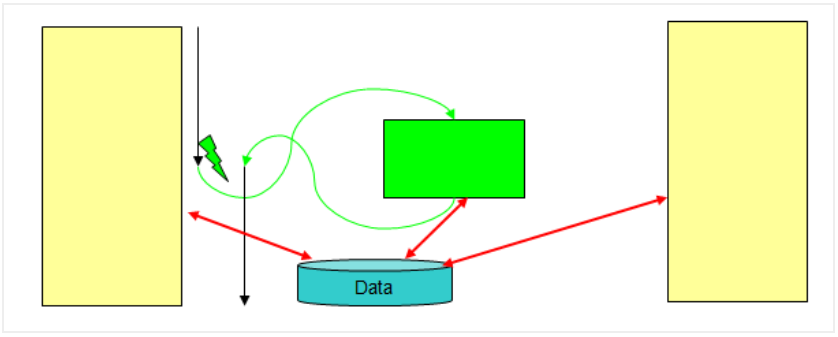
Advertisements

CLion SMART CROSS-PLA C/C++ IDE

EnterCritical() and ExitCritical(): Why Things are Failing Badly

Posted on January 26, 2014 by Erich Styger

I have carefully implemented my firmware. It works perfectly for hours, days, months, maybe for years. I there would not be a problem: the firmware crashes sporadically :-(. Yes, I'm using watchdogs to recover, but hey: **it is a serious problem**. And because it happens only under rare and special conditions, it is hard to track it down or to debug it.



— Accessing shared data from the main application and from an interrupt

The thing is: these nightmares exist, and they are real and nasty. I'm pushing my students hard on this topic: It is about **how to protect** critical sections. And what could go wrong. And here is just yet another example: how it can go badly wrong if you are not careful. And it took me a while too to realize where the problem is. It was not a fun ride….

A Problem?

I had occasional problems with the serial (RS-232) driver I'm using. That code is generated by Processor Expert, and served me very well. Except that I had rare cases where the internal data structure of the driver (a ring buffer) got corrupted. I was considered many causes, from stack overflows to wild pointers. But never was able to truly track it down. Until now.

Critical Sections

A 'critical section' is an area in the code where I need prevent two 'threads' accessing the same data in parallel. In 'parallel' might be code which is executed by the main program, but can be interrupt and accessed by another task or by an interrupt service routine. So I need to make sure that only only one program is executing that sequence. Or that this sequence is executed as such that it cannot be divided up (it needs to be executed in an **atomic** way).

Considering the following implementation of a ring buffer (from RingBuffer component, very similar to what the serial driver is using I mentioned at the beginning):

```
static Rx1_ElementType Rx1_buffer[Rx1_BUF_SIZE]; /* ring buffer */
static Rx1_BufSizeType Rx1_inIdx;  /* input index */
static Rx1_BufSizeType Rx1_outIdx; /* output index */
static Rx1_BufSizeType Rx1_inSize; /* size data in buffer */

byte Rx1_Put(Rx1_ElementType elem) {
  byte res = ERR_OK;

  if (Rx1_inSize==Rx1_BUF_SIZE) {
    res = ERR_TXFULL;
  } else {
    Rx1_buffer[Rx1_inIdx] = elem;
    Rx1_inSize++;
    Rx1_inIdx++;
    if (Rx1_inIdx==Rx1_BUF_SIZE) {
      Rx1_inIdx = 0;
    }
  }
  return res;
}
```

The code accesses global variables (the ring buffer and associated variables for it). Now assume that both my main program and an interrupt service routine can use `Rx1_Put()` to add an element to the same ring buffer. Then it could happen that after executing

```
Rx1_buffer[Rx1_inIdx] = elem;
```

the interrupt could happen, which would call `Rx1_Put()` too. The result would be that the ring buffer data structure would be corrupted.

What I need is to put a critical section in place to prevent that my 'critical' code does not get interrupted.

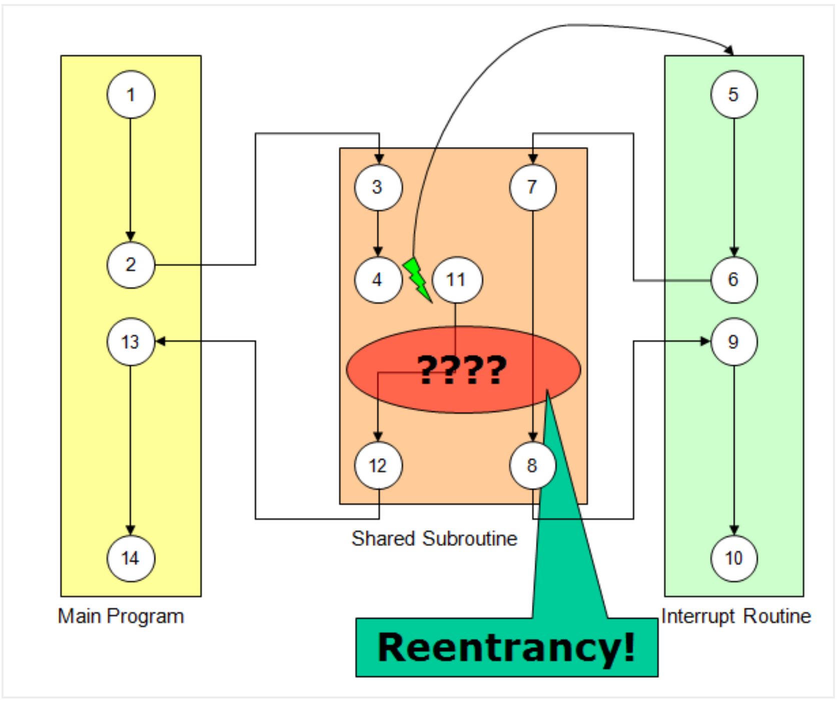
Reentrancy

This critical code section is related to the problem of **Reentrancy**: That different 'programs' can enter (re-enter) a function in such a way that they it does not create a problem. Reentrancy is a concern because

1. Interrupts can happen any time. As such, a running program can be interrupted any time.
2. Shared code (e.g. a function) which is used both from different places like threads or interrupts can be interrupted any time.
3. Access to data from such a function needs to be protected as such that the data integrity is kept.

As such, every shared code between interrupts routines and the main program (or tasks) needs to be re-entrant: that the function can be 'executed in parallel by multiple threads'.

Below is the flow of a program (main program) and an interrupt service routine which use the a shared subroutine:



— Shared Subroutine

In the above picture, there is an interrupt at step 4, causing the interrupt routine to enter the subroutine too. the problem is that every data access inside that subroutine needs to be reentrant.

I recommend to read the article "Reentrancy" by Jack Ganssle:

"A routine must satisfy the following conditions to be reentrant:

1. *It never modifies itself. That is, the instructions of the program are never changed. Period. Under any circumstances. Far too many embedded systems still violate this cardinal rule.*
2. *Any variables changed by the routine must be allocated to a particular "instance" of the function's invocation. Thus, if reentrant function FOO is called by three different functions, then FOO's data must be stored in three different areas of RAM."*
   (Source: http://www.ganssle.com/articles/areentra.htm)

My Ring Buffer uses shared *global* variables, but I cannot different instances (because they are really intended to be shared). So I need to protect the access to the shared variables with a critical section.

Disabling Interrupts

So what can I do? Well, the first thought could be: if the interrupt is the problem, I simply disable the interrupts for my critical section:

```
byte Rx1_Put(Rx1_ElementType elem) {
  byte res = ERR_OK;

  DisableInterrupts();
  if (Rx1_inSize==Rx1_BUF_SIZE) {
    res = ERR_TXFULL;
  } else {
    Rx1_buffer[Rx1_inIdx] = elem;
    Rx1_inSize++;
    Rx1_inIdx++;
    if (Rx1_inIdx==Rx1_BUF_SIZE) {
      Rx1_inIdx = 0;
    }
  }
  EnableInterrupts();
  return res;
}
```

For Disabling/Enabling interrupts Processor Expert generates **EnableInterrupts()** and **DisableInterrupts()** macros.

> ! *Disabling Interrupts increases the interrupt latency time. Means that during this time other (or higher) interrupt priorities are not served. For realtime applications this can be a problem, so keep the critical section as small and fast as possible.*

For the ARM Cortex-M0+ Processor Expert defines the macros as below:

```
/* Macro to enable all interrupts. */
#define EnableInterrupts asm ("CPSIE  i")

/* Macro to disable all interrupts. */
#define DisableInterrupts asm ("CPSID  i")
```

This solution is straight forward, and has several advantages:

- Only one assembly instruction: **fast** and **small** in code size
- **Atomic** operation

But there there is a problem with this approach too:

- It enables interrupts at the end of the critical section, regardless if they were enabled before or not: using such a critical section means that interrupts are always enabled 🙁

EnterCritical() and ExitCritical()

To solve this problem, Processor Expert provides a different set of macros: `EnterCritical()` and `ExitCritical()`. I could use like this to protect my critical section:

```
byte Rx1_Put(Rx1_ElementType elem) {
  byte res = ERR_OK;

  EnterCritical();
  if (Rx1_inSize==Rx1_BUF_SIZE) {
    res = ERR_TXFULL;
  } else {
    Rx1_buffer[Rx1_inIdx] = elem;
    Rx1_inSize++;
    Rx1_inIdx++;
    if (Rx1_inIdx==Rx1_BUF_SIZE) {
      Rx1_inIdx = 0;
    }
  }
  ExitCritical();
  return res;
}
```

The names of the macros tell the purpose: to create a critical section. So this is exactly what I need :-).

So what is behind `EnterCritical()` and `ExitCritical()`? I'm using the macros generated by Processor Expert, and they can be found in the generated Cpu.h. The macros are written in inline assembly as compiler and target microcontroller specific. Here is an example how they look for GNU ARM gcc compiler:

```
volatile uint8_t SR_reg;               /* Current value of the FAULTMASK register */
volatile uint8_t SR_lock = 0x00U;      /* Lock */

/* Save status register and disable interrupts */
#define EnterCritical() \
  do {\
    if (++SR_lock == 1u) {\
      /*lint -save  -e586 -e950 Disable MISRA rule (2.1,1.1) checking. */\
      asm ( \
      "MRS R0, PRIMASK\n\t" \
      "CPSID i\n\t"            \
      "STRB R0, %[output]"  \
      : [output] "=m" (SR_reg)\
      :: "r0");\
      /*lint -restore Enable MISRA rule (2.1,1.1) checking. */\
  }\
} while(0)

/* Restore status register  */
#define ExitCritical() \
  do {\
    if (--SR_lock == 0u) { \
      /*lint -save  -e586 -e950 Disable MISRA rule (2.1,1.1) checking. */\
      asm (                 \
      "ldrb r0, %[input]\n\t"\
      "msr PRIMASK,r0;\n\t" \
      ::[input] "m" (SR_reg)  \
      : "r0");                \
      /*lint -restore Enable MISRA rule (2.1,1.1) checking. */\
    }\
  } while(0)
```

The basic idea of above code is:

1. `EnterCritical()` increases the `SR_lock` variable (which is originally zero): if it gets the value 1, interrupts get disabled and earlier status gets stored in `SR_reg`
2. `ExitCritical()` decreases the `SR_lock` variable. If it gets zero, it restores the original interrupt state from `SR_reg`.

This deserves some further digging:

- First, I was really puzzled about that `do { ... } while(0)` construct. I used `{...}` constructs to have added local variables, but not with a `do-while`. The answer is here. 🙂
- The macros are using *global* variables `SR_reg` and `SR_lock`. This raises reentrancy concerns as every access to global variables from interrupts or the main program can cause reentrancy problems.

To the last point: The `SR_lock` intend is to prevent problems with nested `EnterCritical()` and `ExitCritical()`, such as:

```
void foo(void) {
  EnterCritical();
  ...
     EnterCritical();
     ....
     ExitCritical();
  ...
  ExitCritical();
}
```

Without the `SR_lock` counter, the nested `EnterCritical()` will overwrite the previous state, becaues `SR_reg` is a shared global variable 😟

> ! *Note that earlier versions of Processor Expert did *not* use that SR_lock variable.*

So far, so good. Unfortunately that source code generated by Processor Expert is **wrong!** :-(.

Bug Alarm!

Christian Jost, research and teaching assistant at my university, has pointed me to that problem, and he had found as well this thread in the Freescale Forum which is about exactly that problem, but no resolution :-(.

Two Threads

Let's assume that we have two threads accessing a shared resources. Both threads are protecting access to the shared resource with `EnterCritical()` and `EnterCritical()`:

```
void TreadA(void) {
  ...
  EnterCritical();
  /* critical section */
  ExitCritical();
  ...
}

void TreadB(void) {
  ...
  EnterCritical();
  /* critical section */
  ExitCritical();
  ...
}
```

It does not have to be threads (in an RTOS sense). One thread could be the main program, and the other thread could be an interrupt service routine.

Atomic or not?

The problem is that both

```
if (++SR_lock == 1u) {
```

and

```
if (--SR_lock == 0u) {
```

are ***not*** atomic! The assembly instructions behind it can be interrupted any time (by a thread or interrupt, at least with the code generated for gcc.

EnterCritical() and ExitCritical() for ARM Cortex M0+

Here is the code generated by GNU ARM gcc 4.7.3, with no optimizations set, for `EnterCritical()`. I have added comments if you are not familiar with ARM Cortex M0+ assembly code:

```
EnterCritical();
4:    4b12          ldr    r3, [pc, #72]    ; load address of SR_lock into R3
6:    781b          ldrb    r3, [r3, #0]    ; load value of SR_lock into R3
8:    b2db          uxtb    r3, r3          ; zero extend upper bytes to 32bits (0x000
a:    3301          adds    r3, #1          ; increment by one
c:    b2db          uxtb    r3, r3          ; zero extend again (cut off upper 24 bits
e:    4a10          ldr    r2, [pc, #64]    ; load address of SR_lock into R2
10:   1c19          adds    r1, r3, #0      ; copy r3 (incremented SR_lock value) to r
12:   7011          strb    r1, [r2, #0]    ; safe new SR_lock value
14:   2b01          cmp    r3, #1           ; is SR_lock==1?
16:   d104          bne.n    22             ; If SR_lock!=1, then continue after line
18:   4b0e          ldr    r3, [pc, #56]    ; SR_lock==1! load address of SR_reg into
1a:   f3ef 8010     mrs    r0, PRIMASK      ; load content of interrupt status registe
1e:   b672          cpsid    i              ; disable interrupts in PRIMASK. Interrupt
20:   7018          strb    r0, [r3, #0]    ; store earlier status of PRIMASK for rest
```

In pseudo code, this is what happens for `EnterCritical()`:

```
LOAD SR_lock -> reg
INC reg
STORE reg -> SR_lock
IF (reg==1)
 MOVE PRIMASK -> reg
 DISABLE INTERRUPTS
 STORE reg -> SR_reg
ENDIF
/* critical section starts here, interrupts shall be disabled here */
```

Notice that before DISABLE INTERRUPTS interrupts are still enabled. So the instructions before it could be interrupted.

The code for the `ExitCritical()` is very similar:

```
/* critical section here, interrupts are disabled */
LOAD SR_lock -> reg
DEC reg
```

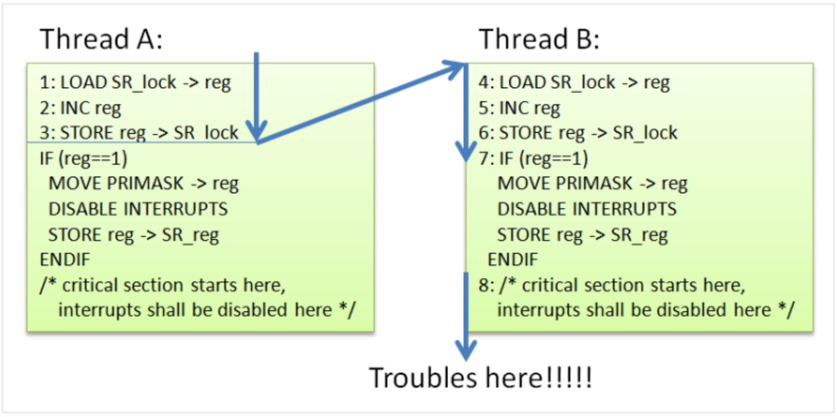
```
STORE reg -> SR_lock
IF (reg==0)
 LOAD SR_reg -> reg
 MOVE reg -> PRIMASK
ENDIF
/* end of critical section starts here */
```

If everything works as intended, then the SR_lock should be either 0 or 1.

Troubles here! EnterCritical() does not create always a Critical Section

My point is that if it can happen that `EnterCritical()` is executed, and that a critical section is NOT established.

Now consider two threads, each using `EnterCritical()` to protect their critical section:



```
Thread A:
1: LOAD SR_lock -> reg
2: INC reg
3: STORE reg -> SR_lock
IF (reg==1)
 MOVE PRIMASK -> reg
 DISABLE INTERRUPTS
 STORE reg -> SR_reg
ENDIF
/* critical section starts here,
  interrupts shall be disabled here */
```

```
Thread B:
4: LOAD SR_lock -> reg
5: INC reg
6: STORE reg -> SR_lock
7: IF (reg==1)
  MOVE PRIMASK -> reg
  DISABLE INTERRUPTS
  STORE reg -> SR_reg
 ENDIF
8: /* critical section starts here,
  interrupts shall be disabled here */
```

— Troubles here!!!!

1. At the beginning, SR_lock is zero. Thread A uses `EnterCritical()` to create a critical section. It load the SR_lock it into the register.
2. Thread A increments the value.
3. Thread A stores the incremented value. SR_lock is now 1 in memory. Now consider that just after this store there is an interrupt or thread switch.
4. Thread B executes, and uses itself `EnterCritical()` to start a critical section. It loads the value of SR_reg (which is now 1) into the register.
5. It increments the value in the register and it becomes 2.
6. It stores the value into memory, and SR_lock gets the value 2
7. Now it compares if the value is 1. It is not, so it assumes that interrupts are already disabled, and sips the interrupt disabling part
8. Thread B enters its critical section and assumes that the section is protected. **It is NOT**! Now any kind of bad things can happen. 😟

Test Program

To prove my point, I have created a simple test program, running on an ARM Cortex-M0+. For this, I have three parts:

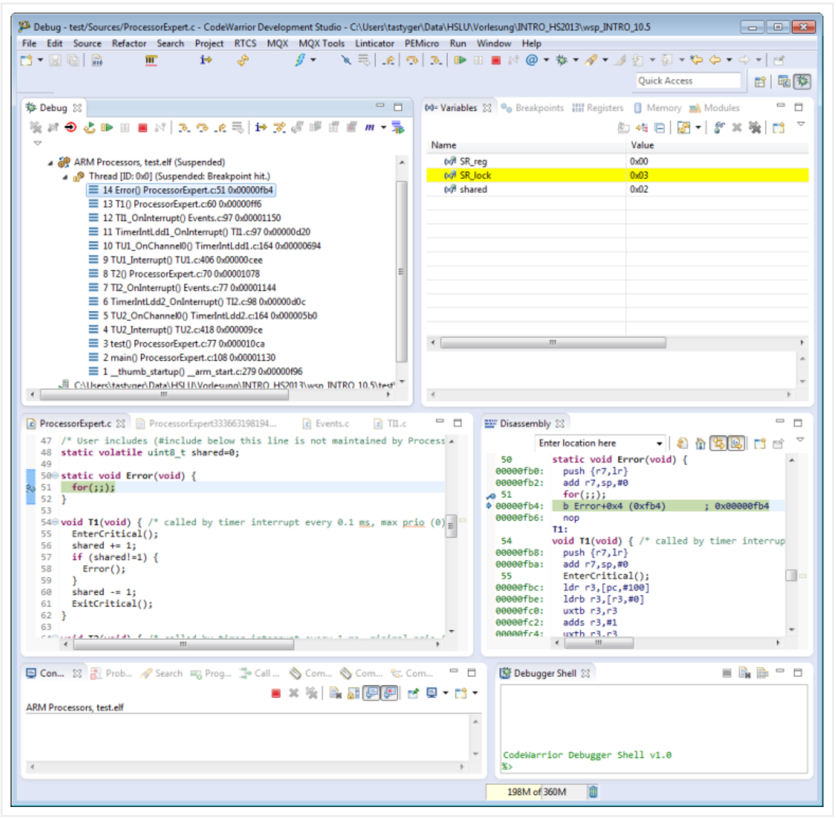
1. My 'main program', which runs from main() and calls test().
2. A periodic timer interrupt T2, called every 1 ms with an interrupt priority of 3 (lowest interrupt priority)
3. A periodic timer interrupt T1, called every 0.1 ms, with an interrupt priority of 0 (maximum interrupt priority)

With that priority setting, I expect that test() will get interrupted by T2 which then again can be interrupted by T1 (which has highest priority). test() gets called from main():

```
static volatile uint8_t shared=0;

static void Error(void) {
  for(;;);
}

void T1(void) { /* called by timer interrupt every 0.1 ms, max prio (0) */
  EnterCritical();
  shared += 1;
  if (shared!=1) {
    Error();
  }
  shared -= 1;
  ExitCritical();
}

void T2(void) { /* called by timer interrupt every 1 ms, minimal prio (3) */
  EnterCritical();
  shared += 1;
  if (shared!=1) {
    Error();
  }
  shared -= 1;
  ExitCritical();
}

static void test(void) {
for(;;) {
  EnterCritical();
  shared += 1;
  if (shared!=1) {
    Error();
  }
  shared -= 1;
  ExitCritical();
  }
}
```

Access to the `shared` variable is protected with `EnterCritical()` and `ExitCritical()` to create a critical section. Inside that critical section I access the `shared` variable, and if the variable is not 1, then clearly the critical section has been violated.

It does not need to run long:



Debug - test/Sources/ProcessorExpert.c - CodeWarrior Development Studio - C:\Users\tastyger\Data\HSLU\Vorlesung\INTRO_HS2013\wsp_INTRO_10.5

```
ARM Processors, test.elf (Suspended)
  Thread [ID: 0x0] (Suspended: Breakpoint hit.)
    14 Error() ProcessorExpert.c:51 0x00000fb4
    13 T1() ProcessorExpert.c:60 0x00000ff6
    12 TI1_OnInterrupt() Events.c:97 0x00001150
    11 TimerIntLdd1_OnInterrupt() TI1.c:97 0x00000d20
    10 TU1_OnChannel0() TimerIntLdd1.c:164 0x00000694
    9 TU1_Interrupt() TU1.c:406 0x00000cee
    8 T2() ProcessorExpert.c:70 0x00001078
    7 TI2_OnInterrupt() Events.c:77 0x00001144
    6 TimerIntLdd2_OnInterrupt() TI2.c:98 0x00000d0c
    5 TU2_OnChannel0() TimerIntLdd2.c:164 0x000005b0
    4 TU2_Interrupt() TU2.c:418 0x000009ce
    3 test() ProcessorExpert.c:77 0x000010ca
    2 main() ProcessorExpert.c:108 0x00001130
    1 __thumb_startup() __arm_start.c:279 0x00000f96
C:\Users\tastyger\Data\HSLU\Vorlesung\INTRO_HS2013\wsp_INTRO_10.5\test\
```

Name	Value
SR_reg	0x00
SR_lock	0x03
shared	0x02

```
47 /* User includes (#include below this line is not maintained by Process
48 static volatile uint8_t shared=0;
49
50 static void Error(void) {
51   for(;;);
52 }
53
54 void T1(void) { /* called by timer interrupt every 0.1 ms, max prio (0)
55   EnterCritical();
56   shared += 1;
57   if (shared!=1) {
58     Error();
59   }
60   shared -= 1;
61   ExitCritical();
62 }
63
```

```
 50          static void Error(void) {
00000fb0:      push {r7,lr}
00000fb2:      add r7,sp,#0
 51            for(;;);
00000fb4:      b Error+0x4 (0xfb4)        ; 0x00000fb4
00000fb6:      nop
             T1:
 54          void T1(void) { /* called by timer interrup
00000fb8:      push {r7,lr}
00000fba:      add r7,sp,#0
 55            EnterCritical();
00000fbc:      ldr r3,[pc,#100]
00000fbe:      ldrb r3,[r3,#0]
00000fc0:      uxtb r3,r3
00000fc2:      adds r3,#1
00000fc4:      uxth r3,r3
```

ARM Processors, test.elf

```
CodeWarrior Debugger Shell v1.0
%>
```

— Test Program shows failure

The stack trace shows my main program `test()` which gets interrupted by TU2 which itself gets interrupted by `TU1`: because the critical section is not working properly, `SR_lock` shows a value of `0x03` and the `shared` counter shows an illegal value of `0x02` :-(.

> ❓ *There might be still the (small?) chance that I'm using the EnterCritical() and ExitCritical() not as intended (Blame me on 'user error')? But looking at how it is used in the drivers provided by Processor Expert, I feel I did nothing wrong?*

Fixing the Problem: Using Local Critical Section Variable

Once knowing a problem, it is much easier to find a solution :-).

What I never liked with that problematic implementation of `EnterCritical()` and `ExitCritical()` is that they use **global** variables. And using global variables can create all kind of reentrancy problems. So a solution is to use local variables instead. For this the macros get rewritten using a local variable which gets defined with macro:

```
#define CpuCriticalVar()  uint8_t cpuSR

#define CpuEnterCritical()              \
  do {                                  \
    asm (                               \
    "MRS   R0, PRIMASK\n\t"             \
    "CPSID I\n\t"                       \
    "STRB R0, %[output]"                \
    : [output] "=m" (cpuSR) :: "r0");   \
  } while(0)

#define CpuExitCritical()               \
  do{                                   \
    asm (                               \
    "ldrb r0, %[input]\n\t"             \
    "msr PRIMASK,r0;\n\t"               \
    ::[input] "m" (cpuSR) : "r0");      \
  } while(0)
```

The local variable needs to be allocated first with `CpuCriticalVar()`. Below is an example usage:

```
void foo(void) {
  CpuCriticalVar(); /* definition of critical section variable */

  /* non-critcal section code */
  CpuEnterCritical();
  /* critical section here */
  CpuExitCritical();
  /* other code outside critical section */
}
```

If you are nesting critical sections, make sure every instance has its own critical section variable.

New Test Program

To test that new approach, I have updated the original program so I can easily switch between the original (failing) version and the new version.

```
#define CpuCriticalVar()  uint8_t cpuSR

#define CpuEnterCritical()              \
  do {                                  \
    asm (                               \
    "MRS   R0, PRIMASK\n\t"             \
    "CPSID I\n\t"                       \
    "STRB R0, %[output]"                \
    : [output] "=m" (cpuSR) :: "r0");   \
  } while(0)

#define CpuExitCritical()               \
  do{                                   \
    asm (                               \
    "ldrb r0, %[input]\n\t"             \
    "msr PRIMASK,r0;\n\t"               \
    ::[input] "m" (cpuSR) : "r0");      \
  } while(0)

#if 0 /* original, buggy version */
  #define DEFINE_CRITICAL() /* nothing */
  #define ENTER_CRITICAL()  EnterCritical()
  #define EXIT_CRITICAL()   ExitCritical()
#else /* fixed version using local variable */
  #define DEFINE_CRITICAL() CpuCriticalVar()
  #define ENTER_CRITICAL()  CpuEnterCritical()
  #define EXIT_CRITICAL()   CpuExitCritical()
#endif

static volatile uint8_t shared=0;

static void Error(void) {
 for(;;);
}

void T1(void) { /* called by timer interrupt every 0.1 ms, max prio (0) */
 DEFINE_CRITICAL();

 ENTER_CRITICAL();
 shared += 1;
 if (shared!=1) {
 Error();
 }
 shared -= 1;
 EXIT_CRITICAL();
}

void T2(void) { /* called by timer interrupt every 1 ms, minimal prio (3) */
 DEFINE_CRITICAL();

 ENTER_CRITICAL();
 shared += 1;
 if (shared!=1) {
 Error();
 }
 shared -= 1;
 EXIT_CRITICAL();
}

static void test(void) {
 DEFINE_CRITICAL();

 for(;;) {
 ENTER_CRITICAL();
 shared += 1;
 if (shared!=1) {
 Error();
 }
 shared -= 1;
 EXIT_CRITICAL();
 }
}
```

And the new version works :-).

Summary

The point here is not about Processor Expert generating wrong code. Or who can say his code has zero bugs? The point is that reentrancy and dealing with interrupts and caring about critical sections is ***critical***. If not, it can cause sporadic problems. Really nasty, hard to debug problems. I'm always amazed how sometimes code can be clearly wrong, but still it ***seems*** to work, only to fail when you really cannot afford have a failing system. Murphy? Maybe.

I'm using the original `EnterCritical()` and `ExitCritical()` in many places of my code, so I better change it now ;-). And if you are interested in a similar topic: read the article by Jack Ganssle about Great Watchdogs.

Happy Entering and Exiting 🙂

PS: Kudos to Christian Jost for implementing the improved EnterCritical() and ExitCritical() featured in this post!

Advertisements

Texas In

Referenzdesign zur Signalkonditionierung und Verarbeitung in AFCI-Schutzeinrichtung zur Erfas Lichtbogensignaturen

Design jetzt anzeigen

TIDesigns

Texas In

SHARE THIS:

Print | Email | Reddit | Twitter | Facebook 4 | LinkedIn 7 | Tumblr | Pinterest | Google | Pocket | WhatsApp | Telegram | Skype

Like

Be the first to like this.

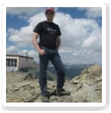
RELATED

Processor Expert, gcc C++ and Kinetis-L with MQXLite
In "Building"

DIY Free Toolchain for Kinetis: Part 4 – Processor Expert for Eclipse
In "Eclipse"

DIY Free Toolchain for Kinetis: Part 1 - GNU ARM Build Tools
In "Boards"

This entry was posted in **Building**, **Debugging**, **Embedded**, **gcc**, **Kinetis**, **Processor Expert**, **Thoughts**, **Tips & Tricks** and tagged **arm gcc**, **Assembly**, **global variables**, **gnu gcc**, **Processor Expert**, **software**, **Thoughts**, **Tips&Tricks** by **Erich Styger**. Bookmark the **permalink [https://mcuoneclipse.com/2014/01/26/entercritical-and-exitcritical-why-things-are-failing-badly/]** .



About Erich Styger

Embedded is my passion....
View all posts by Erich Styger →

48 THOUGHTS ON "ENTERCRITICAL() AND EXITCRITICAL(): WHY THINGS ARE FAILING BADLY"

skywalker2013
on **January 26, 2014 at 16:37** said:

From ARM Infocenter:

The DMB instruction should be used in Semaphore and Mutex operations.

Example 7 shows simple code for getting a lock. A DMB instruction is required after the lock is obtained.

Example 7. Simple code for getting a lock

```
void get_lock(volatile int *Lock_Variable)
{ // Note: __LDREXW and __STREXW are CMSIS functions
int status = 0;
do {
while (__LDREXW(&Lock_Variable) != 0); // Wait until
// Lock_Variable is free
status = __STREXW(1, &Lock_Variable); // Try to set
// Lock_Variable
} while (status!=0); //retry until lock successfully
__DMB(); // Do not start any other memory access
// until memory barrier is completed
return;
}
```

Similarly, code for releasing the lock should have a memory barrier at the beginning.

Example 8 shows simple code for releasing a lock. A DMB instruction is required before the lock is released.

Example 8. Simple code for releasing a lock

```
void free_lock(volatile int *Lock_Variable)
{ // Note: __LDREXW and __STREXW are CMSIS functions
__DMB(); // Ensure memory operations completed before
// releasing lock
Lock_Variable = 0;
return;
}
```

Greetings

★ Like

Laartoor
on **April 12, 2014 at 13:28** said:

On a Cortex-M processor, you will not very often need to use the barrier instructions (DMB, DSB, ISB). They are needed when external memory is changed by someone else than the processor, namely when DMA by-passes caches or in shared memory on a multiprocessor systems. With Kinetis CPUs, you have a very small instruction pipeline, and a 2 entry in-order memory load pipeline. Since three NOPs effectively act as a barrier, I am not ever sure you could write any normal code where you would need to use barriers. I guess you really need to do something stupid like running in RAM and modifying the next instruction to trigger the problem.

★ Like

Erich Styger
on **April 12, 2014 at 13:51** said:

Yes, agreed on the barrier instructions. On the other side, I started to add them just in case I move up to a device which actually needs them. I have now downloaded MCU10.6, and it looks like they have fixed that EnterCritical() and ExitCritical() problem, but I had not had a chance to verify it.

★ Like

Liviu Ionescu (ilg)

on **January 26, 2014 at 17:23** said:

You are generally right, critical sections should nest properly, which means simple CPSID/CPSIE are not enough.

In multi-threaded applications, especially when the critical section does not completely disables all interrupts, but leaves some high priority enabled, things might be even more complicated than you presented. There are several more requirements:

– priorities of all interrupt that trigger context changes or use synchronisation primitives must be chosen to ensure that they are disabled by the critical section
– the yield() function should not switch contexts when called from critical sections

This guarantee that critical sections are atomic for one thread.

In simple applications the safest way is to make critical sections disable all interrupts, which makes the first requirement easy to meet. However, in special applications, it might be needed to classify interrupts in groups (usually two groups are enough), one group with scheduler related interrupts (interrupts that are allowed to use system synchronisation primitives) and the second group with interrupts with higher priorities. In such cases there are two kinds of critical sections; entering a regular critical section will raise the priority to a defined value and entering the high priority critical section (sometimes also called Real Time) raises the priority to the limit. Exiting a critical section always restores the priority existing when the critical section was entered.

The simplest implementation of such critical sections uses local (stack allocated) locations to save the priority.

```
{
// Enter critical section
Mask_t mask = getPriorityMask();
setPriorityMask(mask | CRITICAL_SECTION_MASK);

…

// Exit critical section
setPriorityMask(mask);
}
```

Since I’m a strong C++ promoter, including in embedded applications, I will exemplify how critical sections can be defined and used in C++, using a model inspired from the RAII (Resource Acquisition Is Initialisation) paradigm

```
// Sample usage. The block parenthesis are mandatory.
{
CriticalSection cs;
// once this object is created, the critical section is automatically entered

…
// just before terminating the current block, the critical section is exited
}
```

where the class CriticalSection is defined like:

```
class CriticalSection
{
private:
Mask_t m_mask;

public:
CriticalSection();
~CriticalSection();
}

inline CriticalSection::CriticalSection()
{
m_mask = getPriorityMask();
setPriorityMask(m_mask | CRITICAL_SECTION_MASK);
}

inline CriticalSection::~CriticalSection()
{
setPriorityMask(m_mask);
}
```

For those not used with C++, the language ensures that the destructor is always called when the current block is terminated, so this method ensures that Enter() is always paired with Exit().

Which, you should admit, is quite convenient.

★ Like

Dusty
on **January 26, 2014 at 18:47** said:

The problem with SR_lock++ not being atomic is not a problem as long as EnterCritical() is only called in the foreground ie not from inside an interrupt handler.

The style of coding I use has one timer interrupt, at the minimum period as required to service all peripherals. Typically this will be 100us on down to 10us (depending on the processor and application). So as there is only one interrupt, there is no need for using disabling interrupts from inside interrupt handlers. This method makes embedded programs easy to debug and reliable.

The only reason I can see for using multiple interrupts from various peripherals is for low power operation ie battery. That is, assuming the application is a typical control application suitable for an M-series ARM chip or other such processor, in a control application.

From what I could see, using Processor Expert with a Kinetis M4 and M0+, their code relies heavily on interrupts, and so once you start using multiple Processor Expert modules, things will get messy fast. I ended up re-writing it all.

★ Like

Roberto Paolinelli
on **January 27, 2014 at 09:15** said:

Hi Erich,
I noticed the same problem and solved it rewriting Enter/Exit critical in asm.
In few words the key point is that until you write to a register you are fine because registers are saved by either CPU or compiler.
But when you store a value into a variable, that one will become the section to protect first, so store instruction must be executed with interrupts disabled.

Of course, as you said, if instruction set had been provided with INC memory_addr, no fault would have been occured.

So the conceptual issue is, IMHO Enter/Exit Critical can never be implemented in C code by processor expert, whose aim is to support a wide spectrum of CPUs.

Best Regards
Roberto

★ Like

Pingback: CriticalSection Component | MCU on Eclipse

Bob Paddock
on **March 2, 2014 at 14:41** said:

There is still the problem of Code Motion. An optimizing compiler is free to reorder the code. It could very well move code you think is after the Disable Interrupt instruction to before it. It is important to put in memory barriers that stop code motion across critical section points. Each compiler handles how to do this differently.

For GCC: # define atomic_full_barrier() __asm ("" ::: "memory")

Take a look at how this issues is handled in AVR-LibC:

http://www.nongnu.org/avr-libc/user-manual/group__util__atomic.html

http://www.scs.stanford.edu/histar/src/pkg/uclibc/include/atomic.h

★ Like

Erich Styger
on **March 2, 2014 at 15:06** said:

Yes, this is a very good point, thank you. I still need to wrap my head around how I can make it nice and clean that way. And you are correct with your note about the compiler. Most compilers use any assembly block/instruction as barrier too in order to avoid such problems.

★ Like

Bob Paddock
on **March 2, 2014 at 17:06** said:

Do take a look at how AVR-LibC handles it. No reason to reinvent the wheel.

★ Like

Andrea Collamati
on **March 28, 2014 at 09:06** said:

Hi Erich, we are working on a K21 microcontroller. Our application is using about 10 free rtos task. We have already updated the PE in order to use th CS component. Our code is using the TMOUT to evaluate timeouts in the GI2C. The TMOUT addtick is placed in the ApplicationTick event of free rtos. The generated code is using critical section in the TMOUT, Debug Uart and USB CDC serial component. CS Component is configured to use the "new" version not the original one of critical section.
In this setup we experimented "strange" timeout error in I2C READ of eeprom.
Then we decided to replace the CS component implementation with the FreeRTOS taskENTERCritical and taskEXITCritical, modifying by hand the CS1.h files.
The behaviour of the firmware changes and the error rate was lower.
Finally we also replaced by hand the EnterCritical and ExitCritical calls embedded in the Uart mode. After this "patch" we were able to run our test benchmark without error. The benchmark last about of 12 hours. The test continously reads 500 byte every 100 ms from eeprom and receive every 30 ms from the UART port.
I hope this information can be useful.
Regards
Andrea Collamati

★ Like

Erich Styger
on March 29, 2014 at 21:10 said:

Hi Andrea,
this is indeed interesting. As a result, I have commited an update of the CriticalSection component on GitHub which offers usage of taskENTER_CRITICAL() and taskEXIT_CRITICAL():
https://github.com/ErichStyger/mcuoneclipse/commit/94711daa167ac39b2b4264b3ef9361c45d622d5b
And yes, I had to do the same on my side: replacing calls in the AsynchroSerial component code to use my 'new' version.
However, I'm not sure why the RTOS enter/exitcritical had that effect. At which priority are the UART interrupts? And at which priority is the RTOS running? I ask because you are not allowed to use RTOS functions from interrupts above configMAX_SYSCALL_INTERRUPT_PRIORITY (see http://www.freertos.org/RTOS-Cortex-M3-M4.html). I have seen that violating that might cause subtle problems even after hours.

★ Like

Andrea collamati
on March 31, 2014 at 14:03 said:

Hi Erich, the problem is still there… It appeared again. Just to let you know how subtle is the problem, the bug appears or disappear if I change the files link order or if I add an instruction in section of code that is never reached…
Just tow question about Critical Section. There are some PE Code I use like TMOUT1_AddTick or TU3_ResetCounter that use Critical Section. In general I should pay attention to not use them in a ISR context right or not? For example Is it safe to call it in the FRTOS1_vApplicationTickHook that seems to be called from vPortTickHandler?
Thank you again.
Andrea

★ Like

Erich Styger
on March 31, 2014 at 15:03 said:

Hi Andrea,
my latest published Timeout (TMOUT1_*) component on GitHub already uses the new CriticalSection component.
In question is every code using the default EnterCritical() and ExitCritical() macros from Processor Expert.

★ Like

Andrea collamati
on April 1, 2014 at 09:17 said:

Hi Eric,
finally we found the reason of GI2C timeout. This is the actual implementation of wait interrupt in the ReadBlock routine

```
do { /* Wait until data is received */
isTimeout = TMOUT1_CounterExpired(timeout);
if (isTimeout) {
break; /* break while() */
}
} while (!GI2C_BUS0_deviceData.dataReceivedFlg);
TMOUT1_LeaveCounter(timeout);
```

```
if (isTimeout) {
res = ERR_BUSY;
break; /* break for(;;) */
}
break;
```

In our tasks configuration there were two tasks with high priority that were blocking the task using the GI2C ReadBlock routine. So the timeout expired but also the interrupt had been correctly set.
With the current code only the timeout condition is signalled. In my opinion the code should be changed in the following way

```
do { /* Wait until data is received */
isTimeout = TMOUT1_CounterExpired(timeout);
if (!GI2C_BUS0_deviceData.dataReceivedFlg && isTimeout) {
break; /* break while() */
}
} while (!GI2C_BUS0_deviceData.dataReceivedFlg);

TMOUT1_LeaveCounter(timeout);
if (!GI2C_BUS0_deviceData.dataReceivedFlg && isTimeout) {
res = ERR_BUSY;
break; /* break for(;;) */
}
break;
```

What do you think?

★ Like

stefano
on **March 31, 2014 at 11:14** said:

Hi, could you explain what means these assembler instruction lines:

```
"STRB R0, %[output]" \
: [output] "=m" (SR_reg)\
:: "r0");\
```

What is "output".
I'm not so skilled,…sorry.
thank you.

Stefano M.

★ Like

Erich Styger
on **March 31, 2014 at 13:54** said:

Hi Stefano,
have a look here: http://www.ethernut.de/en/documents/arm-inline-asm.html

★ Like

stefano
on April 3, 2014 at 09:57 said:

Pardon for my ignorance, but I can't visualize the problem in the "real world". If my ISR gets the control means the interrupts were enabled, so at the end of my ISR I must enable "global" interrupt,…because we are talking about "global interrupt", is'n it? If is part of the main to disable and to enable global interrupts to protect part of the code, what is the problem?… specific interrupts are enabled or disabled individually. I know I'm neglecting something,…but what?

Thank you, Erich

★ Like

Erich Styger
on April 3, 2014 at 11:06 said:

Hi Stefano,
the issue is that there can be a race condition: that global variable to count the critial section in the original implementation is not reentrant and not properly protected. It is a very common problem if things are not carefully done. The bad thing is that it only happens under certain circumstances and with special timing. Murphy
🙁

★ Like

Pingback: Processor Expert (Driver Suite/Plugins/KDS) V10.4 with new Component Inspector | MCU on Eclipse

Pingback: Processor Expert Events | MCU on Eclipse

Pingback: NVIC: Disabling Interrupts on ARM Cortex-M and the Need for a Memory Barrier Instruction | MCU on Eclipse

Dorian
on May 30, 2016 at 11:08 said:

Hello Erich
i have a question regarding the "fixed" EnterCritical() code.
#define CpuCriticalVar() uint8_t cpuSR
#define CpuEnterCritical() \
do { \
asm (\
"MRS R0, PRIMASK\n\t" \
"CPSID I\n\t" \
"STRB R0, %[output]" \
…..
Imagine the following scenario:
EnterCritical() gets interrupted after storing PRIMASK into R0,
then the ISR modifies R0 and exits (nothing prevents R0 from beeing modified, or am i wrong?),
then EnterCritical() continues with disabling Interrupts and storing R0 (now with a wrong value) into cpuSR.

Could this scenario not disable all my interrupts and stop my program from working correctly?

PS: assuming this code is implemented in the current version of PE. If not, i'm still interested in the theoretical behaviour.

Best regards,
Dorian

★ Like

Erich Styger
on **May 30, 2016 at 15:05** said:

Hi Dorian,
the ISR has to save/restore the registers modified inside the ISR, and the hardware will push/pop some of the registers anyway. So there is no issue if there is an interrupt just after the MSR instruction.
Erich

★ Like

Dorian
on **May 30, 2016 at 16:19** said:

Hi Erich
ok, i understand. Thank you very much for the explanation.
Dorian

★ Like

edogaldo
on **July 22, 2016 at 00:50** said:

Hello Erich, I see you still reply so a question: I don't understand the need to manage SR_lock in EnterCritical() and Exit Critical().
Wouldn't it be enough to define them like this?

```
volatile uint8_t SR_reg; /* Current value of the FAULTMASK register */

/* Save status register and disable interrupts */
#define EnterCritical() \
do {\
/*lint -save -e586 -e950 Disable MISRA rule (2.1,1.1) checking. */\
asm ( \
“MRS R0, PRIMASK\n\t” \
“CPSID i\n\t” \
“STRB R0, %[output]” \
: [output] “=m” (SR_reg)\
:: “r0”);\
/*lint -restore Enable MISRA rule (2.1,1.1) checking. */\
} while(0)

/* Restore status register */
#define ExitCritical() \
do {\
/*lint -save -e586 -e950 Disable MISRA rule (2.1,1.1) checking. */\
asm ( \
“ldrb r0, %[input]\n\t”\
“msr PRIMASK,r0;\n\t” \
::[input] “m” (SR_reg) \
: “r0”); \
/*lint -restore Enable MISRA rule (2.1,1.1) checking. */\
} while(0)
```

Thanks and bye, E.

★ Like

Erich Styger
on **July 22, 2016 at 14:45** said:

The reason for the SR_lock is simply to have a (local!) variable to store the status. In order to deal with different implementation, the local variable is allocated with a macro.

★ Like

edogaldo
on **July 22, 2016 at 15:34** said:

Maybe I misunderstood the logic: I was referring to variable “SR_lock”, not to variable “SR_reg”; isn’t the latter that stores the status?!

★ Like

Erich Styger
on **July 22, 2016 at 17:31** said:

Yes, SR_reg is used to store the status. SR_lock is used to allow recursive call of the critical section code.

★ Like

Liviu Ionescu (ilg)
on **July 22, 2016 at 11:44** said:

How about this one:

```
inline uint32_t
__attribute__((always_inline))
critical_section_enter (void)
{
uint32_t status;
status = __get_PRIMASK ();
__disable_irq ();
return status;
}

inline void
__attribute__((always_inline))
critical_section_exit (uint32_t status)
{
__set_PRIMASK (status);
}
```

This is a simplified version of the code used in µOS++ IIIe / CMSIS++ (http://micro-os-plus.github.io); the actual code can also use BASEPRI, when available/needed.

★ Like

edogaldo
on **July 22, 2016 at 13:05** said:

Also, why not pushing PRIMASK in the stack instead of writing it into a variable?!
Something like:
enter:
mrs r0, PRIMASK;
cpsid i;
push {r0}
exit:
pop {r0};
msr primask, r0

also this is nice:
static __inline__ uint8_t __iCliRetVal(void)
{
asm volatile("cpsid i");
return 1;
}
static __inline__ void __iRestore(const uint32_t *__s)
{
__set_PRIMASK(*__s);
__asm__ volatile ("" ::: "memory");
}

#define ATOMIC_BLOCK() for (uint32_t PRIMASK_save \
__attribute__((__cleanup__(__iRestore))) = __get_PRIMASK(), __ToDo = __iCliRetVal(); \
__ToDo ; __ToDo = 0)

usage:
ATOMIC_BLOCK() {
instruction 1;
…
instruction n;
}

(inspired by AVR's util/atomic.h).

★ Like

Liviu Ionescu (ilg)
on **July 22, 2016 at 13:20** said:

> why not pushing PRIMASK in the stack

some time ago I also considered a similar solution, but I abandoned it after I encountered compilers that do not cope very well with assembly push/pop in the middle of regular C code.

although it looks more tedious, the safest and most portable solution to implement critical sections is with explicit save/restore:

```
{
uint32_t ics = critical_section_enter();
// ...
critical_section_exit(ics);
}
```

in C++, with the RIAA pattern to encapsulate the enter/exit in constructor/destructor, the use case is:

```
{
interrupts::critical_section ics;

// ...
}
```

(actual code from µOS++ IIIe / CMSIS++)

another advantage of this solution is that it does not use macros at all. macros are evil and must be avoided.

★ Like

Erich Styger
on **July 22, 2016 at 14:47** said:

About macros: as with everything, if used wisely it is not evil. I'm using macros for this Enter and Exit-Critical because I have to use my code with many, many different CPU architectures (and ARM is only one). So using macros I can hide the machine dependent stuff nicely in a mostly portable way. C++ is not a option on small microcontroller targets (say 1 KByte of RAM and 8 KByte of Flash).

★ Like

Liviu Ionescu (ilg)
on **July 22, 2016 at 13:24** said:

RIAA -> RAII (https://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization).

★ Like

Liviu Ionescu (ilg)
on **July 22, 2016 at 15:16** said:

> C89 only

ah, ok. 😉

Flintstones. Meet the Flintstones
They're the modern stone age family…

★ Like

Erich Styger
on **July 23, 2016 at 15:03** said:

I have projects which have to be maintained and be alive for 20 years. This is not about stone age, this is about long term availability, safety and secure systems. Not one of this 'modern' consumer systems which are obsolute already at the time they are out in the market 😉

★ Like

Liviu Ionescu (ilg)
on **July 23, 2016 at 17:26** said:

> projects which have to be maintained and be alive for 20 years.

I understand maintaining old projects for 20 years.

I don't understand using '89 technology in 2016 (2016-1989=27) for new projects that need to be maintained 20 more years.

★ Like

Erich Styger
on **July 23, 2016 at 17:30** said:

Because that compiler for that architecture was created before 1999 (C99). That project has been deployed to the field in 1998-2004. You would be surprised to see how many electronics around you are built with this technology.

★ Like

Liviu Ionescu (ilg)
on **July 22, 2016 at 14:59** said:

> using macros I can hide the machine dependent stuff nicely

did you evaluate using inlined functions?

> C++ is not a option on small microcontroller targets

I just completed the implementation of the third edition of a C++ RTOS. to quote you, "used wisely", there is no overhead, so, on a given set of resources, what you can do in C, you can also do it in C++, just much better structured.

★ Like

Erich Styger
on **July 22, 2016 at 15:01** said:

inlined functions does not exist in C89, and many of my older designs have compilers which support C89 only.
Additionally, these designs only support EC++, so again C++ is not an option.

★ Like

Juan A Luna
on **January 3, 2017 at 09:47** said:

Hi Erich,

does this article applies also to HCS08 mcu? I have created a test project for MC9S08PT16 (I am using CW10.7) and the code generated by PE EnterCritical/ExitCritical default macros is the same than the generated by your component CS1 with PE “Use PE Default” set to “No”

And thanks you for your always interesting articles!

Regards,
Juan

★ Like

Erich Styger
on **January 3, 2017 at 10:40** said:

Hi Juan,
this article only applies to ARM Cortex-M, not the other cores. That’s why (for compatiblity reasons) the component is using the normal Processor Expert Enter/ExitCritical() functions.
The latest Processor Expert drop has fixed the problem discussed in the article, but with a slightly different approach, so you can use one or the other.

Thanks,
Erich

★ Like

Greg
on February 22, 2017 at 15:56 said:

Erich

Is it possible to use those macros with Cortex-M0 (without "plus)?
I have tried check and seems it is the same interrupt model but I am looking confirmation.

Regards
/Greg

★ Like

Erich Styger
on February 22, 2017 at 17:08 said:

Hi Greg,
yes, the M0 and M0+ are very similar (I think only some better low power and some internal pipeline changes), so the same macros apply.
Erich

★ Like

Greg
on February 23, 2017 at 07:56 said:

many thanks

★ Like

Yilmaz KıRCICEK
on November 6, 2017 at 08:45 said:

Hi All,

We are using 10.4 version of PE and problem related with Critical Sections are still on the stage.
In some rare case, we observe that main loop is still running but no any interrupt is serving, debugged the SR_reg and SR_lock and values were corrupted. (ie arbitrary values) What is weird; SR_reg is holding the shadow of the PRIMASK and it can only be '0' or '1' but was NOT !

One of the interrupt source is EXT and if I increase the ext. interrupt trigger signal frequency, fail rate increases dramatically. (0 to 2 KHz)

Now we are digging in to detail but could not find the root cause yet and just want to share if there was any one experiencing the same problem. I will be here again at the end to give a feedback.

Regards.

★ Like

Erich Styger
on November 6, 2017 at 09:48 said:

Hi Yilmaz,
could it be that you face a stack overflow problem, that your interrupt routine or something else is overwriting

that memory area? I suggest you double the stack size to eliminate that possibility. Keep in mind that on the M4 the MSP is used for interrupts.
I hope this helps,
Erich

★ Like

Yilmaz KIRCICEK
on **November 6, 2017 at 10:23** said:

Hi Mr Styger,

We have not implement safety class functions (ie stack test ..) yet but I have kept your suggestion and its seem OK for 10 minutes. We need to obserbe a few couples of day. Thanks for your suggestion and time.

Best Regards.

★ Like