vector has its own individual interrupt enable bit (`GIEL` and `GIEH`, respectively).

112 Note that if the semaphore can be set and reset atomically, the control of the `GIEH` bit in this example is unnecessary. It is shown, however, to remind the reader for the general requirement of protecting global variables.

This approach has a very substantial advantage in that the application can run without Salvo's critical sections affecting interrupts. Yet the runtime performance of signaling a Salvo event is virtually indistinguishable from that of an application built with the interrupt disabled (and its attendant non-zero interrupt latency). This is because Salvo's scheduler processes events all at once, and so it makes little difference as to whether an event is signaled at an arbitrary time[113] or immediately before the scheduler is called.

The disadvantages of this approach are:

- Depending on target architecture, the interrupt source may still need to be disabled, albeit for a very short time (just two instruction cycles in the example of Listing 57 above).
- Event processing no longer occurs in the order that the interrupt occurred, but rather in the order that the event is signaled in the user code when the semaphore is found to have been set. This mainly affects multiple tasks waiting on a single event.
- This involves polling the semaphore prior to every invocation of Salvo scheduler. This is contrary to the purely event-driven (i.e. no polling) operation of Salvo.

For most applications, these disadvantages are outweighed by the advantage of near-zero interrupt latency while still effectively calling a Salvo service from an interrupt.

Note Depending on the target architecture, the (albeit short) disabling and re-enabling of interrupts to protect the semaphore (a global variable) as shown in Listing 57 above can be avoided if the semaphore is set (in the ISR) and reset (after the semaphore test in `main()`) *atomically*. In this case, the total interrupt latency remains 0 cycles – highly desirable. Inspection of the assembly code generated by the compiler will prove whether the desired operations are atomic.

Tip Multiple semaphores from multiple interrupt sources can be combined in this approach. Ideally, each semaphore should be implemented as a single-bit-wide bitfield in C, inside of a structure

---

113 When signaling an event from an ISR, the signaling can happen at any time except during a critical section (because said interrupt is disabled during that critical section).

consisting of a union of all the bits (e.g. in an `int`) and of the individual bits. Therefore all the bits can be tested once (is the `int` non-zero?), and if non-zero, the individual bits can be tested and cleared individually. This minimizes the number of instruction cycles spent polling for a change in the semaphores' status, thereby improving runtime performance and minimizing the use of polling (which is undesirable).

Side Effects of Interrupt Hooks

Salvo's interrupt hooks are called from all Salvo services that contain critical sections. This means that many Salvo services that can be called from ISRs *will call the interrupt hooks while in the ISR*, with attendant changes to the interrupt enable bit(s) of the target.

For the default hook for most targets (see Listing 56 above), this means that interrupts will be enabled at the end of the Salvo service that is called in the ISR. Therefore the interrupts controlled by the interrupt hooks will be enabled prior to the end of the ISR. This could lead to nested interrupts where none are desired, etc.

While this is not usually a problem, it can be solved by explicitly flagging being in an ISR and basing the interrupt hook actions on the flag, as shown in Listing 58:

```
static unsigned int InISR = 0;

void entering_isr ( void )
{
  InISR = 1;
}

void leaving_isr ( void )
{
  InISR = 0;
}

void OSDisableHook ( void )
{
  if (InISR == 0) {
    __disable_interrupt();
  }
}

void OSEnableHook ( void )
{
  if (InISR == 0) {
    __enable_interrupt();
  }
}
```

Listing 58: Interrupt hooks to avoid interrupt nesting.

With this method, any ISR that calls Salvo services begins with `entering_isr()` and ends with `leaving_isr()`. This completely avoids nested interrupts. This user flexibility – no need to change any Salvo code here – is the reason for the introduction of hook functions in Salvo 4.

Tip If the compiler or target provides an automatic means of detecting that code is executing at the ISR / foreground level, this can be used to your advantage in the interrupt hooks.

The Fallacy of Avoiding Critical Sections at the Interrupt Level

Some inexperienced programmers might fall for the notion that preempting a critical section can be avoided by testing for a condition inside an ISR's (i.e. in the foreground) code instead of by disabling the ISR in the critical section in mainline (i.e. in the background) code. The idea is to forego all interrupt control in Salvo's critical sections, in favor of setting a flag, which can then be tested in the ISR to avoid call a Salvo service during the critical section. While the test will in fact work, the rest will not, as *there is no way in the ISR to know when the critical section will complete*. And since the critical section does not progress while in the

ISR, there is in fact no way to know when the ISR can call the Salvo service. In effect, the critical section is being blocked by the ISR, which is the opposite of what is desired.

Therefore the prescribed methods above for configuring Salvo's interrupt hooks for critical sections must be followed.

User Hooks

Salvo has four hook services that can be redefined by the user to suit the chosen target and application. They are:

- Interrupt hooks: `OSDisableHook()`, `OSEnableHook()`
- Watchdog hook: `OSClrWDTHook()`
- Idling hook: `OSIdlingHook()`

Tip Since each user hook is defined in its own source code module, state information can be combined with a hook function by declaring a local static variable in the module, and referencing the variable from the hook function(s).

OSDisableHook(), OSEnableHook()

The use of the interrupt hooks is covered above in *Interrupts*.

OSClrWDTHook()

The watchdog hook provides a simple and integrated way to clear an application's watchdog timer from within the Salvo portion of your application. `OSClrWDTHook()` is called each time Salvo's scheduler is called.

Warning `OSClrWDTHook()` is *not* a failsafe means of properly maintaining a software or hardware watchdog. It is provided as a simple scheme that is useful and applicable to many applications, especially in the early stages of their software development. If a more sophisticated approach to watchdog management is required, the user can either override the hook (by defining it as a dummy function), expand the hook (by replacing the default hook with a more sophisticated version), or augmenting the hook with other watchdog-related application code.

```
void OSClrWDTHook ( void )
{
  WDTCTL = (WDTCTL & 0x00FF) | WDTPW | WDTCNTCL;
}
```

Listing 59: Example watchdog hook. Target is TI's MSP430F1612.

In Listing 59 the watchdog hook clears the target's watchdog timer without any other changes.

Chapter 10 • Porting

With its minimal RAM requirements, small code size and high performance, Salvo is an appealing RTOS for use on just about any processor. Even if it hasn't been ported to your processor and/or compiler, you can probably do the port in a day or two.

If you are interested in porting Salvo to a new target processor and/or compiler, please contact Pumpkin for more details. A comprehensive *Salvo Porting Manual* is available.

Chapter 11 • Tips, Tricks and Troubleshooting

Introduction

If you're having trouble getting your code to work properly with Salvo, here are some suggestions on how to solve your problem.

- • Read and re-read all the relevant portions of this manual.
- • Review the example programs in this manual and in the Salvo distribution. You may find something that is very similar to what you are trying to do.
- • Examine the postprocessed output of your compiler, both in C and in assembly language. Output listings contain a wealth of useful information.
- • Examine any map files generated by your compiler. These files have information containing the location of Salvo routines and variables and their sizes, the calling trees, etc.
- • Use the error codes returned by the user services to verify that the desired Salvo actions are really happening.
- • If your application has the RAM and ROM to support it, use `OSRpt()` to examine the status of the system.
- • If you have access to run-time debugging tools, step through the code in question while monitoring important variables.
- • Examine the Salvo source code – it may contain information not presented elsewhere.

Most importantly, examine your assumptions! Don't assume, for example, that a call to `OSStartTask()` is working until you've confirmed that it is in fact returning an error code of `OSNOERR`.

Compile-Time Troubleshooting

I'm just starting, and I'm getting lots of errors.

Be sure to place

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

at the start of each source file that uses Salvo.

My compiler can't find salvo.h.

Make sure that your compiler's include search paths contain the `Pumpkin\Salvo\Inc` directory.

My compiler can't find salvocfg.h.

Each project needs a project-specific `salvocfg.h`. Create one from scratch or copy one from another project. `salvocfg.h` normally resides in your current working directory – you may need to instruct your compiler to explicitly search this directory.

If you are using a Salvo freeware library, copy its `salvocfg.h` to your working directory and edit it as needed.

My compiler can't find certain target-specific header files.

This problem may arise if your compiler has no generic target processor header file that uses defined symbols to include the appropriate target-specific header file. The solution is to include the target-specific header file in your `salvocfg.h`.

My compiler can't locate a particular Salvo service.

You must either include the Salvo files in your project or link to a Salvo library. See your compiler's *Salvo Compiler Reference Manual* for more information.

My compiler has issued an "undefined symbol" error for a context-switching label that I've defined properly.

This may be happening if you have the context-switching label in unreachable code and your compiler has removed the unreachable code through optimization. For example, `OS_Delay()` below is unreachable because of an innocuous error:

```
if (speed = 0) { // Error – should be "=="
  outPWM = 0;
}
else
{
  outPWM = 1;
  OS_Delay(speed);
  …
}
```

and your compiler may be unable to find label as a result. Change your code to make the context switch reachable[114] and the error should disappear.

My compiler is saying something about OSIdlingHook.

The configuration options in your `salvocfg.h` may be set to enable the user hook function, `OSIdlingHook()`. In a source-code build, you must define a function with this name. For example,

```
void OSIdlingHook(void)
{
  ;
}
```

is a null (i.e. "do-nothing") function that satisfies this requirement.

My compiler has no command-line tools. Can I still build a library?

You can build a library without access to a command-line librarian[115] by creating a project with all of the Salvo source files, and setting the output type of your compiler to be a library file. You will also need a special `salvocfg.h` file that looks something like this:

---

114 Use `if ( speed = = 0 )` instead of `if ( speed = 0 )`.

115 CodeWarrior v3.1 has no command-line tools, but can build a library from a project.

```
#define OSUSE_LIBRARY      TRUE

#define OSLIBRARY_TYPE     OSL
#define OSLIBRARY_CONFIG   OST
#define OSLIBRARY_VARIANT  OSNONE

#undef  OSMAKE_LIBRARY
#define OSMAKE_LIBRARY     TRUE
```

This works as follows: when you set `OSUSE_LIBRARY` to `TRUE` in your project's header file `salvocfg.h`, the library header file `salvolib.h` will be included in your project. By defining the library type, configuration and variant symbols `T`, `C` and `V`, respectively, and by setting `OSMAKE_LIBRARY` to `TRUE`, the Salvo source code is configured for library building.

This method is inefficient for building multiple libraries. For that, refer to Salvo's makefiles.

Run-Time Troubleshooting

Nothing's happening.

Did you remember to:

- • Call `OSInit()`?
- • Set `OSCOMPILER`, `OSTARGET` and `OSTASKS` correctly in your `salvocfg.h`?
- • Create at least one task with `OSCreateTask()`?
- • Choose valid task pointers and task priorities that are within the allowed range?
- • Call the Salvo scheduler `OSSched()` from inside an infinite loop?
- • Task-switch inside each task body with a call to `OS_Yield()`, `OS_Delay()`, `OS_WaitXyz()` or another context-switcher?
- • Structure each task with its body in an infinite loop?

If you've done all these things and your application still doesn't appear to work, you may have a configuration problem (e.g. parts of your `salvocfg.h` do not match those used to create the freeware library you're using) or an altogether different problem.

Also, make sure that you've done a full recompile ("re-make"), and, if you're using some sort of integrated development environment, be sure that you've downloaded your latest compiled code and reset the processor before running the new code.

It only works if I single-step through my program.

This is usually indicative of a problem with interrupts or the watchdog timer. Since both are usually disabled when single-stepping with an in-circuit emulator (ICE) or in-circuit debugger (ICD), your application may work in this mode but not in run mode.

If your application uses interrupts, be sure that any *interrupt flags* are cleared before leaving the ISR. When interrupt sources share the same interrupt vector, failing to clear the interrupt flag will result in an endless loop of interrupt services. In general, vectored interrupts do not have interrupt flags associated with them.

Many target processors enable the watchdog timer by default. If you fail to reset it regularly, your application will appear to be constantly resetting itself. Depending on the watchdog timer's timeout period, this may be a very short (e.g. < 1s) period. Either disable the watchdog timer or use Salvo's `OSCLEAR_WATCHDOG_TIMER()` configuration option.

Note All Salvo projects in the distributions are compiled with `OSCLEAR_WATCHDOG_TIMER()` defined to reset the watchdog timer. This way, even if you forget to disable the watchdog timer[116] in your development environment, the application should still work.

It still doesn't work. How should I begin debugging?

If you have the ability to set breakpoints, a quick way to verify that your application is multitasking is to re-load your executable (e.g. hex) code, place breakpoints at the entry of each task, reset the processor, and Run. If you have successfully initialized Salvo and created tasks (check the error return codes for `OSInit()` and `OSCreateTask()`), the first call to `OSSched()` should eventually result in the processor halting at one of those breakpoints.

116 In the Microchip development tools family, the PICMASTER and the MPLAB-ICE disable the watchdog timer by default, but the MPLAB-ICD enables it by default.

If your application makes it this far, Salvo's internals are probably working correctly, and your problem may have to do with improper task structure and/or use of Salvo's context-switching services. Improper control of interrupts and incorrectly-written interrupt service routines (ISRs) are also a common problem.

If you do not have hardware debugging support, use simple methods (like turning an LED on or off from within a task) to trace a path through your program's execution. On small, embedded systems, "`printf`-style debugging" may not be a viable option, or may introduce other errors (like stack overflow) that will only frustrate your attempts to get at the root of the problem.

My program's behavior still doesn't make any sense.

You may be experiencing unintended interaction with your processor's watchdog timer. This can occur if you've compiled your application with the target processor's default (programmable) configuration, which may enable the watchdog timer. You can avoid this problem by using the `OSCLEAR_WATCHDOG_TIMER()` configuration option in your `salvocfg.h` configuration file. By defining this configuration option to be your target processor's watchdog-clearing instruction, the Salvo scheduler will clear the watchdog each time it's called, and prevent watchdog timeouts.

Compiler Issues

Where can I get a free C compiler?

Borland's C++ compilers can be had for free at:

- http://www.borland.com/bcppbuilder/freecompiler/

They can be used to create 16- and 32-bit PC (x86) applications.

HI-TECH software also offers free C compilers:

- http://www.htsoft.com/
-

Pacific C can be used to create PC (x86) applications, and PICC Lite can be used on the Microchip PIC16C84 family.

Where can I get a free make utility?

You can download the GNU make utility's source code from

- http://www.gnu.org/order/ftp.html

A precompiled DOS/Win32 version is available at

- ftp://ftp.simtel.net/pub/simtelnet/gnu/djgpp/v2gnu/

Look for the `mak*.zip` files. This is a full-featured, UNIX-like make that works well in the Win32 environment.

Where can I get a Linux/Unix-like shell for my Windows PC?

You can download the Cygwin bash shell from RedHat at

- http://sources.redhat.com/cygwin/

A full installation will contain GNU make and many other utilities. It works best on Windows NT / 2000 / XP systems. If you have the Salvo Pro, this shell can be used to generate all of Salvo's libraries on a Windows PC.

My compiler behaves strangely when I'm compiling from the DOS command line, e.g. "This program has performed an illegal operation and will be terminated."

The DOS command line is limited to a maximum of 126 characters. If you invoke your compiler with a longer command line, you may experience very unpredictable results. The solution is to reorganize your project. Consult your compiler's user's manual for more information.

Another possibility is that the environment size on your Windows/DOS PC is inadequate for the DOS program(s) you are running. If you run more than one DOS window under Windows and the environment size is marginal, you may also encounter this problem. You can fix this by adding the shell command to your config.sys file, e.g.:

```
shell = c:\windows\command.com /p /e:nnnnn
```

where nnnnn is the size of the environment, in bytes, from 160 to 32768. The default is 256. See your DOS manual for more information on the DOS command interpreter and the shell command.

My compiler is issuing redeclaration errors when I compile my program with Salvo's source files.

If you create your application by compiling and then linking your files and Salvo's source files all at once, be sure that none of your source files have the same name as any Salvo source file.

HI-TECH PICC Compiler

Salvo has been thoroughly tested with PICC and it is unlikely that you will encounter any problems that are due directly to compiling and linking the Salvo code to your application. However, since it is often difficult to pinpoint the exact cause of a compile-and-link error, you should follow the tips below if you encounter difficulties.

Running HPDPIC under Windows 2000 Pro

Some people like to run HPDPIC[117] in an 80x50 "DOS window" under Windows. Do the following:

- • start HPDPIC
- • right-click on the menu bar and select Properties
- • select Layout
- • choose a Window Size of Width:80 and Height:50
- • select OK, choose "Save properties for future windows with same title", select OK
- • exit HPDPIC (alt-Q)
- • restart HPDPIC

You may want to choose a different font or font size (under Properties → Font) that is better suited to a larger DOS window. If you are having problems with your mouse, instead of changing the window size settings in the procedure above, deselect the QuickEdit mode under Properties → Options.

---

117 The HI-TECH Integrated Development Environment (IDE) for PICC.

Setting PICC Error/Warning Format under Windows 2000 Pro

In Windows 2000 Pro, do either:

- My Computer → Properties → Advanced → Environment Variables ...
-

or

- Start → Settings → Control Panel → System → Advanced → Environment Variables ...
-

then in User Variables for *Userid* do:

- New → Variable, enter HTC_ERR_FORMAT , OK,
  Variable Value, enter Error[ ] %f %l : %s , OK

and

- New → Variable, enter HTC_WARN_FORMAT , OK, Variable Value, enter Warning[ ] %f %l : %s , OK

Then log off and log back on for these changes to take effect. You can see that they are in force by running the MS-DOS Prompt (`C:\WINNT\system32\command.com`) and entering the `SET` command. Type `EXIT` to leave the MS-DOS command prompt.

Note that you must log off and log back on for these changes to take effect. If you change the environment variables without logging off and back on, MPLAB may behave strangely, like do nothing when you click on the error/warning message.

Linker reports fixup errors

If the PICC linker is unable to place variables in RAM, it will report fixup errors. Interpreting these errors can be very difficult. You must successfully place all variables in RAM before attempting to interpret any other PICC link errors. If you're having difficulty, the simplest thing is to place all of Salvo's variables in an unused bank (e.g. Bank 3 on a PIC16C77). Then, by using PICC's bank directives you can move your own variables around until they all fit. A thorough understanding of the bank directives is required, especially when banked (or unbanked) pointers to banked (or un-

banked) objects are involved. Consult the PICC manual for more information, or the Salvo source code for examples of using the bank directives.

See also "Placing Variables in RAM", below.

Placing variables in RAM

Because PICs have generally very little RAM, as your application grows it's likely that you will need to explicitly manage where variables are located in RAM. If your Salvo application has more than a few tasks and events, it's likely that you will want to place the Salvo data structures (e.g. tcbs and ecbs) and other variables in a RAM memory bank other than Bank 0, the default bank for auto variables and parameters. To do this, use the `OSLOC_Xyz` configuration options and recompile your code. The `OSLOC_Xyz` configuration words options not all be the same – for example you can place ecbs in Bank 2, and tcbs in Bank 3.

If you need to use more than one bank to place Salvo's variables in RAM, for best performance place them in bank pairs – e.g. in Banks 2 and 3 only.

Note Your Salvo code will be smallest if you place all of your Salvo variables in Bank 1 and/or Bank 0. PICC places all auto variables in Bank 0. Bank switching is minimized by placing Salvo's variables in the same bank as the auto variables.

Link errors when working with libraries

If you get the following error:

```
HLINK.EXE::Can't open (error): : No such file or
directory
```

while working with multiple projects and libraries, it may go away be simply re-making the project.

Avoiding absolute file pathnames

Use HPDPIC's Abs/Rel path feature when adding source and include files to your project. You'll be able to enter path names much more quickly.

Compiled code doesn't work

Make sure you're using the latest version of PICC, including any patches that are available. Check http://www.htsoft.com for version updates.

PIC17CXXX pointer passing bugs

On the 17C756, in certain cases PICC failed to correctly dereference pointers passed as parameters. This affected Salvo's queueing routines.

Note This was fixed in PICC v7.84.

While() statements and context switches

You may encounter a subtle problem if you use a `while()` statement immediately following a Salvo context switch, e.g.

```
...
OS_Delay(5);
while (rxCount) {
...
```

if rxCount is a banked variable, after optimization the compiler may fail to set the register page bits properly when accessing the variable. This will probably lead to incorrect results. A simple workaround is to add the line

```
rxCount = rxCount;
```

between the context switch and the `while()` statement. This will "force" the proper RP bits.

Note This was fixed in PICC v7.85.

Library generation in HPDPIC

If you are using HPDPIC projects to compile libraries for use with PIC processors with different numbers of ROM and RAM banks (e.g. PIC16C61 and PIC16C77), you may encounter an error when linking your application(s) to one of those libraries. This is because the PICC preprocessor `CPP.EXE` may be fed the wrong processor-selection argument if you're switching between projects with different processors.

The solution is to first load a project whose output is a `.COD` file, and then load a second project destined for the same type of processor and whose output is a library. Make the library (i.e. make the second project), then re-load the first project, and make it, linking to the previously generated library. By loading the first project you correctly set the processor type for the second project.

Note This was fixed in PICC v7.86.

Problems banking Salvo variables on 12-bit devices

On the 12-bit devices (e.g. PIC16C57), Salvo applications don't work when Salvo variables are placed in a RAM bank other than Bank 0. The solution is to upgrade to the latest version of the compiler.

Note This was fixed in PICC v7.86PL4.

Working with Salvo messages

Salvo messages are passed via void pointers. Use the predefined type definition (`typedef`) `OStypeMsgP` when declaring pointers to messages. This type is defined by default as `void *`. In PICC a pointer to a void object points only to RAM. That's fine if your Salvo application has only messages in RAM. But what if you want to send messages which point to objects in ROM (e.g. a string like "STOP' or "GO") as well as RAM? By changing `OSMESSAGE_TYPE` to `const` messages can now point to objects in RAM or ROM. This may add 1 extra byte to the size of each event control block (ecb).

Note `OSMESSAGE_TYPE` must be set to `const` in your `salvocfg.h` if you are using messages and/or message queues and you are accessing message data that's in ROM.

See also *Working with Message Pointers* in this chapter.

Adding OSTimer() to an Interrupt Service Routine

If you are linking to a freeware or custom Salvo library, or if `timer.c` is one of the nodes in your project, and you call `OSTimer()` from within an interrupt routine, PICC automatically

assumes the worst case with regard to register usage within `OSTimer()` and the functions it may call, and automatically adds a large number of register save and restores to your interrupt routine. This makes it large and slow, which is undesirable.

The solution is to change the organization of your source files. Instead of compiling `timer.c` into a linkable object module, *include it in your source file* which contains the call to `OSTimer()`. For example, your `main.c` might now look like this:

```
…
#include "timer.c"

void interrupt intVector( void )
{
  /* handle various interrupts         */
  ...

 /* this happens every 10ms.           */
  if (TMR1IF) {
    /* must clear TMR2 interrupt flag. */
    TMR1IF = 0;

    /* reload TMR1 while it's stopped. */
    TMR1ON = 0;
    TMR1 -= TMR1_RELOAD;
    TMR1ON = 1;

    OSTimer();
  }
}
```

By including `timer.c` in the same source code file as the interrupt routine, PICC is able to deduce exactly which temporary registers must be saved when the interrupt occurs and restored thereafter, instead of assuming the worst case and saving and restoring all of them. The resultant savings in code space and improvement in interrupt execution speed are substantial. If your application uses the Salvo timer, this reorganization is highly recommended.

After including `timer.c` in your interrupt source code file, you may want to recompile your custom Salvo library if you are using one. The Salvo functions will still be able to reference the required queueing functions – they've simply moved from the library to your object modules.

Note You may need to add the switch `-IPumpkin\Salvo\Src` to PICC's command line in order for the compiler and linker to find the `timer.c` source file.

Using the interrupt_level pragma

Whenever you call any Salvo services from both inside an interrupt and from background code (e.g. from within a task), you must insert the following PICC directive prior to your interrupt routine:

```
#pragma interrupt_level 0
```

This alerts the PICC compiler to look for multiple call graphs of functions called from both mainline and interrupt code. This is necessary in order to preserve parameters and auto variables.

Note Placing this PICC pragma before an interrupt routine has no deleterious effects even when multiple call graphs are not generated. Therefore it's recommended that you always do this if you call any functions from within your interrupt routine.

HI-TECH V8C Compiler

Note Support for the V8C compiler has been discontinued as of 2005.

The initial Salvo port to the VAutomation V8-µRISC™ requires an updated V8 assembler, `ht-v8\bin\asv8.exe`, dated 6-21-2001 or later, along with v7.84 of the compiler. Many of the test programs (e.g. `\salvo\test\t41\sysl`) use `printf()` for run-time output for use with the simulators.

Note Since the HI-TECH V8C compiler and its HPDV8 IDE are substantially similar in operation to HI-TECH's PICC compilers and HPDPIC IDE, refer to *HI-TECH PICC Compiler*, above, for related information.

Simulators

Two simulators for the V8-µRISC™ are available – one from HI-TECH (`simv8.exe`) and one from VAutomation (`v8sim.exe`). Salvo applications run on both.

HI-TECH 8051C Compiler

Problems with static initialization and small and medium memory models.

When using the small or medium memory models, the compiler issues the error `Can't generate code for this expression` when faced with the declaration

```
unsigned int counter = 0;
```

This occurs because initialized objects are in ROM for these models, and therefore cannot be changed. The solution is to either declare the variable as near, or explicitly initialize it elsewhere in your code.

IAR PICC Compiler

Target-specific header files

The IAR PICC compiler requires a target-specific header file that contains symbols and addresses for the PICmicro special function registers (SFRs). These files are located in the `inc` subdirectory of the compiler's distribution, and are target-specific.

For example, `\iar\ew23\picmicro\inc\io17c756.h` is the header file for the 17C756 PICmicro. By placing

```
#include "io17C756.h"
```

in your source files, the compiler will be able to correctly resolve certain symbols used throughout the Salvo source code.

Interrupts

The vector for each interrupt must be properly defined. Use the compiler's vector pragma like this:

```
#pragma vector=0x10
__interrupt void intVector(void)
{
  T0IF = 0;
  TMR0 -= TMR0_RELOAD;
  OSTimer();
```

```
}
```

This will place the TMR0 interrupt vector at 0x10 on a PIC17C756.

Mix Power C Compiler

In contrast to usual IBM C call stack programming, which has positive offsets from BP for function arguments and negative offsets from BP for local variables, the Power C compiler uses positive offsets from BP to access both local variables and function arguments. This affects the Salvo context switcher for Power C to the degree that it will only function correctly as long as the call stack for the task is in its simplest form. The key to compiling Salvo applications to run on the PC is to guarantee that each task has the simplest possible Power C entry call stack.

Strict adherence to the Salvo requirement that only static local variables be used in a task is required to avoid run-time errors. Additionally, there are a few other innocuous things ("gotchas") that the Power C programmer might do which violate Salvo's requirement that the call stack remain in its simplest form. Those that are known are outlined below.

Required compile options

When compiling Salvo source code, using the following compile options for `PC.EXE`:

```
/r-
/2
/mm
```

Failure to use these options or to use other incompatible options may prevent your Salvo executable from running properly.

Below is an example line from a makefile:

```
PCopts = /c /o /w /r- /2 /mm /id:Pumpkin\Salvo\Inc
```

Application crashes after adding long C source lines to a Salvo task

If you have source code (e.g. a function with multiple parameters) within a task that is too long to fit on a single line, you must use

the '\' character to continue on the next line, even if it's not necessary for a successful compile. This is because Mix Power C changes the task's entry call stack to one that is incompatible with Salvo's context switcher if the line is not continued with the '\' character. For example, the call to `DispLCD()` below

```
void TaskMsg ( void )
{
  while (1) {
    ...
    DispLCD((char *) ((t_dispMsg *)msgP)->strTop,
      (char *) ((t_dispMsg *)msgP)->strBot);
    OS_Delay((OStypeDelay)
      ((t_dispMsg *)msgP)->delay);
    ...
  }
}
```

will compile successfully, but it will cause the PC application to crash when it runs `TaskMsg()`. By adding the '\' character to the DispLCD() line. e.g.

```
DispLCD((char *) ((t_dispMsg *)msgP)->strTop, \
  (char *) ((t_dispMsg *)msgP)->strBot);
```

the problem is resolved.

Application crashes after adding complex expressions to a Salvo task

Mix Power C changes the task's entry call stack if the expressions in a task exceed a certain level of complexity. For example, placing either

```
char = RxQ[rxHead++];
```

or

```
(dummy = dummy);
```

inside a task will cause problems, whereas replacing them with

```
char = RxQ[rxHead];
rxHead++;
```

and

```
dummy = dummy;
```

will not.

Application crashes when compiling with /t option

Mix Power C changes the task's call entry stack when trace information for the debugger is enabled via the compiler's /t option. This change is incompatible with Salvo's context switcher for Power C. Source code modules which contain Salvo tasks must not be compiled with the /t option.

One way around this problem is to move functionality that does not involve context switching out of the module the task is in and into a separate source code module, and call it as an external function from within the task. A module that does not contain any Salvo tasks can be compiled with the /t option, and hence debugged using Mix Power Ctrace debugger.

Compiler crashes when using a make system

Make absolutely sure that your DOS command line does not exceed 127 characters in length. If it does, the results can be very unpredictable. Simplify your directory structure to minimize pathname lengths when invoking any of the Mix Power C executables (e.g. `PCL.EXE`).

Metrowerks CodeWarrior Compiler

Compiler has a fatal internal error when compiling your source code

Ensure that you do no use duplicate labels in any single source code file. This may occur unintentionally if you duplicate labels for Salvo context-switching macros inside a single function. For example,

```
void Task1( void )
{
  ...
  OS_Delay(1);
  ...
}

void TaskB( void )
{
  ...
```

```
  OS_Delay(1);
  ...
  OS_Yield();
  ...
}
```

may cause a CodeWarrior exception because of the duplicate label a in `Task2()`, whereas

```
void Task1( void )
{
  ...
  OS_Delay(1);
  ...
}

void Task2( void )
{
  ...
  OS_Delay(1);
  ...
  OS_Yield();
  ...
}
```

may not.

Microchip MPLAB

The Stack window shows nested interrupts

The MPLAB Stack window cannot differentiate between an interrupt and an indirect function call. Because Salvo makes extensive use of indirect function calls, you may be seeing a combination of return addresses associated with interrupts and indirect function call return addresses.

Controlling the Size of your Application

The Salvo source code is contained in several files and is comprised of a large body of functions. Your application is unlikely to use them all. If you compile and link the Salvo source files along with your application's source files to form an executable program, you may inadvertently end up with many unneeded Salvo functions in your application. This may prevent you from fitting your application into the ROM of your target processor.

The solution is to compile the Salvo source files separately, and combine them into a single library. You can then link your application to this library in order to resolve all the external Salvo references. Your compiler should extract only those functions that your application actually uses in creating your executable application, thus minimizing its size.

You must always recreate the Salvo library in its entirety whenever you change any of its configuration options.

Refer to your compiler's documentation on how to create libraries from source files, and how to link to those libraries when creating an executable.

See *Chapter 4 • Tutorial* for more information on compiling your Salvo application.

Working with Message Pointers

If you want to use messages as a means of intertask communications, you'll have to be comfortable using Salvo message pointers. Salvo provides predefined type definitions (C `typedefs`) for working with message pointers. The following message pointer declarations are equivalent:

```
OStypeMsg * messagePointer;
```

and

```
OStypeMsgP messagePointer;
```

but you should always use the latter to declare local or global message pointer variables, both static and auto.

In general, Salvo message pointers are of type `void *`. However, you should use the predefined types to avoid problems when a void pointer is not correct for a message pointer. This occurs mainly with processors that have banked RAM.

When passing an object that is not already a message pointer, you'll need to typecast the object to a message pointer in order to avoid a compiler error. The following two calls to `OSSignalMsg()` are equivalent:

```
OSSignalMsg(MSG1_P, (OStypeMsg *) 1);
```

and

```
OSSignalMsg(MSG1_P, (OStypeMsgP) 1);
```

The typecast above is required because 1 is a constant, not a message pointer. Here are some more examples of passing objects that are not message pointers:

```
char letter = 'c';
OSSignalMsg(MSG_CHAR_VAR_P, (OStypeMsgP) &letter);

const char CARET = '^';
OSSignalMsg(MSG_CHAR_CONST_P, (OStypeMsgP)
&CARET);

unsigned int * ptr;
OSSignalMsg(MSG_UINT_P, (OStypeMsgP) ptr);

void Function(void);
OSSignalMsg(MSG_FN_P, (OStypeMsgP) Function);
```

Once an object has been successfully passed via a message, you will probably want to extract the object from the message via `OS_WaitMsg()`.[118] When a task successfully waits a message, Salvo copies the message pointer to a local message pointer (`msgP` below) of type `OStypeMsgP`. To use the contents of the message, you'll need to properly typecast and dereference it. For the examples above, we have:

```
char:            * (char *) msgP

const char:      * (const char *) msgP

unsigned int *:    (unsigned int *) msgP

void * (void):     (void * (void)) msgP
```

Failing to properly typecast an object (e.g. using `(char *)` instead of `(const char *)` when dereferencing a constant) will have unpredictable results. Please see *Salvo Application Note* **Error! Reference source not found.** for more information on dereferencing pointers.

NOTE When working with message pointers, it's very important to ensure that Salvo's message pointer type `OStypeMsgP` is properly configured for the kinds of messages you wish to use. On most

---

[118] An exception occurs when you are not interested in the contents of the message, but only that it has arrived.

targets, the default configuration of `void *` will suffice … but there are some exceptions.

For example, the HI-TECH PICC compiler requires 16 bits for `const char` pointers, but only 8 bits for `char` pointers. Therefore the Salvo code (whether in a library or in a source-code build) must be configured to handle these larger pointers or else you will encounter runtime errors.

Appendix A • Recommended Reading

Salvo Publications

A variety of additional Salvo publications are available to aid you in using Salvo. They include Application Notes, Where applicable, some are included in certain Salvo distributions. Application Notes, Assembly Guides, Compiler Reference Manuals, Conference proceedings & presentations, and others. They are all available online at http://www.pumpkininc.com.

Learning C

K&R

Kernighan, Brian W., and Ritchie, Dennis M., *The C Programming Language*, Prentice-Hall, New Jersey, 1978, ISBN 0-13-110163-3.

Of Interest This book is the definitive, original reference for the C programming language.

C, A Reference Manual

Harbison, Samuel P. and Steele, Guy L., Jr., *C, A Reference Manual*, Prentice-Hall, NJ, 1995, ISBN 0-13-326224-3.

Of Interest A modern C language reference.

Power C

Mix Software, *Power C, The High-Performance C Compiler*, 1993.

Of Interest Mix Power C is a very inexpensive, full-featured ANSI-compatible C compiler for use on the PC. Its excellent 600+-page manual contains comprehensive tutorial and reference sections. Library source code is available.

Real-time Kernels

µC/OS & MicroC/OS-II

Labrosse, Jean J., *µC/OS, The Real-Time Kernel*, R&D Publications, Lawrence, Kansas, 1992, ISBN 0-87930-444-8.

Labrosse, Jean J., *MicroC/OS-II, The Real-Time Kernel*, R&D Books, Lawrence, Kansas, 1999, ISBN 0-87930-543-6.

Of Interest This book and its greatly expanded and well-illustrated successor provide an excellent guide to understanding RTOS internals. It also demonstrates how even a relatively simple conventional RTOS requires vastly more memory than Salvo. Its task and event management is array-based. Source code is included.

CTask

Wagner, Thomas, *CTask, A Multitasking Kernel for C*, public domain software, version 2.2, 1990, available for download on the Internet.

Of Interest The author of this well-documented kernel takes a very hands-on approach to describing its internal workings. CTask is geared primarily towards use on the PC. As such, it is *not* a real-time kernel. Its task and event management is primarily queue-based. Source code is included.

Embedded Programming

Labrosse, Jean J., *Embedded Systems Building Blocks*, R&D Publications, Lawrence, Kansas, 1995, ISBN 0-13-359779-2.

Of Interest This book provides canned routines in C for a variety of operations (e.g. keypad scanning, serial communications and LCD drivers) commonly encountered in embedded systems programming. RTOS- and non-RTOS-based approaches are covered. The author also provides an excellent bibliography. Source code is included.

LaVerne, David, *C in Embedded Systems and the Microcontroller World*, National Semiconductor Application Note 587, March 1989, http://www.national.com.

Of Interest The author's comments on the virtues of C programming in embedded systems are no less valid today than they were in 1989.

RTOS Issues

Priority Inversions

Kalinsky, David, "Mutexes Prevent Priority Inversions," *Embedded Systems Programming*, Vol. 11 No. 8, August 1998, pp.76-81.

Of Interest An interesting way of solving the priority inversion problem.

Microcontrollers

PIC16

Microchip, *Microchip PIC16C6X Data Sheet*, Section 13.5, Interrupts, 1996.

Of Interest A special method for disabling the global interrupt bit GIE is required on the PIC16C61/62/64/65. Set `OSPIC16_GIE_BUG` to `TRUE` when using these and certain other processors. The later versions (e.g. PIC16C65A) do not require this

fix. Below is a response from Microchip to a customer query on this issue:

The GIE issue is not a 'bug' in the part it relates more to an operational consideration when the GIE bit is handled in software to disable the interrupt system and the fact that during execution of that operation it is possible for an interrupt to occur. The nature of the MCU core operation means that whilst the current instruction is flowing through the device an asynchronous interrupt can occur. The result of this is that the processor will vector to the ISR disable GIE, handle the Interrupt and then enable GIE again. The result of this is of course that the instruction to disable GIE has been overridden by the processor vectoring to the interrupt and disabling then enabling the interrupt. This is a very real possibility and AN576 is explaining a method to ensure that, in the specific instance where you wish to disable GIE in software during normal execution that your operation has not been negated by the very action you wish to stop.

The app note is related to the disabling of GIE in software. The disabling and re-enabling of GIE when an interrupt occurs is performed in hardware by the processor and the execution of the RETFIE instruction. The GIE check is a safeguard to ensure your expected/desired operation has occurred and your program can then operate as expected/desired without the unexpected occurrence of an interrupt. This issue remains on the current range of parts since it is related to the operation of the core when the user wishes to take control of the interrupt system again.

BestRegards,

UK Techhelp

Appendix B • Other Resources

Web Links to Other Resources

Here are some web sites for information and products related to Salvo and its use:

- • http://www.atmel.com/ – Atmel Corporation, supplier of 8051 architecture and AVR 8-bit RISC microcontrollers
-
- • http://www.circuitcellar.com/, "The magazine for Computer Applications," – lots of information on computer and embedded computer programming
-
- • http://www.cygnal.com/ – Cygnal Integrated Products, supplier of advanced in-system programmable, mixed-signal System-on-Chip products
-
- • http://www.embedded.com/ – Home of *Embedded Systems Programming* magazine
-
- • http://www.gnu.org/ – The Free Software Foundations GNU[119] project web server
-
- • http://www.htsoft.com/ – HI-TECH Software LLC, home of the PICC, PICC Lite, PICC-18 and V8C compilers.
-
- • http://www.iar.com/ – IAR Systems, makers of embedded computing tools including C compilers, Embedded Workbench IDE and C-SPY debugger
-

119 GNU is a recursive acronym for ``GNU's Not Unix"; it is pronounced "guh-NEW".

- • http://www.imagecraft.com/ – ImageCraft, makers of ANSI C tools combined with a modern GUI development environment
-
- • http://www.keil.com/ – Keil Software, makers of C compilers, macro assemblers, real-time kernels, debuggers, simulators, integrated environments, and evaluation boards for the 8051
-
- • http://www.metrowerks.com/ – Metrowerks Corporation, home of the CodeWarrior compiler and integrated development environment
-
- • http://www.microchip.com/ – Microchip Corporation, supplier of PIC microcontrollers
-
- • http://www.mixsoftware.com/ – Mix Software, Inc., home of the Power C compiler
-
- • http://www.motorola.com/ – Motorola, Inc., makers of M68HCxx single-chip microcontrollers and providers of the Metrowerks CodeWarrior IDE
-
- • http://www.mixsoftware.com/ – Mix Software, Inc., home of the Power C compiler
-
- • http://www.quadravox.com/ – Quadravox, Inc., makers the AQ430 Development Tools for TI's MSP430 line of ultra-low-power microcontrollers
-
- • http://www.redhat.com/ – Provider of a well-known Linux distribution, and also home of the Cygwin[120] project.
-
- • http://www.rowley.co.uk.com/ – Rowley Associates, makers development tools for TI's MSP430
-
- • http://www.ti.com/ – Texas Instruments, makers of the TMS320C family of DSPs as well

120 Search site for "Cygwin".

as the MSP430 line of ultra-low-power microcontrollers

-
- • http://www.vautomation.com/ – VAutomation, Inc., home of the V8-µRISC™ synthesizeable 8-bit core

Appendix C • File and Program Descriptions

Overview

Each Salvo distribution contains a variety of files in different formats. Most (e.g. Salvo libraries and project files) are intended for use with a particular set of tools and on a particular target, although some – e.g. the Salvo source code – are often universal.

Each distribution has an organized file hierarchy. Directories (i.e. folders) include subdirectories (i.e. subfolders), etc. Files that are higher up in a particular directory tree are more general, and those towards the bottom are more specific for a particular target, compiler and / or Salvo distribution.

If you have installed only one Salvo distribution on your PC, it will contain files for just your compiler and / or target processor. If you have installed multiple Salvo distributions, all of their files will be distributed in subdirectories below the root Salvo directory.

Online File Locations

Salvo Distributions

Unless otherwise noted, each complete Salvo distribution is distributed as a Windows self-extracting executable,[121] and is available online exclusively at Pumpkin's website http://www.pumpkininc.com for download. Salvo Lite is freely downloadable. Salvo LE and Salvo Pro are only available to those Salvo customers who have purchased the corresponding Salvo license(s).

[121] In Salvo v3 and earlier, Salvo's installer was built using MindVision's Installer VISE product. As of Salvo 4, Salvo installers are built using the NSIS system.

Local/User File Locations

By default, Salvo is always installed in the `\Pumpkin` directory of the user's hard disk. Therefore Salvo's various installation directories are `\Pumpkin\Salvo`, `\Pumpkin\Salvo\Doc`, `\Pumpkin\Salvo\Inc`, etc.

Note Due to a variety of problems that may be encountered[122] if installing Salvo to the Windows `Program Files` directory (due to the space character in said directory's name), installation to a root directory other than `\Pumpkin\Salvo` is *not* recommended.

Salvo Uninstaller(s)

The Salvo uninstaller(s) are located in the `Salvo` directory. There is an uninstaller for each unique Salvo distribution installed on the user PC.

Salvo Documentation

Salvo documents – when included in a Salvo distribution – are located in `Salvo\Doc`.

Note Not all Salvo documents are included in every Salvo distribution. For example, the Salvo User Manual is not included, due to its size. An alias file with a link to the on-line (and therefore most up-to-date) version of the Salvo User Manual is included in each Salvo distribution.

Salvo Header Files

Salvo's header files are located in the `Salvo\Inc` directory. All Salvo header files (`*.h`) are in written in C.

Salvo Source Files

Salvo's source files located in the `Salvo\Src` directory. Most of Salvo's source files (`*.c`) are in written in C. The remaining source files (`*.asm`, `*.s`, etc.) are written in target- and com-

[122] E.g. problems for Salvo Pro users with Salvo makefiles. Also, certain compilers cannot properly handle spaces in e.g. the names of include paths.

piler-specific assembly language, and are located in designated subdirectories.

Salvo Libraries

Salvo's target- and compiler-specific libraries (`*.lib`, `*.a`, etc.) are located in the `Salvo\Lib` directory. Where compiler versions impact the format of libraries, there may be multiple directories of libraries for a particular series of compilers.

Salvo Applications

Depending on the particular distribution, a Salvo installation may include applications related to Salvo's tutorials, examples, test code or other applications. They are located in the `Salvo\Example` directory, and are normally in the form of *projects* (see *Projects*, below) for the associated software toolset.

Salvo Graphics Files

The Salvo installers require some graphics files. These are located in `Salvo\Gfx`.

Other Pumpkin Products

Salvo is just one of Pumpkin's software products involving. Other Pumpkin products will usually be installed alongside Salvo under the `\Pumpkin` directory.

Target and Compiler Abbreviations

Salvo employs a shorthand notation when referring to files that are specific to a particular target and compiler combination. These abbreviations are usually a combination of an abbreviation of the toolset vendor name and of the target's name. The implied compiler and target are usually self-explanatory.

Projects

Nomenclature

Nearly all Salvo applications are built using projects.[123] Usually the project type is the one native to the tool being used, e.g. Microchip MPLAB projects (`*.mcp`) or Keil µVision2 (`*.uV2`) projects.

Programs can be built using Salvo libraries or Salvo source code. Projects follow the naming convention shown below:

- *projectname*`lite.*`: uses Salvo Lite (freeware) libraries
- *projectname*`le.*`: uses Salvo LE or Pro (standard) libraries
- *projectname*`pro.*`: uses source code from a Salvo Pro distribution
- *projectname*`pro-lib.*`: uses Salvo LE or Pro (standard) libraries with embedded debugging information

Each project has a single, unique `salvocfg.h` configuration file associated with it.

Wherever possible, relative pathnames have been used for maximum installation flexibility.

In general, projects designed for a particular target and compiler system can be easily modified to work with other, similar target processors. For example, a project for the NXP LPC2129 ARM7TDMI-based MCU could be rebuilt for the NXP LPC2106 with minor changes.

---

123 Some applications may be built via simple makefiles and via the command line.

Project Files

The source files for a project are generally unique to a project, though they may be substantially similar to those of similar projects. The only target-specific code contained in a project's source code is unique to the intended target.[124]

Wherever possible, the projects used to generate the applications have organized the project's files into abstracts, help files, source files and libraries that are unique to the project; Salvo help files, configuration file, source files and libraries; and other files (e.g. map files, hex files, etc.).

Additionally, where several projects are grouped together (e.g. the Salvo Lite, LE and Pro versions of the tutorial project `tut5`), files that are common to all of the projects are located in the parent directory of the project files.

[124] This methodology differs substantially from that used in Salvo v3 and earlier. In the earlier projects, target-specific code for the intended target was enabled via the preprocessor, and all other target-specific code was ignored. This created substantial confusion among Salvo users, to the point where it was deemed detrimental to overall comprehension of the Salvo applications.

Index

D

E

F

H

I

K

L

M

O

P

R

S

T

U

V

W

Y

Notes