PUMPKIN™
REAL-TIME SOFTWARE

RM-KCARM
Reference Manual

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

Salvo Compiler Reference Manual – Keil CARM

Salvo™
The RTOS that runs in tiny places.™

Introduction

This manual is intended for Salvo users who are targeting ARM7TDMI MCUs with Keil's (http://www.keil.com/) CARM C compiler.

Related Documents

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

Salvo User Manual
Application Note AN-31

Example Projects

Example Salvo projects for use with Keil's CARM C compiler and Keil's µVision3 IDE can be found in the:

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

directories of every Salvo for ARM family distribution.

Features

Table 1 illustrates important features of Salvo's port to Keil's CARM C compiler.

general	
available distributions	Salvo Lite, LE & Pro for ARM7TDMI family
supported targets	all ARM7TDMI derivatives
header file(s)	`portkcarm.h`
other target-specific file(s)	`portkcarm.s`
project subdirectory name(s)	`SYSAG`
salvocfg.h	
compiler auto-detected?	yes[1]
libraries	
`\salvo\lib` subdirectory	`kcarm`
CPU mode intermixing supported?	yes
context switching	
method	function-based via `OSDispatch()` & `OSCtxSw()`
`_OSLabel()` required?	no
size of auto variables and function parameters in tasks	total size must not exceed 65,532 8-bit bytes
interrupts	
controlled via	`OSDisableInts()`, `OSEnableInts()`, `OSRestoreInts()`, `OSSaveInts()`
interrupt status preserved in critical sections?	yes, with appropriate user functions
method used	see user functions
debugging	
source-level debugging?	only in source-code builds
compiler	
bitfield packing support?	no
printf() / %p support?	yes / yes
va_arg() support?	yes

Table 1: Features of Salvo Port to Keil's CARM C Compiler

Libraries

Nomenclature

The Salvo libraries for Keil's CARM C compiler follow the naming convention shown in Figure 1.

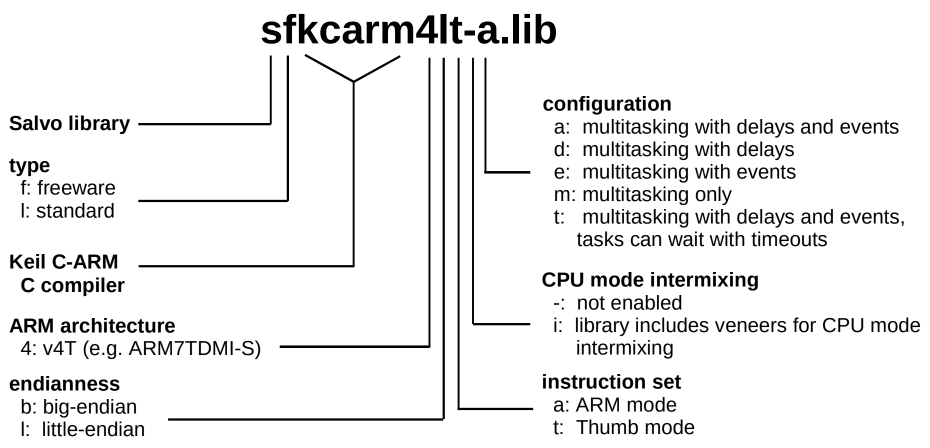

Figure 1: Salvo Library Nomenclature – Keil's CARM C Compiler

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.

ARM Architecture

The libraries can be used with all microcontrollers based on ARM7, ARM7TDMI(-S), ARM720 etc. cores.

Note The Salvo libraries may also be compatible with newer ARM architectures that offer binary compatibility with ARM's v4T architecture.

Endianness

Indicates the endianness (big-endian or little-endian) of the library.

Instruction Set

Two sets of libraries are provided – one in the ARM instruction set, and one in the Thumb instruction set.[2]

Note Unlike the library configuration option specified in the `salvocfg.h` file for a library build, none is specified for the selected instruction set. Therefore particular attention must be paid to the instruction set settings used to build an application when linking to a Salvo library that does not include CPU mode interworking (see below). The instruction set is usually specified on a project-wide basis in the µVision3 IDE.

CPU Mode Intermixing

Keil's CARM compiler supports CPU mode intermixing via the `INTERWORK` directive. Libraries with CPU mode intermixing enabled can be linked to applications built in either ARM or Thumb mode.

Note Applications built with CPU mode intermixing libraries will be slightly larger than those built with "pure" ARM or Thumb code.

Configuration

Different library configurations are provided for different Salvo distributions and to enable you 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 Keil's CARM 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 2.

compiled limits for Salvo Lite libraries	
max. number of tasks	3
max. number of events	5
max. number of event flags[3]	1
max. number of message queues[4]	1
target-specific settings for all Salvo libraries	
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 2: Limits, Build Settings and Overrides for Salvo Libraries for Keil's CARM C Compiler

Note When building ARM7TDMI applications from Salvo libraries, the numbers of tasks, events, event flags and message queues are limited only in Salvo Lite libraries. There is no limit to their numbers in all other Salvo libraries. See the *Libraries* chapter of the *Salvo User Manual* for more information.

Available Libraries

There are 40 Salvo libraries for Keil's CARM C compiler. 20 are built with the ARM instruction set, and 20 with the Thumb instruction set. Each Salvo for ARM 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 ARM distributions targeting a Philips LPC2129 single-chip microcontroller.

Salvo Lite Library Build

```
#define OSUSE_LIBRARY              TRUE
#define OSLIBRARY_TYPE             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 sfkcarm4lt-a.lib

Salvo LE & Pro Library Build

```
#define OSUSE_LIBRARY              TRUE
#define OSLIBRARY_TYPE             OSL
#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 slkcarm4la-a.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[5]	total ROM[6]	total RAM[7]
tu1lite	660	42
tu2lite	868	42
tu3lite	948	42
tu4lite	1572	66
tu5lite	2272	95
tu6lite	2416	97
tu6pro[8]	2272	93

Table 3: ROM and RAM requirements for Salvo Applications built with Keil's CARM C Compiler

Special Considerations

Target Independence

The Salvo code for ARM is *pure* ARM7TDMI code. It does not make any assumptions on or references to the manufacturer-specific peripherals found in each ARM7TDMI-based microcontroller.

Note Because of this target independence, it's unnecessary to include target-specific header files (e.g. `#include <LPC21xx.h>`) in any Salvo source or configuration files. Target-specific header files are only required in user files that access on-board peripherals, etc.

Auto Variables in Salvo Tasks

Due to implementation details, the total size of all auto / local variables in each Salvo task is limited to 65,532 bytes. It is unlikely this will cause any problems in real-world applications. There are no limits on the numbers and sizes of auto / local variables in any other functions in a Salvo application.

ARM and Thumb CPU Modes

Library Builds

Applications that run solely in ARM or Thumb modes should use the ARM-only and Thumb-only Salvo libraries to reduce object code size. If an application mixes the two modes, a Salvo library with the CPU mode intermixing veneer (a *mixed-mode* Salvo library) should be used instead. In all cases, the linker will issue warning messages if the appropriate Salvo function (ARM or Thumb mode)[9] cannot be found.

Note Dummy versions of Salvo's user-defined control functions (see User Control Functions, below) are included in each Salvo library. In order to avoid link-time warnings and/or errors when building library projects that use mixed-mode Salvo libraries. the

INTERWORK directive must be applied to the user's source module(s) that contain the control functions.

Source-Code Builds

Keil's CARM C compiler has command-line directives for compiling in ARM (ARM) or Thumb (THUMB, the default) modes, and for enabling the CPU mode intermixing veneer (INTERWORK). These directives are also available in the µVision3 IDE as project options. In a source-code build of a Salvo application, these directives should be applied equally to all of the Salvo source files in the project.

Salvo's context switcher (portkcarm.s) is written in assembly language and can be configured via two defined symbols, MAKE_FOR_ARM and MAKE_FOR_INTERWORK. A non-zero value for MAKE_FOR_ARM results in ARM code. Any other value, or no value, results in Thumb code. Similarly, a non-zero value for MAKE_FOR_INTERWORK results in a module that supports the CPU mode intermixing veneer, and can be called from ARM and Thumb mode, regardless of the setting of MAKE_FOR_ARM. Any other value, or no value, results in a module without the veneer.

symbol	value	effect
MAKE_FOR_ARM	0	Thumb mode
MAKE_FOR_ARM	> 0	ARM mode
MAKE_FOR_INTERWORK	0	CPU mode intermixing disabled
MAKE_FOR_INTERWORK	> 0	veneer for CPU mode intermixing enabled

Table 4: Symbol Values and Effects when Assembling portkcarm.s

These symbols are defined via the assembler's SET directive, e.g:

```
AA.EXE … SET(MAKE_FOR_ARM=0,MAKE_FOR_INTERWORK=1)
```

These symbols can be set from within the µVision3 IDE.

User Control Functions

Interrupts for Critical Sections

You have total control of interrupts in a Salvo application. Salvo for ARM distributions require that four user functions be defined for the control of interrupts in Salvo's critical sections. They are:

```
OSDisableInts()
OSEnableInts()
OSRestoreInts()
OSSaveInts()
```

More information on these user functions can be found in the *Salvo User Manual*.

An example will illustrate the use of these functions. Assume that Salvo's `OSTimer()` is called by a Philips LPC2129 ARM7TDMI's Timer 0 every 10ms. The LPC2129's Timer 0's interrupt enable bit is located in `VICIntEnable[4]`.[10] No other Salvo services are called from the foreground / interrupt level. A simple interrupt control scheme that ensures proper Salvo operation and re-enables the Timer 0 interrupt is shown below:

```
void OSDisableInts(void)
{
      VICIntEnClr  |= 0x00000010;
}

void OSEnableInts(void)
{
      VICIntEnable |= 0x00000010;
}

void OSSaveInts(void)
{
      ;
}

void OSRestoreInts(void)
{
      VICIntEnable |= 0x00000010;
}
```

This set of functions ensures that on entry (`OSSaveInts()`, `OSDisableInts()`) to Salvo's critical sections, Timer 0 interrupts are disabled, and on exit (`OSRestoreInts()`) they are re-enabled. You can call `OSEnableInts()` at the beginning of the Salvo application to enable Timer 0 interrupts.

The functions above are particularly simple for the LPC2129 microcontroller because of its ability to independently set and clear individual interrupt enable bits. The scheme can easily be extended to multiple interrupt sources if additional Salvo services (e.g. `OSSignalBinSem()`) are called from other ISRs.

In this example, there's no need to save and later restore the overall state of interrupt enable bits, because of the LPC2129's ability to individually set and clear those bits. Other interrupt enable bits that are not associated with Salvo services will not be affected by the functions above.

Note If no Salvo services are called from the foreground, then interrupt disabling during Salvo critical sections is not required.

Watchdog Timer

The target's watchdog timer (if present) can be automatically cleared with each invocation of the Salvo scheduler via suitable definition of the user function `OSClrWDT()`.

Source Code Builds

Salvo Pro users wishing to minimize code size can build applications and custom libraries by redefining the following macros in the project's `salvocfg.h`, thereby avoiding function calls to the user functions listed above. The reduction in code size will be minimal, however, and may not be portable between different ARM7TDMI-based microcontrollers.

```
OSDi()
OSEi()
OSEnterCritical()
OSLeaveCritical()
OSResumeCriticalSection()
OSSuspendCriticalSection()
```

[1] This is done automatically through the `__KEIL__` and `__CA__` symbols defined by the compiler.

[2] Thumb mode is CARM's default mode.

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

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

[5] Salvo v3.3.0-dev8 with DK-ARM v1.4b. CARM in Thumb mode, set to Optimization Level 7, with an emphasis on favoring code size.

[6] In bytes, as reported under `Program Size: code=cc`. Includes CARM `Startup.s` startup file (over 250 bytes).

[7] In bytes, `DATA`, as reported under `Memory Map` in the project's `.map` file. This is the total amount of RAM Salvo and the application requires. Does not include RAM automatically allocated by the compiler (1168 bytes) to the stack. Note that `Program Size: data=dd` includes stack size.

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

[9] Thumb-mode functions have `?T` suffixes appended to their names, and ARM-mode functions have `?A` appended to theirs.

10 Bits are set via writes of 1's to `VICIntEnable`, and cleared via writes of 1's to `VICIntEnClr`. Writes of 0's have no effect.