GPIO BSRR

- LED_GPIO->ODR |= BV(pin);
  - LED_GPIO->BSRR = BV(pin);
- LED_GPIO->ODR &= ~BV(pin);
  - LED_GPIO->BSRR = BV(pin + 16);

Bit Banding

- The System bus interface contains logic that controls bit-band accesses as follows:
  - It remaps bit-band alias addresses to the bit-band region.
  - For reads, it extracts the requested bit from the read byte, and returns this in the LSB of the read data returned to the core.
  - For writes, it converts the write to an atomic read-modify-write operation.
  - The processor does not stall during bit-band operations unless it attempts to access the System bus while the bit-band operation is being carried out.

Bit-banging (is not Bit-banding)

- Informally bit-banging is any method of data transmission that employs software as a substitute for dedicated hardware to generate transmitted signals or process received signals.
- Frequently switching GPIO pin state as per protocol requirement.

Bit-banding (Cortex-M architecture feature)

- Bit-banding maps each bit of a register to a word in the address space
- Typically two bit-band regions (each of 1 MB) are available on CM3/CM4 devices, which are mapped to corresponding bit-band alias regions (each of 32 MB).
- Region1 - SRAM:
  - 0x20000000 - 0x22000000 (alias)
- Region2 - Peripherals:
  - 0x40000000 - 0x42000000 (alias)
- bit_word_addr = bit_band_alias_base + (byte_offset * 32) + (bit_number * 4);

```
#define BB_BASE 0x40000000UL
#define BB_ALS_BASE 0x42000000UL

//#define BB_ALS_ADDR(regr,bit) (BB_ALS_BASE + ((regr-BB_BASE) * 32) + (bit * 4))
#define BB_ALS_ADDR(regr,bit) (BB_ALS_BASE + ((regr-BB_BASE) << 5) + (bit << 2))
#define BB_ALS(regr,bit) (*(uint32_t*)BB_ALS_ADDR((uint32_t)regr,bit))
```

Weak Functions - GCC

- Available in Assembly and C programming on GCC toolchain.
- The weak function is assigned to a temporary function.

    - If no function is implemented with that name, the temporary function definition is invoked when the weak function is called.
    - If function is implemented exactly with same name, that definition is consider (invoked).

External Interrupt

- All GPIO pins can be configured to produce interrupt on
    - Rising edge
    - Falling edge
    - Rising & Falling edge
- External interrupt controller consists of up to 23 edge detectors for generating interrupts.
    - PORTx.n -> EXTIn (0 <= n <= 15)
- EXTI16 to EXTI22 are used for RTC, USB and Ethernet.

External Interrupt registers

- EXTI registers
    - IMR: EINT Interrupt masked (0) or not masked (1)
    - EMR: EINT Event masked (0) or not masked (1)
    - RTSR: EINT Rising edge trigger disabled (0) or enabled (1)
    - FTSR: EINT Falling edge trigger disabled (0) or enabled (1)
    - PR: EINT Not Pending (0) or Pending (1). Pending interrupt cleared by writing 1.
- SYSCFG registers
    - EXTICR1 (index 0) -> EXTI0-3 [4 bits for port id: PA=0, PB=1, ..., PI=8]
    - EXTICR2 (index 1) -> EXTI4-7 [4 bits for port id: PA=0, PB=1, ..., PI=8]
    - EXTICR3 (index 2) -> EXTI8-11 [4 bits for port id: PA=0, PB=1, ..., PI=8]
    - EXTICR4 (index 3) -> EXTI12-15 [4 bits for port id: PA=0, PB=1, ..., PI=8]
- NVIC registers
    - ISERx -> NVIC_EnableIRQ(irq);

External Interrupt Configuration

- Configure GPIO pin as input with appropriate push/pull resistor config.
- Configure GPIO External interrupt into system config.
    - EXTICR config for GPIO pin.
- Configure & enable interrupt into External Interrupt controller (peripheral).
    - Falling and/or Rising edge config into FTSR and/or RTSR.
    - Unmask interrupt from IMR.
- Enable interrupt in NVIC.
- Implement interrupt handler.
    - Function name must match with the name given in interrupt vector.
    - Acknowledge the interrupt.
    - Interrupt handler should be ideally short & non-blocking

Interrupt handling

- When ISR is running, all interrupts with lower priority are masked/disabled. This will increase their interrupt latency.
- ISR should be short and "non-blocking".
- If interrupt handling needs any blocking operation to be performed, it should be moved to some other function which can execute after ISR.

Optimization level 0 : (-O0)

```
uint32_t ext_flag;
void EXTI0_IRQHandler(void)
{
    // ack to peripheral
    EXTI->PR |= BV(0);
    // raise the flag
    ext_flag = 1;
        /*
        LDA r7, = ext_flag
        LDA r5, #1
        STA r5, [r7]
        */
}
while(ext_flag == 0)
    ;
    /*
    LDA r7, = ext_flag
    loop:
        LDA r6, [r7]
        CMP r6, 0
        beq loop
    */
ext_flag = 0;
//
```


Optimization level 3 : (-O3)

```
uint32_t ext_flag;
void EXTI0_IRQHandler(void)
{
    // ack to peripheral
    EXTI->PR |= BV(0);
    // raise the flag
    ext_flag = 1;
        /*
        LDA r7, = ext_flag
        LDA r5, #1
        STA r5, [r7]
        */
```

```
}

while(ext_flag == 0)
    ;
    /*
    LDA r7, = ext_flag
    LDA r6, [r7]
    loop:
        CMP r6, 0
        beq loop
    */
ext_flag = 0;
//
```

register and volatile keyword

- If a variable is used repeatedly in a loop, compiler cache it into CPU register in order to optimize the execution of the program.
- Meanwhile the variable may be modified outside the current execution context, which will be ignored by the system. This may lead to unexpected results.
- Such optimization can be disabled by declaring variables as volatile. In this case compiler always fetch it from its memory location (not from cached register).
- Loop variables which may change suddenly out of current execution context should be declared as volatile.
    - Interrupt handler / ISR
    - Another thread of execution
    - Memory mapped IO

static, const and volatile keyword

- static keyword restrict the scope of the variable to the declared function or file and place it in data section. Variable can be modified.
- const keyword instructs compiler that the variable is not intended to be modified in source code, so that compiler disallow using certain operators on the variable (that may modify value of the variable).
- volatile keyword disable optimization of the variable, so that it will be always fetched from its RAM location. Variable may be modified suddenly out of the current context.
- static const volatile i = 1;
    - Scoped variable that is not allowed to be modified within the source code and its optimization (caching in CPU register) is disabled.
    - Typically variable may get modified due to memory-mapped IO