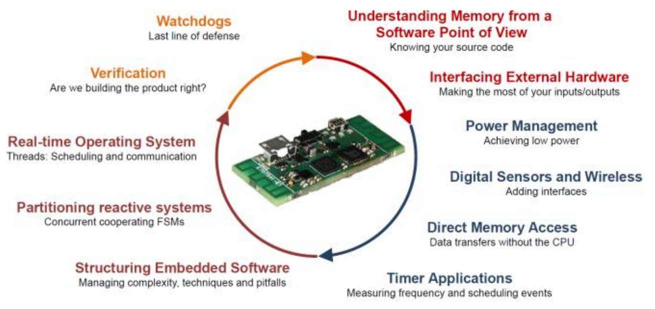
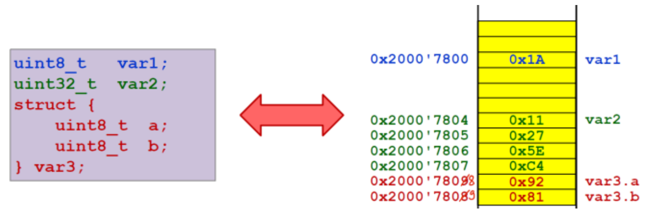
MC1–Strickler Frederic & Janko Uehlinger

1. Memory from a Software POV

1.1. Memory object in C



Watchdogs
Last line of defense
Understanding Memory from a Software Point of View
Knowing your source code
Verification
Are we building the product right?
Interfacing External Hardware
Making the most of your inputs/outputs
Power Management
Achieving low power
Real-time Operating System
Threads: Scheduling and communication
Digital Sensors and Wireless
Adding interfaces
Partitioning reactive systems
Concurrent cooperating FSMs
Direct Memory Access
Data transfers without the CPU
Structuring Embedded Software
Managing complexity, techniques and pitfalls
Timer Applications
Measuring frequency and scheduling events



```
uint8_t   var1;
uint32_t  var2;
struct {
    uint8_t  a;
    uint8_t  b;
} var3;
```

- **Data type** Implies size, i.e. number of bytes in memory
- **Name** Used to access the memory region
- **Value** Content/data stored in the memory region
- **Address** Location in memory where the variable resides
- **Scope** Part of the source code in which the name is visible (known)
- **Lifetime** When is the variable created (allocation of memory) and when is it destroyed (deallocation of memory)

Sizes of integer types depend on architecture and compiler!

Only Rule: Int must be bigger than char

stdint.h needs different h-files for different Platforms so the program can stay the same for different archtitectures

Same sizes for all architectures / compilers

Size of pointers is platform dependent

1.2. Name and data type

Definition:

- Introduces name/data type **and** allocates storage space
- Function body

Declaration:

- Introduces name/data type
- Does **not** allocate storage space

1.3. Structs

Order of elements may influence number of required bytes in memory if elements have different size
The compiler or programmer can optimize storage

```
typedef struct {
} large_struct_t;
```

Large makes holes in storage because halfword always uses addresses dividable by 4.

```
typedef strsuct {
} compact_struct_t;
```

1.4. Header Files

Interface -> header file

- Only declarations -> NO definitions
- No other #include statements 1)

Implementation -> .c file

- Provides definitions of items that have been declared in header file

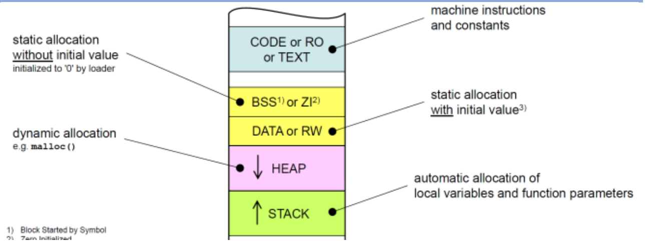
1.5. Value

Content/data stored in the memory region

Content of variable is interpreted based on data type
E.g. casting: Bit representation stays the same, interpretation changes

%: takes Value at Storage position and interprets it dependent on the symbol that follows
%x **int** and output as **unsigned Hex number**
%d **int** and output as **signed decimal number**
%u **int** and output as **unsigned decimal number**

1.6. Address -> Memory Sections



machine instructions and constants
CODE or RO or TEXT
static allocation without initial value
initialized to '0' by loader
BSS1) or ZI2)
DATA or RW
static allocation with initial value3)
dynamic allocation
e.g. malloc()
HEAP
automatic allocation of local variables and function parameters
STACK
1) Block Started by Symbol
2) Zero Initialized

In Code (color-coded):

```
uint32_t bss_a;
uint32_t data_a = 0x12;
static uint32_t bss_b;
static uint32_t data_b = 0x34;

void foo(void)
{
    uint32_t *p;
    uint32_t stack_a;
    static uint32_t bss_c;
    static uint32_t data_d = 0x56;
    const uint32_t read_only_c = 0x78;
    ...
    p = (uint32_t*) malloc(sizeof(uint32_t));
    if (p == NULL) { /* handle error */ }
    ...
    free(p);
}
```

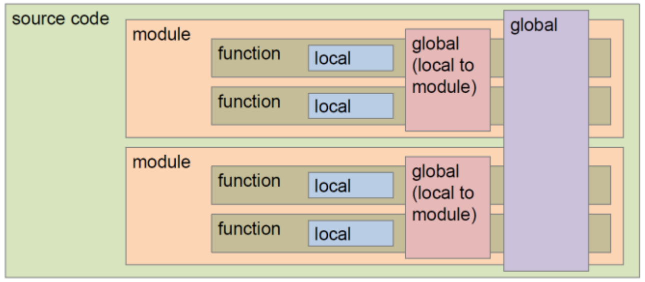
1.7. Scope

Keep the scope as small as possible

Terms 'local variables' and 'automatic variables' are used interchangeably

Visibility of names in **source code**

- An arrangement between compiler/assembler and programmer
- Not relevant for object code



Scope of automatic variables

- Automatic variables have local scope
- Begins at point of definition
- Ends at end of containing block

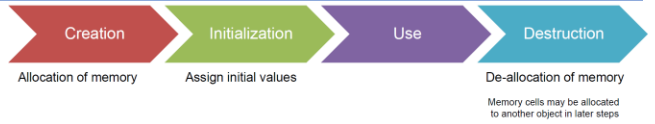
Scope of variables with static allocation

- **Static** variables within function
    - Local scope
- **Static** variables outside functions
    - Module wide scope
    - Starting from point of definition
- **Global** variables
    - Visible within whole source code

Use **static** to provide encapsulation

Make functions and global variables **private** !!!
Add qualifier static for all objects that are not in header file

1.8. Lifetime



Type	Creation	Initialization	Destruction
Automatic	Each time the program enters the function in which it is defined	Default: No initialization If definition contains an assignment: Each time program enters block	On each return from function
Static	*Once*: When program is first loaded into memory	*Once*: Just before start of program	*Once*: At program termination
Dynamic	By calling `malloc()`	Responsibility of programmer: Source code has to explicitly write initial values	By calling `free()`

1.8.1. Automatic variables ex.

```
void foo(void)
{
    uint32_t a;
    uint32_t b = 0xAB;
    ...
    for (a = 0; a < 8; a++){
        uint32_t c = 0xCD;
        ...
    }
    ...
}
```

- Creation:
    - Memory for a, b, c allocated on stack, when foo() is entered
    - Recursion: several instances exist
- Initialization
    - a: no initialization!
    - b: each time foo() is entered
    - c: each time for block is entered
- Destruction: at the end of function, memory is released (not deleted!)

1.8.2. Static variables ex.

```
// visible to all modules
// even when not part of header!
uint32_t s;
uint32_t t = 0x78;

// visible within module bar
static uint32_t u;
static uint32_t v = 0x9A;

void bar(void)
{
    // visible only within bar()
    // inititialized ONCE at prg start
    static uint32_t w;
    static uint32_t x = 0xBC;
    ...
}
```

- Creation
    - Memory is allocated, when program is loaded into memory
- Initialization
    - Just before program starts
    - t, v, x: initialized to specified value
    - s, u, w: initialized to zero
- Destruction
    - Memory released when program terminates

1.8.3. Dynamic Memory ex.

```
void bar(void)
{
    uint32_t *p;
    p = (uint32_t*) malloc(sizeof(uint32_t));
    if (p == NULL) { /* handle error */ }
    *p = 0xEF;    // manual init
    ...
    free(p);      // release memory
}
```

- Creation
    - By calling malloc()
    - Use sizeof() operator
    - Check return value of malloc() →memory available?
- Initialization
    - To be done by program
- Destruction
    - By calling free()
    - Responsibility of programmer

1.9. Selected Implications

Common coding error

- Returning a pointer to an automatic variable

Alternative

- Allocate memory in calling function
- Pass the address

Passing read-only data

- Make function parameter a pointer to **const**
- Shows that foo() is not supposed to change the data
- Compiler enforces read-only
-

Use const Whenever Possible

2. Interfacing External Signals

Issues for 'digital' signals

- Voltage levels of input signals outside allowed range
- Slow edges
- Spikes and glitches
- Noise on signal and/or supply

- Bouncing
- Transient oscillations ('Einschwingen')

2.1.1. Voltage level of GPIOs on STM32F4xx

- User defined through VDD
- CMOS voltage levels
- Most GPIOs are 5V tolerant

Ensure compatibility of logic levels

- Voltage levels of output and input stage have to be compatible -> data sheets
- VOL/VOH of driver -> VIL/VIH of input stage

2.1.2. Schmitt-Trigger

because of noise so it doesn't trigger on slight noise. Hysteresis helps on noisy signals and on signals with slow edges.

2.2. Potential Hardware issues

1. Avoid floating inputs (on unused pins)

Noise is a source of random power consumption by making the buffer switch randomly Connect unused pin to VSS/VDD on PCB or use pull-up/pull-down

2. Avoid use of pull-up if voltage on I/O pin exceeds VDD

Causes cross-voltage domain leakage, e.g. when a debugger probe driving 5V is connected

Voltage protection may cause leakage current if VDD of STM32 is switched off

E.g. If the STM32 is switched off (VDD = 0V), but the external circuit is still powered and provides a voltage on the I/O pin. This may also occur in the other direction, i.e. STM32 on, but external circuit not supplied

2.3. Edge Detection

Change of IO state -> event (Falling/Rising)

Hardware edge detection through interrupts

Polling in software: Example on CT_BOARD

- Repeatedly read buttons and compare to last value read

Read input only once and safe it to edit it multiple times. Don't read out input multiple times shortly after one another and expect the same input

State of objects: Allocate memory on the stack and pass by reference to a function

2.4. Debouncing

Switches and buttons are mechanical

- Preloaded spring
- Contacts bounce up to a few milliseconds

Key press: Every bounce is seen as input change

Hardware Debouncing: Often not available

Polling at low sample rates -> Read signal once during bouncing

Disadvantages of polling at low sample rates

- Maximum sample rate defined by bouncing
- High jitter -> often unacceptable
- Long reaction times
- Potential missing of on/off sequences

Polling at higher frequency

- Problem: Every bounce detected

2.4.1. Option 1: Sliding window filter

- Filter for inputs
- Store last n samples in array
- Edge detected only when there

is a preliminary sequence of samples with same values

Disadvantages

- Polling
- Length of window depends on uC frequency

2.4.2. Option 2: Debouncing timer

Block input after first occurrence for bouncing time tB

- Flexible solution
- No dependency on clock frequency
- Doesn't need processing power
- Works also using interrupts
- More complex
- (Software-) timer needed

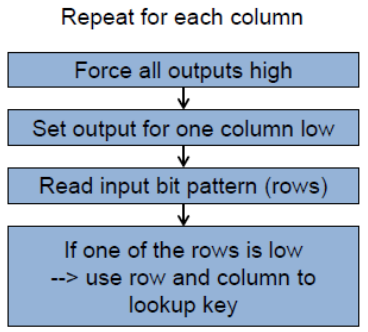
3. Matrix Keyboards

Direct connection to GPIO input -> many GPIO ports

- One input port per switch
- *n* input ports for *n* switches

Connecting multiple switches

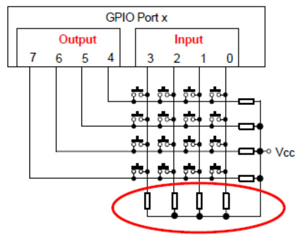
- Most efficient way for more than 5 switches
- Best layout: number of rows = number of columns


Repeat for each column
Force all outputs high
Set output for one column low
Read input bit pattern (rows)
If one of the rows is low --> use row and column to lookup key

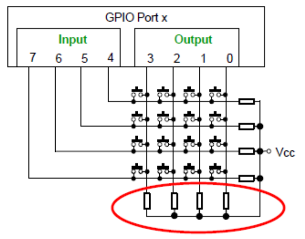
3.1.1. Fast scanning algorithm

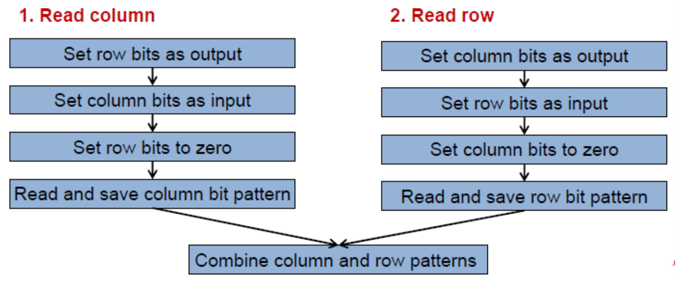
- Read in only two steps
- Not possible with each microcontroller (Changing input / output)

1. Read column



2. Read row





3.1.2. Using interrupt to avoid polling

Detect key stroke

- All outputs are forced low (initialization)
- No key pressed -> all inputs read high
- Using interrupt to detect key stroke
- Key pressure will start scanning algorithm

Scanning algorithm

- Both algorithms possible

4. Power Management and Low Power App.

4.1. Characteristics of low-power systems

- **Low dynamic/operating power**
  - Dominated by transistor switching current
  - Processor, memory, clocking circuits, other analog circuits on chip
- **Low standby power**
  - Caused by leakage
  - Circuits, clocking circuits, active peripherals, analog systems, RAM retention power
- **High energy efficiency**
  - Ratio between processing capability and power consumption
  - Balance between performance and power usage
- **Wakeup latency**
  - Delay before processor can resume operation after sleep mode
  - Can be critical for some applications

4.2. Power consumption basics

Power to switch capacitive load C:

$$P_{dyn} = V_{DD} * I_{dyn} = V_{DD}^2 * C * f_{clk}$$

Switching power is

- Proportional to $\boldsymbol{C}, \boldsymbol{fclk}$
- Proportional to the square of $\boldsymbol{VDD}$

4.2.1. Shrinking semiconductor process dim.

Increase performance

- Allows higher switching frequencies

Lower the supply voltage

- Reduces the dynamic power consumption
- $P_{1.8} = \frac{1.8^2}{2.5^2} * P_{2.5} = 0.52 * P_{2.5}$

Increase the power density

- I.e. the chip gets hot-spots

Increase leakage currents

Operate at low voltage to minimize dynamic power

$$P_{dyn} = V_{DD}^2 * C * f_{clk}$$

4.3. Static power P_{stat}

(1) Leakage of CMOS gates

- Subthreshold conduction
- Transistors have a small current even if they are off
- Smaller process geometries have lower $\boldsymbol{VDD}$ but higher leakage currents
- Ballpark for leakage currents of a chip: micro-Amperes (μA)

(2) Power consumed by non-CMOS components

E.g. on a typical microcontroller there are sense amplifiers, voltage references, constant current sources, voltage regulators that contribute to overall static power.

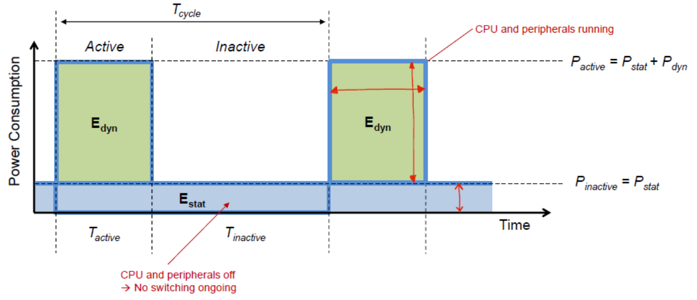
$$P_{stat} = V_{DD} * I_{stat}$$

Static power is:

- Proportional to $\boldsymbol{VDD}$
- Independent of switching frequency
- The power you have even if no switching is going on

4.4. Dynamic and static power consumption

Typical application: Active (run) and inactive phases alternate



Energy per cycle

$$E_{dyn} = P_{dyn} * T_{active}$$

$$E_{stat} = P_{stat} * (Tactive + T_{inactive})$$

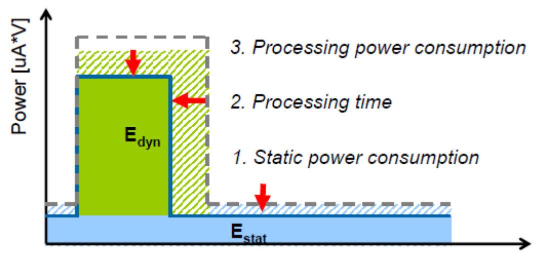
4.4.1. Units and Orders of Magnitude

Voltage	V Volt		0.9V-5V
Current	A Ampere		Dyn: mA Stat: μA-mA
Power	W Watt	W=V*A =J/s	Dyn: mW Stat: μW-mW
Energy	J Joule	J=w*s	Dyn (1s): mJ Stat (1s): μJ-mJ

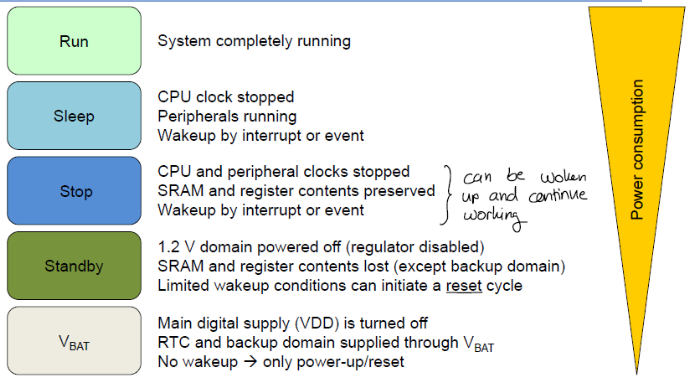
the multi-meter measures the average consumption because it is to slow to measure exact values.

4.5. Reducing energy demand

Reduce the area below the curve



4.6. STM32F4 Low Power Modes



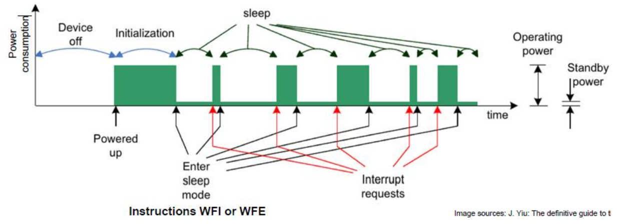
Run	System completely running
Sleep	CPU clock stopped Peripherals running Wakeup by interrupt or event
Stop	CPU and peripheral clocks stopped SRAM and register contents preserved Wakeup by interrupt or event } can be woken up and continue working
Standby	1.2 V domain powered off (regulator disabled) SRAM and register contents lost (except backup domain) Limited wakeup conditions can initiate a reset cycle
V_{BAT}	Main digital supply (VDD) is turned off RTC and backup domain supplied through V_{BAT} No wakeup → only power-up/reset

Turn-off the CPU clock – CPU still powered
Peripherals powered and clocked

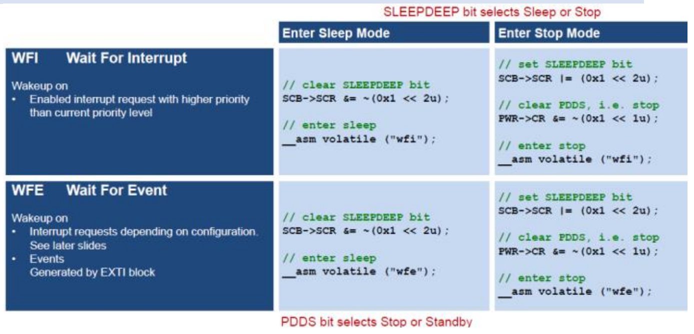
4.7. Entering Low Power Modes and Wakeup

Interrupt-driven system

- CPU completes task and enters sleep, stop or standby mode
- Assembly instructions WFI and WFE suspend execution until wakeup by interrupt request or event
- At wakeup, CPU resumes execution ➔ quite often by directly calling the associated ISR



4.7.1. Enter Sleep Stop Mode with WFI and WFE



SLEEPDEEP bit selects Sleep or Stop

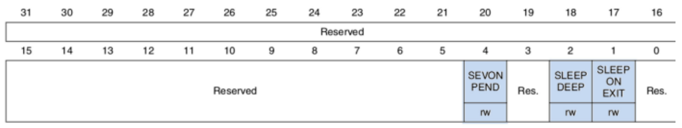
	Enter Sleep Mode	Enter Stop Mode

WFI Wait For Interrupt
Wakeup on
• Enabled interrupt request with higher priority than current priority level
	`// clear SLEEPDEEP bit`
	`SCB->SCR &= ~(0x1 << 2u);`
	`// enter sleep`
	`__asm volatile ("wfi");`
		`// set SLEEPDEEP bit`
		`SCB->SCR |= (0x1 << 2u);`
		`// clear PDDS, i.e. stop`
		`PWR->CR &= ~(0x1 << 1u);`
		`// enter stop`
		`__asm volatile ("wfi");`

WFE Wait For Event
Wakeup on
• Interrupt requests depending on configuration. See later slides
• Events Generated by EXTI block
	`// clear SLEEPDEEP bit`
	`SCB->SCR &= ~(0x1 << 2u);`
	`// enter sleep`
	`__asm volatile ("wfe");`
		`// set SLEEPDEEP bit`
		`SCB->SCR |= (0x1 << 2u);`
		`// clear PDDS, i.e. stop`
		`PWR->CR &= ~(0x1 << 1u);`
		`// enter stop`
		`__asm volatile ("wfe");`

PDDS bit selects Stop or Standby

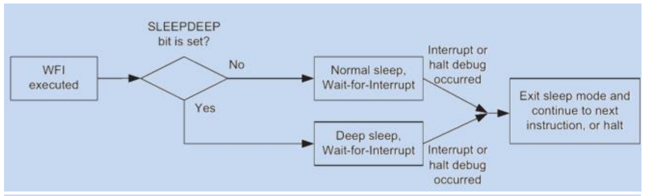
4.7.2. System Control Register (SCR)

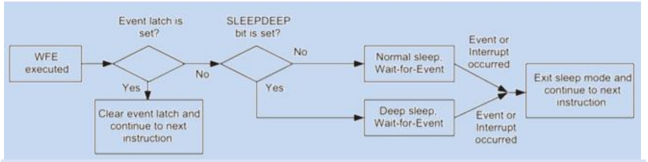


31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															

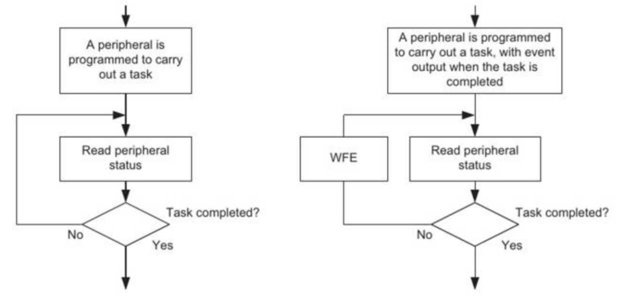
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved											SEVON PEND	Res.	SLEEP DEEP	SLEEP ON EXIT	Res.
											rw		rw	rw	

- Address: 0xE000ED10 (this is a register defined by ARM and not by ST)
- SEVONPEND: Send Event on Pending bit only relevant for WFE, not for WFI
  - 0: Only enabled interrupts or events can wakeup the processor
  - 1: Enabled events and all interrupts, including disabled interrupts, can wakeup the processor
- SLEEPDEEP
  - 0: Sleep mode
  - 1: Deep sleep mode
- SLEEPONEXIT
  - 0: Do not sleep when returning to Thread mode.
  - 1: Enter sleep, or deep sleep, on return from an interrupt service routine.

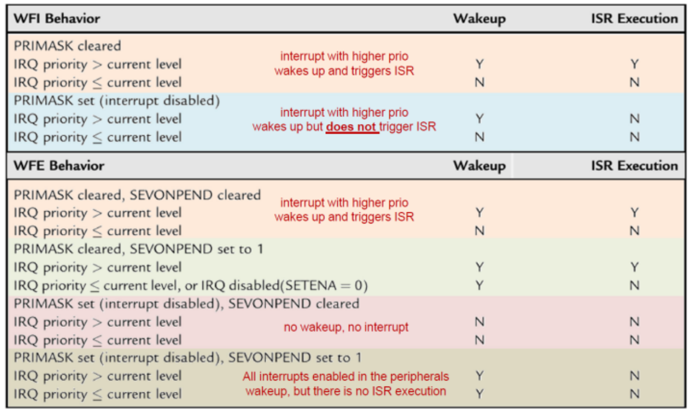

WFI executed
SLEEPDEEP bit is set?
No
Yes
Normal sleep, Wait-for-Interrupt
Deep sleep, Wait-for-Interrupt
Interrupt or halt debug occurred
Interrupt or halt debug occurred
Exit sleep mode and continue to next instruction, or halt


WFE executed
Event latch is set?
Yes
No
Clear event latch and continue to next instruction
SLEEPDEEP bit is set?
No
Yes
Normal sleep, Wait-for-Event
Deep sleep, Wait-for-Event
Event or Interrupt occurred
Event or Interrupt occurred
Exit sleep mode and continue to next instruction

4.7.3. Using WFE in polling loops


A peripheral is programmed to carry out a task
Read peripheral status
Task completed?
No
Yes
A peripheral is programmed to carry out a task, with event output when the task is completed
WFE
Read peripheral status
Task completed?
No
Yes

Without WFE, a polling loop consumes power and results in lower energy efficiency
With WFE, power consumption by polling loop is significantly reduced

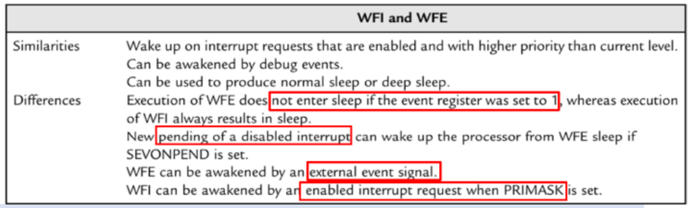


WFI Behavior		Wakeup	ISR Execution
PRIMASK cleared			
IRQ priority > current level	interrupt with higher prio wakes up and triggers ISR	Y	Y
IRQ priority ≤ current level		N	N
PRIMASK set (interrupt disabled)			
IRQ priority > current level	interrupt with higher prio wakes up but **does not** trigger ISR	Y	N
IRQ priority ≤ current level		N	N
WFE Behavior		**Wakeup**	**ISR Execution**
PRIMASK cleared, SEVONPEND cleared			
IRQ priority > current level	interrupt with higher prio wakes up and triggers ISR	Y	Y
IRQ priority ≤ current level		N	N
PRIMASK cleared, SEVONPEND set to 1			
IRQ priority > current level		Y	Y
IRQ priority ≤ current level, or IRQ disabled(SETENA = 0)		Y	N
PRIMASK set (interrupt disabled), SEVONPEND cleared			
IRQ priority > current level	no wakeup, no interrupt	N	N
IRQ priority ≤ current level		N	N
PRIMASK set (interrupt disabled), SEVONPEND set to 1			
IRQ priority > current level	All interrupts enabled in the peripherals wakeup, but there is no ISR execution	Y	N
IRQ priority ≤ current level		Y	N

- PRIMASK: Priority Mask Register in the NVIC Enable/disable all interrupts
- SEVONPEND Send Event on Pending bit in system control register (SCR)
- NVIC Nested Vectored Interrupt Controller

All interrupts enabled in the peripherals wakeup. However, execution of ISR depends on configuration in NVIC

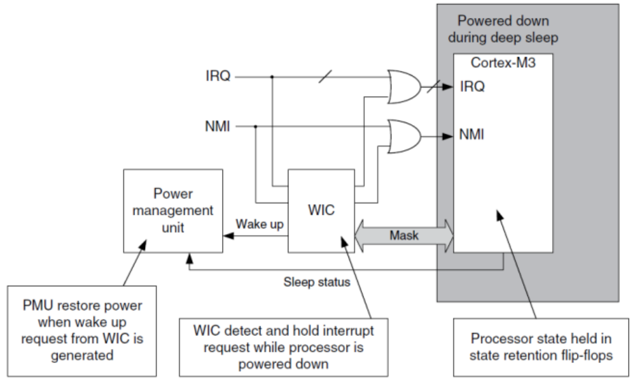
Additionally, in case of WFE, the system will wake up in case of an event in EXTI.



	WFI and WFE
Similarities	Wake up on interrupt requests that are enabled and with higher priority than current level. Can be awakened by debug events. Can be used to produce normal sleep or deep sleep.
Differences	Execution of WFE does not enter sleep if the event register was set to 1, whereas execution of WFI always results in sleep. New pending of a disabled interrupt can wake up the processor from WFE sleep if SEVONPEND is set. WFE can be awakened by an external event signal. WFI can be awakened by an enabled interrupt request when PRIMASK is set.

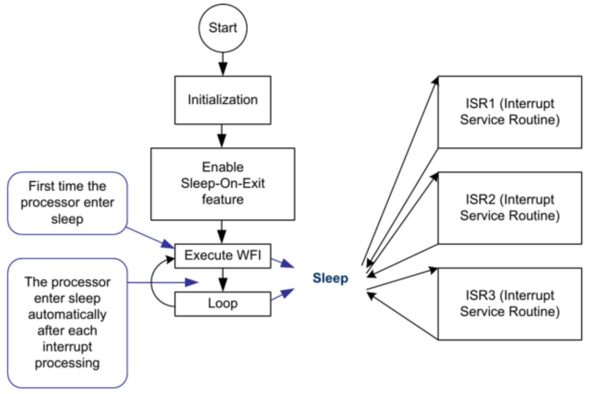
4.7.4. Wakeup Interrupt Controller (WIC)

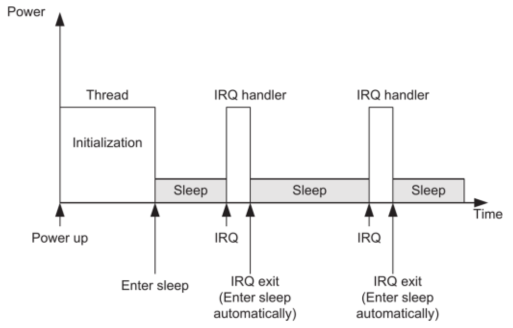
- A standardized ARM peripheral
- WIC mirrors the Interrupt detection function when Cortex-M in low power mode
- Restore power and clock when binterrupt / event detected


Powered down during deep sleep
Cortex-M3
IRQ
NMI
IRQ
NMI
Power management unit
Wake up
WIC
Mask
Sleep status
PMU restore power when wake up request from WIC is generated
WIC detect and hold interrupt request while processor is powered down
Processor state held in state retention flip-flops

4.7.5. Sleep on exit

- Enter Wait-for-Interrupt (WFI) sleep mode when exiting an exception handler
- System control register (SCR) -> SLEEPONEXIT bit


Start
Initialization
Enable Sleep-On-Exit feature
First time the processor enter sleep
Execute WFI
Loop
The processor enter sleep automatically after each interrupt processing
Sleep
ISR1 (Interrupt Service Routine)
ISR2 (Interrupt Service Routine)
ISR3 (Interrupt Service Routine)



4.8. Sources for Wakeup

Wakeup examples in Sleep Mode

- CPU clock stopped
- Peripherals running
  - They generate interrupts or events
- transmission of byte completed on UART
- timer expires
- analog watchdog triggers i.e. analog value is outside programmed thresholds
- DMA transfer completed
- byte received on SPI
- edge on GPIO

4.8.1. Wakeup sources in Stop mode

- CPU and peripherals in low power mode ➔ i.e. peripherals do not generate interrupts
- EXTI provides 23 sources for wakeup
- EXTI: External Interrupt Controller

4.9. Real Time Clock

Runs from independent 32kHz clock

Slow clock consumes low amount of power

Two programmable alarms A + B
→ Generate interrupt / wakeup event when date and time is reached

Calendar function

DR	Date Register	Year YY
		Week day
		Month MM
		Day DD
TR	Time Register	AM/PM
		Hours HH
		Minutes MM
		Seconds SS
SSR	Sub Second Register	

Wake-up timer
→ 16-bit programmable auto-reload down-counter
→ Generates periodic interrupt / wake-up event

Interrupt-driven system in stop

- Use RTC alarms for scheduled wake-ups
- Use wake-up counter for periodic wake-up
- Use GPIOs (through EXTI) to trigger wake-up through external events

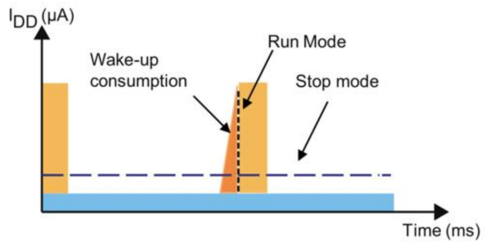
Alternatively, connect external RTC through GPIO/EXTI for wake-up

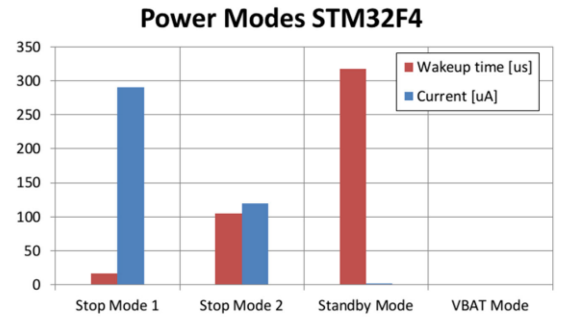
4.10. Power Saving Techniques

4.10.1. Using low-power modes

- Keep the device as much as possible in low-power modes
- Best power management approach
  - Switching between different power modes
  - Simultaneously taking into account the application requirements

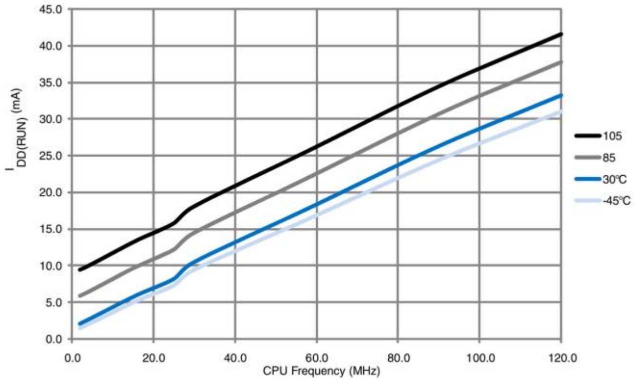
▶ power consumption
▶ wakeup sources/time
▶ peripherals





System clock

- Power/current consumption depends on switching frequency
- Static power strongly depends on temperature (offset of curves)



The dynamic energy required to complete a task with a defined number of clock cycles is independent of the clock frequency

$$E_{dyn} = P_{dyn} * T_{active} = V_{DD}^2 * C * f_{clk} * \frac{N}{f_{clk}}$$
$$= V_{DD}^2 * C * N$$

High clock frequency allows to go to sleep earlier Reduces Tactive

Individual choice of prescaler dividers can save energy. Not all the parts require the same clock frequencies

4.10.2. Reduce switching on pins

Chip pins typically have high supply voltage (3.3V) and high capacitive load

$$I_{SW} = V_{DD} * f_{SW} * C$$

Reduce switching on clock pins by using a PLL (Phase Locked Loop

4.10.3. Disable unused peripherals

E.g. in RCC APB1 peripheral clock enable register (RCC_APB1ENR)

I/O configuration

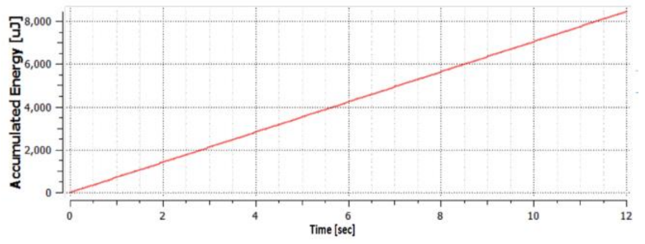
- Unused Pins Configure as analog inputs
  - Schmitt trigger input disabled Zero consumption for I/O pin
- Avoid pull-up and pull-down activation if not used
- Output I/O speed frequency at lowest possible value
- Disable clock output pins if not used

4.11. Power Debugging

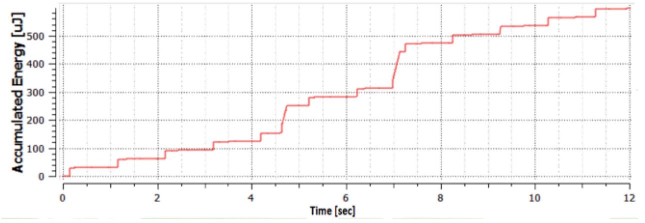
Potential issues

- Floating I/O
  - Highly variant, especially at low temperatures or high humidity
- Unexpected current sinking/sourcing
  - Voltage-level mismatch
  - Powered off ICs
  - LEDs
- Analog peripherals
  - Constant current when enabled
  - Often active at sleep
- Spurious wakeups
  - Unexpected interrupts reduce time in low-power modes

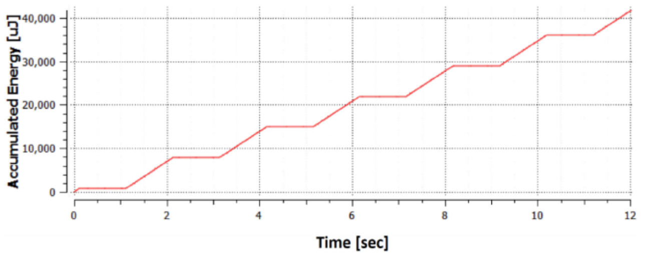
Typical static power consumption
Gradient indicates high static power consumption



Unexpected wakeups requiring different amounts of energy



Unwanted periodic high energy drain



Dynamic Power ➔ Reduce processing power consumption

4.12. Examples

5. Digital Sensors

5.1. MEMS Sensors

→Micro Electro Mechanical Systems

- Integrate mechanical & electronic parts in single component
  - Directly placed on PCB
- Digital output
  - No ADC required

5.1.1. Advantages

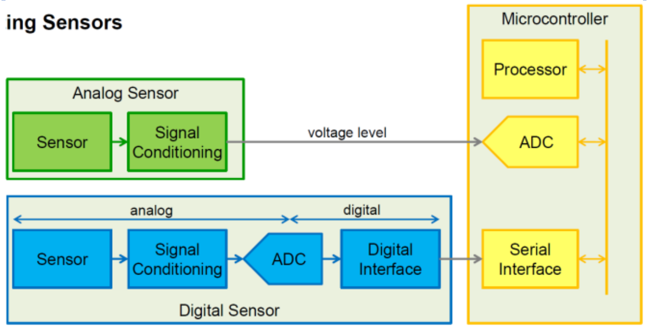
- Small, miniaturized, highly integrated
- Low-cost, low power

5.1.2. Disadvantages

- Less accurate than bigger sensors

5.2. Connecting Sensors

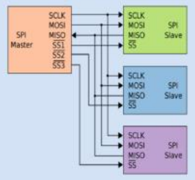
ing Sensors



5.2.1. SPI

SPI → 4 wires

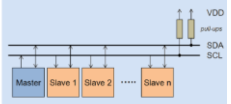
- Serial bus for on-board connections
- Master (single master)
  - Generates clock (SCLK)
  - Initiates data transfer ($\overline{SS}$)
- MOSI: Master Out Slave In
- MISO: Master In Slave Out
- Full-duplex
- Up to 10 Mbit/s



5.2.2. I2C

I2C → 2 wire interface

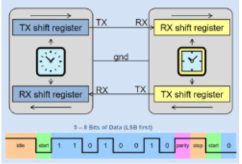
- Clock → SCL
- Data → SDA
- Each slave addressable through unique 7/10-bit address
- 8-bit oriented data transfers
- Half-duplex
- Up to 3.4 Mbit/s
- Derivative: SMBUS Simple communication in PC: Fan, temperature, etc.



5.2.3. UART

UART

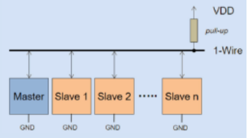
- Serial port
- Asynchronous
  - Synchronization of each byte with start/stop bit
- Full-duplex
- Typical baud rates of 9600, 19200, 38'400, 115'200 … bit/s
- PC terminal programs



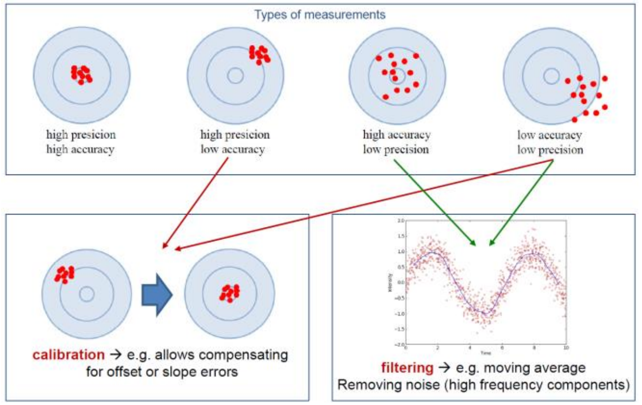
5.2.4. 1-Wire

1-Wire

- Communication through single wire (and common ground)
- Dallas Semiconductor (Today Maxim Integrated)
- Each device has unique 64-bit address
- 1-Wire line powers slaves
- Few compatible devices
- Half-duplex
- Internal clock, 100 kbit/s



5.3. Processing in MC


Types of measurements
high presicion
high accuracy
high presicion
low accuracy
high accuracy
low precision
low accuracy
low precision
calibration → e.g. allows compensating for offset or slope errors
filtering → e.g. moving average
Removing noise (high frequency components)
Intensity
Time

5.3.1. Calibration

Compensate for offsets by:

- Scaling: Linear transformation
- Conversion: conversion of units / data format

5.3.2. Filtering

- Removing noise

5.4. Accessing Registers

5.4.1. Writing

```
#include "hal_spi.h"
#define  CTRL1_XL_ADDR    0x10
...

static example_write(void)
{
    accelerometer_reg_write(CTRL1_XL_ADDR, 0x04);
}

static void accelerometer_reg_write(uint8_t address, uint8_t value)
{
    uint8_t tx_data[2];
    uint8_t rx_data[2]; // will be ignored - but needed as param in

    tx[0] = address;
    tx[1] = value;

    hal_spi_read_write(2, tx_data, rx_data);
}
```

Name
CTRL1_XL

```
void hal_spi_read_write(uint16_t nr_of_bytes, uint8_t *tx_buffer, uint8_t *rx_buffer);
```

5.4.2. Reading

```
static example_read(void)
{
    uint8_t reg_value;
    reg_value = accelerometer_reg_read(CTRL1_XL_ADDR);
    ...
}

static uint8_t accelerometer_reg_read(uint8_t address)
{
    uint8_t tx_data[2];
    uint8_t rx_data[2];

    tx[0] = address | 0x80;   // bit 7 = 1 for read
    tx[1] = 0;

    hal_spi_read_write(2, tx_data, rx_data);
    return rx_data[1];
}
```

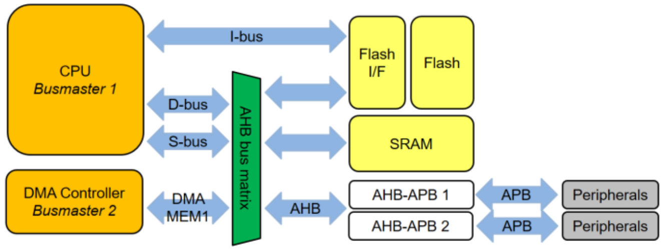
6. DMA - Direct Memory Access

How do I transfer data between SPI and memory?
Polling or Interrupt

- Direct Memory Access (DMA) as an option to reduce the load on the CPU

DMA functionality

- Provide high-speed data transfer between one or more units
- No CPU action
- Keeps CPU resources free for other operations
- DMA is an additional bus master



- Direct memory access (DMA) used to provide high-speed data transfers
  - peripherals-to-memory
  - memory-to-peripheral
  - memory-to-memory (only DMA 2)
- DMA moves Data without any CPU interventions
  - Keeps CPU resources free for other operations

- DMA controller combines powerful dual AHB master bus architecture with independent FIFO to optimize the bandwidth of the system
  - Based on a complex bus matrix architecture.
- The two DMA controllers have 16 streams in total (8 for each controller)
  - Each dedicated to managing memory access requests from one or more peripherals.
  - Each stream can have up to 8 channels (requests) in total. And each has an arbiter for handling the priority between DMA requests.

6.1. Peripheral-to-memory DIR = '00'

- 'Interrupt' condition (request) on peripheral triggers transfer
- Load data from peripheral into FIFO
  - Source address defined by peripheral address register (SxPAR)
  - Byte, half-word or word depending on configuration
- Store data to memory
  - Immediately in direct mode, or otherwise if programmed FIFO level reached
  - Destination address defined by memory address register (SxM0AR)
  - Byte, half-word or word depending on configuration

6.2. Memory-to-peripheral DIR = '01'

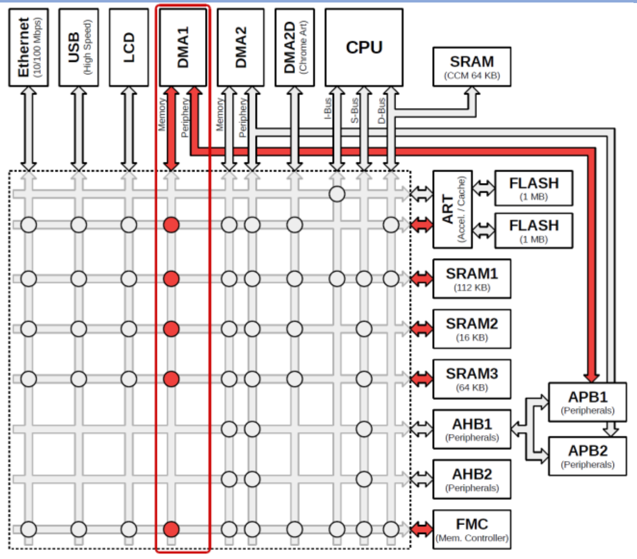
- Transfer starts immediately after stream enable
- Pre-load data from memory into FIFO
  - Source address defined by memory address register (SxM0AR)
  - Byte, half-word or word depending on configuration
- Each time a peripheral request occurs, data from FIFO is stored to peripheral
  - Destination address defined by peripheral address register (SxPAR)
  - Byte, half-word or word depending on configuration
- Reload of FIFO from memory
  - Direct mode: Immediately after store
  - Otherwise as soon as FIFO level is 10 below programmed level

6.3. Memory-to-memory DIR = '10'

- Transfer starts immediately after stream enable
- Load data from memory into FIFO
  - Source address defined by peripheral address register (SxPAR)
  - Byte, half-word or word depending on configuration
- When programmed FIFO level is reached, data is stored to memory
  - Destination address defined by memory address register (SxM0AR)
  - Byte, half-word or word depending on configuration

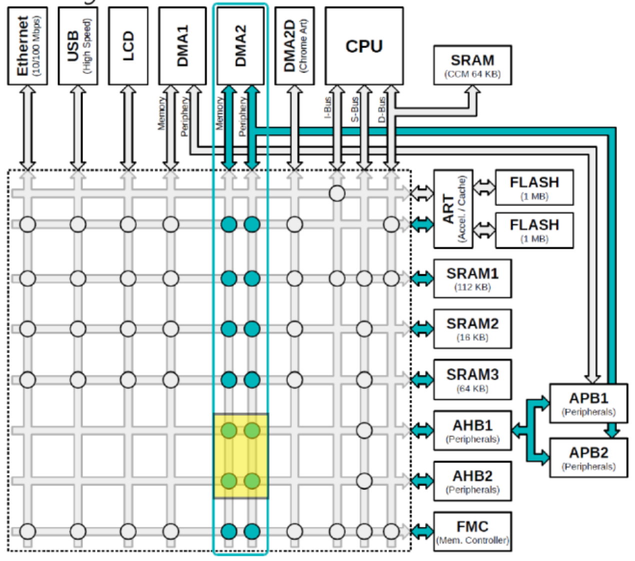
Direct mode and circular mode are not allowed

6.4. Supported Transfers and Requests



DMA2 Allows memory-to
memory access

AHB Advanced High-performance Bus
APB Advanced Peripheral Bus



DMA2: Peripherals on APB2 OR any other component on the bus matrix

DMA 1 Request Mapping

	Stream 0	Stream 1	Stream 2	Stream 3	Stream 4	Stream 5	Stream 6	Stream 7
Channel 0	SPI3 RX		SPI3 RX	SPI2 RX	SPI2 TX	SPI3 TX		SPI3 TX
Channel 1	I2C1 RX		TIM7 Up		TIM7 Up	I2C1 RX	I2C1 TX	I2C1 TX
Channel 2	TIM4 Ch. 1		I2S3 Ext RX	TIM4 Ch. 2	I2S2 Ext TX	I2S3 Ext TX	TIM4 Up	TIM4 Ch. 3
Channel 3	I2S3 Ext RX	TIM2 Up TIM2 Ch. 3	I2C3 RX	I2S2 Ext RX	I2C3 TX	TIM2 Ch. 1	TIM2 Ch. 2 TIM2 Ch. 4	TIM2 Up TIM2 Ch. 4
Channel 4	UART5 RX	USART3 RX	UART4 RX	USART3 TX	UART4 TX	USART2 RX	USART2 TX	UART5 TX
Channel 5	UART8 TX	UART7 TX	TIM3 Ch. Up TIM3 Ch. 4	UART7 RX	TIM3 TRG TIM3 Ch. 1	TIM3 Ch. 2	UART8 RX	TIM3 Ch. 3
Channel 6	TIM5 Up TIM5 Ch. 3	TIM5 TRG TIM5 Ch. 4	TIM5 Ch. 1	TIM5 TRG TIM5 Ch. 4	TIM5 Ch. 2		TIM5 Up	
Channel 7		TIM6 Up	I2C2 RX	I2C2 RX	USART3 TX	DAC1	DAC2	I2C2 TX

DMA 2 Request Mapping

	Stream 0	Stream 1	Stream 2	Stream 3	Stream 4	Stream 5	Stream 6	Stream 7
Channel 0	ADC1	SAI1 A	TIM8 Ch. 1 TIM8 Ch. 2 TIM8 Ch. 3	SAI1 A	ADC1	SAI1 B	TIM1 Ch. 1 TIM1 Ch. 2 TIM1 Ch. 3	
Channel 1		DCMI	ADC2	ADC2	SAI1 B	SPI6 TX	SPI6 RX	DCMI
Channel 2	ADC3	ADC3		SPI5 RX	SPI5 TX	CRYP Out	CRYP In	HASH In
Channel 3	SPI1 RX		SPI1 RX	SPI1 TX		SPI1 TX		
Channel 4	SPI4 RX	SPI4 TX	USART1 RX	SDIO		USART1 RX	SDIO	USART1 TX
Channel 5		USART6 RX	USART6 RX	SPI4 RX	SPI4 TX		USART6 TX	USART6 TX
Channel 6	TIM1 TRG	TIM1 Ch. 1	TIM1 Ch. 2	TIM1 Ch. 1	TIM1 TRG TIM1 COM TIM1 Ch. 4	TIM1 Up	TIM1 Ch. 3	
Channel 7		TIM8 Up	TIM8 Ch. 1	TIM8 Ch. 2	TIM8 Ch. 3	SPI5 RX	SPI5 TX	TIM8 TRG TIM8 COM TIM8 Ch. 4

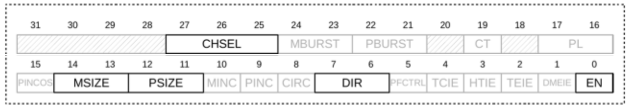
Only 8 predefined requests can trigger an individual stream

E.g. it is not possible to trigger Stream2 by ADC1

6.5. DMA Configuration

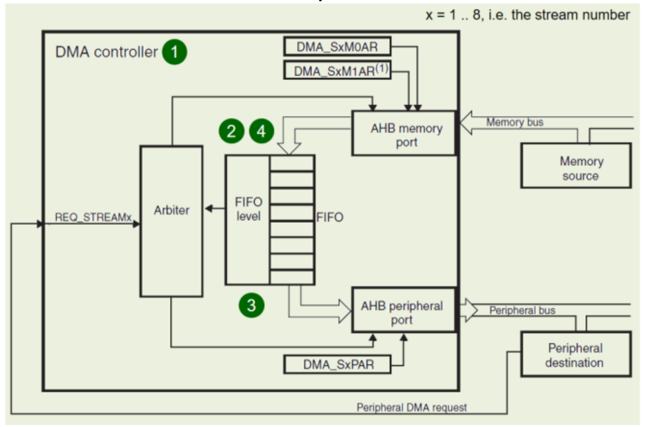
DMA stream x configuration register (DMA_SxCR)

- CHSEL[2:0] Channel selection 0-7
- MSIZE[1:0] Memory data size
- PSIZE[1:0] Peripheral data size
  - 00: 8-bit / 01: 16-bit / 10: 32-bit
- MINC/PINC Memory/Peripheral increment mode
  - 0: address pointer is fixed
  - 1: address pointer is incremented after each data transfer
- DIR[1:0] Data transfer direction
  - 00: Peripheral-to-memory
  - 01: Memory-to-peripheral
  - 10: Memory-to-memory
- EN Stream enable



Pointer increment – MINC and PINC bits in SxCR

- Configure in SxCR whether source and/or destination addresses need to be incremented
- Increment is done after transfer
- Value of increment depends on transfer size



1. Disable stream, i.e. reset EN bit in SxCR Read SxCR until EN == 0 ⍰ all transfers finished 1) Clear interrupt status registers LISR and HISR
2. Set peripheral address register SxPAR
3. Set memory address register SxM0AR
4. Configure number of transfers in SxNDTR
5. Select DMA channel request in CHSEL[2:0] in SxCR
6. Configure stream priority PL[1:0] in SxCR
7. Configure FIFO usage
8. Configure data transfer direction, data widths, interrupts, increments and others in SxCR
9. Activate the stream by setting EN in SxCR

ST HAL: Using structs and functions

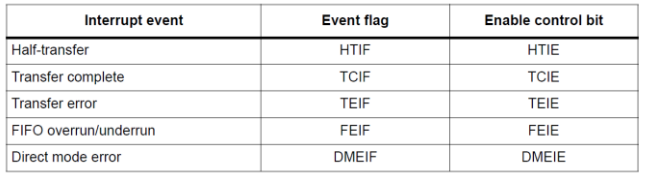
- Instantiate a handle
- Use handle to configure and start DMA

DMA FIFOs

- Each stream has an independent 4-word FIFO, i.e. a total of 16 bytes
- Transfer sizes on source and destination can be different
  - byte vs. half-word vs. word
- Threshold levels can be programmed
- Direct mode does not use FIFO

6.5.1. DMA interrupts

- For each stream, an interrupt can be produced in the following events ⍰ registers LISR and HISR
- Transfer complete
  - The programmed number of transfers has been completed
  - Tell the CPU to take back control
- Transfer error
  - E.g. a bus error during a DMA access on the bus



Interrupt event	Event flag	Enable control bit
Half-transfer	HTIF	HTIE
Transfer complete	TCIF	TCIE
Transfer error	TEIF	TEIE
FIFO overrun/underrun	FEIF	FEIE
Direct mode error	DMEIF	DMEIE

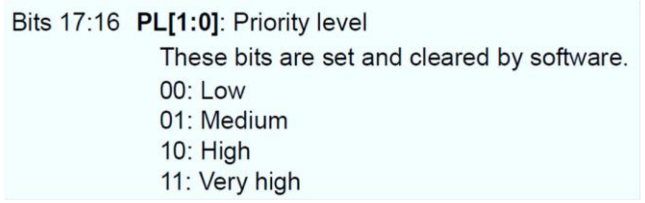
6.6. Arbitration

CPU and DMA1 both require the bus to SRAM1

- Several masters (including CPU and DMA1) can request AHB bus in case they have data to transfer
- An arbiter decides who gets the bus based on a round-robin scheme (bus grant)

DMA internal arbiter

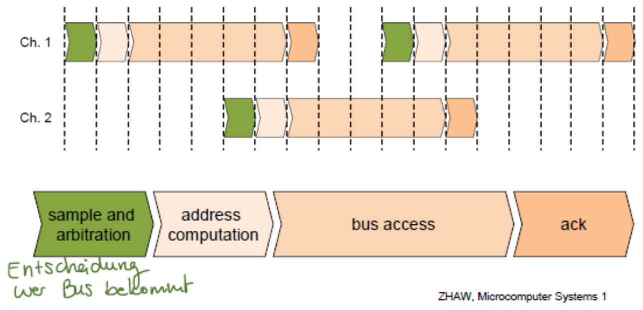
- Manages the 8 streams based on the programmed priority
- Then, generates request to AHB bus
- Priority levels can be programmed per stream



Bits 17:16 **PL[1:0]**: Priority level
These bits are set and cleared by software.
00: Low
01: Medium
10: High
11: Very high

6.6.1. Parallel transfers

Pipeline Transfer



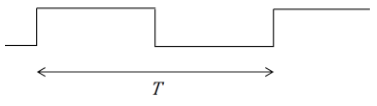
6.7. Power savings with DMA

Sleep with DMA

- Put CPU into sleep
- Let DMA service “interrupts” from peripherals
  - E.g. transfer data received on I2C to memory
- DMA issues an interrupt after programmed number of transfers
  - Wakes up the CPU

7. Timer Applications

Frequency of Digital Signals



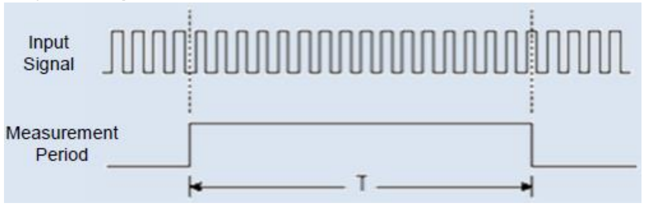
$$f = \frac{1}{T} \qquad T = \text{Period}$$

$$frequency = \frac{number\ of\ events}{time\ in\ seconds} \qquad Hz = \frac{1}{s}$$

Direct frequency counting

Count number of edges of an input signal over a defined period of time (T)

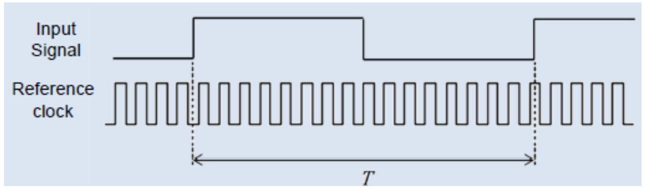
ARR – Auto Reload Register defines measurement period. Adapt measurement window (ARR) depending on Timer 2 values



Reciprocal frequency counting

- Count periods of a fast reference clock
- Edges of input signal start and stop counter
- Objective: Measure *fsignal on GPIO pin* ⍰ choose *fcounter >> fsignal*
- Rising edge on GPIO pin captures value of Counter and resets Counter
- CCR contains the number of internal clock pulses during a period on the input pin

I.e., choose Reference clock much faster than Signal to be measured



Advantages / disadvantages reciprocal counting

- More complex, higher hardware requirements
  - ‘Floating point’ routines to calculate the frequency
- Higher resolution
  - Fixed in multiples of counter clock
  - Independent of input signal frequency (as long as *fcounter >> fsignal*)

Note: Accuracy depends on the clock stability of the unit

7.1. Frequency Multiplication

- Use frequency multiplier to generate an output with a higher frequency
- Measure frequency using timer 1
  - Reciprocal frequency counting
- Generate output with timer 2
  - Reload timer 2 with capture value of timer 1
  - Timer 2 runs on higher frequency (fcounter_2)

7.2. Software Timers

Problem

- Applications often need multiple timing elements
- A microcontroller only has a limited number of HW-timers
- Implementing numerous timing elements with individual HW-timers results in complex interrupt structures
  - Many interrupts
  - Potential problems during run time
  - Complex testing

Solution: Software timer

- A single hardware timer with a single interrupt service routine (ISR)
- Timing elements implemented as software timers

8. Structuring Embedded Software

8.1. Modularity

- Partition functionality into manageable chunks
- Hierarchical design
- Group together what belongs together
- Lean external interface
- Each module fulfills a **single** defined task

High cohesion

- Strength of relationship between functions and data of a module
- Module functions and data shall have much in common

Low coupling

- Minimize dependencies between modules ⍰ Avoid circular dependencies
- Lean interfaces

8.1.1. Encapsulation and Data Hiding

- Split interface from implementation (see also previous lecture on definition vs. declaration)
- Do not disclose implementation details
- Maintain freedom to change implementation at any time

8.1.2. Layering

- Hierarchical dependencies
- Each layer provides a specific service to the upper layer
- Upper layer uses a service from the lower layer

8.1.3. Layering – Call-backs

- Inversion of control
    - Lower layer calls function in higher level
- Specify function to be called as an argument, i.e., a function pointer

10 Qualities of Portable Firmware

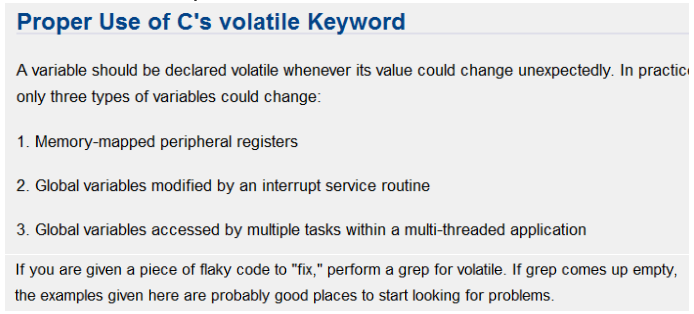
Portable Firmware ….

1) is modular
2) is loosely coupled
3) has high cohesion
4) is ANSI-C compliant
5) has a clean interface
6) has a Hardware Abstraction Layer (HAL)
7) is readable and maintainable
8) is simple
9) uses encapsulation and abstract data types
10) is well documented

8.2. Passing Data from/to ISR

8.2.1. Global variable

- Code works fine with optimization level 0
    - Number of executed interrupts is correctly displayed on LED
- Then you crank up the optimization level of your compiler
    - LEDs always show a constant value



Proper Use of C's volatile Keyword

A variable should be declared volatile whenever its value could change unexpectedly. In practic only three types of variables could change:

1. Memory-mapped peripheral registers

2. Global variables modified by an interrupt service routine

3. Global variables accessed by multiple tasks within a multi-threaded application

If you are given a piece of flaky code to "fix," perform a grep for volatile. If grep comes up empty, the examples given here are probably good places to start looking for problems.

However, volatile does not prevent data consistency issues of shared objects

8.2.2. Register a private variable

- Memory for counter is allocated in main program
- Address is passed as a parameter in **init_irq()**
- Module *interrupt* stores the parameter in a private variable

`static` is a storage class specifier for the variable being defined, here this is `count_ptr`. Therefore `count_ptr` is only visible in module interrupt `volatile` is a type qualifier for the type where `count_ptr` points to.

8.2.3. Getter function (accessor)

Provides encapsulation

Advantage: Access through single channel

- Avoids accesses by mistake e.g. by confusing variable names
- Easier debugging
    - Allows setting a breakpoint or adding a logging function1)
    - Find out from which module an access takes place
- Setter: Allows implementing a validation of input parameters, e.g., a range check, at a single location
    - As opposed to duplicating your range checks all over your code
- Make access atomic: Possibility to turn-off/on interrupts in central place

8.2.4. Register a call-back

ISR calls a function in main module
handler() can be changed without modification of module interrupt

8.2.5. Queue

- **main()** instantiates queue and passes the pointer to module interrupt
- Module *interrupt* enqueues data; **main()** dequeues data
- Synchronization needs to be solved in module *queue*

8.3. ISR Data Consistency Issues

Interrupts may occur at any time during execution

- Between any two assembly instructions
- **Within** a C-statement

Interrupt service routines (ISR)

- Have to be treated as parallel threads1)
  - I.e., as a sequence of instructions that can run in parallel to the main program flow
- Require synchronization to the main program

Data consistency needs special attention

- Verification of interrupt service routines is challenging
- Timing constraints
- Priorities

Keep ISRs brief use as few parallel interrupts as required

Problem may prove difficult to reproduce / debug

- Chances are it only occurs every few hours or even every few days
- Before you fix the problem, you should reproduce it
  - In a reliable and (if possible) automated way
- What can you do to reproduce it?
  - You do not want to spend hours looking at the displays

Critical section

- A piece of a program that may not be interrupted
- Access to the shared resource needs to be protected
- ISR may not update **count** as long as main is reading it
- Reading **count** should be an *atomic* operation
  - I.e. the operation may not be interrupted

Possible fix: Disable interrupts when copying

- Drawback: Increases interrupt latency

Use disabling and re-enabling interrupts with caution

- If you use it too generously funny things can happen

Call-backs don't make shared objects go away

- ISR does not even access the variable count
- But the callback may still have a data consistency problem

9. Partitioning reactive systems

Reactive System: An embedded system reacting to external, asynchronous events. The events can occur in parallel.

9.1. Cooperating FSMs

Port

- Defines the messages that can be sent and received by an FSM
- Output message action of the FSM
- Input message event of the FSM

Link

- Defines a connection for sending messages

The actions of FSM A and of FSM B both become events for FSM C

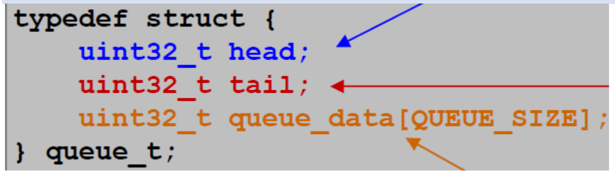
FSM with event queue

- Collect events generated by different FSMs / objects
- Buffered in event queue (FIFO)
  - Avoids losing events
  - Decouples event generation and processing of events
- FSM processes one event after the other
- Events are deleted after processing

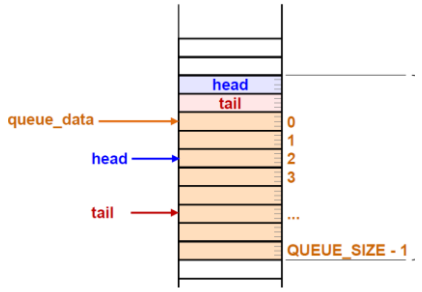
9.2. IMPLEMENTING QUEUES

- FIFO – First In, First Out
- Implemented as ring buffer / circular buffer avoids copying data
- Tail – indicates where the next element shall be inserted (write)
- Head – indicates the element that is next in line (read)

9.2.1. The data structure of a Queue



```
typedef struct {
    uint32_t head;
    uint32_t tail;
    uint32_t queue_data[QUEUE_SIZE];
} queue_t;
```



9.2.2. Methods of a Queue

void queue_init(queue_t *queue)

→initialize queue before first use

uint32_t queue_enqueue(queue_t *queue, uint32_t data)

→enqueues data at tail of queue

uint32_t queue_dequeue(queue_t *queue)

→removes element at head & moves the head tot he next element.

9.3. Code Structure of an FSM

9.3.1. FSM interface

Other tasks put events into an event queue. The scheduler periodically calls a handle_event function of the FSM, which reacts to the next event in the queue.

9.3.2. Implementation

```
#include "example_fsm.h"
#include "queue.h"

/* enumerate the states of the FSM */
typedef enum {
    STATE_A,
    STATE_B,
    STATE_C
} example_fsm_state_t;

/* event queue for this FSM */
static queue_t example_fsm_queue;

/* current state of the FSM */
static example_fsm_state_t state = STATE_A;

/*  see header file */
void example_fsm_put_queue(example_fsm_events_t event)
{
    queue_enqueue(&example_fsm_queue, event);
}
```

```
/*  see header file */
void example_fsm_handle_event(void)
{
    uint32_t event;

    event = queue_dequeue(&example_fsm_queue);

    switch (state) {
        case STATE_A:
        switch (event) {
            case EXAMPLE_FSM_EVENT_X:
                state = STATE_B;
                // ... actions
                break;

            default:
                ;   // no change
        }
      ...
```

9.4. COOPERATIVE SCHEDULER

Scheduler calls **`handle_event()`** functions

10. RTOS

- Scheduling of threads
  - Predictable and deterministic: Execute within time bounds
  - Fair access to resources
  - Provide time reference
- Communication
  - Exchange of events and data between threads
- Synchronization
  - Signaling
  - Critical sections, mutual exclusion, semaphores

Thread of execution: An independent flow of control that can be scheduled individually

10.1. Scheduler

Loads and dispatches threads to processor

- Threads get their share of execution time until
  - They block on (wait for) some I/O
  - Sleep deliberately/wait on some event
  - A higher priority thread wants to run
  - Used up their maximum time slice
- Then put into wait-queue until it's their turn again
- Scheduling algorithm
  - Various choices
  - None is "perfect"
  - Challenge is that no thread is starving, i.e., not executed due to other threads consuming all the slots

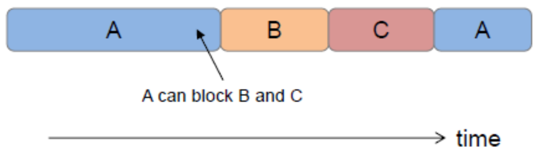
10.2. RTOS – Concepts

10.2.1. Cooperative scheduler

Picks thread and lets it run

- Until it blocks either on I/O or waiting for another process
- Or until it voluntarily releases the CPU

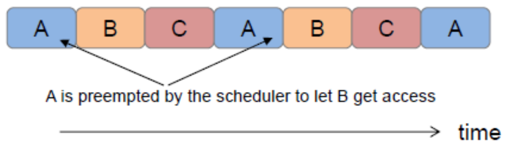
Threads will not be forcibly suspended



10.2.2. Preemptive scheduler

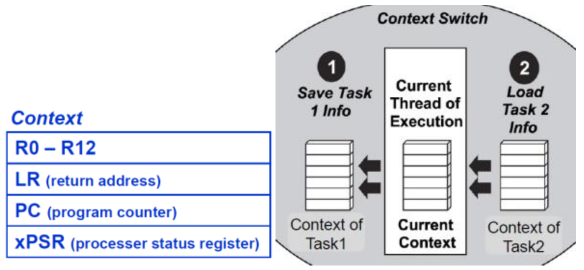
In addition, a preemptive scheduler

- May preempt the execution of a thread in favor of another thread
- E.g., the running thread is suspended for a thread with higher priority
- E.g., after it used up its time slice, the running thread is suspended for another thread of same priority

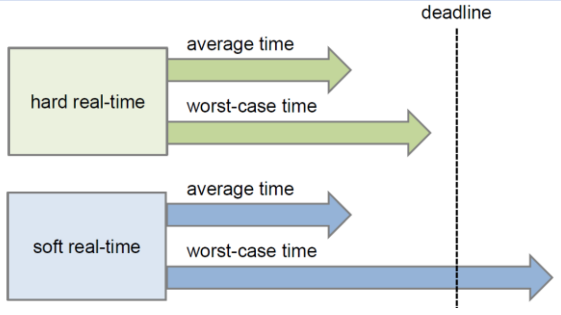


10.2.3. Context switch

Saves the context of the preempted thread



10.2.4. Hard vs. soft real-time



10.3. Thread Management

RUNNING Currently running thread Only one thread at a time
READY Ready to run
WAITING Waiting for an event to occur
INACTIVE Not created or terminated

Priority-based preemptive scheduler
At each *systick*, the active thread with the highest priority becomes the **RUNNING** thread provided it does not wait for any event.

10.3.1. CMSIS_RTOS RTX configuration

Without "Round-Robin Thread switching"

- A thread will only be pre-empted by a thread with a higher priority
- Among threads with equal priority: Control will only be passed to another thread if the thread in control yields or goes waiting

With "Round-Robin Thread switching"

- Threads with equal priority are pre-empted in round-robin scheme
- Round-robin Timeout [ticks] Specifies the number of ticks a thread is allowed to run before the round-robin preempts

10.4. Inter-thread Communication

10.4.1. Event types

Support communication between multiple threads and/or ISR

Signal is a flag that may be used to indicate specific conditions to a thread. Signals can be modified in an ISR or set from other threads.
Message is a 32-bit value that can be sent to a thread or an ISR. Messages are buffered in a queue. The message type and queue size is defined in a descriptor.
Mail 1) is a fixed-size memory block that can be sent to a thread or an ISR. Mails are buffered in a queue and memory allocation is provided. The mail type and queue size is defined in a descriptor.

10.4.2. Signal events

- Trigger execution between threads
- Functions to control or wait for signal flags

ISRs can call osSignalSet() but ISRs **cannot** call osSignalClear() or osSignalWait()

10.4.3. Message queue

- One thread sends data explicitly, while another thread receives it
- FIFO-like operation
- Functions to control, send, receive, or wait for messages

10.5. Resource sharing

Mutex - mutual exclusion

- Protects access to shared resources
  - Use resource only by one thread at a time
  - E.g communication channels, memory, files
- Mutex is passed between threads
  - Threads can acquire and release mutex

10.6. Generic Wait Functions

10.6.1. Add a time delay

`osStatus osDelay(uint32_t millisec);`

Thread goes into state **WAITING** and waits for a specified
time period in **millisec**.

10.6.2. Wait for unspecified events

`osEvent osWait(uint32_t millisec);`

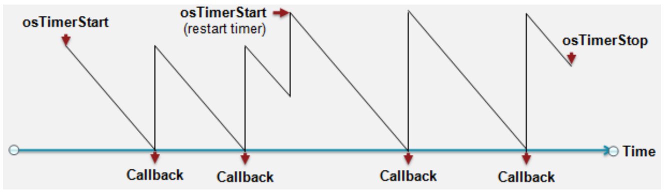
The **osWait** function puts a thread into the state **WAITING**
for at most the time period specified in **millisec** and waits for any of the following events:

- A signal sent to that thread explicitly
- A message from a message object that is registered to that thread

10.7. Timer Management

Create and control timer and timer callback functions

- Timer objects can trigger the execution of a function (not threads)
- When timer expires, a callback function is executed to run associated code



Combining threads and ISRs

- Applying priority levels

10.8. Thread Management Revisited

Prioritizing ISRs and threads

- Interrupt handlers (ISR) signal the threads when an interrupt occurs
- Processing is done in Threads not in ISRs
- Thread priority level defines which thread gets scheduled by the kernel

11. Structuring Embedded Software – Part 2

Possible approaches for embedded software

- Many real-life programs apply combinations of the presented approaches *image source: colourbox*
- Which approach is best suited depends on the application at hand
- No general rule that one approach is superior to the other

11.1. Bare metal

Running the embedded software directly on hardware as opposed to running it on top of an Operating System (OS)

11.1.1. Polling – Round Robin

After a certain period of time has passed, the thread gets switched to the next one

11.1.2. Fully interrupt driven

11.1.3. ISR flags

11.1.4. ISR pass data through queues

11.1.5. Co-operative Scheduler

11.2. Firmware bridge SPI slave and UART

Continuous streams with same nominal bitrate but subject to clock tolerances

- CPU calls conversion functions on each byte

Discuss potential firmware approaches

- Identify challenges
- Which missing information may influence your decision?

11.3. Shared Memory

Sharing objects may create consistency issues

- Working concurrently requires special measures

Recognizing shared objects
-> finding potential programming errors

Properties

- Asynchronous
  - No predictable time relationship among instruction sequences
  - E.g., main program and interrupt service routine
  - E.g., multi-threaded programs
- Access to shared memory requires coordination
  - Risk of data corruption ⍰ requires protection measures
- Challenges
  - Identification of shared memory ⍰ find unintentional, hidden cases
  - Minimize use of shared memory to necessary cases

When two or more asynchronous instruction sequences (threads) access the same data, that data is called *shared memory*.

11.3.1. Recognizing Shared Objects

- Important: **How** an object is used
- Scope or allocation method do not allow conclusions
- Three possible mechanisms for shared objects
  - Shared global data
  - Shared private data
  - Shared functions

Try to minimize shared data

- Reduces risk of data corruption issues
- Reduces complexity of the software

Apply careful design to protect unavoidable shared objects

Read-only data

- E.g., a table of constants
- If shared data is read but never written ⍰ no data corruption can occur
- However, as a program evolves the requirements might change
- Risk that a program modification changes the code to read-write
  - E.g., hard-coded data could be loaded from a file in a later version
  - Use a comment in your source code that the read-only table is shared

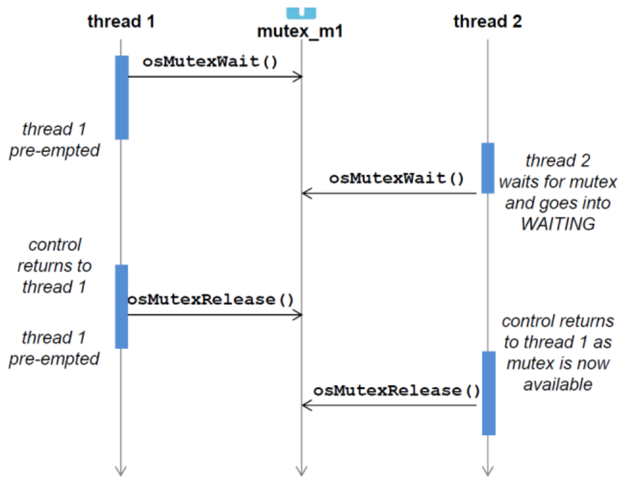
11.3.2. Data Consistency Issues

Call-backs don't make shared objects go away

- ISR does not even access the variable count
- But the callback may still have a data consistency problem

11.4. Mutex

Ensures that only one thread at a time can access a resource, like communication channels, memory or files. Threads can acquire and release a mutex. While one thread has acquired it, the others can't access until it is released.


thread 1
mutex_m1
thread 2
osMutexWait()
thread 1 pre-empted
osMutexWait()
thread 2 waits for mutex and goes into WAITING
control returns to thread 1
osMutexRelease()
thread 1 pre-empted
control returns to thread 1 as mutex is now available
osMutexRelease()