



# Synchronization: Reentrancy & Interrupts

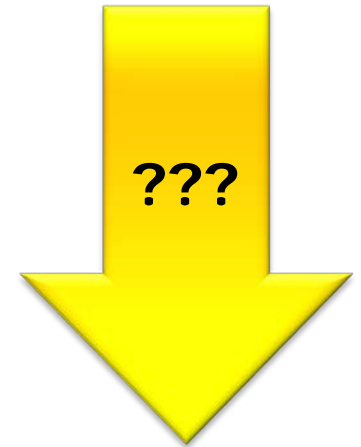
*„The thing with interrupts is bugging me.“*

Prof. Erich Styger  
[erich.styger@hslu.ch](mailto:erich.styger@hslu.ch)  
+41 41 349 33 01

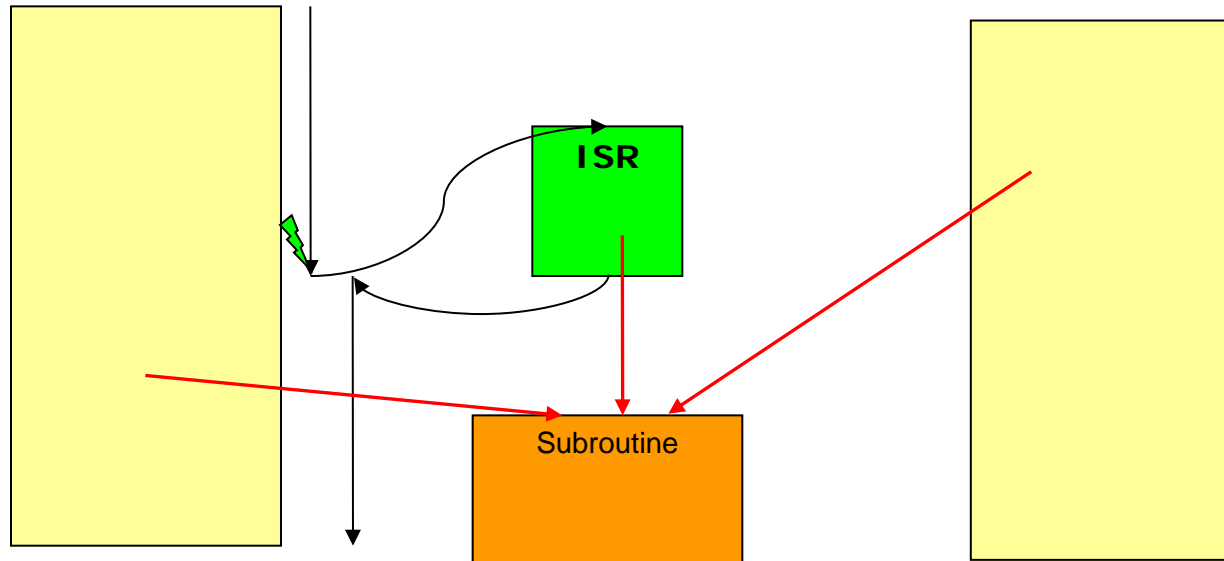
**Scriptum:  
Synchronization, Interrupts  
using C, Multiple Interrupts,  
Reentrancy, Interrupt Latency**

# Learning Goals

- Problem: Infrastructure for Synchronization: Reentrancy and Interrupts
- Common Subroutines
- Shared Data
- Interrupt Systems
- Reentrancy

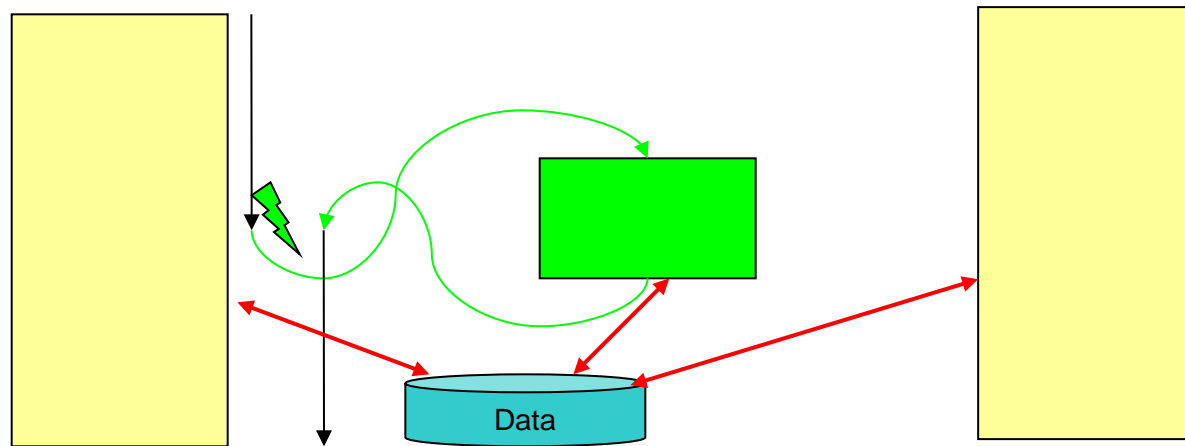


# Common Subroutines



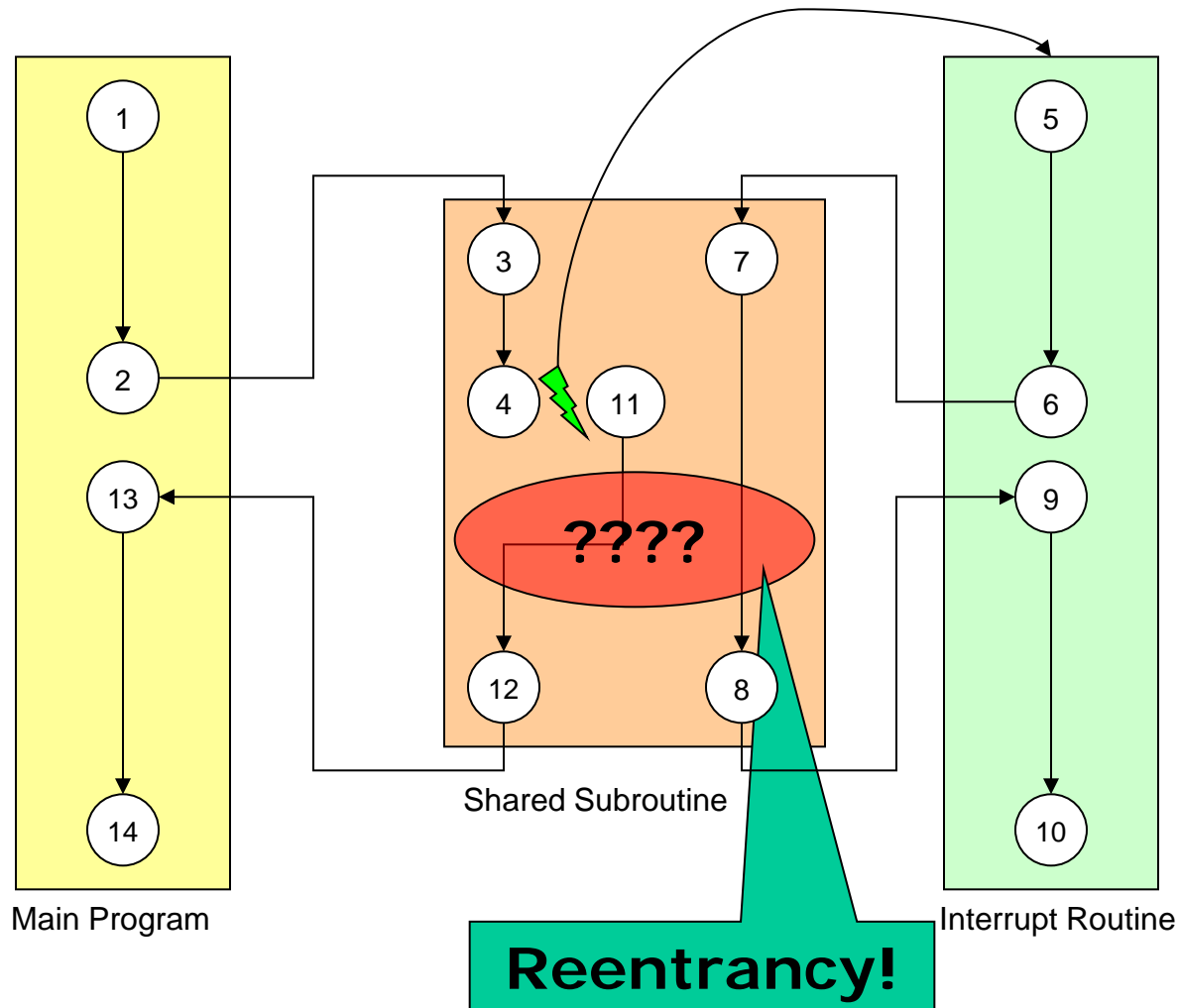
- Modularization
- Common functionality
- What happens with shared data?

# Shared Data



- Caution! Inconsistent states possible!
- Need to protect access to data!

# Common/Shared Subroutines



# Reentrancy

***An interrupt can happen **any time**.***

***As a consequence every subroutine  
(which can be called from an  
interrupt routine) has to be  
**reentrant**.***

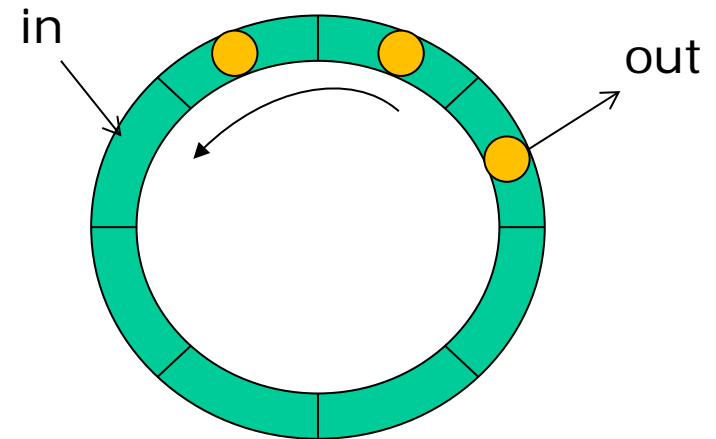
Read: <http://www.ganssle.com/articles/areentra.htm>

# Reentrancy

```
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 */
```

```
uint8_t Rx1_Put(Rx1_ElementType elem) {
    uint8_t 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;
}
```



# Priorities

- The main program can be interrupted any time
- Interrupt routines can be interrupted

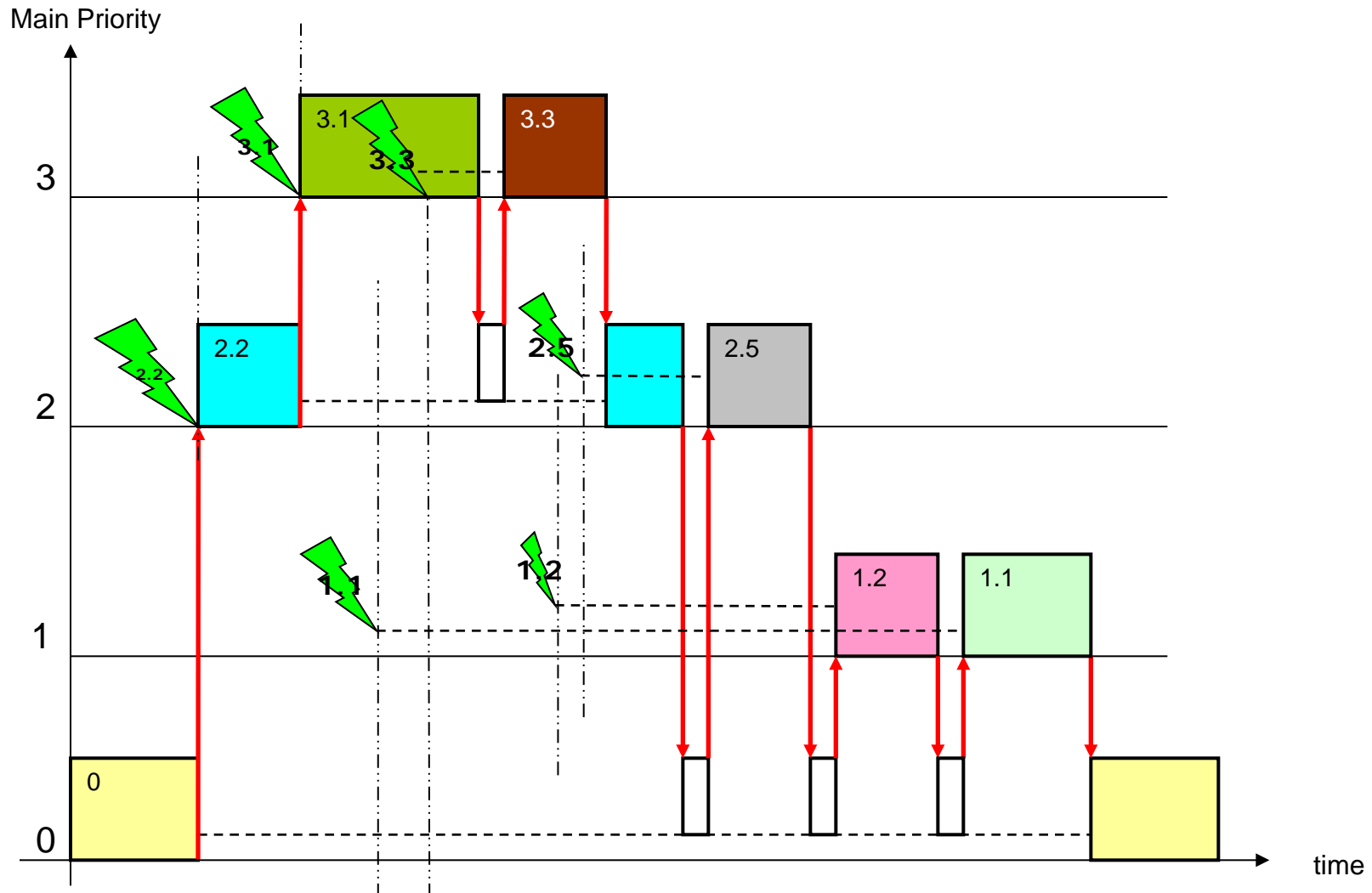




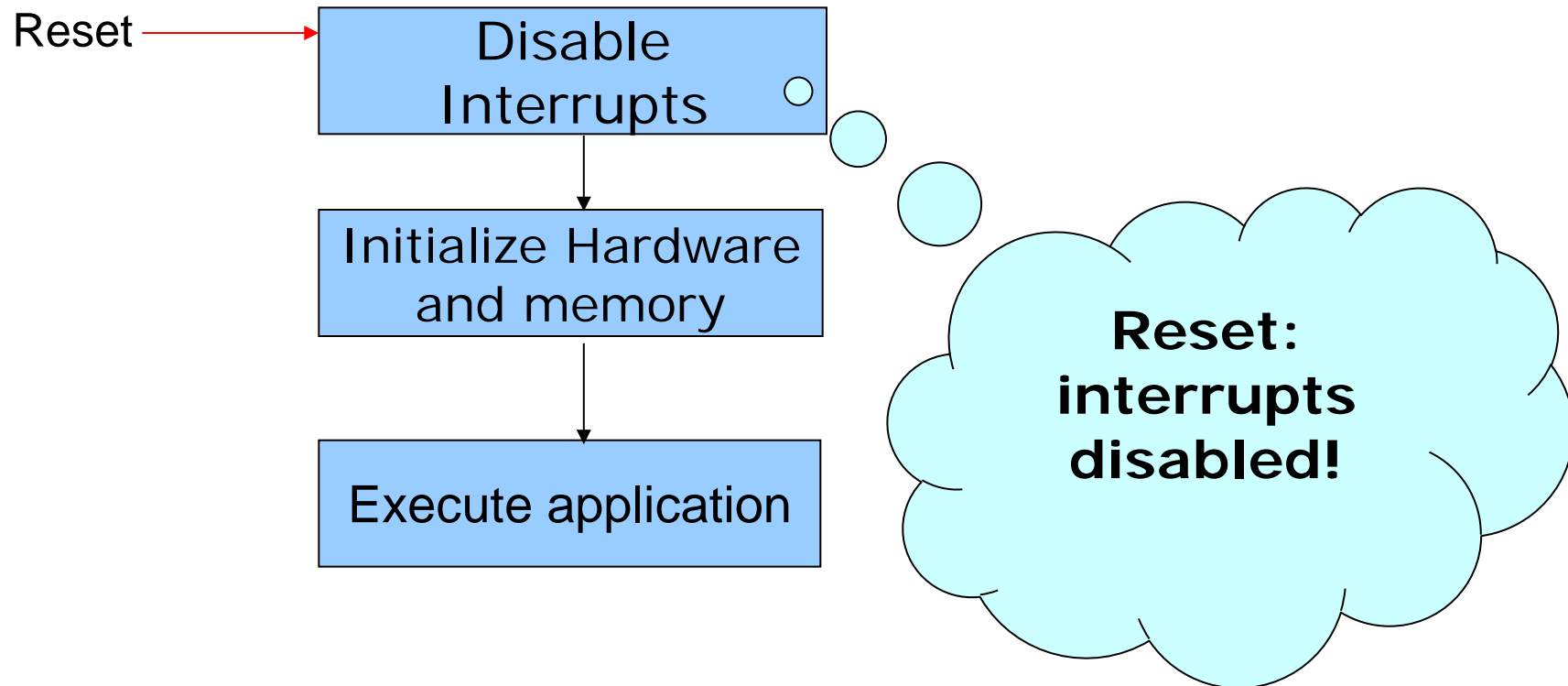
# Interruption Rules

- Priorities (Main- and Sub-Priorities)
  - Main program: Base Priority
  
  - *fn*: currently executed program
  - *in*: interrupted program
  - *MP*: Main Priority
  - *SP*: Sub Priority
  - *S*: Signal
  - *WS*: Waiting Signal
- 
1.  $MP(S) > MP(fn)$ : interrupt
  2.  $MP(S) \leq MP(fn) \Rightarrow S \rightarrow WS$
  3.  $SP(S) > SP(fn) \Rightarrow S \rightarrow WS$
  4.  $MAX(SP(WS)) \Rightarrow WS \rightarrow fn$
  5.  $MP(WS) > MP(in) \Rightarrow fn \rightarrow (in) \rightarrow WS \rightarrow fn$

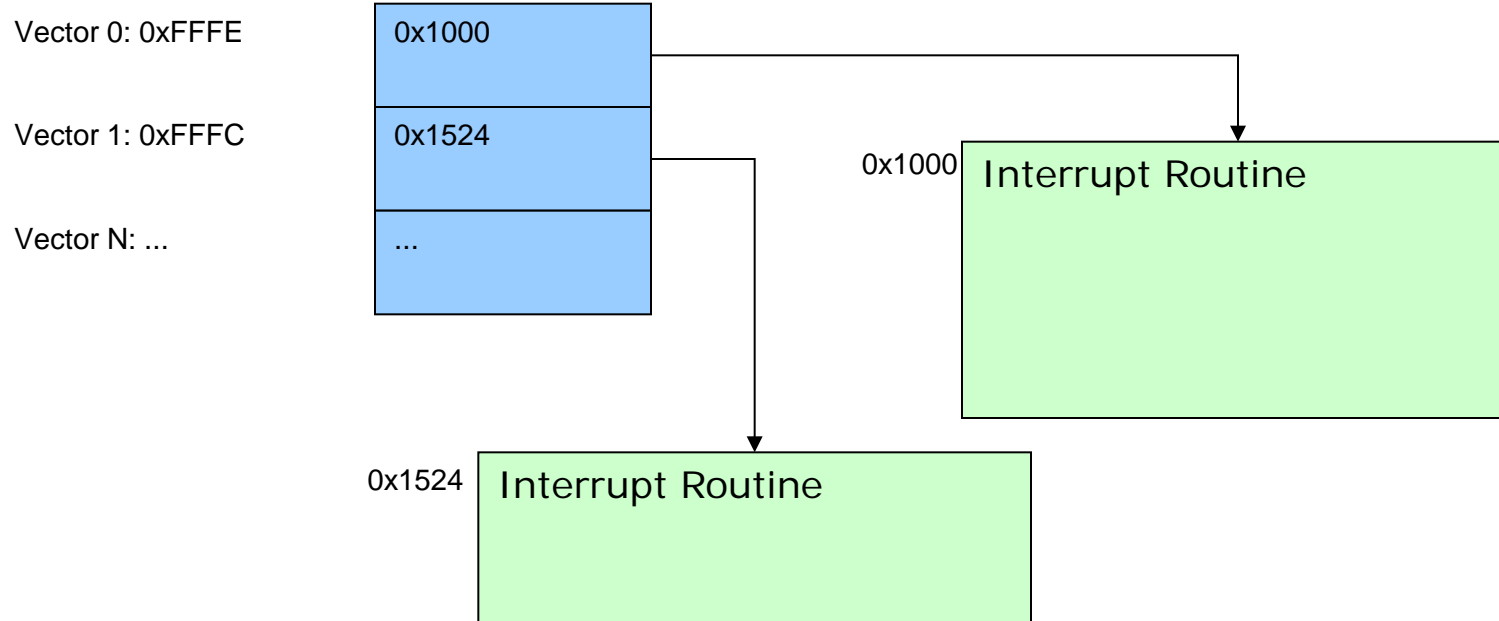
# Example



# Example System Startup



# Interrupts: Vector Table



- Memory
- Indirection
- Examples
  - S08 (needs RTI)
  - ARM Cortex (normal function)

# ARM GCC Inline Assembler Cookbook

```
__asm("mov r0,r0"); /* NOP */
```

```
__asm(  
    "mov r0,r0    \n\t"  
    "mov r0,r0    \n\t"  
    "mov r0,r0"  
);
```

```
__asm(code : output operand list : input operand list : clobber list);
```

```
uint8_t srcVar, dstVar;
```

```
__asm(  
    "ldrb r1, %[src] \n\t" /* load src into r1 */  
    "add r1,#1          \n\t" /* increment r1 */  
    "strb r1,%[dst] \n\t" /* store r1 in dst */  
    : [dst] "=m" (dstVar) /* result in memory */  
    : [src] "m" (srcVar)  /* src from memory */  
    : "r1"               /* clobber, changed register */  
);
```

<http://www.ethernut.de/en/documents/arm-inline-asm.html>

# Critical Sections

- Need for Mutual Exclusion!
- **DisableInt; {CS} EnableInt; → PRIMASK** | Bit Setting  
    \_\_asm volatile("cpsid i"); /\* set interrupt disable flag \*/  
    {Critical Section}  
    \_\_asm volatile("cpsie i"); /\* set interrupt enable flag \*/
- EnterCritical(); {CS} ExitCritical();  
    DisableInterruptsAndStoreCurrentInterruptStatus;  
    {Critical Section}  
    ReEnableInterruptsWithPreviousStatus;
- *TaskEnterCritical(); {CS}; TaskExitCritical();*
- *xSemaphoreTake(); {CS}; xSemaphoreGive();*



# Correct Critical Section?

- <http://mcuoneclipse.com/2014/01/26/entercritical-and-exitcritical-why-things-are-failing-badly/>

```
volatile uint8_t SR_reg;
volatile uint8_t SR_lock = 0x00U; /* Lock */

#define EnterCritical() \
    if (++SR_lock == 1u) {\
        asm ("MRS R0, PRIMASK\n\t" \
            "CPSID i\n\t" \
            "STRB R0, %[output]" \
            : [output] "=m" (SR_reg)\
            :: "r0");\
    }

#define ExitCritical() \
    if (--SR_lock == 0u) { \
        asm ("ldrb r0, %[input]\n\t" \
            "msr PRIMASK,r0;\n\t" \
            ::[input] "m" (SR_reg) \
            : "r0"); \
    }
```

```
LOAD SR_lock -> reg
INC reg
STORE reg -> SR_lock
IF (reg==1)
    MOVE PRIMASK -> reg
    DISABLE INTERRUPTS
    STORE reg -> SR_reg
ENDIF
/* CS starts */
```

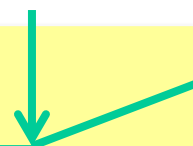
```
/* CS here */
LOAD SR_lock -> reg
DEC reg
STORE reg -> SR_lock
IF (reg==0)
    LOAD SR_reg -> reg
    MOVE reg -> PRIMASK
ENDIF
/* end of CS */
```

# Reentrancy: Race Condition

- Output is dependent on the sequence/timing

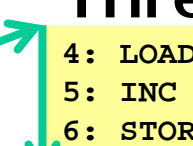
## Thread A:

```
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!!!!**



# CriticalSection Component

- <http://mcuoneclipse.com/2014/02/08/criticalsection-component/>

CS1:CriticalSection

CriticalSection

EnterCritical

ExitCritical

```
#define CS1_CriticalVariable() \
    uint8_t cpuSR; /* variable to store current status */
```

```
#define CS1_EnterCritical() \
do { \
    asm ( \
        "MRS R0, PRIMASK\n\t" \
        "CPSID I\n\t" \
        "STRB R0, %[output]" \
        : [output] "=m" (cpuSR) :: "r0"); \
} while(0)
```

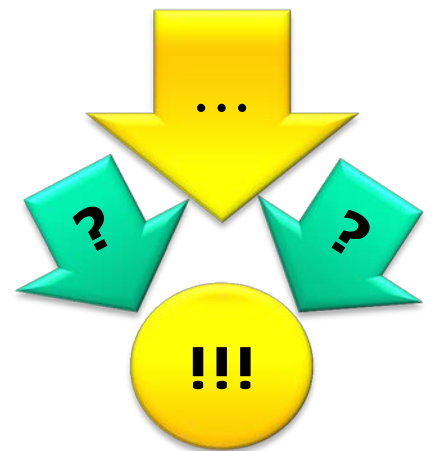
```
#define CS1_ExitCritical() \
do{ \
    asm ( \
        "ldrb r0, %[input]\n\t" \
        "msr PRIMASK,r0;\n\t" \
        ::[input] "m" (cpuSR) : "r0"); \
} while(0)
```

```
void foo(void) {
    CS1_CriticalVariable() /* declaration of storage for system status */

    CS1_EnterCritical(); /* start of critical section */
    /* critical section here */
    CS1_ExitCritical(); /* critical section ends here */
}
```

# Summary

- Interrupt Processing
- Shared Data
- Shared Subroutines
- Reentrancy
- Priorities
  - Depends on hardware
  - No/few/complex priorities
- Interrupt Vectors
- Design Criteria's
  - Latency
  - Use with care
  - Interrupts for things which cannot wait
  - Make timing analysis/map



## Lab: Critical Sections

- Use CriticalSection component
- Inspect assembly code

