PUMPKIN™
REAL-TIME SOFTWARE

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

Optimizing Salvo for Use with the HI-TECH PICC-18 C Compiler

Introduction

Through judicious use of Salvo's configuration options, substantial memory savings and performance improvements can be realized when compiling Salvo applications with the HI-TECH PICC-18 C compiler.

Where To Make Changes

All of Salvo's configurability is driven by the configuration options that appear as defined symbols in your project's `salvocfg.h`. You will not need to modify any other Salvo source files.

Note If you are upgrading from an existing PICC (PIC16 MCU) application, you must ensure that certain, PICC-specific symbols (e.g. `bank1`) do not appear in your `salvocfg.h` file when compiled under PICC-18. Failure to remove such symbols may result in compiler errors.

How to Optimize

You should always use obvious techniques like applying local and global compiler optimizations and using the smallest variable type possible. See the PICC-18 documentation for more information.

To optimize an application further, it's instructive to identify which parts of the application will benefit most from optimization, and where further optimization is actually possible. For example, all Salvo libraries are supplied with the most up-to-date configurations to minimize Salvo's ROM and RAM footprints and maximize performance. But since the libraries are precompiled, they cannot be optimized further. In a situation like this, your only option may

be to apply the techniques outlined below to your own source code, and use the optimized Salvo libraries.

If you are building your application with the Salvo source code, then you have much more flexibility, as you can define the various Salvo configuration parameters as befits your application.

Control Object Placement and Type

Salvo uses static global variables for pointers, task control blocks, semaphore values, timers, etc. Every Salvo object has an `OSLOC_XYZ` configuration option that can be used to specify the object's memory type. The greatest improvements to run-time performance and code size will come from careful selection of *special type qualifiers* for Salvo objects. You can control where these variables are placed in memory – and in some instances, what their types will be – by defining the configuration options with appropriate type qualifiers in `salvocfg.h`.

Salvo also has some other configuration options (e.g. `OSOPTIMIZE_FOR_SPEED`, etc.) that may have beneficial effects on run-time performance. Their effect is usually minor, and can only be determined by before-and-after comparisons.

Finally, functions may have parameters and / or local auto variables. The memory allocation of these variables is under direct control of the compiler and cannot be influenced by the user.

Use the Access Bank

Perhaps the greatest improvement to both program ROM size and RAM utilization is the use of PICC-18's `near` memory type qualifier. By declaring a variable as `near`, the code to access it is smaller and faster because `near` objects are represented by 8-bit pointers, and the access bank is always accessible.

The total RAM used is smaller, too, since Salvo makes extensive use of pointers to manage tasks and events. To qualify all of Salvo's objects as `near`, add this line to your `salvocfg.h`:

```
#define OSLOC_ALL        near
```

Avoid Redundant Initialization Code

PICC-18 provides the `persistent` memory type qualifier. This is useful in Salvo applications that need not preserve RAM contents

between successive power-ups, resets, wakes-from-sleep, etc. When used on Salvo objects, it reduces ROM slightly because the Salvo objects no longer need be initialized by the PICC-18 startup code. To qualify all of Salvo's objects as `near` and `persistent`, add this line to your `salvocfg.h`:

```
#define OSLOC_ALL        near persistent
```

Qualify Variables Intelligently

In larger applications, it may not be possible to apply the same optimizations to all of the variables because of the limited size of the access bank. You should qualify objects that are accessed most often as `near`, and leave those that are rarely accessed in other banks.

You can do this with Salvo objects, too, by qualifying certain ones individually. For instance, to place Salvo's event control blocks and counters in normal memory and leave the rest in the access bank, add this to your `salvocfg.h`:

```
#define OSLOC_ALL        near persistent
#define OSLOC_ECB        persistent
#define OSLOC_COUNT      persistent
```

This way you can selectively place Salvo's objects in RAM and still have room left over in the access bank for critical variables in your own source code.

Results

As an example that illustrates these optimizations, Salvo demo program `d1` (`salvo\demo\d1\sysf`) was compiled with different configuration options. This is a complex application with an 80x25 terminal screen interface, eight tasks and five semaphores, interrupts, a system timer, sampled keys, etc. It displays the run-time behavior of two distinctly different operating modes on the LEDs of the Microchip PICDEM™-2 Demonstration Board. The results are shown below.

ROM and RAM Utilization

Table 1 illustrates the dramatic improvement to code size that occurs when Salvo objects are declared with the `near` type qualifier.

OSLOC_ALL	Program ROM	RAM data	Common RAM	ROM data
(undefined)	10,394	184	62	2136
near	8548	50	157	2128
near persistent	8528	50	157	2128

Table 1: ROM and RAM Utilization of Salvo Demo d1 for different values of OSLOC_ALL (all data in bytes)

Note that the use of the `persistent` memory type qualifier only affects program ROM size. Since `d1` calls `OSInit()`, all Salvo objects are explicitly initialized in the application, and the compiler's startup initialization code *for Salvo objects only* is therefore unnecessary.

The overall compile- and run-time improvements from the methods described above are shown in Table 2, again for `d1` (4MHz). The performance parameter is a measure the application's context-switching rate.

parameter	PIC18C452 (unoptimized)	PIC18C452 (optimized)	PIC16C77 (optimized)
total ROM	6,256 words (12,530 bytes)	5,328 words (10,656 bytes)	7049 words
total RAM	246	207	194
performance, mode 1	2,018/s	2,462/s	2,644/s
performance, mode 2	2,989/s	3,715/s	3,401/s

Table 2: Results of Optimizations on Salvo Demo d1

Of particular interest are the reduced ROM size (-15%), the reduced RAM size (-16%) and the improved run-time performance (+22% and +24%) by declaring `OSLOC_ALL` to be `near persistent`.

For comparative purposes, the results of `d1` compiled under PICC for a PIC16 target are also shown. The PIC18 application is *much* smaller, ROM-wise (-24%), the RAM usage is nearly identical, and the difference in performance is clearly application-dependent.

Conclusion

Substantial performance gains and code size reductions can be achieved through PICC-18's special type qualifiers when applied to a Salvo application via the `OSLOC_XYZ` configuration options.