PUMPKIN™
REAL-TIME SOFTWARE

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

Interrupts and Salvo Services

Introduction

Writing reliable microcontroller applications that use interrupts is easy if you understand the effect that interrupts can have on mainline code. This Application Note explains how to control interrupts when using Salvo™ user services.

Critical Sections

Most real-time operating system (RTOS) services contain *critical sections* of code. Critical sections may read or modify global (i.e. shared) RTOS variables. In order to protect these variables against possible corruption by interrupt service routines (ISRs), interrupts are disabled prior to the critical section and restored thereafter.

Note A basic tenet of RTOS design is to minimize the time during which interrupts are disabled. In other words, critical sections of code should be as short as possible.

As an example of the need for protecting critical sections, an RTOS might maintain a linked list of objects. Removing an object from this linked list might be one of the actions performed by an RTOS service – `SvcM()` – that is callable by the user. If an unrelated interrupt (e.g. a UART receive buffer full interrupt) were to occur during the execution of this RTOS service, it would have no effect.[1]

However, if the application calls another RTOS service – `SvcI()` – in an ISR, and that service operates on same the linked list, then it's entirely possible that data corruption will occur if `SvcI()` is called while `SvcM()` is interrupted. To protect this critical section of code, each service would be written as shown in Listing 1.

created by Andrew E. Kalman on Aug 17, 2001 updated on Aug 17, 2001

```
{
  …
  disable interrupts;
  operate on linked list; /* critical section */
  restore interrupts;
  …
}
```

Listing 1: Protecting A Critical Section

This way, an interrupt that modifies the linked list[2] can only happen before or after the operation on the linked list – it cannot happen during that time. All Salvo services disable interrupts during critical sections of code.

Don't Worry, Be Happy

The overview above sums up the issue of critical sections and interrupts for a conventional RTOS. But Salvo is different – *it neither uses nor requires a software stack*. This is one reason for Salvo's miniscule RAM requirements. In certain situations, this has important implications on the issue of interrupts and critical sections.

But first, note that there are two situations where you needn't worry about interrupts in a Salvo application: 1) if you neither have nor use interrupts,[3] and 2) if the C compiler you are using passes parameters on the stack. The vast majority of compilers operate this way due to their target processors having support for stack-based operations. Salvo supports certain processors and compilers like this. The ability to save (*push*) the interrupt state on the stack and restore (*pop*) it later, together with the compiler's ability to pass function parameters and return values on the stack, ensure that the method outlined above for protecting critical sections of code is sufficient.

A Stack! A Stack! My Kingdom for a Stack![4]

C compilers for processors without a general-purpose software stack (we'll call them *stackless compilers*) are able to provide full C functionality by using *dedicated RAM* for storing parameters, return values, auto variables, etc. To minimize RAM usage, objects are *overlayed* when not in use at the same time. The compiler does this automatically by analyzing the *call graphs* of all the functions in the application. If two functions are never in the same call graph (i.e. neither one is a child of the other), then their parameters can share the same RAM at runtime.

In Listing 2 we see how this is done.[5] Prior to calling the subroutine of interest, the subroutine's second parameter (an unsigned 8-bit value) is cleared to 0, and its first parameter (an 8-bit pointer to RAM) is passed to it via a register.[6]

```
                main.c: 231: OSCreateBinSem(&OSecbArea[(1-1)], 0);
03B6  01AF                   clrf    ?_OSCreateBinSem
03B7  30AA                   movlw   OSecbArea
03B8  120A  158A  2668       fcall   _OSCreateBinSem
```

Listing 2: Example of Stackless Parameter Passing

For functions called from interrupts, a separate area in RAM is dedicated for parameters, etc., since these functions can execute at any time. Again, the compiler analyzes the call graphs for the opportunity to overlay in RAM. The map file (not shown) reveals that `OSCreateBinSem()`'s second parameter is overlayed with one of `OSReturnBinSem()`'s parameters and with `OSSignalBinSem()`'s lone auto variable – they all share the same location in RAM.

Applications with nested interrupts complicate things further by requiring even more dedicated RAM. All of this is handled automatically by the compiler, and the user is unaware that anything differs from a normal compiler … except in one important instance.

Be Afraid, Be Very Afraid

Imagine the situation with a stackless compiler where a function is called at both the mainline (background) and interrupt (foreground) levels. The compiler must decide where to locate and overlay the parameters, etc. in RAM. Since interrupts are involved, it will probably place them with those of other interrupt-level functions. This is all well and good, except that *the function's parameters* (and return values) *are no longer protected against corruption!*

Review Listing 2 carefully. If this code is called both at the mainline and interrupt levels, what will happen? If the interrupt happens after line 0x03B6 and before the mainline code enters `OSCreateBinSem()` and disables interrupts (typically 5-7 instruction cycles later), it will overwrite the function's second parameter.[7] This can have catastrophic results.

The solution, as the compiler vendors are careful to point out, is to *disable interrupts in mainline code before calling functions with multiple callgraphs*. An example is shown in Listing 3.

```
                main.c: 278: GIE = 0; /* disable interrupts */
03B5  138B                   bcf      11,7
                main.c: 279: OSCreateBinSem(&OSecbArea[(1-1)], 0);
03B6  01AF                   clrf     ?_OSCreateBinSem
03B7  30AA                   movlw    OSecbArea
03B8  120A  158A  2668       fcall    _OSCreateBinSem
                main.c: 280: GIE = 1; /* re-enable interrupts */
03BB  178B                   bsf      11,7
```

Listing 3: Protecting Parameters Passed without a Stack

Note that the issues of parameter passing and critical section protection are not directly related – even functions that have parameters but no critical section are affected. Disabling interrupts inside a function in order to protect a critical section is *too late* to protect the parameters of the function. Similarly, if interrupts are re-enabled inside the function after the critical section, return values[8] may be corrupted by the interrupt-level function. Therefore interrupt control must be done *outside* the function.

It would be desirable to hide the need for this external interrupt control from the Salvo user. One possibility would be to create macros for mainline code that would first disable interrupts, then execute the desired Salvo service, and then restore interrupts. Unfortunately this method *is incompatible with functions that have return values.*

Comparison of Methods

Table 1 lists the methods whereby Salvo controls interrupts for critical sections.

service is called from:	conventional compiler	stackless compiler
background only	inside	inside
foreground only	inside	inside
anywhere	inside	outside

Table 1: Location of RTOS Service Interrupt Control for Protecting Critical Region

Table 1 makes clear that the need to protect critical regions by external control of interrupts is required in only one situation. Salvo provides two macros – `OSProtect()` and `OSUnprotect()` – expressly for this purpose. Their use is illustrated in Listing 4 for a Salvo service called at the mainline (background) level. Services called from ISRs do not require these macros.

```
OSProtect();
OSSignalBinSem(BINSEM_TXBUFF_P);
OSUnprotect();
```

Listing 4: Protecting Salvo Services with Multiple Call Graphs

This protection is simply an external form of disabling and the restoring interrupts around a function with multiple call graphs. By using these macros with both conventional and stackless compilers, no changes to the source code are required when porting from one development environment to another.

Conclusion

An RTOS should support the calling of certain services from both mainline and interrupt levels. Stackless compilers require control of interrupts external to functions with multiple call graphs in order to avoid parameter corruption. Salvo's `OSProtect()` and `OSUnprotect()` macros must be used in these situations.

---

1 Assuming the compiler implements proper context save and restore for the interrupt in question.
2 With potentially disastrous results if it were to occur at the wrong time.
3 Believe it or not, there are useful microcontrollers with no interrupts, e.g. Microchip PIC12 PICmicro family.
4 With apologies to Wm. Shakespeare. Richard III, act 5, sc. 7.
5 `salvo\demo\d5\main.c` compiled for PIC16C77 PICmicro® MCU.
6 `W` (working) register. PIC16C77 is RISC-like.
7 Since the first parameter is passed in a register, and registers are preserved by the interrupt handler, only the second parameter will be affected.
8 Only those that are not passed in registers will be affected.