Efficiently ensure data consistency

Background, examples and HowTos

Version 4

Contents

- Motivation, examples for inconsistent data
- Data consistency with **single-core**
  - Protection mechanisms
  - Concepts removing the need for protection altogether

  Solutions S1..S5
- Data consistency with **multi-core**
  - Spinlocks
  - Concepts removing the need for protection altogether

  Solutions M1..M5
- Runtime consumption (= ‘costs’)
- Summary

Motivation, examples for inconsistent data

Example 1: calculating the absolute value

- Simple function:

x = | y |

- Simple implementation:

```
if (y<0)
  x = -y;
else
  x = y;
```

What can happen if code gets preempted here?

Example 1 compiled for Atmel AVR

```
volatile int y = -4;

int main(void)
{
    int x;

    if (y<0)
        x = -y;
    else
        x = y;

    return x;
}
```

Compiler

Assume code gets preempted here…

… and preempting code writes y=5

AVR: potential problem

```
main:
        lds r24,y
        lds r25,y+1
        tst r25
        brge .L2
        lds r24,y
        lds r25,y+1
        neg r25
        neg r24
        sbc r25,__zero_reg__
        std Y+2,r25
        std Y+1,r24
        rjmp .L3
.L2:
        lds r24,y
        lds r25,y+1
        std Y+2,r25
        std Y+1,r24
.L3:
        ldd r24,Y+1
        ldd r25,Y+2
```

Result: x = | y | = – 5

Example 1 compiled for TriCore (AURIX)

```c
volatile int y = -4;

int main(void)
{
    int x;

    if (y<0)
        x = -y;
    else
        x = y;

    return x;
}
```

Compiler

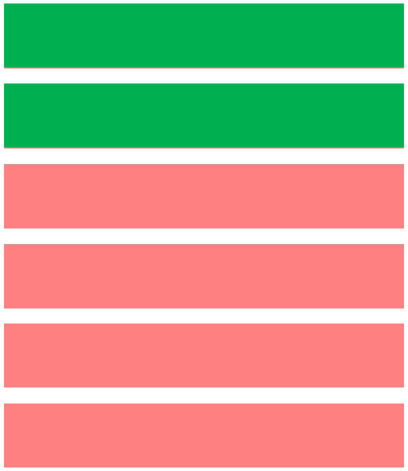
AURIX: no problem

```
main:   ld.w    d15,y
        abs     d2,d15
        ret
```

Example 2: accessing data structures

- Preemption while reading
- The preempting code updates the data structure.
- When the preempted code continues, it uses inconsistent data (partly old, partly new)

struct {



}

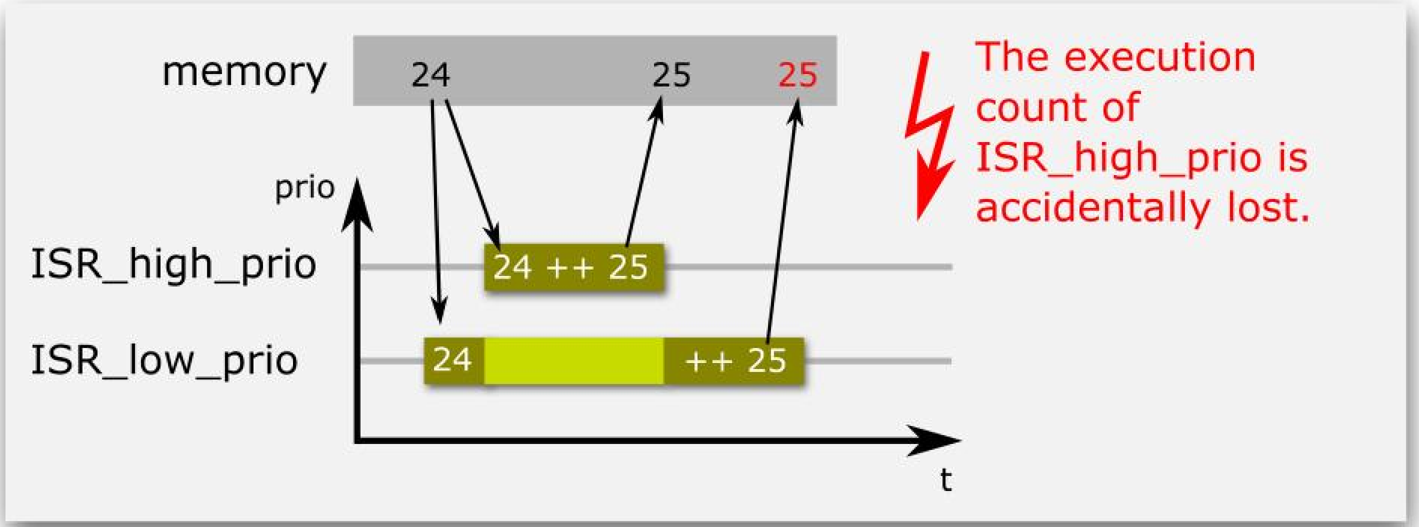
Example 3: count interrupts

- Let’s assume an application needs to know the total number of ISRs executed.

- Problem: sometimes the execution of `ISR_high_prio` is not considered.

```
unsigned int counterISR = 0;

void __interrupt(0x05) ISR_low_prio (void)
{
    _enable(); // globally enable interrupts
    counterISR++;
    DoSomething();
}

void __interrupt(0x30) ISR_high_prio (void)
{
    _enable(); // globally enable interrupts
    counterISR++;
    DoSomethingElse();
}
```

Example 3: how are interrupts ‘lost’?



Ensuring data consistency on single-core

Solution S1: suspending interrupts

- Introduce interrupt suspension:

```
  __disable();
  // Execute critical code with interrupts
disabled
  __enable();
```

- Advantage: easy to implement

- Problems
  - Critical sections often difficult to identify
  - Interrupts/tasks with higher priority get delayed.
  - Even interrupts/tasks which do not write to the critical data/resource get delayed.

Solution S2: Priority Ceiling Protocol

- Priority Ceiling Protocol: ‘suspension up to the lowest required prio’

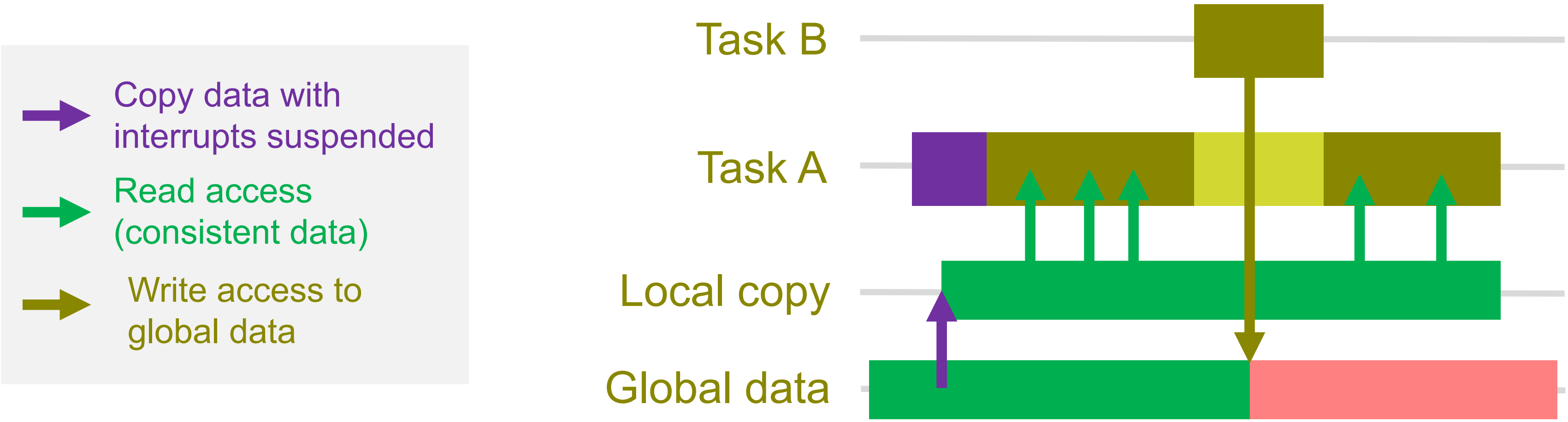
```
GetResource(myResource1);
// Execute critical code; the CPU runs with
// the (lowest possible) prio that ensures, no
// other code writing to myResource1 can preempt.
ReleaseResource(myResource1);
```

- Advantage compared to interrupt suspension: no delay of tasks/interrupts above the ‘*Ceiling Prio*’.

- Problems
  - All other problems stated for interrupt suspension remain.
  - OS or at least implementation of the Priority Ceiling Protocol required

Solution S3: use copies I

- At the beginning of each task, copies of all critical data are created.
  → 'critical' means here: data gets accessed from code with higher priority.
- The actual copy process is protected through interrupt suspension of PCP.
- The task uses the copy of the data only.



Solution S3: use copies II

- Typically implemented as some part of ‘all inclusive’ solution
  - Automated discovery of critical sections/data
  - Code generation of copies and copy routines

- Example: AUTOSAR RTE (**R**un-**T**ime **E**nvironment)

- Advantage: User does not need to bother about data-consistency

  ➔ As long as run-time consumption and RAM requirements are irrelevant.

- Problems
  - Often high costs by means of run-time and RAM
  - Increased latency due to protected copy routines

Solution S4: polling I

```
#include <avr/io.h>
#include <avr/interrupt.h>

void InitHardware(void)
{
    DDRB = (1<<PB0); /* pin connected to LED is output pin */

    /* initialize timer 1 */
    TCCR1B = (1<<CS11) | (1<<CS10); /* prescaler = clk/64 */
    TIMSK |= (1<<TOIE1); /* enable overflow interrupt */
}

ISR(TIMER1_OVF_vect) /* timer 1 overflow interrupt */
{
    PORTB ^= (1<<PB0); /* toggle LED */
    // DoSomePeriodicalStuff();
}

int main(void)
{
    InitHardware();
    sei(); /* globally enable interrupts */
    while(1) {
        // DoSomeBackgroundStuff();
    }
}
```

Solution S4: polling II

```
#include <avr/io.h>
#include <avr/interrupt.h>

void InitHardware(void)
{
    DDRB = (1<<PB0); /* pin connected to LED is output pin */

    /* initialize timer 1 */
    TCCR1B = (1<<CS11) | (1<<CS10); /* prescaler = clk/64 */
}

int main(void)
{
    InitHardware();
    while(1) {
        // DoSomeBackgroundStuff();
        if (TIFR & (1<<TOV1)) {
            TIFR |= (1<<TOV1); /* clear pending flag by
                                  writing a logical 1 */

            PORTB ^= (1<<PB0); /* toggle LED */
            // DoSomePeriodicalStuff();
        }
    }
}
```

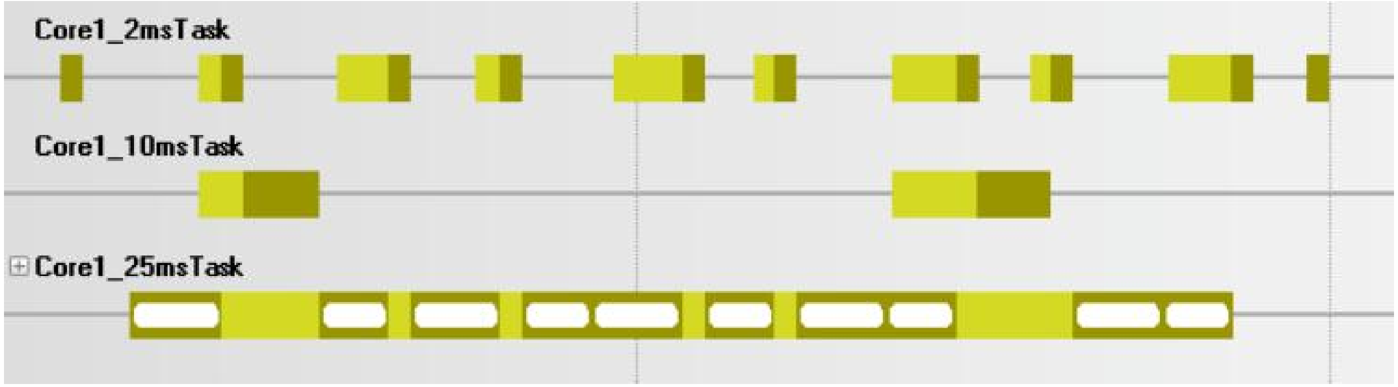
Implementation using **polling**

Solution S4: polling III

- Advantage: interruptions avoided altogether, data consistency conceptionally ensured
- Problems
  - Increased latency
  - Big (and thus risky) modification when introduced to existing software.

Solution S5: cooperative multitasking I

Core1_2ms_Task **is preemptive**



Core1_2ms_Task **is cooperative**

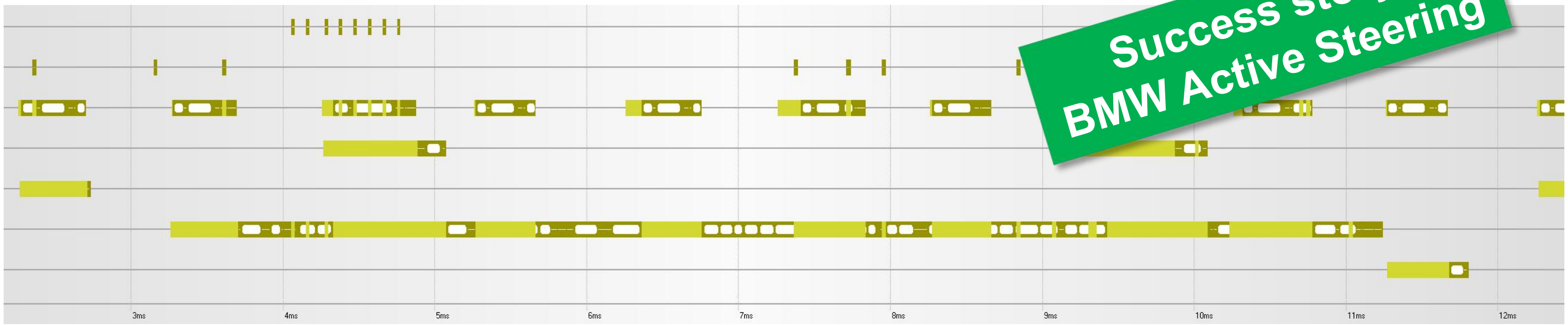
Solution S5: cooperative multitasking II

- (Cooperative) task switches are possible at ‘Schedule-Points’ only.
- Schedule-Points: call of OS service function `OS_Schedule( );`
- `OS_Schedule( )` call not necessarily required after each runnable/function

```
OS_TASK( Core1_25msTask )
{
    Core1_25msRunnable0( );
    OS_Schedule( );
    Core1_25msRunnable1( );
    OS_Schedule( );
    Core1_25msRunnable2( );
    OS_Schedule( );
    Core1_25msRunnable3( );
    OS_Schedule( );
    Core1_25msRunnable4( );
    OS_Schedule( );
    Core1_25msRunnable5( );
    OS_Schedule( );
    Core1_25msRunnable6( );
    OS_Schedule( );
    Core1_25msRunnable7( );
    OS_Schedule( );
    Core1_25msRunnable8( );
    OS_Schedule( );
    Core1_25msRunnable9( );
}
```

Solution S5: cooperative multitasking III

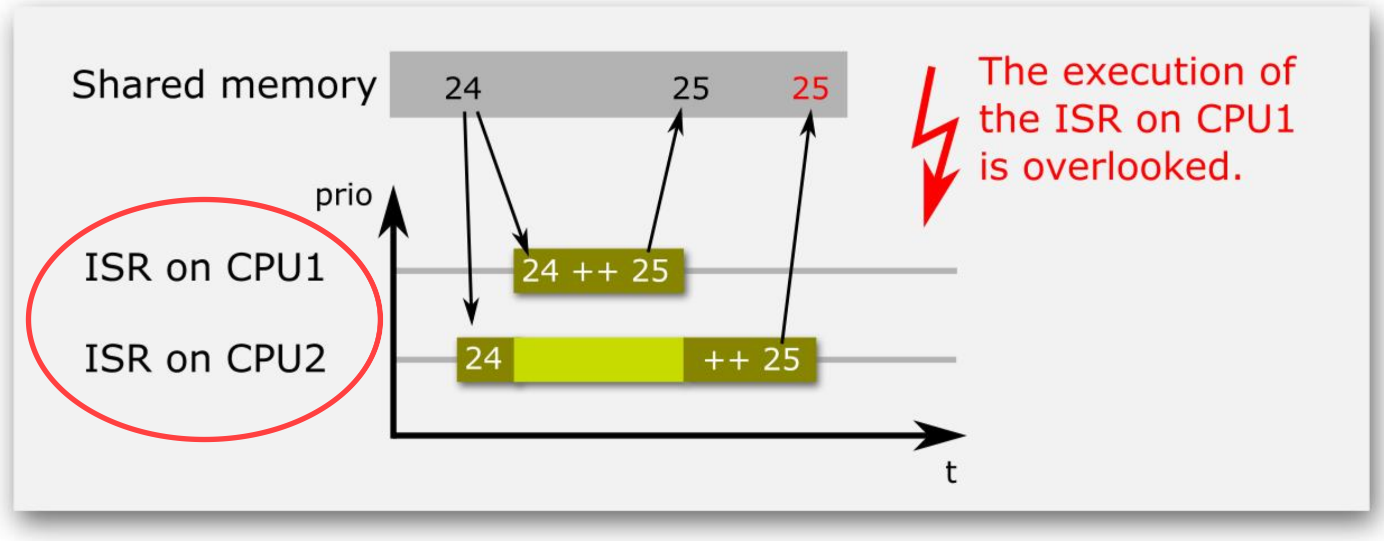
- Advantage
  - No preemption of runnables, i.e. application code, data consistency conceptionally ensured
  - RAM required for stack typically drastically reduced.
  - More efficient cache usage → run-time optimization
- Problem
  - Latency of tasks with higher priority depends on execution time of runnables in tasks with lower priority.



Success story:
BMW Active Steering

Ensuring data consistency on multi-core

Previous example 3 now with multi-core



- All previous approaches for ensuring data consistency do not work.

Solution M1: Spinlocks

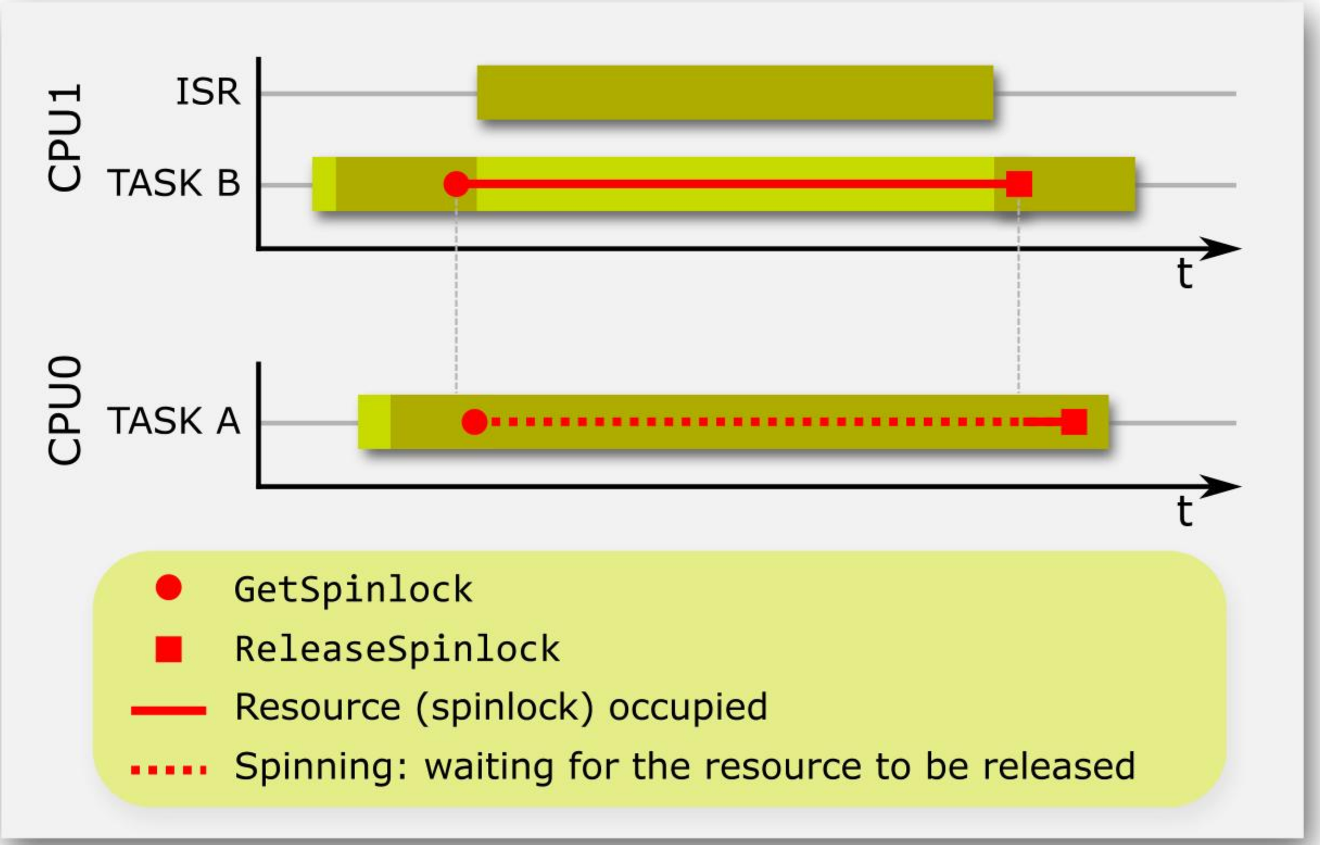
Interface (services)

```
StatusType GetSpinlock      ( SpinlockIdType SpinlockId      );

StatusType ReleaseSpinlock  ( SpinlockIdType SpinlockId      );

StatusType TryToGetSpinlock ( SpinlockIdType SpinlockId,
                              TryToGetSpinlockType* Success );
```

Usage

```
GetSpinlock(spinlock);
/* Execute critical code here */
ReleaseSpinlock(spinlock);
```

Problem #1 with `GetSpinlock`



Problem #2 with `GetSpinlock`

Pseudo solution

```
DisableAllInterrupts();
GetSpinlock(spinlock);
/* Execute critical code here */
ReleaseSpinlock(spinlock);
EnableAllInterrupts();
```

Using Spinlocks correctly and efficiently

```
TryToGetSpinlockType success;
DisableOSInterrupts( );
(void)TryToGetSpinlock( spinlock, &success );
while( TRYTOGETSPINLOCK_NOSUCCESS == success )
{
    EnableOSInterrupts( );
    /* Interrupts and high prio tasks can preempt here */
    DisableOSInterrupts( );
   (void)TryToGetSpinlock( spinlock, &success );
}
/* Execute critical code here */
ReleaseSpinlock( );
EnableOSInterrupts( );
```

The previously mentioned problems #1 and #2 are solved.

Solution M2: GLIWA double-buffer

- Idea: make any data structure behave like an atomic global variable
  - Writing overwrites old value
  - Reading gives you the last value written
- Accesses surrounded by `Get` and `Finish` methods.
- No blocking (interrupt locks or spinlocks) in *between* these methods!
- Very short sections of blocking *within* these methods

```
dataToprotect_t* pWr;

pWr = GetWrPtr();
If (NULL != pWr) {
    // write access, e.g.
    // pWr->data[0] = 'A';
    // pWr->data[1] = 'B';
    FinishWr();
}
```

How to write

```
dataToprotect_t* pRd;

pRd = GetRdPtr();
// read access, e.g.
// someVar  = pRd->data[0];
// otherVar = pRd->data[1];
FinishRd();
```

How to read

Free! Code openly available, ask us!

Solution M3: Lamport's bakery algorithm

- Cf. bakery: only one customer served at the counter. Others have to queue.
- Advantage: works without interrupt locks!

- https://en.wikipedia.org/wiki/Lamport%27s_bakery_algorithm
- https://www.geeksforgeeks.org/bakery-algorithm-in-process-synchronization/

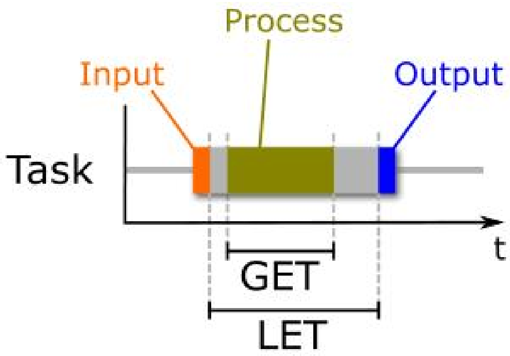
Compile with `gcc -pthread`

```
void lock(int thread)
{

    // Before getting the ticket number
    //"choosing" variable is set to be true
    choosing[thread] = 1;

    MEMBAR;
    // Memory barrier applied

    int max_ticket = 0;

    // Finding Maximum ticke
    for (int i = 0; i < THREA

        int ticket = tickets[
        max_ticket = ticket >
    }
    // Allotting a new ticket
    tickets[thread] = max_tic
    MEMBAR;
    choosing[thread] = 0;
    MEMBAR;
    // The ENTRY Section starts from here
    for (int other = 0; other < THREAD_COUNT; ++other) {

        // Applying the bakery algorithm conditions
        while (choosing[other]) {
        }

        MEMBAR;

        while (tickets[other] != 0 && (tickets[other]
                                        < tickets[thread]
                                     || (tickets[other]
                                            == tickets[thread]
                                        && other < thread))) {

        }
    }
}
```

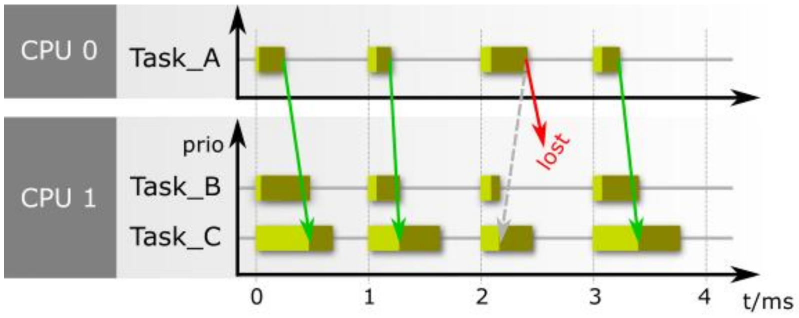
Image by dandelion_tea from Pixabay

Solution M4: Logical Execution Time (LET)

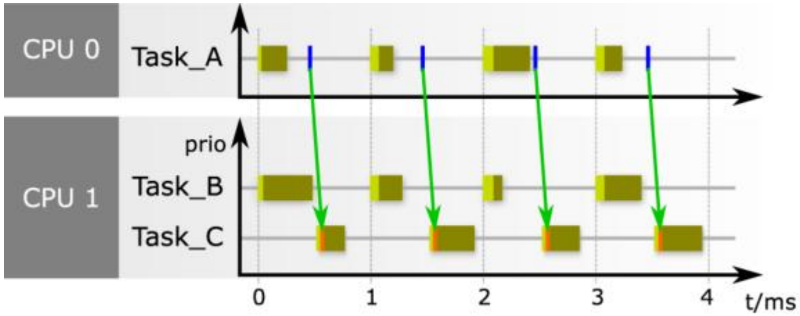
- **Idea:** reserve certain time slots on the time axis for sending/writing and for receiving/reading data.
  → writing and reading is decoupled by design, hence no further protection necessary



Example *without* LET (data loss)



Same example *with* LET (no data loss)

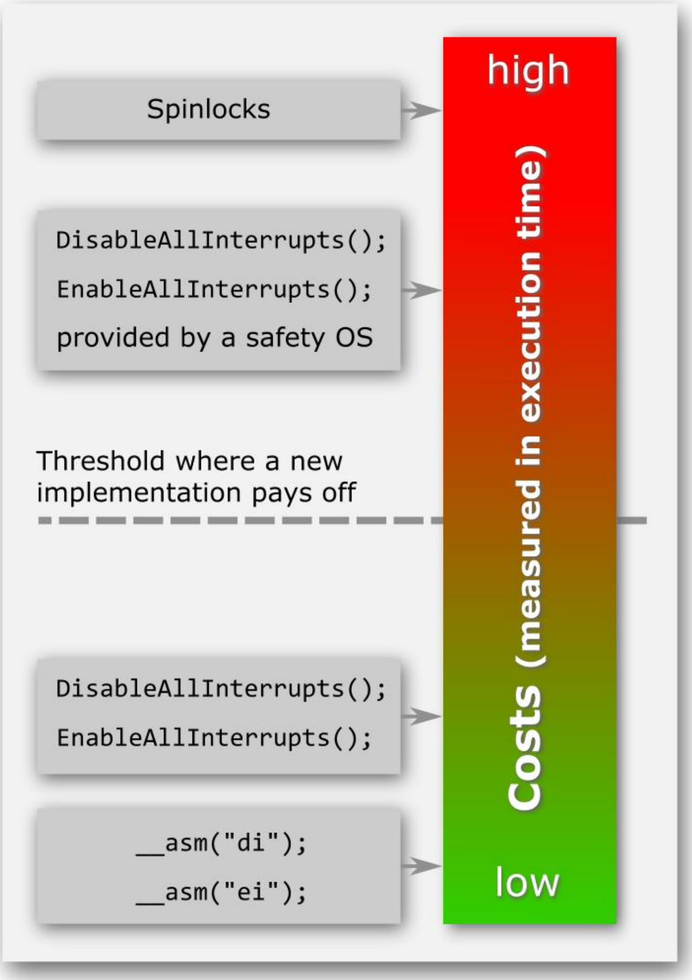


Solution M5: avoid the need for protection

```
unsigned int counterISR_low_prio = 0;
unsigned int counterISR_high_prio = 0;

void ISR_low_prio (void) __attribute__ ((signal,used));
void ISR_low_prio (void)
{
    _enable(); // globally enable interrupts
    counterISR_low_prio++;
    DoSomething();
}

void ISR_high_prio (void) __attribute__ ((signal,used));
void ISR_high_prio (void)
{
    _enable(); // globally enable interrupts
    counterISR_high_prio++;
    DoSomethingElse();
}

unsigned int GetCounterSum(void)
{
    return counterISR_low_prio + counterISR_high_prio;
}
```

“The best data protection mechanism is the one you do not need”

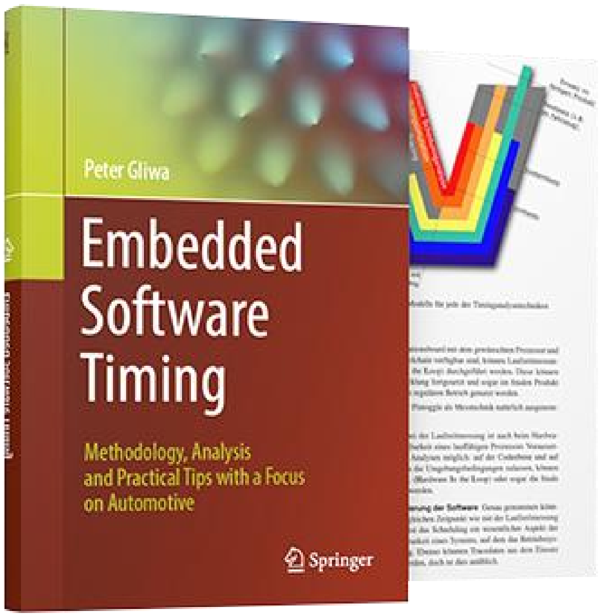
Run-time costs of data protection mechanisms



Summary

GLIWA
embedded systems

Book *Embedded Software Timing*



Contents

- Basics (Compilers, RTOSs, processors)
- Timing theory
- Timing analysis techniques
- Examples from automotive projects
- Timing optimization
- Multi-core, many-core
- AUTOSAR
- Safety, ISO 26262

Available in EN, DE, CN and KR

Summary

- Safe and reliable software only possible with consistent data

- General rules
  - Multiple readers are **not critical**
  - Nested reads are **not critical**
  - Multiple writers are typically a design flaw → **do not do it!**
  - Nested or simultaneous reading and writing are critical → **need protection**

My personal recommendation: use cooperative multitasking on single-core and LET on multi-core!

- All protection mechanisms come with advantages and disadvantages.
  - **For making the right choice, you need to know them!**
  - Knowing how your code generator (e.g. RTE) works carries great optimization potential.

- When using Spinlocks, apply the `TryToGetSpinlock` approach shown!

- The best data protection mechanism is the one you do not need


GLIWA
Peter Gliwa
Dipl.-Ing. (BA)
Geschäftsführer (CEO)
GLIWA GmbH embedded systems
Pollinger Str. 1
82362 Weilheim i.OB.
Germany
fon +49 - 881 - 13 85 22 - 10
fax +49 - 881 - 13 85 22 - 99
mobile +49 - 177 - 2 57 86 72
peter.gliwa@gliwa.com
www.gliwa.com

Thank you