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

RM-KC51
Reference Manual

Salvo Compiler Reference Manual – Keil Cx51

Salvo™
The RTOS that runs in tiny places.™

Introduction

This manual is intended for Salvo users who are targeting 8051 family MCUs with Keil's (http://www.keil.com/) Cx51 C compiler.

Related Documents

The following Salvo documents should be used in conjunction with this manual when building Salvo applications with Keil's Cx51 C compiler:

Salvo User Manual
Application Note AN-13
Application Note AN-16

Example Projects

Example Salvo projects for use with Keil's Cx51 C compiler and Keil's µVision2 IDE can be found in the:

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

directories of every Salvo for 8051 family distribution.

Features

Table 1 illustrates important features of Salvo's port to Keil's Cx51 C compiler.

general	
available distributions	Salvo Lite, LE & Pro for 8051 family
supported targets	all 8051 derivatives
header file(s)	`portkc51.h`
other target-specific file(s)	`port8051.c`
project subdirectory name(s)	`SYSI`
salvocfg.h	
compiler auto-detected?	yes[1]
libraries	
`\salvo\lib` subdirectory	`kc51`
context switching	
method	function-based via `OSCtxSw()`
`_OSLabel()` required?	no
size of auto variables and function parameters in tasks	unrestricted
memory	
memory models supported	`compact`, `small` and `large`
memory types for Salvo's global objects	`data`, `idata` and `xdata`
interrupts[2]	
controlled via	EA
interrupt status preserved in critical sections?	configuration-dependent
method used	configuration-dependent
nesting limit	configuration-dependent
alternate methods possible?	3 different methods are used[3]
debugging	
source-level debugging?	only in source-code builds
compiler	
bitfield packing support?	yes
printf() / %p support?	yes / yes
va_arg() support?	yes

Table 1: Features of Salvo Port to Keil's Cx51 C Compiler

Compiler Optimizations

Incompatible Optimizations

Optimizer level 9 (common block subroutine packing) is incompatible with `OSCtxSw()` (present in all Salvo tasks). However, this is handled automatically[4] in `portkc51.h`, so the user need not be concerned with it. I.e. optimizer level 9 can be applied globally to any Salvo source files or source files that call Salvo services.

Libraries

Nomenclature

The Salvo libraries for Keil's Cx51 C compiler follow the naming convention shown in Figure 1.

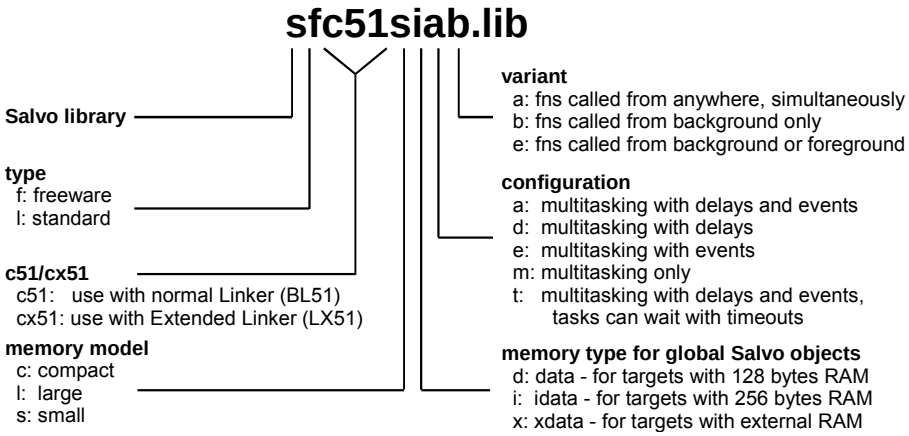

Figure 1: Salvo Library Nomenclature – Keil's Cx51 C Compiler

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 (c51/cx51)

No target-specific identifiers are required. However, when using the BL51 standard linker, the appropriate libraries contain `c51` in their names. When using the LX51 extended linker, the appropriate libraries contain `cx51` in their names. A mismatch will cause DATA TYPES DIFFERENT warnings.

Memory Model

Keil's Cx51 C compiler's `compact`, `small` (Cx51 default) 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 – a mismatch will generate a link-time error with an obvious message. For source-code builds, the same memory model 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. 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 µVision2 IDE.

Memory Type for Global Salvo Objects

Salvo's global objects can be placed anywhere within the 8051's RAM data space, as shown in Table 2.

memory type code	description
`d / OSD:`	Salvo's global objects will be placed within the first 128 bytes of RAM (the `data` RAM area)
`i / OSI:`	Salvo's global objects will be placed within the first 256 bytes of RAM (the `idata` RAM area)
`x / OSX:`	Salvo's global objects will be placed anywhere within external RAM (the `xdata` RAM area)

Table 2: Memory Types for Salvo Libraries – Keil's Cx51 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 the internal RAM of the 8051 is often used for the system's stack, function parameters and auto variables, in larger applications it may be necessary to place Salvo's global objects in external 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.

Variant

The Salvo libraries for Keil's Cx51 C compiler implement different methods of controlling interrupts. Each is designed for minimal code size. The different variants – and how they control interrupts – are outlined in Table 3.

variant code	description
a / OSA:	Applicable services can be called from *a*nywhere, i.e. from the foreground and the background, simultaneously. Global interrupts will be disabled (EA = 0) during critical sections, and restored thereafter. Unlimited nesting is permitted. OSPRESERVE_INTERRUPT_MASK is TRUE.
b / OSB:	Applicable services may only be called from the *b*ackground (default). Global interrupts will be blindly disabled (EA = 0) during critical sections. Nesting is not permitted. OSPRESERVE_INTERRUPT_MASK is FALSE.
e / OSE:	Applicable services may only be called from *e*verywhere, i.e. from the foreground or the background, but not simultaneously.. Global interrupts will be disabled (EA = 0) during critical sections, and restored thereafter. Nesting is not permitted. OSPRESERVE_INTERRUPT_MASK is TRUE.

Table 3: Variants for Salvo Libraries – Keil's Cx51 C Compiler

If your application does not call any Salvo services from within interrupts, use the *b* variant. If you wish to call applicable services from within interrupts or from the background level – but never simultaneously and without any nested interrupts – use the *e* variant. If you don't want any restrictions placed on how you call Salvo services, 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.

The *a*-variant libraries are the most versatile, but they are also the largest because of the on-stack saving of IE and the `reentrant` keyword that is applied to appropriate Salvo services. The *e*-variant libraries are smaller because reentrancy is not used. The *b*-variant libraries are smaller still.

See the `OSCALL_OSXYZ` configuration parameters for more information on calling Salvo services from interrupts.

Build Settings

Salvo's libraries for Keil's Cx51 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[5]	1
max. number of message queues[6]	1
target-specific settings	
delay sizes	8 bits
idling hook	enabled
interrupt-enable bits during critical sections	`EA = 0`
message pointers	can point to `idata`
Salvo objects	as per library's memory type
system tick counter	available, 32 bits
task priorities	enabled
watchdog timer	not affected

Table 4: Build Settings and Overrides for Salvo Libraries for Keil's Cx51 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 540 Salvo libraries for Keil's Cx51 C compiler. 270 are for use with the BL51 linker, and 270 for use with the LX51 linker. Each Salvo for 8051 family 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 8051 distributions targeting a generic 8051.

Note When overriding the default number of tasks, events, etc. in a Salvo library build, `OSTASKS` and `OSEVENTS` (respectively) *must also be defined* in the project's `salvocfg.h`. If left undefined, the default values (see Table 4) will be used.

Salvo Lite Library Build

```
#define OSUSE_LIBRARY           TRUE
#define OSLIBRARY_TYPE          OSF
#define OSLIBRARY_GLOBALS       OSD
#define OSLIBRARY_CONFIG        OSA
#define OSLIBRARY_VARIANT       OSB
```

Listing 1: Example salvocfg.h for Library Build Using sfc51sdab.lib

Salvo LE & Pro Library Build

```
#define OSUSE_LIBRARY           TRUE
#define OSLIBRARY_TYPE          OSL
#define OSLIBRARY_GLOBALS       OSD
#define OSLIBRARY_CONFIG        OSA
#define OSLIBRARY_VARIANT       OSB
```

Listing 2: Example salvocfg.h for Library Build Using slc51sdab.lib

Salvo Pro Source-Code Build

```
#define OSENABLE_IDLING_HOOK    TRUE
#define OSENABLE_SEMAPHORES     TRUE
#define OSEVENTS                1
#define OSLOC_ALL               data
#define OSTASKS                 3
```

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

Performance

Memory Usage

tutorial memory usage[7]	total ROM[8]	total RAM[9]
`tu1lite`	151	23
`tu2lite`	275	23
`tu3lite`	305	25
`tu4lite`	674	35
`tu5lite`	1026	52
`tu6lite`	1278	57
`tu6pro`[10]	1159	53

Table 5: ROM and RAM requirements for Salvo Applications built with Keil's Cx51 C Compiler

Special Considerations

8051 Target Compatibility

An extraordinary variety of 8051-type MCUs are available. Many have extended memory ranges beyond the typical `data`, `idata` and `xdata` spaces.

Salvo is source-code compatible with all of the 8051 family devices that Keil's Cx51 C compiler supports.

Salvo's libraries are compatible with all 8051 variants that have RAM in the `data`, `idata` or `xdata` spaces.

Alternate Memory Mappings for Source-Code Builds

To place Salvo's global objects in a different RAM space in a source-code build, redefine `OSLOC_ALL` (default: `idata`) or individual `OSLOC_XYZ` configuration options.

To change what Salvo's message pointers can point to, redefine `OSMESSAGE_TYPE` (default: `idata`).

Memory Map Issues Involving RAM Address 0x0000

Certain Cx51 memory attributes (e.g. the `xdata` space) include the ability to place global variables at an address of 0x0000. If the global Salvo object placed at address 0x0000 is pointed to

anywhere in the Salvo code, Salvo's runtime error checking will interpret the pointer as a NULL pointer and will reject the operation (e.g. creating an event flag whose event flag control block is located at 0x0000). This can cause confusion, as code that locates some or all of Salvo's global objects in one memory space (e.g. the `data` space) may not work correctly when the memory space is changed (e.g. to `xdata`) because of a difference in the valid address ranges in the two data spaces.

All of Salvo's global objects are located in Salvo's `mem.c`. The objects are arranged in `mem.c` such that the first object, `OScTcbP`, can be located at a memory address of 0x0000 without causing any runtime problems.[11] *No other Salvo global objects may be located at address 0x0000.*

Effect of Random Linktime Global Object Arrangement

As an example, Figure 2 illustrates the memory locations for Salvo's global objects for a Salvo project built with `OSLOC_ALL` set to `xdata` in the project's `salvocfg.h`. By default, the BL51 linker arranges the global objects as it sees fit, *not necessarily in the order in which they are declared in the source code*:

```
[SNIP]
-------         MODULE        MEM
C:0000H         SYMBOL        _ICE_DUMMY_
X:0000H         PUBLIC        OSefcbArea
X:0004H         PUBLIC        OSecbArea
X:000EH         PUBLIC        OSdelayQP
X:0010H         PUBLIC        OStcbArea
X:0022H         PUBLIC        OSsigQinP
X:0024H         PUBLIC        OSsigQoutP
X:0026H         PUBLIC        OSeligQP
X:0028H         PUBLIC        OSlostTicks
X:0029H         PUBLIC        OScTcbP
-------         ENDMOD        MEM
[SNIP]
```

Figure 2: Salvo's Global Objects Located in the xdata Memory Space in Random Order

In Figure 2, the first of Salvo's event flag control blocks (efcbs) in the `OSefcbArea` array is located at 0x0000 (equivalent to NULL when used as a pointer). The project compiles successfully, but does not operate correctly with regard to the first event flag (at `OSEFCBP(1)`). This is because Salvo's event flag services, which take a pointer to the efcb as a parameter, reject a NULL pointer as being invalid, and return an error code.

Solution #1: Keep Salvo's Global Objects in Order

By selecting Keep variables in order under µVision's Project → Options → C51,[12] Salvo's global objects in `mem.c` will be arranged in the intended order, with `OScTcbP` first:

```
[SNIP]
-------         MODULE        MEM
C:0000H         SYMBOL        _ICE_DUMMY_
X:0000H         PUBLIC        OScTcbP
X:0002H         PUBLIC        OStcbArea
X:0014H         PUBLIC        OSeligQP
X:0016H         PUBLIC        OSecbArea
X:0020H         PUBLIC        OSsigQinP
X:0022H         PUBLIC        OSsigQoutP
X:0024H         PUBLIC        OSefcbArea
X:0028H         PUBLIC        OSdelayQP
X:002AH         PUBLIC        OSlostTicks
-------         ENDMOD        MEM
[SNIP]
```

Figure 3: Salvo's Global Objects Located in the xdata Memory Space in Specified Order

By ensuring that `OScTcbP` is at 0x0000, all issues with NULL pointers and Salvo's global objects are avoided.

Solution #2: Specify xdata Memory Range(s)

By selecting Off-chip Xdata memory Start and Size under µVision's Project → Options → Target, one can avoid placing any of Salvo's global objects at 0x000. For example, by starting the `xdata` space at 0x0004, we get the memory map below:

```
[SNIP]
-------         MODULE        MEM
C:0000H         SYMBOL        _ICE_DUMMY_
X:0004H         PUBLIC        OSefcbArea
X:0008H         PUBLIC        OSecbArea
X:0012H         PUBLIC        OSdelayQP
X:0014H         PUBLIC        OStcbArea
X:0026H         PUBLIC        OSsigQinP
X:0028H         PUBLIC        OSsigQoutP
X:002AH         PUBLIC        OSeligQP
X:002CH         PUBLIC        OSlostTicks
X:002DH         PUBLIC        OScTcbP
-------         ENDMOD        MEM
[SNIP]
```

Figure 4: Salvo's Global Objects Located in the xdata Memory Space with a Specified Non-zero Start Address

Once again, since there is no Salvo global object at 0x0000, all issues with NULL pointers and Salvo's global objects are avoided.

Solution #3: Reserving xdata Space

By reserving some RAM in the xdata space via the `DS` assembly-language directive, you can ensure that none of Salvo's global objects are placed at 0x0000:

```
xseg at 0
XNULL: DS 0x10
```

Figure 5: Assembly-Language Directives to Avoid placement of any Salvo Objects at 0x0000

By placing the snippet in Figure 5 in an assembly-language module that is built as part of the project, an absolute `xdata` segment that's 16 bytes long will be created, and the linker will place it at address `xdata`:0000. Other relocatable segments (e.g. the segment for Salvo's `mem.c`) will be located after it, thus guaranteeing that none of Salvo's global objects are located at 0x0000 and all issues with NULL pointers and Salvo's global objects are avoided.

Bank Switching

Salvo is compatible with Cx51's *code banking* (also called *bank switching*), with certain restrictions.

Overview

The dispatching of tasks via the Salvo scheduler and context-switcher is likely to be incompatible with the bank switching implemented via Cx51's `L51_BANK.A51` assembly-language module. Therefore the Salvo scheduler, context-switcher and tasks themselves *must be located in the common (code) area*.

Note To maximize the available space in the common area, it's recommended that user modules containing Salvo tasks contain no other user functions.

In all cases, user functions called by Salvo tasks are not restricted in terms of their code locations – they can be in the common area, or in banked areas.

Recommended Practice

It is recommended that all Salvo modules, as well as all modules containing Salvo tasks, be placed in the common area. This will maximize Salvo's performance by avoiding the bank-switching overhead that would occur if any of Salvo's user services or

internal services are located in a bank. Because of Salvo's small ROM footprint, many applications can accommodate Salvo and Salvo tasks in the common area.

Note The BL51/Lx51 linker/locator always places program sections of runtime libraries in the common area. Therefore applications built with Salvo libraries will automatically have the Salvo code placed in the common area. In this case, the user must also ensure that the Salvo tasks themselves are placed in the common area.

Alternative Practice

If space in the common area is at a premium, one possible alternative for Salvo Pro users is to do a source-code build, and locate all of Salvo's source modules *except those containing the scheduler and the context-switcher* in a banked area. Salvo tasks would still need to be located in the common area.

Preserving Interrupt Masks

When `OSPRESERVE_INTERRUPT_MASK` is set to `TRUE` (the default), Salvo initially saves and later restores the register `IE` on the stack as part of disabling interrupts in critical sections. This has a substantial impact on Salvo code size – therefore it should only be used in source-code builds if / when you intend to call one or more Salvo services from the foreground / interrupt level.

Multiple Callgraphs, Reentrancy, etc.

Keil's Cx51 C compiler *does not* pass parameters and auto variables on the stack unless a function is declared as reentrant. Salvo services that can be called from the foreground / interrupt level are made reentrant by defining the associated `OSCALL_OSXYZ` configuration option to be `OSFROM_ANYWHERE`.

---

1 This is done automatically through the `__C51__` and `__CX51__` symbols defined by the compiler.

2 See Variant.

3 `OSEnter|LeaveCritical()` can be re-written to control particular interrupt bits when `OSPRESERVE_INTERRUPT_MASK` is `TRUE`.

[4] Level 9 optimizations are automatically disabled in any source module that contains a Salvo context switch when `OSPRESERVE_INTERRUPT_MASK` is `TRUE`.

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

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

[7] Salvo v3.2.0 with Cx51 v7.00A

[8] In bytes, as reported under `Program Size: data=dd xdata=xx code=cc`.

[9] In bytes, `idata`, as reported under `Program Size: data=dd xdata=xx code=cc`.

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

[11] `OScTcbP` is never referenced by pointer in the Salvo code.

[12] The command-line control directive `ORDER` has the same effect.