PUMPKIN™
REAL-TIME SOFTWARE
750 Naples Street • San Francisco, CA 94112 • (415) 584-6360 • http://www.pumpkininc.com

RM-MCC18
Reference Manual

Salvo Compiler Reference Manual – Microchip MPLAB-C18

Salvo™
The RTOS that runs in tiny places.™

Introduction

This manual is intended for Salvo users who are targeting Microchip (http://www.microchip.com/) PIC18 PICmicro® MCUs with Microchip's (http://www.microchip.com/) MPLAB-C18 C compiler.

Related Documents

The following Salvo documents should be used in conjunction with this manual when building Salvo applications with Microchip's MPLAB-C18 C compiler:

Salvo User Manual
Application Note AN-12 (obsolete)
Application Note AN-25

Note MPLAB-C18 users are strongly advised to upgrade to Microchip's MPLAB IDE v6.30 or later. Use *AN-25* in place of *AN-12*.

Example Projects

Example Salvo projects for use with Microchip's MPLAB-C18 C compiler and the Microchip MPLAB IDEs v5 and v6 can be found in the:

```
\salvo\ex\ex1\syse
\salvo\tut\tu1\syse
\salvo\tut\tu2\syse
\salvo\tut\tu3\syse
\salvo\tut\tu4\syse
\salvo\tut\tu5\syse
\salvo\tut\tu6\syse
```

directories of every Salvo for Microchip PICmicro® MCUs distribution.

Features

Table 1 illustrates important features of Salvo's port to Microchip's MPLAB-C18 C compiler.

general	
available distributions	Salvo Lite, LE & Pro for Microchip PICmicro® MCUs
supported targets	PIC18 PICmicro® MCUs
header file(s)	`portmcc.h`
other target-specific file(s)	`portpic18.c`
project subdirectory name(s)	SYSE
salvocfg.h	
compiler auto-detected?	yes[1]
libraries	
`\salvo\lib` subdirectory	`mcc18`
default storage class	`auto`
context switching	
method	via `OSCtxSw(label)`
`_OSLabel()` required?	no
size of auto variables and function parameters in tasks	unrestricted
memory	
memory models supported	`small` and `large`
stack models supported	`single-bank` and `multi-bank`
interrupts	
controlled via	`GIEL` and/or `GIEH` bits. Controlled via `OSPIC18_INTERRUPT_MASK` configuration option
interrupt status preserved in critical sections?	yes
method used	relevant `GIE` bits are saved to software stack on entry, interrupts are disabled, and relevant `GIE` bits are restored from software stack on exit
nesting limit	unlimited[2]
alternate methods possible?	yes[3]
debugging	
source-level debugging?	only in source-code builds
compiler	
bitfield packing support?	no
printf() / %p support?	no / no
va_arg() support?	yes[4]

Table 1: Features of Salvo Port to Microchip's MPLAB-C18 C Compiler

Libraries

Nomenclature

The Salvo libraries for Microchip's MPLAB-C18 C compiler follow the naming convention shown in Figure 1.

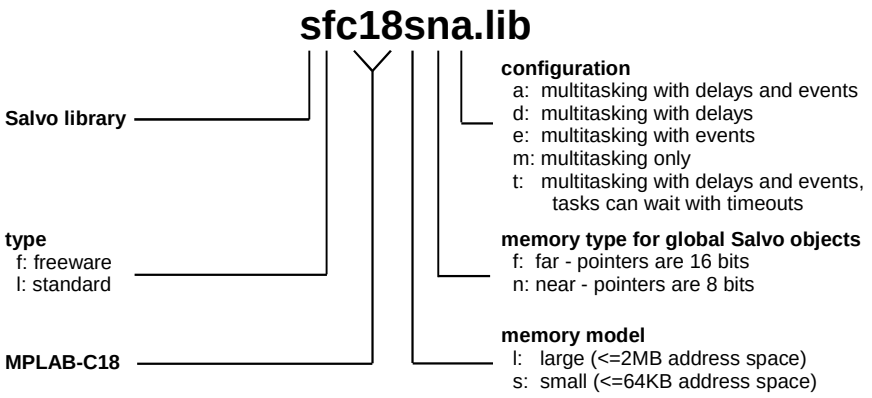

Figure 1: Salvo Library Nomenclature – Microchip's MPLAB-C18 C Compiler

Type

Salvo Lite distributions contain *freeware* libraries. All other Salvo distributions contain *standard* libraries. See the *Libraries* chapter of the *Salvo User Manual* for more information on library types.

Target

The PIC18's core architecture[5] is consistent throughout the entire family. Every Salvo library compiled under the small memory model can be used with any member of the PIC18 family, provided that the size does not exceed 32K (64KB). Salvo libraries compiled under the large memory model can be used with any member of the PIC18 family.

Memory Model

The Microchip MPLAB-C18 C compiler's `small` and `large` memory models are supported. In library builds, the memory model applied to all of the source files must match that used in the library. For source-code builds, the same memory model must be applied to all of the source files.

memory model code	description
`l / OSL:`	*L*arge memory model. Program space is a maximum of 1M words (2MB).
`s / OSS:`	*S*mall memory model. Program space is a maximum of 32K words (64KB).

Table 2: Memory Models for Salvo Libraries – Microchip's MPLAB-C18 C Compiler

Note Unlike the library configuration and variant options specified in the `salvocfg.h` file for a library build, none is specified for the selected memory model. Therefore particular attention must be paid to the memory model settings used to build an application. The memory model is usually specified on a node-by-node basis inside an IDE (e.g. MPLAB).

Memory Type for Global Salvo Objects

You can choose the memory type for Salvo's global objects in your application by choosing the appropriate library. `near` type objects can be accessed the fastest, but consume precious RAM in the Access Bank. `far` type objects will be placed in banked RAM, which will result in slower accesses. The global object codes are listed in Table 3.

memory type code	description
`f / OSF:`	Salvo objects are declared as type `far`, and will be located in banked RAM.
`n / OSN:`	Salvo objects are declared as type `near`, and will be located in the first 128 bytes of internal RAM (i.e. in access RAM).

Table 3: Memory Types for Salvo Libraries – Microchip's MPLAB-C18 C Compiler

The code required to access Salvo's global objects (e.g. the task control blocks, or `tcbs`) will vary in size and speed depending on where the objects are located.

Since there are only 128 bytes of access RAM in the PIC18 architecture, in larger applications it may be necessary to place Salvo's global objects in banked RAM.

Configuration

Different library configurations are provided for different Salvo distributions and to enable the user to minimize the Salvo kernel's footprint. See the *Libraries* chapter of the *Salvo User Manual* for more information on library configurations.

Build Settings

Salvo's libraries for Microchip's MPLAB-C18 C compiler are built using the default settings outlined in the *Libraries* chapter of the *Salvo User Manual*. Target-specific settings and overrides are listed in Table 4.

compiled limits	
max. number of tasks	3
max. number of events	5
max. number of event flags[6]	1
max. number of message queues[7]	1
target-specific settings	
delay sizes	8 bits
idling hook	enabled
interrupt-enable bits during critical sections	`GIEH` = `GIEL = 0`
message pointers	can point to ROM or RAM
Salvo objects	far
system tick counter	available, 32 bits
task priorities	enabled
watchdog timer	cleared in `OSSched()`.

Table 4: Build Settings and Overrides for Salvo Libraries for Microchip's MPLAB-C18 C Compiler

Note The compiled limits for tasks, events, etc. in Salvo libraries can be overridden to be less (all Salvo distributions) or more (all Salvo distributions except Salvo Lite) than the library default. See the *Libraries* chapter of the *Salvo User Manual* for more information.

Available Libraries

There are 40 Salvo libraries for Microchip's MPLAB-C18 C compiler. Each Salvo for Microchip PICmicro® MCUs distribution contains the Salvo libraries of the lesser distributions beneath it.

salvocfg.h Examples

Below are examples of `salvocfg.h` project configuration files for different Salvo for PICmicro® MCUs distributions targeting the PIC18C452.

Salvo Lite Library Build

```
#define OSUSE_LIBRARY           TRUE
#define OSLIBRARY_TYPE          OSF
#define OSLIBRARY_GLOBALS       OSF
#define OSLIBRARY_CONFIG        OSA
#define OSTASKS                 2
#define OSEVENTS                4
#define OSEVENT_FLAGS           0
#define OSMESSAGE_QUEUES        1
```

Listing 1: Example salvocfg.h for Library Build Using sfc18lfa.lib

Salvo LE & Pro Library Build

```
#define OSUSE_LIBRARY           TRUE
#define OSLIBRARY_TYPE          OSL
#define OSLIBRARY_GLOBALS       OSF
#define OSLIBRARY_CONFIG        OSA
#define OSTASKS                 7
#define OSEVENTS                13
#define OSEVENT_FLAGS           3
#define OSMESSAGE_QUEUES        2
```

Listing 2: Example salvocfg.h for Library Build Using slc18lfa.lib

Salvo Pro Source-Code Build

```
#define OSENABLE_IDLING_HOOK    TRUE
#define OSENABLE_SEMAPHORES     TRUE
#define OSTASKS                 9
#define OSEVENTS                17
#define OSEVENT_FLAGS           2
#define OSMESSAGE_QUEUES        4
```

Listing 3: Example salvocfg.h for Source-Code Build

Performance

Memory Usage

tutorial memory usage[8]	total ROM[9]	total RAM[10]
`tu1lite`	644	34
`tu2lite`	976	36
`tu3lite`	1026	38
`tu4lite`	1914	47
`tu5lite`	2708	59
`tu6lite`	2988	62
`tu6pro`[11]	2770	58

Table 5: ROM and RAM requirements for Salvo Applications built with Microchip's MPLAB-C18 C Compiler

Special Considerations

Variables Inside Tasks

Due to architectural limitations of the Salvo context switcher for Microchip's MPLAB C18 C compiler, *auto variables inside tasks are not supported and will lead to various runtime problems*. Therefore all variables inside tasks should be declared as `static` when using Microchip's MPLAB C18 C compiler to build a Salvo application.

Incompatible Optimizations

The Microchip MPLAB-C18 C compiler's *procedural abstraction* optimization is incompatible with Salvo tasks due to its potential effect on the Salvo context switcher `OSCtxSw()`. Therefore *this optimization must be explicitly disabled* (`-Opa-`) in any source file that contains Salvo tasks.[12]

Note In cases where the effects of procedural optimization are otherwise beneficial to an application, it is recommended that source code (`*.c`) modules be dedicated to holding Salvo tasks and no other functions. Thus, the scope of disabling the procedural abstraction optimizations can be limited to just Salvo tasks, thereby allowing the use of this optimization on other functions where it is appropriate.

Storage Classes

Microchip's MPLAB-C18 C compiler supports three different default global storage classes for local variables and function parameters: `auto`, `static` and `overlay`.[13] The storage class for all modules in a project must match in order to avoid link-time errors.

All of Salvo's libraries are built with the `auto` default storage class. Should you wish to build a Salvo application with a different default storage class, you'll need Salvo Pro to either do a source-code build with the alternate default storage class in force, or to create a custom Salvo library with the alternate default storage class.[14]

The different storage classes are set at compile time via the `-sca`, `-scs` and `-sco` command-line options.

Stack Issues

For architectural reasons, Microchip's MPLAB-C18 C compiler passes parameters on a software stack, and uses the PIC18's hardware stack for `call…return` addresses. While the compiler supports both reentrant and static overlay models, Salvo's implementation for this compiler is compatible only with the reentrant model.

Stack Size

By default, Salvo is configured to work with MPLAB-C18's software stack of any size, i.e. the single-bank and multi-bank (`-Ls-`) stack models. If the single-bank model is used (MPLAB-C18's default), then Salvo's `OSMPLAB_C18_STACK_SIZE` can be set to 256 or less and a minor savings in Salvo code size can be realized.

Warning If MPLAB-C18's multi-bank stack model is chosen with Salvo's `OSMPLAB_C18_STACK_SIZE` set to 256 bytes or less, errors will occur.

Salvo's Global Objects

Locating Objects in Near/Access or Banked RAM

With Microchip's MPLAB-C18 C compiler, Salvo's global objects can be located *en masse* in one of two areas – either in banked RAM or in access RAM. With `OSMPLAB_C18_LOC_ALL_NEAR` set to `FALSE` (the default), all of Salvo's objects are placed in banked RAM. To locate all of the objects in access RAM in a Salvo Pro source-code build, set `OSMPLAB_C18_LOC_ALL_NEAR` to `TRUE`.

Note Because of the small size of the PIC18's access RAM, locating Salvo's global objects in access RAM is rarely appropriate.

There are no provisions for uniquely placing selected Salvo global objects in access or banked RAM. Salvo's `OSLOC_XYZ` configuration parameters do not apply.

Limits on the Numbers of Tasks, Events, etc.

By default, each of Salvo's larger global objects (e.g. the array of task control blocks, or `tcbs`) is placed in its own *data section*.[15] This enables the use of the largest possible array size (256 bytes) supported by the compiler, and enables the compiler to efficiently pack all of Salvo's global objects into available RAM.

Note Because of this 256-byte limit on the size of arrays, the numbers of Salvo tasks, events, message queues, etc, is limited not by the amount of total RAM on the PIC18 processor, but by the number of elements that will fit within a 256-byte array (i.e. within one bank).

For example, the maximum value for `OSTASKS` is 42 in a Salvo configuration that supports multitasking and events, because 42 tasks require an array of `0xFC` (252) bytes.[16] The same application can also concurrently support up to 51 events.

Interrupt Service Routines

By default, Microchip's MPLAB-C18 C compiler preserves basic context in ISRs. To properly declare an ISR that calls a Salvo service, the `save=` clause of the `interrupt` pragma should be used thusly:

```
#pragma interrupt ISR save=PROD,section(".tmpdata")

void ISR( void )
{
  …
  OSTimer();
  …
}
```

Note Additional functions in the ISR may require additional arguments to the `save=` clause. See the *MPLAB-C18 User's Guide* for more information.

Interrupt Control

The PIC18 architecture supports two distinct priority levels. When enabled, two separate global-interrupt-enable bits, `GIEH` and `GIEL`, are used to control high- and low-priority interrupts, respectively.

Interrupts are automatically disabled within Salvo's critical sections. By default, both `GIEH` and `GIEL` are reset (i.e. made `0`) during critical sections. This is controlled by Salvo's `OSPIC18_INTERRUPT_MASK` configuration option (default value: `0xC0`).

Salvo Pro users can reconfigure the way in which interrupts are disabled during critical sections by redefining `OSPIC18_INTERRUPT_MASK` in the project's `salvocfg.h`. For example, *if Salvo services* (e.g. `OSTimer()`) *are called only from low-priority interrupts*, then a value of `0x40` for `OSPIC18_INTERRUPT_MASK` ensures that only low-priority interrupts are disabled during a Salvo critical section. In this configuration, high-priority interrupts will therefore be unaffected by Salvo. This is especially useful when high-rate interrupts are present.

Note Salvo Pro users have the option of building *custom libraries* with interrupt disabling and re-enabling controlled by non-default values for `OSPIC18_INTERRUPT_MASK`. See the *Salvo User Manual* for more information.

[1] This is done automatically through the `__18CXX` symbol defined by the compiler.

[2] Though the PIC18 architecture has a hardware `call…return` stack depth limit of 32 levels.

[3] Via either in-line assembly or a function call.

[4] As of MPLAB-C18 v2.20.

[5] I.e. the non-peripheral SFRs, like `TOSU|H|L`, `STKPTR`, `PCLATH|L`, `PCL`, `FSR0H|L`, etc.

[6] Each event flag has RAM allocated to its own event flag control block.

[7] Each message queue has RAM allocated to its own message queue control block.

[8] Salvo v3.2.1 with MPLAB-C18 v2.2x.

[9] In program addresses (words).

[10] In bytes, all banks, `udata`. Does not include stack (default: 0x100 bytes). Salvo global objects are in banked RAM (`far`).

[11] Salvo Pro build differs slightly from Salvo Lite build due to configuration – see tutorial's `salvocfg.h`.

[12] By default, all of MPLAB-C18's optimizations are enabled.

[13] `overlay` applies only to local variables, not function parameters.

[14] Note that this may require changes to Salvo's makefile system, since an additional command-line argument will need to be passed to the compiler.

[15] As of Salvo v3.2.4.

[16] E.g. with an `e`-configuration library.