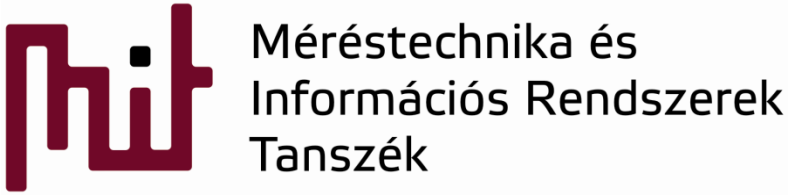
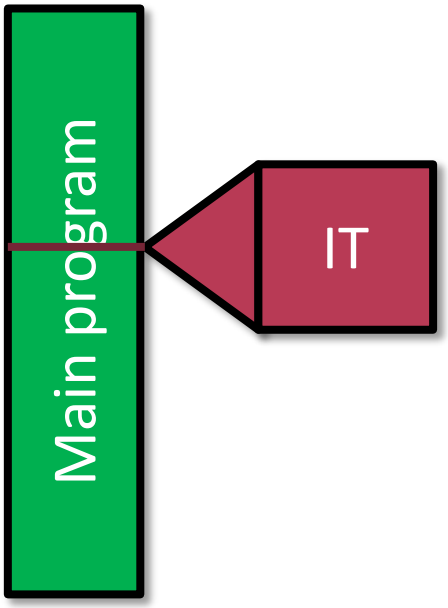
Embedded and ambient systems 2022.10.18.

Interrupts



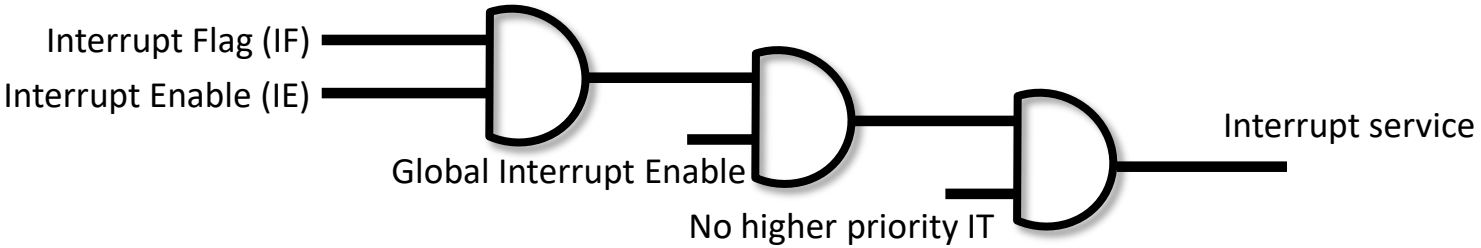
Interrupts

- Running of the program is interrupted due to an external event, and the code that belongs to the interrupting event starts running
- The code of the interrupting event is “inserted” into the main program
- Returning of the main program from the interrupted state the main program should not “notice” that it had been interrupted. To assure that:
  - Work registers have to be restored
  - Processor status registers have to be restored
  - Stack has to be restored
  - In short: context change has to be done
- In embedded systems: several different architectures and solutions exist, therefore general considerations have to be completed in a device-specific manner



Hierarchy of interrupts

- Process for execution of interrupts:
  - IT event created → IT flag belonging to the event (e.g. pushbutton is pushed) is set to 1
  - If IT is enabled for the event, than global IT flag is also set to 1
  - If global IT flag is enabled than interrupt can take effect
    - In case of priority-based interrupt service routine IT of higher priority can interrupt that of lower one (preemptive)
    - In case of non priority-based interrupt service routine IT-s cannot interrupt each other

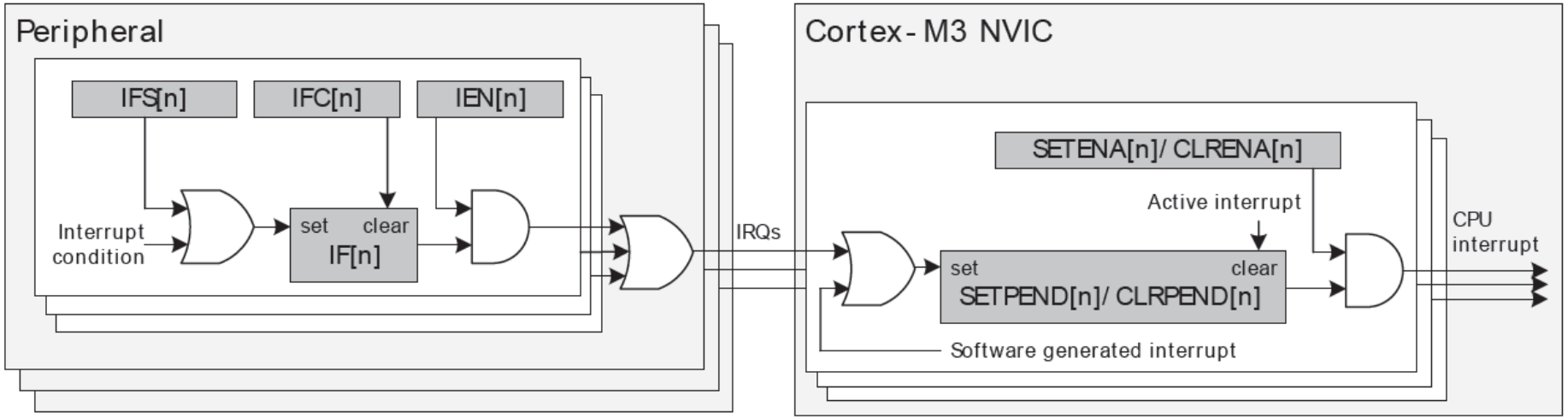


Interrupt Service Routine

- In general IT-s are vector-based
  - A table of vectors is found in the memory. Interrupt service can be operated in different ways, e.g.:
    - In case of interrupt commands are executed placed in the IT vector table
      - Jump (or Call) command to the memory address where function of interrupt service routine is placed
      - Rare case to handle IT rapidly by asm operation placed in the vector table
    - In case of interrupt the program continue running on the address placed in the IT vector table (this is faster in case of longer service since the program jumps where the whole IT routin can be executed)
  - It is possible even in case of vector-based interrupt that more than one event belong to a single IT
    - E.g.: on EFM32 development board two IT belongs to GPIOs: an even and an odd. It has to be given in the interrupt service routine which GPIO was the source

Example: IT of EFM32 + Cortex-M3

- Two-level IT:
  - Peripheral
    - Interrupt condition
    - IFS: Interrupt Flag Set: IT generated in SW
    - IFC: Interrupt Flag Clear
    - IEN: Interrupt ENable
  - Cortex-M3 core (NVIC: Nested Vector Interrupt Controller)
    - IT events from the peripherals are mapped into a given IRQ (Interrupt Request) line
    - In case of IT, PEND bit becomes 1, and deleted when stepped into IT routine



Example: IT of EFM32 + Cortex-M3

- IT table of processor core
  - Negative IRQ# belong to the processor core
  - Positive IRQ# belong to the peripherals
  - A priority belong to the ITs that can be set

Table 2.1. EFM32 Interrupts

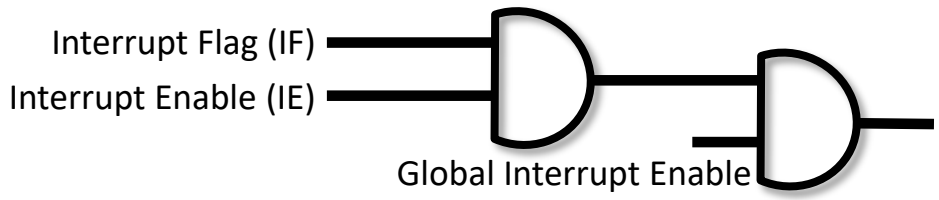
IRQ #	Source
-	Reset
-14	Non-maskable interrupt (NMI)
-13	Hard fault
-12	Memory management fault
-11	Bus fault
-10	Usage fault
-5	SVCall
-2	PendSV
-1	SysTick
0	DMA
1	GPIO_EVEN
2	TIMER0
3	USART0_RX
4	USART0_TX
5	ACMP0/ACMP1
6	ADC0
7	DAC0
8	I2C0
9	GPIO_ODD
10	TIMER1
11	USART1_RX
12	USART1_TX
13	LESENSE
14	LEUART0
15	LETIMER0
16	PCNT0
17	RTC
18	CMU
19	VCMP
20	LCD
21	MSC
22	AES

Example: AVR (ATmega128)

- Simple vector-based IT
- In case of IT the program continues running based on the program address found in the IT table
- Global IT is disabled automatically
- In ASM code RETI command is used to return from IT (global IT is automatically enabled), and C compiler has to be informed about the return (see later).
- When IT routine is called the corresponding IT flag is disabled automatically

Table 23. Reset and Interrupt Vectors

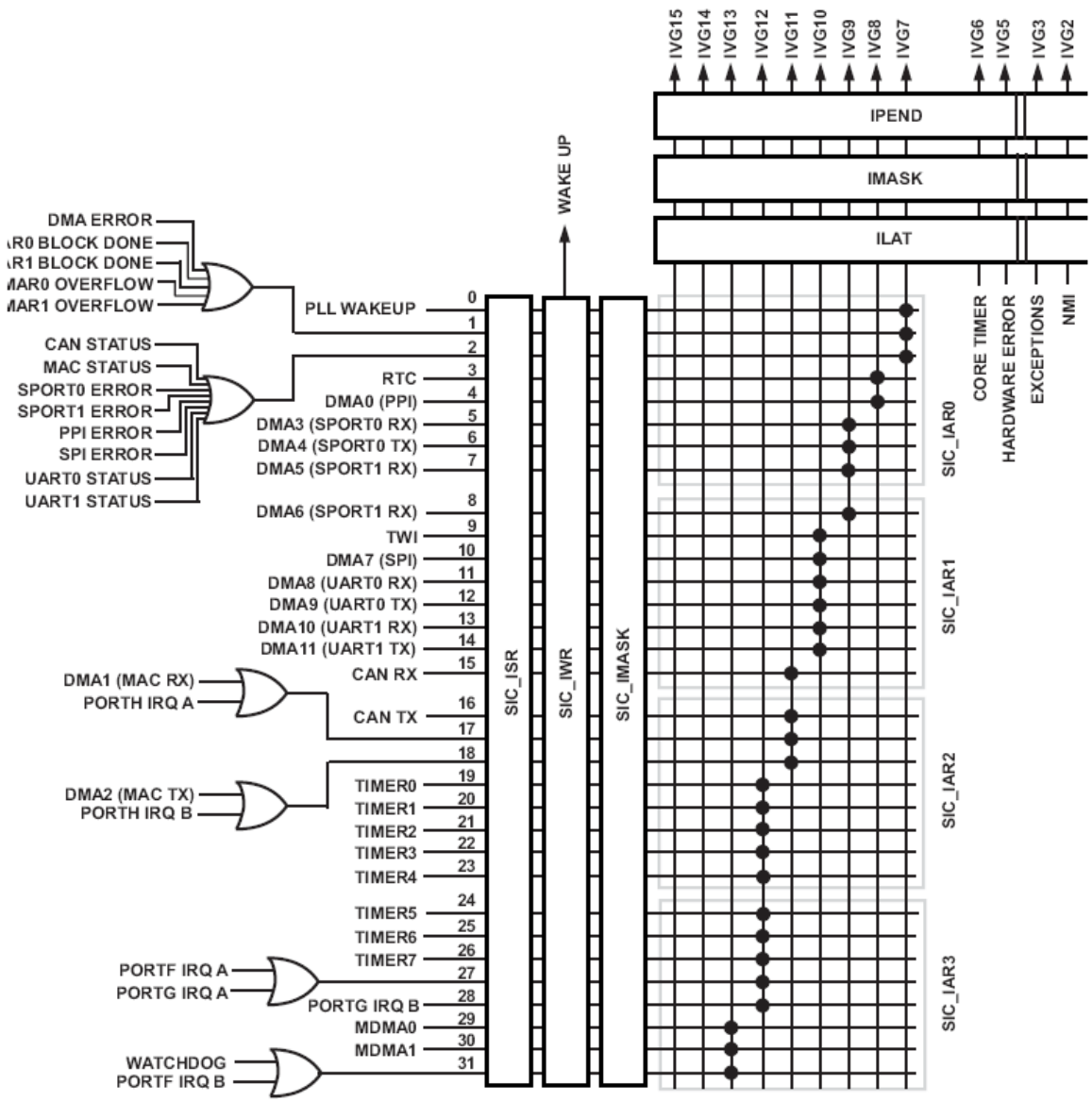
Vector No.	Program Address(2)	Source	Interrupt Definition
1	$0000(1)	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$0002	INT0	External Interrupt Request 0
3	$0004	INT1	External Interrupt Request 1
4	$0006	INT2	External Interrupt Request 2
5	$0008	INT3	External Interrupt Request 3
6	$000A	INT4	External Interrupt Request 4
7	$000C	INT5	External Interrupt Request 5
8	$000E	INT6	External Interrupt Request 6
9	$0010	INT7	External Interrupt Request 7
10	$0012	TIMER2 COMP	Timer/Counter2 Compare Match




```
Address  LabelsCode          Comments
$0000        jmp    RESET     ; Reset Handler
$0002        jmp    EXT_INT0  ; IRQ0 Handler
$0004        jmp    EXT_INT1  ; IRQ1 Handler
$0006        jmp    EXT_INT2  ; IRQ2 Handler
$0008        jmp    EXT_INT3  ; IRQ3 Handler
$000A        jmp    EXT_INT4  ; IRQ4 Handler
$000C        jmp    EXT_INT5  ; IRQ5 Handler
$000E        jmp    EXT_INT6  ; IRQ6 Handler
$0010        jmp    EXT_INT7  ; IRQ7 Handler
$0012        jmp    TIM2_COMP ; Timer2 Compare Handler
```

Example: BF537 (DSP)

- IT hierarchy of BF537 type DSP from Analog Devices
- SIC: System Interrupt Controller
- IT lines of peripherals wired from the left side
- SIC_ISR: storing the primary ITs (IT flag)
- SIC_IWR: enabling to wake up the processor
- SIC_MASK: enabling primary IT sources
- Switch matrix: which line is mapped into which input IT vector
- ILAT: IT latch register
- IMASK: IT mask register
- IPEND: IT pending
- IVGnn: IT lines



Example: BF537 (DSP)

- In case of IT, the appropriate bit of SIC_ISR (Interrupt Status Register) set to 1
- SIC_ISR register cannot be cleared in SW, but when the peripheral IT request has been handled the appropriate bit of SIC_ISR is disabled automatically (peripheral clears it)
- Even more than one input IT sources can be connected to an output IT line (IVGnn), the processor has to determine the source
  - During development if the number of ITs are only a few (we try to achieve this in general), then every source can be assigned to a distinct IT vector, therefore the source will be unambiguous
- RTI instruction is used to return from the function belonging to a certain IT line and that IT line will be enabled (during handling an IT is is disabled to avoid the generation further interrupts) and IPEND register is also cleared
- Some special ITs: NMI (non-maskable), reset, HW error, core timer

IT initialization for a peripheral

- Initialization of IT in a general case:
  - Enabling peripheral (turn perif. on, config., etc.)
  - Determination of IT-handling function
  - Clear of IT flag belonging to the certain IT
    - An IT request may be stuck from a previous state that can cause problem since after enabling IT a false interrupt can take action. A stuck IF can be the consequence of a non-initialized peripheral (e.g. IT occurs on a floating input)
  - Enabling the IT of a certain peripheral
  - Clearing of global IT flag (if needed)
  - Enabling of global IT

IT handling for a peripheral

- Execution of IT, the proposed way:
  - Rapid execution inside a function
  - Larger tasks are handled outside the function
  - When a peripheral is used more than only once in the program mutual exclusion has to be assured (e.g. when part of our code sends data using UART, and data is also sent in an IT routine, then the two data can be messed up)
    - It is advised to use flags for such kind of tasks, and execute them in the main program it they are not time-critical
  - Tightly connected tasks for IT handling (take care of the order!):
    - Peripheral handling (e.g. data of UART must be read or GPIO state must be read, etc.),
    - Clearing the corresponding IT flag (not needed when done automatically, but better done twice than never),
    - Enabling the IT (done automatically in most cases)

IT handling example

- Example

volaitle bool button_IT_flag = false;
int buttonpushed;

volatile type is important: program calling of Button_IRQ function is not seen by the compiler in the program, therefore unaware of it so it may change in the background. Therefore it may happen that when reading of button_IT_flag comes, new data is not read. Every variable that appears in an IT function MUST be volatile.

```
// somehow we define that this function will handle interrupt: see later…
void Button_IRQ(void){  // IT handling function
    button_IT_Flag = true; // rapid handling = a flag is set that an IT happened
    clear_button_IF(); // clear button IT flag
}
```

IT function

```
int main(void){

initButtons();          //1. init of buttons
clear_button_IF();      //2. IT flag clear
Button_IT_enable(); //3. IT enable for buttons
clear_global_IF();      //4. IT flag clear
global_IT_enable();  //5. enable global IT
```

Done at the beginning of the main program for initialization purposes

```
while(1){
    if (button_IT_Flag ){
        button_IT_Flag = false;
        printLCD(”buttonpush: %d”, buttonpushed++); // putting on the screen is slow, done in the main program
    }                                                // (not in the IT function otherwise uC may be blocked until
  }                                                  // writing on the display is not done)
}
```

Implementation of IT handling

- Basic rule: „No rule”
  - Lots of solution exist
  - Always has to be check how IT handling should be done depending on the compiler/processor (RTFM)
- IT handling is not supported by C compiler by default, therefore as many solution may exist as many compiler and uC-based system available
- Despite the fact that general rule cannot be given, there exist some recipes, and best practices

Implementation of IT handling– EFM32

- Startup code provided by the uC manufacturer defines the addresses of IT vector
- A default IT handler is assigned to every address (an infinite loop)
- In our own code a non-parametric function has to be given owing the same name as the one in the vector table
- In the startup code the functions have *weak* attribute. Attribute *weak* means repeated implementation of the function in our code will cause no error
- When IT event occurs our own function is called from the vector table (the linker will substitute the default function with weak attribute defined in the startup code with our own function address)

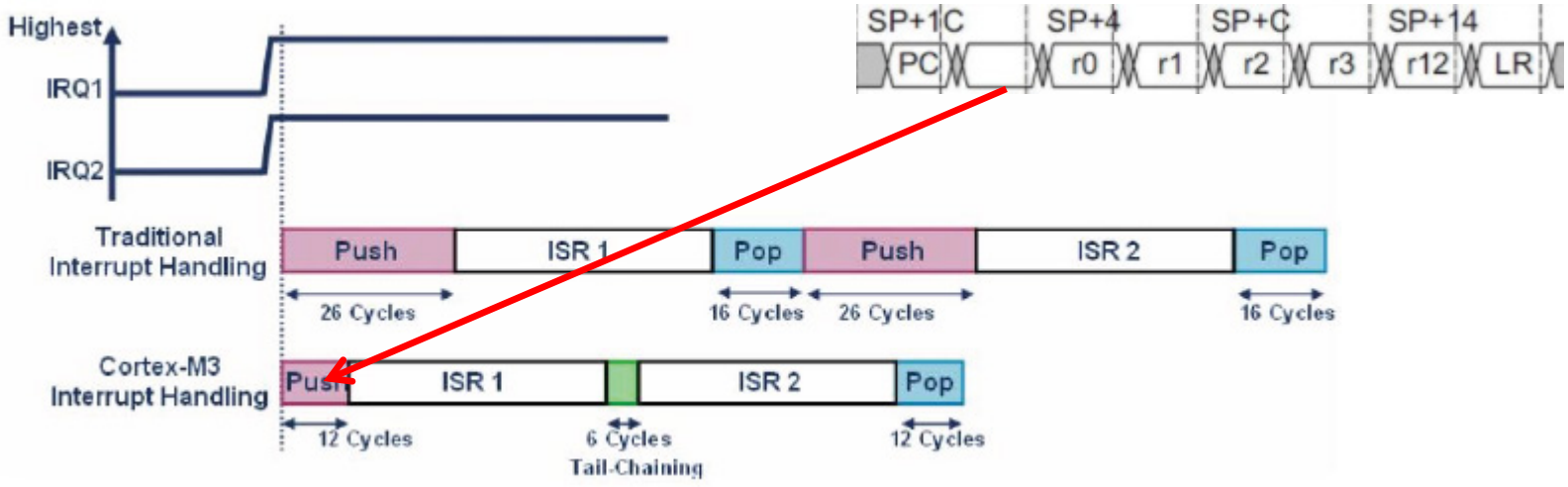
```
    .section    .vectors
    .align      2
    .globl      __Vectors
__Vectors:
    .long       __StackTop              /* Top of Stack */
    .long       Reset_Handler           /* Reset Handler */
    .long       NMI_Handler             /* NMI Handler */
    .long       HardFault_Handler       /* Hard Fault Handler */
    .long       MemManage_Handler       /* MPU Fault Handler */
```

```
    .align  1
    .thumb_func
    .weak   Default_Handler
    .type   Default_Handler, %function

Default_Handler:
    b       .
    .size   Default_Handler, . - Default_Handler


/*    Macro to define default handlers. Default handler
 *    will be weak symbol and just dead loops. They can be
 *    overwritten by other handlers */
    .macro  def_irq_handler handler_name
    .weak   \handler_name
    .set    \handler_name, Default_Handler
    .endm

    def_irq_handler     NMI_Handler
    def_irq_handler     HardFault_Handler
    def_irq_handler     MemManage_Handler
```

Implementation of IT handling– EFM32

- Simple IT handling:
  - Cortex – M3 processors save several important registers when IT occurs (R0…R3, R12, PC, Link Reg.) therefore IT handling can be done by a simple function
  - Interrupt latency (time elapsed between IT occurrence and starting of IT handling) is short
  - Example (GPIO IRQ handling):

```
void GPIO_ODD_IRQHandler(void){ …
    Function-implementation…}
```



Implementation of IT handling– EFM32

- Safe solution: let the compiler be informed somehow that this is a special IT handling function, since this way it is sure that all register state will be saved that have been modified
  - Mainly important to do so if the program is planned to be run on other type of processors as well
- How to inform the compiler about an IT handling function?
  - Using **__attribute__** special option different features of functions and variables can be set that are processed by the compiler
  - Definition of IT routine (e.g. GPIO IRQ handler):

```
void GPIO_ODD_IRQHandler(void) __attribute__ ((interrupt ("IRQ")));
void GPIO_ODD_IRQHandler(void){ …
    Function-implemetation…}
```

Implementation of IT handling– EFM32

- What code is compiled in case of a normal function and that of a function with IRQ attribute?
  - Several registers are saved (see push instruction), some of them in a redundant way since processor saves them by default
  - If the user does not know the processor well it is a more safe solution (it may happen that C code will use such a register which had not been saved…)

__attribute__ ((interrupt ("IRQ")))

```
13          void GPIO_ODD_IRQHandler(void){
            GPIO_ODD_IRQHandler:
000004d8:     mov     r0,sp
000004da:     bic     r1,r0,#0x7
000004de:     mov     sp,r1
000004e0:     push    {r0,r3,r7,lr}
000004e2:     add     r7,sp,#0x0
14                GPIO_IF_value = GPIO->IF;
000004e4:     ldr     r3,[pc,#0x20] ; 0x504
000004e6:     ldr.w   r3,[r3,#0x114]
000004ea:     ldr     r2,[pc,#0x20] ; 0x508
000004ec:     str     r3,[r2]
15                GPIO_IntClear(1<<PUSH1_PIN); // if commented, IF
000004ee:     mov.w   r0,#0x200
000004f2:     bl      0x00000320
16                GPIO_PinOutToggle(LED1_PORT, LED1_PIN);
000004f6:     movs    r0,#0x4
000004f8:     movs    r1,#0x2
000004fa:     bl      0x00000444
17          }
000004fe:     mov     sp,r7
00000500:     pop.w   {r0,r3,r7,lr}
00000504:     mov     sp,r0
00000506:     bx      lr
00000508:     str     r0,[r0]
0000050a:     ands    r0,r0
0000050c:     lsls    r4,r4,#2
0000050e:     movs    r0,#0x0
```

normal function

```
13          void GPIO_ODD_IRQHandler(void){
            GPIO_ODD_IRQHandler:
000004d8:     push    {r7,lr}
000004da:     add     r7,sp,#0x0
14                GPIO_IF_value = GPIO->IF;
000004dc:     ldr     r3,[pc,#0x18] ; 0x4f4
000004de:     ldr.w   r3,[r3,#0x114]
000004e2:     ldr     r2,[pc,#0x18] ; 0x4f8
000004e4:     str     r3,[r2]
15                GPIO_IntClear(1<<PUSH1_PIN); // if commented, IF is
000004e6:     mov.w   r0,#0x200
000004ea:     bl      0x00000320
16                GPIO_PinOutToggle(LED1_PORT, LED1_PIN);
000004ee:     movs    r0,#0x4
000004f0:     movs    r1,#0x2
000004f2:     bl      0x00000444
17          }
000004f6:     pop     {r7,pc}
000004f8:     str     r0,[r0]
000004fa:     ands    r0,r0
000004fc:     lsls    r4,r4,#2
000004fe:     movs    r0,#0x0
```

Implementation of IT handling– AVR

- Example: ATmega128 (8-bit uC)
- IT name and program code must be given:

ISR(vector_name)

Example:

```
ISR(USART1_RX_vect) {
    UART1_read_data();
    …  // other things to do is possible
    crear_IF();
}
```

- Other options may exist, e.g. „ISR_NOBLOCK”: in IT routine the compiler enables the interrupts.
  - Example: ISR(USART1_RX_vect, ISR_NOBLOCK){…}

Implementation of IT handling– AVR

- How it works?

```
#  define ISR(vector, ...)            \
   void vector (void) __attribute__ ((signal,__INTR_ATTRS)) __VA_ARGS__; \
   void vector (void)
```

So using „ ISR(USART1_RX_vect){…}” command the following is compiled after extracting the macro:

```
void __vector_30 __attribute__ ((signal));  //function declaration with attribute
void __vector_30{     //function implementation
      code
}
```

Explanation (somewhere else the following definitions can be found):

```
#define USART1_RX_vect            _VECTOR(30)
…
#define _VECTOR(N) __vector_ ## N
```

Address __vector_30 is found in crtm128.o startup file, which are re-defined

Implementation of IT handling – BF537

- The development environment provided by the manufacturer offers functions that can be used to register IT handling functions:

```
register_handler_ex(interrupt_kind kind, ex_handler_fn fn, int enable);
```

Example:

```
register_handler_ex(ik_timer, timer_handler, EX_INT_ENABLE);
```

Funtion has to given in the following way, e.g.:

```
EX_INTERRUPT_HANDLER(timer_handler){
    Timer_IT_Number ++;
}
```

As an IT handler of the timer function named timer_handler is given and IT is enabled.

- How it works?

Implementation of IT handling – BF537

- How the function is inserted into the vector table? (crt\reghdlr.h and sys\exception.h)
  - A function pointer is set to the start address of the vector table
  - Function pointer defines an array whose interrupt_kind element has to be replaced by the address of the given function. From now it will be called by the processor

```
register_handler_ex(interrupt_kind kind, ex_handler_fn fn, int enable);

typedef void (*ex_handler_fn)(); // definition of type of function pointer
#define EX_EVENT_VECTOR_TABLE  0xFFE02000 // start address of IT vector table
// function array is set to the start address of the vector table
ex_handler_fn *evt = (volatile ex_handler_fn *)EX_EVENT_VECTOR_TABLE;
// the specific element of the vector table (depending of the type of the IT) is replaced by
    the function pointer of our own function
evt[kind] = fn;
```

Implementation of IT handling – BF537

- How IT handling function declaration works? Now it is “known” by the processor which function has to be called (the address of that one has been stored in the vector table) but how the compiler will know that that function should be compiled in a different way? Lets see the following macros (\sys\exception.h)

```
#define EX_INTERRUPT_HANDLER(NAME)  EX_HANDLER(interrupt,NAME)
#define EX_HANDLER(KIND,NAME) \
_Pragma(#KIND) \
void NAME ()
```

Therefore

```
EX_INTERRUPT_HANDLER(timer_handler){
     Timer_IT_Number ++;
}
```

Expanding of function definitions are the following after substitution of the macros

```
_Pragma(interrupt)  // now the compiler knows that it is an IT handling function
void timer_handler(){ // here the function is given by the normal mode
    Timer_IT_Number ++;
}
```

Implementation of IT handling – BF537

- Interruptable function can also be given by the following macro:

```
EX_REENTRANT_HANDLER(NAME)
```

- After expanding the macro:

```
#define EX_REENTRANT_HANDLER(NAME) \
_Pragma("interrupt_reentrant") \
EX_HANDLER(interrupt,NAME)
```

- So,
  _Pragma("interrupt_reentrant")
  compiling directive is used to let the compiler be informed that this function is such an IT handling function that can be interrupted

Implementation of IT handling – BF537

- Summary:
  - There exists such a library function that replace (rewrite) the start address of the IT routine found in the IT vector table with the function defined by us
  - Using a macro provided by the manufacturer such function can be defined that is known by the compiler to be an IT handler function (Pragma should be used for that)

Implementation of IT handling–ADSP21364

- Example: ADSP21364: 32-bit, floating-point DSP
- Callback function: functions called based on certain events (see e.g. Java)
- Manufacturer provides an IT dispatcher function. The function runs at every IT event, saves the status of the processor and calls the function assigned (by us) to the IT (similar to callback)
- IT handler is a normal function, context change is done by the dispatcher
- Several options exist. The following functions can be used to register the IT handling functions:
  - interruptcb(): during IT every special DSP HW status is saved (e.g. circular buffer, loop counter, secondary registers), it takes approximately 200 CLK cycles
  - interrupt(): during IT most special DSP HW status is saved it takes approximately 150 CLK cycles
  - interruptf(): very fast handing only some special DSP HW status is saved
  - Etc…: consequency is that the actual status of the processor can be saved at different depth. The more processor registers are saved the more context change happens (=slower) but the more reliable then. In case of rapid context change extra care is needed what can be done by the function.

Implementation of IT handling–ADSP21364

- Example: UART IT handling
  - It has to be found in the documentation of the compiler that in case of a certain processor how the peripheral belonging to UART is named. In our case it is SIG_SP4 (SP: serial port)
  - Our function has to be registered, e.g. UARTrec
  - Our function has to be implemented

```
interrupt (SIG_SP4 , UARTrec); // SIG_SP4 type and handling function

// handling function is given by:
void UARTrec(void){
   // handling of IT comes here
}
```

Implementation of IT handling - summary

- **General tasks:** configuring peripheral, clear of IT flags, enabling of IT (that of peripheral, and global), hanfling of peripheral in IT routine (in many cases manual clear of IT flag is needed)
- **EFM32-Cortex M3:** find the name of IT handler in startup file and using this name an own function has to be defined (interrupt attribute may used). E.g.:

```
void GPIO_ODD_IRQHandler(void) __attribute__ ((interrupt ("IRQ")));
void GPIO_ODD_IRQHandler(void){
   Function-implementation…}
```

- **AVR 8-bit uC:** find the name belonging to a specific IT from documentation and using ISR() macro the function can be defined. E.g.:

```
ISR(USART1_RX_vect) {
   Function-implementation…}
```

- **ADSP BF357 (DSP):** The IT function has to be registered in the IT vector table and it has to be implemented using macro EX_INTERRUPT_HANDLER:

```
register_handler_ex(ik_timer, timer_handler, EX_INT_ENABLE);
EX_INTERRUPT_HANDLER(timer_handler){
    Function-implementation…}
```

- **ADSP 21364 (DSP):** The IT function has to be registered at the dispatcher:

```
interruptcb(SIG_IRQ1,IRQ1_handler); // IRQ1 type and handling function
     void IRQ1_handler(int x){
        Function-implementation…}
```

Implementation of IT handling - summary

- It is necessary to always read the documentation of the processor and compiler:
  - Is it necessary to indicate the special IT function? (pl. #pragma, __attribute__)
  - Does the processor clear IT flag automatically? If not it must be done, but in general it is a good idea to do so all the time
  - Does the processor disable IT when an IT handling started?
  - Multi level IT exist?
  - Is it necessary to enable IT when return from IT handling?
- Is a special peripheral, variable used? If yes, than mutual exclusion is assured?
- IT routine should not be too long
- IT handling functions generally have no parameters. If a parameter is needed it has to be solved by ourselves.
- The type of variables used in IT routines always has to be **volatile**
  - Volatile variables are considered by the compiler such variables whose value can change any time, therefore their value are always read even if they are not seem to have changed