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

RM-IARAVR

Reference Manual

Salvo Compiler Reference Manual – IAR AVR C

Salvo™

The RTOS that runs in tiny places.™

Introduction

This manual is intended for Salvo users who are targeting Atmel (http://www.atmel.com/) AVR® and MegaAVR™ microcontrollers[1] with IAR's (http://www.iar.com/) Embedded Workbench for C AVR.

Related Documents

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

- *Salvo User Manual*

Example Projects

Example Salvo projects for use with IAR's AVR C compiler and the Embedded Workbench IDE can be found in the:

```
\Pumpkin\Salvo\Example\AVR\AT90S8515
```

directories of every Salvo for Atmel AVR and MegaAVR distribution.

Tip These example projects can be easily modified for any AVR or MegaAVR device.

Features

Table 1 illustrates important features of Salvo's port to IAR's AVR C compiler.

General	
Abbreviated as	`IARAVR`
Available distributions	Salvo Lite, LE & Pro for Atmel AVR and MegaAVR
Supported targets	entire AVR and MegaAVR family
Header file(s)	`salvoportiaravr.h`
Other target-specific file(s)	`salvoportiaravr.c`
salvocfg.h	
Compiler auto-detected?	yes[2]
Include target-specific header file in `salvocfg.h`?	yes
Libraries	

Located in	`Lib\IARAVR-v2`
	(for v2.x compilers)
	`Lib\IARAVR-v3`
	(for v3.x compilers)
	`Lib\IARAVR-v4`
	(for v4.x compilers)

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 254 8-bit bytes
Memory & Registers	
Internal and external RAM supported?	yes / yes (see section on external memory)
Register usage	follows assembly language interface
Interrupts	
Interrupt latency in context switcher	?? cycles
Interrupts in critical sections controlled via	user hooks
Default behavior in critical sections	see example user hooks
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 IAR's AVR C compiler

Libraries

Nomenclature

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

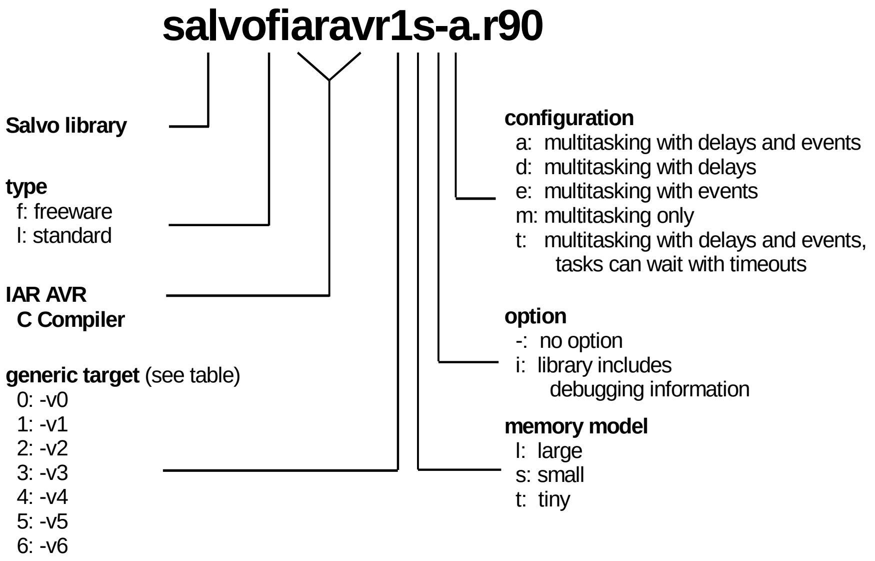
salvofiaravr1s-a.r90
Salvo library
type
f: freeware
l: standard
IAR AVR
C Compiler
generic target (see table)
0: -v0
1: -v1
2: -v2
3: -v3
4: -v4
5: -v5
6: -v6
configuration
a: multitasking with delays and events
d: multitasking with delays
e: multitasking with events
m: multitasking only
t: multitasking with delays and events, tasks can wait with timeouts
option
-: no option
i: library includes debugging information
memory model
l: large
s: small
t: tiny

Figure 1: Salvo Library Nomenclature – IAR's AVR C Compiler

Note Each successive version of IAR's AVR C/C++ compiler *uses different library formats*. Therefore independent sets of Salvo libraries are available for each compiler version – see *libraries* in Table 1, above.

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

Each library is built using generic options and is intended for one or more specific processors, based on the compiler's -v option appropriate for the selected processor. Table 2 lists the correct library for each AVR derivative.

Target Code	Processor(s)
`0:`	AT90S2313, AT90S2323, AT90S2333, AT90S2343, AT90S4433, ATtiny13, ATtiny22, ATtiny26, ATtiny2313, generic –v0
`1:`	AT90S4414, AT90S4434, AT90S8515, AT90S8534, AT90S8535, ATmega8, ATmega48, ATmega8515, ATmega8535, generic –v1
`2:`	generic –v2
`3:`	ATmega16, ATmega32, ATmega103, ATmega128, ATmega161, ATmega162, ATmega163, ATmega168, ATmega169, ATmega323, generic –v3
`4:`	generic –v4
`5:`	generic –v5
`6:`	generic –v6

Table 2: Processors for Salvo libraries – IAR's AVR C compiler

Note Table 2 is not exhaustive – if you are unsure which target code to use with a Salvo library, refer to the *Processor variant* section of the IAR AVR C compiler's *Reference Manual*.

Additionally, you can inspect the command-line options passed to the compiler when building a Embedded Workbench project to discover which processor variant is associated with your AVR target.

When building your application with a Salvo library, you can either specify a specific target (e.g. `--cpu=8515`) or a generic target (e.g. `–v1`) to IAR's AVR C compiler.[3] In either case, the generic option associated with your target must match the Salvo library's target code.

Memory Model

IAR's AVR C compiler's `tiny`, `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 – 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 option 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 Embedded Workbench IDE.

Option

Salvo Pro users can select between two sets of libraries – standard libraries, and standard libraries incorporating source-level debugging information.[4] The latter have been built with IAR's AVR C compiler C compiler's `--debug` command-line option. This adds source-level debugging information to the libraries, making them ideal for source-level debugging and stepping in the C-SPY debugger. To use these libraries, simply select one that includes the debugging information (e.g. `salvoliaravr1tia.r90`) instead of one without (e.g. `salvoliaravr1t-a.r90`) in your Embedded Workbench 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 IAR's AVR 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.

Target-specific Settings	
Delay sizes	8 bits
Idling hook	dummy, can be overridden
Interrupt hook	disables then restores GIE bit, can be overridden
Watchdog hook	clears WDT without other changes, can be overridden
System tick counter	available, 32 bits
Task priorities	enabled

Table 3: Build settings and overrides for Salvo libraries for IAR's AVR C compiler

Note Salvo Lite libraries have precompiled limits for the number of supported tasks, events, etc. Salvo LE and Pro libraries have no such limits. See the *Libraries* chapter of the *Salvo User Manual* for more information.

Available Libraries

There are a total of 330 Salvo libraries for IAR's AVR C compiler – 165 for AVR C v2.x, and 165 for AVR C v3.x. Each Salvo for Atmel AVR and MegaAVR distribution contains the Salvo libraries of the lesser distributions beneath it.

Target-Specific Salvo Source Files

The source file `salvoportiaravr.c` is needed for Salvo Pro source-code builds.

salvocfg.h Examples

Below are examples of `salvocfg.h` project configuration files for different Salvo for Atmel AVR and MegaAVR distributions.

Salvo Lite Library Build

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

Listing 1: Example salvocfg.h for library build using salvofiaravr1s-t.r90

Salvo LE & Pro Library Build

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

Listing 4: Example salvocfg.h for library build using salvoliaravr3t-t.r90 or salvoliaravr4tit.r90

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 5: Example salvocfg.h for source-code build

Performance

Memory Usage

Example Application[5]	Program Memory Usage[6]	Data Memory Usage[7]
`\AVR\…\tut5lite`	xx	xx
`\AVR\…\tut5le`	xx	xx
`\AVR\…\tut5pro`	xx	xx

Table 4: ROM and RAM requirements for Salvo applications built with IAR's AVR C compiler

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 1: Default Salvo idling hook for IAR's AVR C compiler

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

Interrupt

The default interrupt hooks in `salvohook_interrupt.c` is shown below.[8]

```
static unsigned char s;

void OSDisableHook(void)
{
  unsigned char s_local;

  s_local = __save_interrupt();
  __disable_interrupt();
  s = s_local;
}


void OSEnableHook(void)
{
  __restore_interrupt(s);
}
```

Listing 2: Default Salvo interrupt hooks for IAR's AVR C compiler

These functions clear the `GIE` bit (i.e. disable global interrupts) across Salvo's critical section, and restore the bit to its previous value thereafter. These hooks are suitable for *all* applications. These hooks work very well within Salvo services called from interrupts, as the GIE bit is automatically cleared upon entry to an interrupt. Therefore interrupts are *not* re-enabled at the end of a Salvo service that is called in an ISR. This avoids unnecessary interrupt nesting. The use of the auto variable `sreg_local` avoids issues that would affect the shared global `sreg` when a Salvo service is called from within an ISR.

Note 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.c` is shown below.[9]

```
void OSClrWDTHook ( void )
{
  __watchdog_reset();
}
```

Listing 3: Default Salvo watchdog hook for IAR's AVR 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

Runtime Models and Compatible Libraries

The runtime models used by Embedded Workbench for AVR have evolved over the years. When building an application with Salvo libraries, it's necessary to link to the libraries compatible with the version of Embedded Workbench for AVR that you are using. Table 5 lists the locations of Salvo libraries as a function of the Embedded Workbench for AVR version.

Embedded Workbench for AVR Version	IAR C Compiler Version	IAR Runtime Model	Salvo Library Location
`2.x`	v2.x	1	`Lib\IARAVR-v2`
`3.x`	v3.x	1	`Lib\IARAVR-v3`
`4.x`	v4.x	2	`Lib\IARAVR-v4`

Table 5: Compiler versions, runtime models and Salvo library locations for IAR's AVR C compiler

Incompatible Optimizations

There are no known incompatibilities between IAR's AVR C compiler and Salvo.

Special Considerations

Stack Issues

IAR's AVR C compiler uses two separate stacks – one for return addresses (the hardware stack, `RSTACK`, which uses `SP`) and one for local storage (the data stack, `CSTACK,` which uses `Y`).

Compared to a non-Salvo, non-multitasking application with similar call trees, the corresponding Salvo application will require an additional 6 bytes (i.e. two return addresses and two saved registers) in the hardware stack[10].

The hardware stack and the software stack are set to the same location. However the hardware stack grows downwards, and the software stack grows upwards.

External RAM

Salvo's global objects[11] can be placed in internal or external RAM. The placement of data objects is controlled by linker options – see the IAR documentation for more information.

Memory Models and Salvo's Global Objects

The compiler's *default memory attributes* and *default pointer types* are automatically applied to Salvo's global objects based on the memory model selected. Therefore Salvo's `OSLOC_XYZ` configuration options should not be used in Salvo Pro source-code builds.

Data Storage

Table 6 illustrates the effect of the selected memory model on Salvo's global objects.

memory model employed	maximum object size	pointer size	address range
`tiny`	127 bytes	1 byte	0x0-0xFF
`small`	32 Kbytes	2 bytes	0x0-0xFFFF
`large`	32 Kbytes	3 bytes	0x0-0xFFFFFF

Table 6: Effect of Memory Model on Salvo's Global Objects

The `tiny` memory model is useful when wishing to minimize the amount of RAM used by Salvo, and the Salvo code's ROM requirements. With the `tiny` memory model, the maximum number of tasks and events is severely restricted.[12]

The `small` memory model is required when the size of Salvo's global objects exceeds 127 bytes.

The large memory model is likely to be unnecessary for most applications, but is included for completeness. If used, the Salvo objects cannot cross a 64K boundary.

Note A user's application must be built with the same memory model as the Salvo library in use. However, the memory attributes of the user's data *need not be the same* as those of Salvo's global objects.

For example, one could build a small memory model application with the __tiny attribute applied to some of the user's data, in order to maximize speed and minimize ROM associated with that user data.

Function Pointers

The Salvo context switcher uses the AVR's IJMP instruction to implement indirect function calls. Since IJMP only supports PC(15..0), all Salvo tasks must be __nearfunc (the compiler's default),[13] and must therefore be located within the first 128KB of program space. The compiler's __farfunc function memory attribute is not supported for use on Salvo tasks.

Optimizations

Salvo is compatible with IAR's AVR code optimizations at all levels in both source and library builds.

Global Register Variables

Registers R4-R15 are available to the user as *global register variables* via the compiler's --lock_regs option.

[1] tinyAVR devices are not supported because of their lack of RAM.

[2] This is done automatically through the __GNUC__ and __AVR __ symbols defined by the compiler.

[3] Either in Embedded Workbench or in a command-line build.

[4] The Salvo libraries provided with Salvo Lite and LE do not contain C-SPY-compatible debugging information because this requires the inclusion of source file listings.

[5] Salvo 4.1.0-rc0 with v3.42A compiler.

[6] In bytes. Salvo code only.

[7] In bytes. Salvo objects only.

[8] This hook is valid for all AVR and MegaAVR targets because the register and GIE bit locations are the same for all targets.

[9] This hook is valid for all AVR and MegaAVR targets because the watchdog control register is the same for all targets.

[10] Salvo Pro application can reduce this by 2 bytes (one return address) by inlining `OSSched()`.

[11] E.g. task control blocks, queue pointers, counters, etc.

[12] Especially since the stack is also located in low RAM.

[13] I.e. two bytes are used for function pointers.