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

RM-MCC30
Reference Manual

Salvo Compiler Reference Manual – Microchip MPLAB C30

Salvo™
The RTOS that runs in tiny places.™

created by Andrew E. Kalman on Aug 21, 2007 updated on Aug 2, 2013

Introduction

This manual is intended for Salvo users who are targeting Microchip's 16-bit PIC24® and dsPIC® single-chip microcontrollers with Microchip's (http://www.microchip.com/) MPLAB C30 C compiler.

As of Salvo 4.3.0, Salvo supports both the original (e.g., dsPIC33) and Enhanced Core (e.g., PIC24E) parts in this family of 16-bit MCUs and DSCs.

Related Documents

The following Salvo documents should be used in conjunction with this manual when building Salvo applications with Microchip's MPLAB C30 C compiler:

- *Salvo User Manual*

Example Projects

Example Salvo projects for use with Microchip's MPLAB C30 C compiler can be found in the:

```
\Pumpkin\Salvo\Example\PIC\PIC24
```

directories of every Salvo for Microchip PIC24® MCUs and dsPIC® DSCs distribution.

Features

Table 1 illustrates important features of Salvo's port to Microchip's MPLAB C30 C compiler.

General	
Abbreviated as	MCC30
Available distributions	Salvo Lite, LE & Pro for Microchip 16-bit MCUs and dsPIC® DSCs
Supported targets	all PIC24, PIC24E, dsPIC30F, dsPIC33 & dsPIC33E devices
Header file(s)	`salvoportmcc30.h`
Other target-specific file(s)	`salvoportmcc30-sm.s`, `salvoportmcc30-lm.s`, `salvohook_interrupt_PIC24_IRQ.c`
salvocfg.h	
Compiler auto-detected?	yes[1]
Include target-specific header file in `salvocfg.h`?	recommended
Libraries	
Located in	`Lib\MCC30-v3` (for v3.x compilers and later linker)
Behavior of user hooks in libraries	do nothing (dummy functions)
Same libraries for PIC24 & dsPIC families?	yes
Format	COFF
Context Switching	
Method	function-based via `OSDispatch()` & `OSCtxSw()`
Labels required?	no
Size of auto variables and function parameters in tasks	total size must not exceed 65,535 8-bit bytes
Interrupts	
Interrupt latency in context switcher	0 cycles
Interrupts in critical sections controlled via	`OSDisableHook()`, `OSEnableHook()`, `OSRestoreHook()`, `OSSaveHook()`
Interrupt status preserved in critical sections?	optional, via appropriate user functions
Method used in critical sections	see example user functions
Memory	
Memory models supported	`small` and `large`
Debugging	
Source-level debugging with Pro library builds?	yes
Compiler	
Bitfield packing support?	no
printf() / %p support?	yes / yes
va_arg() support?	yes

Table 1: Features of Salvo port to Microchip's MPLAB C30 C compiler

Libraries

Nomenclature

The Salvo libraries for Microchip's MPLAB C30 C compiler C compiler follow the naming convention shown in Figure 1.

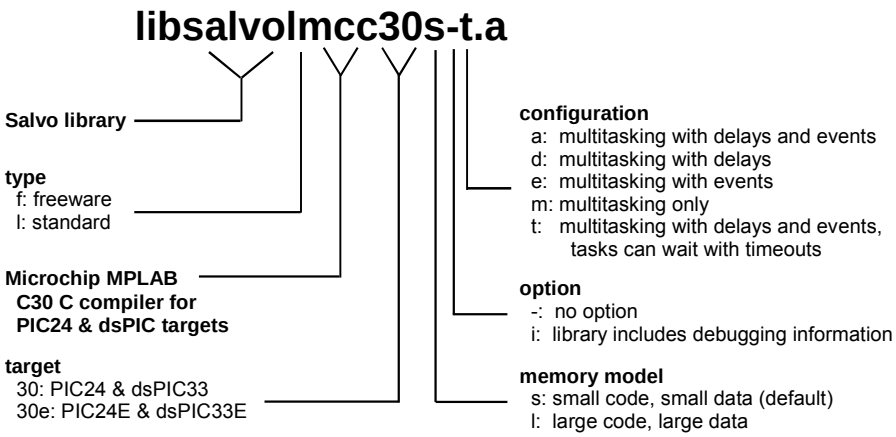

Figure 1: Salvo library nomenclature – Microchip's MPLAB C30 C compiler

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.

Note The libraries are supplied in COFF format. Salvo Pro users can rebuild the libraries in ELF format by editing the Salvo library makefiles.

Target

The standard and enhanced 16-bit MCUs and DSCs require different (and unique) libraries.

Note Attempting to build a project with a mismatched library (e.g., when building for the PIC24EP256MC206 and using the Salvo library `libsalvolmcc30sit.a`) may result in an inability to link or in runtime errors.

Memory Model

The MPLAB C30 C compiler's `small` (default) and `large` code and data memory models are supported:

Code Models	
small	up to 32K words program memory
large	over 32K words of program memory

Table 2: Code models for Microchip's MPLAB C30 C compiler

Data Models	
small	up to 8KB of data memory
large	over 8KB of data memory

Table 3: Data models for Microchip's MPLAB C30 C compiler

Note Salvo libraries for Microchip's MPLAB C30 C compiler C compiler are compiled with either *small code and small data models*, or *large code and large data models*.

If additional flexibility is required, a Salvo Pro user can build an application with a different combination of memory models, e.g., the small code and the large data models by using Salvo source code.

See Configuring for Different Memory Models for information on properly selecting the different memory models.

Option

Salvo Pro users can select between two sets of libraries – standard libraries, and standard libraries incorporating source-level debugging information. The latter have been built with the appropriate command-line options. This adds source-level debugging information to the libraries, making them ideal for source-level debugging and stepping in the MPLAB IDE. To use these libraries, simply select one that includes the debugging information (e.g. `libsalvolmcc30lit.a`) instead of one without (e.g. `libsalvolmcc30l-t.a`) in your MPLAB project.

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 C30 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	4
Max. number of events	8
Max. number of event flags	1
Max. number of message queues	1
Target-specific Settings	
Delay sizes	8 bits
Idling hook	enabled
Interrupt-enable bits during critical sections	controlled via user functions
System tick counter	available, 32 bits
Task priorities	enabled
Watchdog timer	controlled via user functions

Table 4: Build settings and overrides for Salvo libraries for Microchip's MPLAB C30 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

Salvo Lite for Microchip PIC24® MCUs and dsPIC® DSCs contains a single freeware library in a single configuration. Salvo LE for Microchip PIC24® MCUs and dsPIC® DSCs adds standard libraries in multiple configurations. Salvo Pro for Microchip PIC24® MCUs and dsPIC® DSCs adds standard libraries in multiple configurations with debugging information included.

Each Salvo for Microchip PIC24® MCUs and dsPIC® DSCs distribution contains the Salvo libraries of the lesser distributions beneath it. Additionally, Salvo Pro distributions contain makefiles for all possible library configurations.

Target-Specific Salvo Source Files

Depending on the desired code model, Table 5 illustrates that different target-specific source files are required for Salvo Pro source-code builds.

Target-specific Source Files	
small	`Src\MCC30\salvoportmcc30-sm.s`
large	`Src\MCC30\salvoportmcc30-lm.s`

Table 5: Target-specific files required for different code models of Microchip's MPLAB C30 C compiler

Note These files are independent of the `OSMCC30_LARGE_CM` symbol.

salvocfg.h Examples

Below are examples of `salvocfg.h` project configuration files for various different Salvo distributions and the PIC24HJ256GP610.

Salvo Lite Library Build

```
#define OSUSE_LIBRARY           TRUE
#define OSLIBRARY_TYPE          OSF
#define OSLIBRARY_CONFIG        OST
#define OSTASKS                 3
#define OSEVENTS                4
#define OSEVENT_FLAGS           0
#define OSMESSAGE_QUEUES        1
```

Listing 1: Example salvocfg.h for library build using libsalvofmcc30s-t.a

Salvo LE & Pro Library Build

```
#define OSUSE_LIBRARY           TRUE
#define OSLIBRARY_TYPE          OSL
#define OSLIBRARY_CONFIG        OSA
#define OSTASKS                 7
#define OSEVENTS                11
#define OSEVENT_FLAGS           0
#define OSMESSAGE_QUEUES        4
```

Listing 2: Example salvocfg.h for library build using libsalvolmcc30s-a.a or libsalvolmcc30sia.a

Salvo Pro Source-Code Build

```
#define OSEVENTS                    9
#define OSEVENT_FLAGS               1
#define OSMESSAGE_QUEUES            2
#define OSTASKS                     17

#define OSENABLE_BINARY_SEMAPHORES  TRUE
#define OSENABLE_TIMEOUTS           TRUE
#define OSBYTES_OF_DELAYS           4
#define OSBYTES_OF_TICKS            4
```

Listing 3: Example salvocfg.h for source-code build

Performance

Interrupt Latencies

Since Salvo's context switcher for Microchip's MPLAB C30 C compiler does not need to control interrupts, Salvo applications can easily be created with zero total interrupt latency for interrupts of interest.

In a properly-configured application, only those interrupts that call Salvo services will experience interrupt latency from Salvo's operations. Users must ensure that these interrupt sources are disabled (and re-enabled) via the user interrupt hooks.

Disabling and re-enabling interrupts globally in the user interrupt hooks (i.e., the default user interrupt hook behavior) is of course permitted, but will result in non-zero interrupt latencies for all interrupt sources, even those that do not call Salvo services. See the target-specific source files of this distribution for examples.

See also User Hooks, below.

Memory Usage

Examples of the total memory usage of actual Salvo-based applications are listed below.

Example Application[2]	Program Memory Usage[3]	Data Memory Usage[4]
`tut5lite (for PIC24)`	3813	84
`tut5le (for PIC24)`	3747	84
`tut5pro (for PIC24)`	3651	82

Table 6: Program and data memory requirements for Salvo applications built with Microchip's MPLAB C30 C compiler

Special Considerations

Configuring for Different Memory Models

When building a Salvo application with MPLAB C30, the memory models of all objects linked together to form an application must be consistent. The memory models are specified in the MPLAB IDE under Project → Build Options → Project → MPLAB C30 → Memory Model.[5]

In library builds, the memory models applied to all of the source files must match that used in the library – a mismatch may generate link-time errors and or runtime errors. For source-code builds, the same memory models must be applied to all of the source files.

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(s). Therefore particular attention must be paid to the memory model settings used to build an application. The memory model is usually specified on a project-wide basis in the MPLAB IDE.

Code Models

Use of the large code model requires that the Salvo symbol OSMCC30_LARGE_CM be defined for all Salvo code modules.[6] Symbols can be defined in the MPLAB IDE under Project → Build Options → Project→ MPLAB C30 → General → Preprocessor Macros.

Note OSMCC30_LARGE_CM should *not* be defined when using the small code model.

See also Target-Specific Salvo Source Files, above.

Data Models

No symbols are required for the small or large data models.

User Hooks

Overriding Default Hooks

In library builds, users can define new hook functions in their projects and the linker will choose the user function(s) over the default function(s) contained in the Salvo library.

In source-code builds, users can remove the default hook file(s) from the project and substitute their own hook functions.

Idling

The default idling hook in `salvohook_idle.c` is a dummy function, as shown below.

```
void OSIdlingHook(void)
{
  ;
}
```

Listing 4: Default Salvo idling hook for Microchip's MPLAB C30 C compiler

Users can replace it (e.g. with a directive to put the PIC24 to sleep) by building their own version with their application.

Interrupt

The default interrupt hooks in `salvohook_interrupt_MCC30_IRQ.c` are shown below.[7]

```
void OSDisableHook(void)
{
  __asm__ volatile("disi #0x3FFF");
}

void OSEnableHook(void)
{
  __asm__ volatile("disi #0x0000");
}
```

Listing 5: Default Salvo interrupt hooks for Microchip's MPLAB C30 C compiler

These functions represent the simplest way to disable global interrupts across Salvo's critical section, and then restore them thereafter. Interrupts are *not* re-enabled inside of ISRs with these functions, thereby avoiding unnecessary or unwanted interrupt

nesting. Therefore these default interrupt hooks are suitable for *all* applications.

Note The `DISI` instruction is rather unusual in the embedded microcontroller world, in that it disables interrupts for a maximum of 16,384 (i.e., 0x3FFF) instruction cycles. Once 16,384 instruction cycles have passed or the `DISI 0x0000` instruction is executed, interrupts will be re-enabled. In some cases it may be possible that interrupts will be re-enabled before Salvo's critical section has ended – this would be a run-time fault.

`DISI` "only disables interrupts with priority levels 1-6 only. Priority 7 interrupts and all trap events still have the ability to interrupt the CPU when the `DISI` instruction is active."[8]

When using the `DISI` instruction, it's imperative that any interrupts that call Salvo services be at interrupt level 6 or lower, and the overall Salvo application must not have any critical section that approach 16,384 instruction cycles.

Therefore the default interrupt hooks as shown in Listing 5 (above) *should only be used in the simplest of Salvo applications*, and the user should implement targeted interrupt hooks (see below) *as soon as possible.*

For greater runtime performance and to unambiguously avoid the problems with the `DISI` instruction, users can replace the default interrupt hooks with targeted interrupt-control functions by building their own version with their application. For example, if the Salvo service `OSTimer()` is the *only* Salvo service called from the foreground (i.e. interrupt) level, *and* it's called (only) via the Timer2 ISR, then the following functions to disable and re-enable Timer2's interrupt enable bit are all that is required, e.g.:

```
void OSDisableHook(void)
{
  DisableIntT2;
}

void OSEnableHook(void)
{
  EnableIntT2;
}
```

Listing 6: Example of user interrupt hooks where only Timer2 ISR calls Salvo services

With the hooks shown in Listing 6, only the Timer2 interrupt is disabled during Salvo's critical sections. All other interrupts

sources (and global interrupts in general) are untouched. This allows other interrupts that do not call Salvo services to proceed with *zero interrupt latency* due to Salvo.

Warning Not disabling all source of interrupts that call Salvo services during critical sections will cause the Salvo application to fail.

Watchdog

The default watchdog hook in `salvohook_wdt_MCC30_clrwdt.c` is shown below.[9]

```
void OSClrWDTHook(void)
{
  __asm__ volatile("clrwdt");
}
```

Listing 7: Default Salvo watchdog hook for Microchip's MPLAB C30 C compiler

Users can replace it (e.g. with a dummy function – this would stop Salvo from clearing the watchdog timer and allow the user to clear it elsewhere) by building their own version with their application.

Compiler Issues

Incompatible Optimizations

It is recommended that optimizations – in particular, procedural abstraction – be disabled for all user code modules that contain Salvo tasks. There are no restrictions on applying optimizations on any of Salvo's modules, or any other user code.

Use with MPLAB XC16 Compiler

Microchip's successor to the MPLAB C30 compiler is the MPLAB XC16 compiler. A few issues must be addressed when working with MPLAB XC16 and Salvo:

1. XC16 uses a different set of predefined macros (`XC16`, `_XC16`, etc.) to identify itself to the preprocessor – these are auto-detected by Salvo in `salvoadc.h`.

2. For Linux users, there may be some case sensitivity issues among Salvo files.

3. MPLAB X's default object format is ELF. Earlier MPLAB products used COFF as the default object file format. Ensure that you are properly configured to link to the correct type of object module. Salvo libraries are currently supplied in COFF format only.

---

1 This is done automatically through the `C30`, `__C30` and/or `__C30__` symbols defined by the compiler.

2 Salvo 4.0.0.

3 In bytes. Includes `.reset`, `.ivt`, `.aivt`, `.text`, `.dinit` & `.isr` sections.

4 In bytes. Includes `.nbss` section. This represents all of Salvo's objects. Does not include RAM allocated to the heap or stack. Salvo applications typically require the same (small) stack size as simple, non-multitasking applications.

5 The MPLAB IDE passes command-line arguments to the MPLAB C30 compiler and linker as part of the build process. E.g., `-mlarge-code -mlarge-data` for large code and data models.

6 Unfortunately the MPLAB C30 compiler does not emit any symbols that identify which memory model(s) are in use … therefore Salvo users must define this symbol, which drives conditional compilation of the Salvo source code that is affected by the memory model settings.

7 This hook is valid for all 16-bit MCU and DSC targets.

8 From Section 8.2.3 of Microchip document DS39707A, *Section 8. Interrupts*.

9 This hook is valid for 16-bit MCU and DSC targets because the watchdog clearing instruction is the same for all targets.