# General Informations

## Exam Admission (Attestation)

**Recap** 🡪 Summary of a topic from previous week, **Tips** 🡪 Tips and Tricks for the next students, **Functionality** 🡪 Defined Robot + Remote Task (Snake Game in FS2017)

## Lab Material

**tinyK20** 🡪 Programmer/Debug Probe, USB & SWD Cable, used to program robot and remote, debugging interface

**K22 Zumo Robot** 🡪 V1 (2014) and V2 (2016), **K22FX512 (ARM Cortex-M4F**), 120 MHz, 512Kbyte of FLASH, 64 Kbyte of RAM, USB, 2 LED’s, 1 Buzzer, Reset + user button, 1:75 DC Motors, **Optical (V1) or Magnetic (V2) Quadrature Encoder**, IR Line Sensor, Arduino Headers

**K20 Remote** 🡪 included tinyK20, **K20DX128** (**ARM Cortex-M4)**, 50 MHz, 128 Kbyte FLASH, 16 Kbyte RAM, nRF24L01+ 2.4 GHz Transceiver, Nokia 64x48 BW **LCD**, **7 Buttons** (Joystick Buttons (4way + center), 2 side buttons), 260 mA LiPo Battery,

ICharging: 195mA

# Build and Debug

## Eclipse Workspace

- Where Ecplise stores the ‘meta data’ (folder .metadata) 🡪 1. **‘global’ options** across projects, 2. **NEVER move/share meta data**

- Wsp (Workspace) could contain project folders, but it’s recommended to keep wsp and projects separate because it’s easier for the VCS (Version Control System) **Example: wsp** (C:\user\wsp\_kds) and **projects** (C:\user\projects)

|  |  |
| --- | --- |
| Workflow **make**  🡪 .mk file 🡪 Beinhaltet Regeln  für Abhängigkeiten zwischen den Files | Parallel Build Reducing Compile Time in Eclipse with Parallel Build. This is especially useful for host machines having multiple cores or CPU: such as each CPU then could do a compilation and balance the build load across all available CPUs to cut the build time. |
| Sharing Debug Configuration Normally the debug configurations are not stored in the project settings. If I zip that project or share it with a VCS, then the debug configurations are not shared.  **To share debug configurations** with my project, I need to enable ‘Shared file’ in the configuration: that way the configuration gets stored in a .launch file inside the project. |
| PE Code Generation In the processor expert project options you can say “don’t’ generate code before build automatically” to speed up PE projects |
| Debug without Build Uncheck the Option “Build (if required) before launching” to debug without build. This speeding up the debug/launch. |

# Version Control System (VCS)

## A good VCS

**Backup and Restore** (wiederherstellen), **Synchronization, Short- and Long-term undo, Track changes and ownership** (wer hat was wieso geändert), **Sandbox** (You can make temporary changes in an isolated area, test and work out the kinks before “checking in” your changes), **Branching and Merging** (A larger sandbox. You can **branch** (verzweigen) a copy of your code from the main trunk into a separate area and modify it in isolation (tracking changes separately). Later, you can **merge** (vereinigen, zusammenführen) your work back into the common area (main trunk).)

## Typical VCS

- Server(s) with data base(s)

- Clients connect to server

-locally or remote

- Single Server or distributed

|  |  |
| --- | --- |
| **‘Optimistic’ approach (e.g. GitHub)** | **‘Pessimistic’ approach** |
| Assumes rarely conflicts  Different clients can work on the same file concurrently | avoid conflicts  Just one client can work on a file |
| **Centralized** | **Distributed** |
| typically single repository server, data is just on server, **local copy of current snapshot on client,** commit/compare when connected, Example: CVS, **SVN** (SubVersion) | Repository server (can be multiple), Data is on server and on client, **Local copy contains full repository history**, commit/compare even when not connected, Commits to local repo 🡪 then sync with server, Example: **GIT** |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| What to share and what not to share with VCS **Share:** 🡪 everything needed to build the project  - **.project** and **.cproject** (project files and build options  - source files (**\*.c,\*.h, etc.)** and source file folders  - **Project\_Settings** folder and files (linker files, startup files)  - **\*.launch** (contains launch/debugger settings)  - **ProcessorExpert.pe** (contains component settings) | | | | **Not share:** 🡪 generated or derived (abgeleitete) resources  - generated documentation/log files  - Build output files **\*.o** (named as the build target)  - PE 🡪 **Documentation and Generated\_Code**  - **.ProcessorExpert.g\_c** and **.ProcessorExpert.g\_x** (contain information about the generated files) | | | |
| GitAbout Git - By Linus Torvalds  - **Index:** collection of added and file changes  - Staged and unstaged files can co-exist    (commit and ‘check in’ is the same) | | | Typical Git Workflow | | | | Other Actions - **Discard/Revert**  🡪undo a local change  - **Delete**  🡪 deletes a file from the index/disk  **- Tag**  🡪 Mark files with a label, e.g. to check out all files with the same label |
| PE Important VCS Note - Agree on group change in advance! (component change)  - User A: Commit/Push  - User B: Closes project, remove \*.pe file, then pull file  - Otherwise merge |
| Ignoring files - .gitignore File ignores files and folders, wich shouldn’t be shared  - **Recommendation:** one .gitignore File per project  - **NOTE:** path in .gitignore File is relative to ignore file location | | | .gitignore Format - #: starts comment  - line item: ignore file/folder/pattern | | | |
| Systems & RealtimeTransforming Systems - Data processing quality  - Troughput (Durchsatz)  - Optimized system load  - Optimized Memory Usage | Reactive Systems - External events are driving system  - guaranteed response time  - Control loop,  - Realtime | | | | | Realtime - System Interaction with the environment  - System has to deal with the time constraints (Grenzen) of the real world (real time)  Not 🡪 as fast as possible  Instead 🡪 at the right time  **Realtime System Requirements:**  Correctness and External time conditions compliance (Zustimmung, Einhaltung)  **Examples:** Train system schedule computation (Abfahrtsplan),Railroad switch | |
| Interactive Systems - short response time  - High system load  - Human-Machine Interaction (HMI) | | | | |
| Realtime for Computer Systems A computer is classified as Realtime if it can react on external events in the real world:  - With the correct result  - At the correct time  - Independent of current system load  - In a deterministic (foreseeable) way  **Claims:** Timeliness and Concurrency | | Timeliness (Rechtzeitigkeit) For all processing stages: **Input 🡪 Process 🡪 Output**  Categories: **absolute and relative** | | | Concurrency (Nebenläufigkeit) - Real world is concurrent 🡪 Problem: Computers are sequential | | |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Reaction Time Realtime Systems require a defined absolute or relative reaction time  - **Interactive Systems** 🡪 seconds  - **Transforming and Reactive Systems** 🡪 milli- or microseconds  **System load defined with**  - Number of concurrent events/tasks  - Interval of events  - Reaction time for events  - Processing time for events | | Hard Realtime - Incorrect if correct Result does not meet time conditions 🡪 outside the blue marked area, the data is useless (unbrauchbar) 🡪 Error! | | | | | | | | Soft Realtime Degradation (Verschlechterung), if correct result does not meet the time conditions 🡪 outside the blue area still ok, but nut very good | |
| Processor Expert - Problem: No time to deal with the very low level Embedded Components - Implemented in a C like scripting language  - Functionality separated into small objects  - Components have interface  **o Methods** 🡪 Procedures that can be executed  🡪 Function Calls  **o Events** 🡪 Indication of State Changing  🡪 usually implementation of ISRs  **o Properties** 🡪 Modify/Customize object behavior  🡪 Set during design-time  Importing Components as \*.PEupd files | | | | | Component Model Development Flow | | | | | | |
| Project StructureSystem Blocks: many in common use linked Folder in Eclipse projects (like Team\_Common) where the common source files (drivers) are stored | Common Library Structure (example) - Common drivers in Team\_Common  - Drivers guarded by PL\_CONFIG macro #if PL\_CONFIG\_HAS\_BUZZER in source file  - **Platform.h** maps dependencies  #define PL\_CONFIG\_HAS\_BUZZER (1 && !defined(PL\_LOCAL\_CONFIG\_HAS\_BUZZER\_DISABLED) && PL\_CONFIG\_BOARD\_IS\_ROBO  - **Platform\_Local.h** can turn turn off/on functionality and defines board  #define PL\_LOCAL\_CONFIG\_HAS\_BUZZER\_DISABLED  #define PL\_LOCAL\_CONFIG\_BOARD\_IS\_ROBO(1) | | | | | | | | | | Smart Way to #define - in Platform.h (common file)    - To define Number of LEDs dependent on the used board |
| PreprocessorMacros/#define - Definition of a Macro with #define BLUE 0  - Compiler is replacing Macros textually  **Why Macros?**  - Names instead of ‚magic‘ numbers  #define DELAY\_TIME\_MS 10  - Configuration  #define DEBUG\_ME 1  - Portability (Übertragbarkeit)  #define ENABLE\_INTERRUPTS \_\_asm(“CPSIE”)  - Optimization | | | Inlining with Macros **Pros:** Faster and Smaller Code  **Cons:** Interface, Encapsulation, Debugging Tip using Macros Use as much parenthesis (Klammern) as possible:  #define DELAY ((PRE\_DELAY) + (POST\_DELAY)) | | | | | | | | |
| #include (Preprocessor Macro) **Header/Source, what is where?**  - Use header files for declarations which shall be known to other implementation files.  - Do not place definitions (memory allocation) in header files.  - Use extern in header files for variable declarations. (normalerweise zu vermeiden)  - Do not use extern for definitions in the implementation file. | | | | | | | #include Directive -**Textual inclusion of files** (with #include)  - Result is **‘compilation unit’** | | | | |
| #ifndef - #define - #endif (Protection against multiple declarations/definitions and recursive includes) | | | | | | | | | | | |
|  | | | | | -‘Protection’ symbol | | | | 🡪 Avoid name conflicts | | |
| 🡪 Convention: \_\_<FileName>\_H\_ | | |
| 🡪 Double Underscore: ‘reserved names’ | | |
| What and Where? Self-Containment Header file should be **‘self contained’** **1.** Users of the interface should only need to include that Interface **2.** Using an interface/header file shall not depend on include order What not to do… 'Reducing' includes in the wrong place is a bad thing 🡪 better protect | | | | | | |
| LED **Goal:** ‘same’ driver for multiple platforms Implementation #1 **Goal:** one Function (LED\_On) for Multiple LED’s  **Problem:** Multiple LED’s as parameters | | | | | | Implementation #2 - one argument  - LEDs are encoded 🡪 LED0: 0x01, LED1: 0x02, LED2: 0x04, LED3: 0x08 | | | | Implementation #3 - one argument  - LED’s masks defined as symbols (#define LED\_LED0 0x01, etc.) | |
| Implementation #4 - One argument  - LED’s used in a symbolic way  - LED’s as type | | | | | | Implementation #5(how PE Component LED works) - **Interface for each LED**  🡪 Flexible Anode/Cathode  🡪for few LED’s | | | | | |
| Microcontroller Pin on Anode or Cathode?   or smarter implementation | | | | Device Driver Flow - **Init()** 🡪 Initialization/allocation of memory, data structure, …  - **Open()** 🡪 Lock device, get device handle, …  - **Close()** 🡪 Free device, return device handle, …  - **Deinit()** 🡪 Free memory, reset to default, … Device Handle - One interface for all devices  - Need to pass device information (handle) 🡪 flexible, overhead | | | | | | | |
| Synchronization Real World 🡪 concurrent and continuous  Computer World 🡪 sequential and discrete  🡪 Real Time Systems: Need for Synch  Computer operates in different time scale  **if slower than reality** 🡪 Result too late: incorrect 🡪 **Problem!**  **if faster than reality** 🡪 Result too early: incorrect 🡪 **Synchronization**  Computer has to synchronize with (real world time- ) process  🡪 Examples: A/D converter, keyboard | | | | | | | | A) Realtime Synchronization The synchronization method is named Realtime Synchronization because it is using real time to wait  - What is the needed waiting time?  - Inefficient  - Problems different compiler or clock rate (using for loop to wait some time can be more or less fast)  - **Example A/D – Converter** (wait for Tconv) | | | |
| B) Gadfly Synchronization The principle of Gadfly Synchronization is to check a flag (usually a hardware flag) which indicates if the device is in DONE state. This **'polling'** is usually implemented in a loop where the program checks the status of this flag until it detects DONE or an error condition.  - Blocks further execution  - Processing power needed to checking the flag (because the flag will be checked “everytime” not just e.g. every 10ms) | | | | | | | | | | | |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| C) Interrupt Synchronization **Realtime and Gadfly synchronizations have one big disadvantage:** they require cycles on the processor while they are waiting. If the waiting time is only very short (means: only a few processor cycles), then this might be acceptable. However, if the waiting time is rather random or might be long, it would be a waste of CPU or Microcontroller Unit (MCU) performance. As soon as you have multiple things to do, you want to use the processor  cycles for something else while you have to wait for a device. **🡪 do something else until**  **you get a notification (e.g. from an interrupt)** | | | | | Interrupt Execution and Execution Speed - ISR as efficient and straight forward as possible  **Possible Approach:**  -ISR just add events in Event Array/Queue  - Event Handler in the e.g. main loop does the heavy workload ( does something dependent on event) | | | | | |
| Summary: Synchronization Methods **A) Realtime Synchronization**  **B) Gadfly Synchronization**  **C) Interrupt Synchronization** | | | Interrupts 🡪 important notes **The entry and exit sequence of the interrupt needs to be atomic**: this means that the save and restore operations shall not be interrupted itself.  An important aspect for Realtime applications is the **interrupt latency time**: this time ideally should be as small as possible, and is defined as the time between the cause of an interrupt cause (e.g. pressing a key) and the time when the ISR starts executing.  **Using Volatile for ISR flags** 🡪 extern volatile int ISR\_Flag;  The value of volatile variables may change from outside the program (e.g. from an ISR). For example, you may wish to read an A/D converter or a port whose value is changing. Often your compiler may eliminate code to read the port as part of the compiler's code optimization process if it does not realize some outside process is changing the port's value. You can avoid this by declaring the variable volatile. | | | | | | | |
| Interrupts and ReentrancyCommon Subroutines Using subroutines or functions which can be used from multiple places is good programming style as it avoids code duplication. But as **an interrupt might happen anytime**, it might be that the execution of the common subroutine executed by the main program is interrupted too.  As in figure both the main program and the ISR can call the same function, the **shared subroutine needs to be reentrant**. Reentrant means that the code can "re-enter" a function without interfering another portion of the application which is already executing that code. | | | | | | Shared data Interrupt programs need special care if both the ISR and the main program are sharing (global) data as in figure. **As long as both are just reading the data, that works out fine.** But as soon as one is reading the data while another is writing the data, there might be the problem of inconsistent data.    **Need to protect access to data** | | | | |
| Interrupt Priorities - The **main program (Base Priority**) can be interrupted any time  - The **interrupt routines** also can be interrupted 🡪 **need Rules!**  - Main- and Sub-Priorities are possible | | | | | Interruption Rules **-fn:** currently executed program **-in:** interrupted program  **-MP:** Main Priority **-SP:** Sub Priority **-S:** Signal **-WS:** Waiting Signal  🡪 Signal ist in diesem Kontext ein Interrupt der aufgetreten ist  **Rules:**  **1.**MP(S) > MP(fn): interrupt **2.**MP(S) <= MP(fn)  S -> WS  **3.**SP(S) > SP(fn)  S -> WS **4.**MAX(SP(WS))  WS -> fn  **5.**MP(WS) > MP(in)  fn->(in)->WS -> fn | | | | | |
| Example System Startup **Reset 🡪**  **Interrupts disabled** Interrupt Vector Table In an vector based system as in the figure there is a table of function pointers (vector table) mapped on a memory address. | | | | |
| Critical Sections (code and also some pseudo Code) | | | | **🡪 Need for Mutual Exclusion** | | | | | | |
| **PRIMASK I Bit Setting**  //set interrupt disable flag  \_\_asm volatile("cpsid i");  {Critical Section}  //set interrupt enable flag  \_\_asm volatile("cpsie i"); | **EnterCritical/ExitCritical**  //DisableInterruptsAndStoreCurrentInterruptStatus  EnterCritical();  {Critical Section}  //ReEnableInterruptsWithPreviousStatus  ExitCritical(); | | | | | | | **Task CS**  TaskEnterCritical();  {Critical Section}  TaskExitCritical(); | | **Semaphore CS**  xSemaphoreTake();  {Critical Section}  xSemaphoreGive(); |
| CriticalSection Component und deren Anwendung **- Kein Semikolon nach CritcalVariable(),** da schon in #define gemacht  The component source implements three macros:  1. CriticalVariable() is a macro who declares and defines a local variable to store the current interrupt state.  2. EnterCritical() is a macro to create a critical section.  3. ExitCritical() leaves the critical section. | | | | | | | | | | |
|
| Reentrancy: Race Condition  🡪 Output is dependent on the sequence/timing  🡪 **reg** wird in diesem Fall zuerst von Thread A und danach von Thread B incrementiert und ist dann bei der Abfrage (if reg == 1) nich eins sondern zwei und somit werden die Interrupts nicht disabled und dies führt zu Problemen, da anschliessend eine Critical Section startet | | | | | | | | | | |
| ARM Cortex-M0+/M4(F) - ARM Inc. (Cambrige UK)  - **32bit RISC** (Reduced Instruction Set Computer  - ARM Cortex-**A** family 🡪 Applications processors  - ARM Cortex-**R** family 🡪 Embedded processors for real-time signal processing, control applications  - ARM Cortex-**M** family 🡪 Microcontroller-oriented processors for MCU and SoC applications | |  | | | | | | | Instruction Set - Upward compatibility  - THUMB(2) 🡪 Code Density, 16bit Instructions (subset of ARM instructions), **Odd** Address Bit | |
| CPU Registers (eine Auswahl) - RO-R12 GPR (32bit)  - R13 🡪 Main Stack Pointer (MSP) (default), Process Stack Pointer (PSP)  - R14 Link Register (LR)  - R15 Program Counter  - CPSR (Current Program Status Register) | |
| Masking Interrupts 🡪 **PRIMASK and BASEPRI Register 🡪 BASPRI not for the M0/M0+!**  **PRIMASK:**  - 1 Bit Register  - Masks (disable) interrupts  - NMI and HardFault are not maskable (negative prio!)  **BASEPRI:**  - masks interrupts with priority ≥ BASEPRI value 🡪 masks interrupt with lower or equal urgency | | | | | | | **Example BASEPRI** 🡪 hypothetically 8 Priority Bits  - BASEPRI set to a value of 3 🡪 **disables** interrupts with values (priorities) 3,4,5,6,….,255 🡪 **allows** any interrupts with value (priority) of 0, 1 and 2  **Note:**  Because BASEPRI is a mask register: setting it to zero means interrupts are not masked and therefore enabled. It means that **BASEPRI cannot mask/disable interrupts with priority zero! 🡪 use PRIMASK instead** | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| Interrupt Vectors Table The ARM Cortex-M is using a **NVIC** (Nested Vectored Interrupt Controller) and it means that it uses a **vector table**  - The Table is vectored, because the 32 bit entries in it (e.g. Hard Fault vector at address 0x0C) point to the corresponding interrupt service routine  - The exception numbers 1 to 15 are defined by ARM (they are part of the core) and the exception above 15 are vendor specific (herstellerspezifisch z.B. UART/I2C/USB/etc.). In other words the negative IRQ numbers (from -1 (SysTick) to -14 (NMI) plus reset) are defined by ARM. | Interrupt Priorities 🡪 **the lower the number, the higher the urgency** - I can assign a priority of each Exception/Interrupt. Except that **Reset, NMI** (Non Maskable Interrupt) **and HardFault** **have a fixed (negative) priority and cannot be disabled**. The interrupt with a priority of 0 is the most urgent one.  Note: The SysTick is optional for the ARM-Cortex-M0 but most vedors have it implemented  The number of vendor implemented exceptions (IRQ1, IRQ2, …) depend on the implementation and the core  - **ARM Cortex-M0** can have up to 48 exceptions (16 core specific and 32 vendor specific)  - **ARM Cortex-M3/4/7** can have up to 256 exceptions (16 core specific and 240 vendor specific) | | |
| Priority Bits 🡪 8 bit Priority Register - The number of bits implemented is vendor specific  **- min 2 bits for the M0/M0+**  **- min 3 bits for the M3/4/7** | | Sub-Priorities 🡪 **just on M3/M4/M7, not M0+** The number of subpriority bits is configured by the **PRIGROUP** register. The PRIGROUP can be changed at run-time.  For example if PRIGROUP has value of 5 and number of priority bits are 3, then there are **2 main/pre-emption priorities and one sub priority**  The Preempt Prio defines if an interrupt can nest/interrupt an already running interrupt. The **Sub-Priority is used** when multiple interrupts with the same Preempt Prio are pending, then the one with the lower sub-priority (higher urgency) will be executed first. |
| Shifted Priority Bits The implemented priority bits are **left-aligned** 🡪 keeps priority values compatible between different implementations  **for three implemented bits**, it means we can have 2^3 (8) priority levels, with **the following shifted values**:  Hex: 0x00, 0x20, 0x40, 0x60, 0x80, 0xA0, 0xC0, 0xE0 | |
| NVIC Interrupt Configuration 🡪 The NVIC offers several registers to configure the interrupts  On the **Cortex M0/M0+** there are the following (Cortex- M3/4/7 hat noch ein zusätzliches)  - **NVIC ICER** (Interrupt Clear Enable Register): disable interrupt bit, one bit for each interrupt  - **NVIC ISPR** (Interrupt Set Pending Register): mark interrupt as pending bit, one bit for each interrupt  - **NVIC ICPR** (Interrupt Clear Pending Register): clear pending flag bit, one bit for each interrupt  - **NVIC IPRx** (Interrupt Priority Register): interrupt priority (8bit for each interrupt, 4 interrupts in a 32bit register)  The **Cortex M3/4/7** has one register more in addition  **- NVIC IABR** (Interrupt Active Bit Register): set if an interrupt is running, one bit for each interrupt | | PEx: ARM Core Interrupt Settings - Settings in CPU component properties / - Vector table in vectors.c (Generated\_Code)  - 0x00: initial SP (Stackpointer) / - 0x04: initial PC (Program Counter) | |
| Fault Exception - **Bus Fault** 🡪 Instruction or data error, stacking error  - **Memory Fault** 🡪 Write to Read-Only, outside of memory map  - **Usage Fault** 🡪 Illegal instruction, invalid ISR return, unaligned memory access  - **Hard Faults** 🡪 can be called by above fault conditions  🡪 **Hard Fault Handler** (PE Component) 🡪 Interrupt handler to find out location of problem | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Events - **Synchronous** (Timer Interrupt, Periodic Task output) and **Asynchronous** (Button pressed, Transceiver packet received, Beep after button press) **Events**  - need infrastructure 🡪 **Set/Clear/check** if event happened  - Possible Implementation with an **Event Array**/-Queue/- List | | Decoupling Event and Processing - ISR or Polling Loop just Set Event (flags) 🡪 ‘fast’  - Main loop (Event handler) does the heavy workload 🡪 ‘slow’ | | EVNT Array - Array of Bytes  - Set event  - Considerations 🡪 Bit Order (Little or Big Endian)  🡪 Size of base memory unit: uint8\_t, uint16\_t, … |
| Storing Events - Using as few memory as possible  - using event ‘flags’  - 🡪 mapping from ‘numbers’ to bits/flags  - **numbering can be priority**  **-** with#define or typedef enum (symbolic names instead of #define) | Handling Events from Main loop **1. Extract Event** (e.g. Loop)  - See if there is an event  - Event ‘number’ or bit position could be used as priority  - Extract bit/event  **2. Handle Event** (e.g. Switch) 🡪 Act according to event  **Advantage**: simple, **Disadvantage**: Long if/else/switch, Order of event handling needs to be defined, **need to protect against concurrent access** (weil z.B. eine ISR den Event im Array setzt und der Main loop das Event Array abfragt) | | EVNT Interface | |
|  | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| State Machines Design pattern 🡪 states and Transitions between states | 1. Implementation: **Functions** - Each function implements state  - Transition between states by function calls  - **No global state variable needed**  - **Disadvantage** 🡪 possible Recursion and Stack overflow (Wenn sich die States immer wieder gegenseitig aufrufen) | | | | 2. Implementation: **switch or if-else** - Periodic Process function call (e.g. from main loop or from a task)  - **global state variable needed 🡪 Problem**  - Changing state during processing or from outside 🡪 **reentrancy!** (während die state machine läuft oder von ausserhalb kann der state auch geändert werden)  - better structure with switch instead of if-else | |
| 3. Implementation: **Table Implementation** (Mealy Table) - condensed (verdichtet) way to implement a state machine from a given mealy table | | | | Hierarchical state machine - combination of multiple FSM |
| Mealy Sequential Machine - Table driven sequential state machine  - Input 🡪 Transition with output (ein definierter Input kann zu einem Zustandswechsel führen und während dem Wechsel ist ein gewisser Output vorhanden) | |
| Clock and TimerRealtime Systems: Timeliness - Categories 🡪 **Absolute or Relative**  - Need 🡪 Time base, Clock, Interrupt Synchonization  - Derived (abgeleitet) 🡪 Timer, Time (all the clocks of the system are derived with dividers and multipliers from system clock (either internal or external reference clock) | | What do we need? - Linkage to the real time 🡪 ns, us, ms, s, h  - **Periodic Ticks 🡪** 1. Known real time tick period 2. External or internal source  3. System/CPU/Bus clock  - **Synchonization with counter(s)** 🡪 Events, Flags | | | Timer Component and Interface - **Not use LPTMR (low power timer) as periodic interrupt source!** 🡪 in the Timer PE component  **Note:** Some timer on the ARM run only in low power (LP) mode. As we are not running low power modes 🡪 **use a non-low power timer as the FTM** (Flex Timer Module) | |
| Summary - Different clock sources 🡪 External Crystal/Oscillator (8MHz Crystal on Remote)  - **CPU Clock vs Bus Clock** (**CPU Clock** 🡪 rate of doing operations like adding two numbers or move value between CPU registers,  **Bus Clock** 🡪 refers to how much data can move across the bus simultaneously | | |
| KeysKey Scanning - Key press detection with Polling or Interrupts  - Debounce 🡪 long and short key detection | | | Buttons (Pull-Up, Pull-Down) - **Need defined logical level** 🡪 Pull-Up or –Down  - No hardware pull-up resistor for USR\_BTN in Robot V2 and for different buttons in Remote 🡪 **use microcontroller internal pull up** | | | Port Interrupt Sharing - Cortex M0+ 🡪 only 32 interrupt sources  - One Interrupt for all port pins (e.g. one interrupt for PORTA)  - **Need to check in ISR which pin triggered the ISR** |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Keyboard Driver Solutions simple **Realtime-/Gadfly-Synchronization** and Polling the Key    - **Interrupt Synchronization** 🡪 Requires that pin is able to generate interrupt | **Control and Data Flow with Polling and Interrupts** | | | | |  |
| Console - Console (Terminal) connection from Device to Host  - Using SCI (Serial Communication Interface)  - Robot V2 and Remote 🡪 RX/TX on SWD  - **Robot V1 🡪 Segger RTT 🡪 ´virtual´ communication through debug interface (SWD)** | | Shell PE Component Console Shell 🡪 **- Serial (SCI/RS232), - RTT, -(USB)**  - Uses 🡪 Wait, Utility, CriticalSection  **Blocking Send:** 🡪 if the other side not respond or not receive  **timeout:** maximum blocking time for a character  **wait time:** waiting time before resend if sending was blocked (output buffer full) | | | AsynchroSerial UART Interface - Define UART Interface in Shell Component as Default Serial  **Define in PE Component:**  Blocking send or not, Channel (e.g. UART0), ISR with ring buffer (input/ output buffer size), Port for Rx and Tx, Baudrate | |
| Shell Standard I/O - I/O structure with callbacks (function pointers)  🡪 **Stdin** (read char), **Stdout** (write char), **Stderr** (write char), **KeyPressed** (char in stdin?) | | | Writing Strings/Numbers - **CLS1\_SendStr without Blocking Send!** | | | |
| Utility: Safe String Routines - Buffer size as parameter  - **Unlike normal strcpy(), does \*not\* cause buffer overflow**  - Buffers always zero byte terminated | | | | Problem Windows USB CDC -Standard Windows CDC Driver Problem  -Problem if USB CDC COM Port open  -**Device stops communicating**  -Cable gets unplugged  -Otherwise: COM port is blocked  -**Solutions 🡪** Proprietary Serial driver (mbed.org, N/A) or:  1. Have COM port closed (in Terminal Program)  2. Unplug cable, 3. Plug cable in again, 4. Open COM Port  -**Windows 10: much better** | | |

|  |  |  |
| --- | --- | --- |
| Triggers - We want the application to trigger at a given time in the future  - We want e.g. to flash an LED every 500ms, turn on LED for 200ms or start sounder for 500ms when button pressed  - **Need for a common infrastructure** 🡪 minimal memory usage, one time (reuse), universal infrastructure  Design Idea  🡪 based on a periodic interrupt (e.g. every 10ms) 🡪 gives the time base : **the tick counter**  The idea is that instead implementing things directly in the ISR, we implement a ‘trigger’ module which maintains the jobs. **The application (e.g. main loop) could set Triggers and the periodic timer interrupt routine simply add Ticks and check if there is any pending ‘trigger’ to execute and then call the callback (function pointer).**  - the trigger will be cleared before the callback is called 🡪 clearing approach 🡪 the trigger needs to be re-installed | Trigger Descriptor (data structure for a single Trigger) **triggerTick** 🡪 the time when the trigger shall be executed  **callback** 🡪 the information what to execute -> implemented with a function pointer  **\*data** 🡪 a void data pointer to pass any kind of parameters to the callback | |
| Data Allocation 🡪 store the Triggers in an Array **Static way** 🡪 using a fixed array of descriptors (Triggers) 🡪 number of triggers need to be known in advance to define the array | |
| Trigger Interface - **TRG\_SetTrigger** 🡪 setting up a specific Trigger from e.g. the main loop  - **TRG\_AddTick** 🡪 will be called from an ISR (e.g. periodic timer interrupt)  - **both Functions need to be reentrant** because they both using the Array with the Triggers inside | |
| Trigger Examples **Blinking 2 different LED’s**  🡪 using Parameter ledP  passing the data pointer to our Trigger 🡪 **use static variable for data (somit ist die Variable nicht auf dem Stack sondern im RAM)** 🡪 the data is valid all the time 🡪 **if we don’t use static** 🡪 the variable is only available as long as we stay in function foo() | | **Beep:**  In the example blinking LED’s **we used a static local variable** to ensure that the memory address we pass to the trigger is always valid. **This increases the amount of RAM** used. If we just want to pass a value as parameter, it would be possible to pass that value directly, instead of using the address of it |
| Relative Time Triggers 🡪 Accuracy (Genauigkeit) depends on Timer Resolution (e.g. 10ms) |
| Bit Banged PWM Buzzer with Trigger 🡪 Software PWM | |
| **Realtime Aspects** 🡪 The ISR of the tick timer is calling our Trigger Module (TRG\_AddTick()), which then will execute any pending Trigger 🡪 Keep the Number of Triggers as small as possible, Do something small (e.g. Toggling a Pin) in a trigger you would otherwise do in a ISR, every shared code between ISR and main application needs to be reentrant. |

|  |  |  |  |
| --- | --- | --- | --- |
| Debouncing **Idea:** Filter 🡪 Hard- or Software  Filter duration 🡪 Empirical or Measure Bouncing - Keys on platform might bounce  - Mechanical problem  - Possibility of raising Interrupt  🡪 Need a filter over time  🡪 Need relative time base or e.g. “do something in 500ms”  - **Solution** 🡪 State Machine & Trigger | Debouncing State Machine - Debouncing key presses  - Measure duration of key press ( long or short press)  - define state kinds with typedef enum | | Keyboard Program Flow **(Interrupts)** |
| Keyboard Program Flow **(Polling)** |
| State Machine Details - State machine entered through e.g. button pressed interrupt 🡪 KEYDBNC\_Process();  - **Cannot Stay in state machine! 🡪uses Trigger to re-enter**  FSM with Trigger Transistions: | | Reentrancy 🡪 ein Problem?   Nachdem ein Interrupt ausgelöst wurde von einem Key (Button pressed), werden die Key Interrupts alle ausgeschaltet und der Debounce Process wird gestartet im Idle State. Somit kann, solange das Debouncing geht, kein weiterer Key den Process starten. Wenn das Debouncing abgeschlossen ist werden die Key Interrupts wieder eingeschaltet 🡪 so funktioniert die Reentrancy.  Im Debounce Process wird dann ein Trigger verwendet, welcher nach einer gewissen Zeit wieder in den Process einsteigt (callback) und die Parameter des callbacks sind die Daten der FSM (**DBNC\_FSM Data**). | | |

|  |  |  |
| --- | --- | --- |
| Webinar 1 RTOS Getting StartedBarmetal Scheduling (without RTOS) 🡪 non pre-emptive **Round Robin Scheduling (Superloop) with Interrupts:**  - every time changing the code (e.g. add more tasks) the real time behaviour will be changed  - doing soft real time things in main loop and hard real time things (things which shall be did at the correct time 🡪 high priority things) in the ISR  - because interrupts can occur 🡪 we don’t know exactly when a task will be executed  - round robin scheduling don’t allow pre-emption 🡪 e.g. Task 2 cannot pre-empt (unterbrechen) Task 1  **Cooperative Scheduling:**  - This scheduling technique checks e.g. every 1 ms if a task shall run and then execute the task  - interrupts with hard real time (high priority events) can also occur  - cooperative scheduling don’t allow pre-emption 🡪 Task 2 cannot pre-empt (unterbrechen) Task 1  - it is important that all Tasks are done within the system tick time (e.g. 1 ms) | RTOS Characteristics A RTOS can make Tasks appear (erscheinen) to be concurrent 🡪 to make them look like they are all running together in parallel  **six important characteristics of an RTOS:**  - Reliability (Zuverlässigkeit), Predictability (Vorhersehbarkeit), Performance, Compactness, Scalability (Skalierbarkeit 🡪 usable in many different types of devices), Multi-Tasking (pre-empt Tasks) | |
| FreeRTOS Task Fundamentals **Task Control Block (TCB):**  - every single task is like its own application 🡪 has its own stack and also an infinite loop (for(;;)) to keep the task running  **Creating a Task in FreeRTOS:**  - We used NULL instead of 0 in our INTRO-programs  - if **stack depth** is e.g. 200 (Minimal Stack Size) 🡪 that means on a 32bit microprocessor 200\*32bit= 200\*4Byte= 800Byte  **Context Switching:** (Task 1 (lowest prio), Task 4 (highest prio))  - Context Switching means change between tasks depend on their priority  - be aware of average switch time (Durchschnittszeit zum Wechseln zwischen Tasks), because if there are many context switches over the time 🡪 CPU time gets wasted | |
| Baremetal or RTOS? 🡪 7 reasons to choose an RTOS - Concurrency, Pre-emption, Available RAM, Available flash, Synchronization tools, 3rd party software (), Ease of use |
| Webinar 2 RTOS Fundamentals using FreeRTOSSynchronization Fundamentals **Semaphore (Sync and Notify):**  - using semaphores in order to synchronize tasks or to notify that events have occurred  - Typically we have a flag or token that we are giving and taking from a task to the next. So may have a task which gives the token first then another task can take the token.  **Mutex (Mutual Exclusion):**  - special type of a semaphore  - trying to protect a resource 🡪 is going to give (unlock) or to take (lock) a resource 🡪 resource could be e.g. memory or a peripherial (e.g. LCD-Display)  - only the task which has the key can use the resource 🡪 **ownership** | | **Message Queues (Communication):**  - not much using for synchronization tasks 🡪 more likely used to transmit messages from one task to another  - typically the type of information that we will sending is perhaps pointers to data structures  **Event Flags (Synchronization):**  Event bits are used to indicate if an event has occurred or not. Event bits are often referred to as event flags. For example, an application may:  🡪 Define a bit (or flag) that means "A message has been received and is ready for processing" when it is set to 1, and "there are no messages waiting to be processed" when it is set to 0. |

|  |  |  |
| --- | --- | --- |
| Semaphore Fundamentals counting semaphore (e.g. 0 to 255) or binary semaphore (0 or 1)  **3 different possible ways to use a semaphore:**  **1.** a task takes a particular semaphore 🡪 no one else can take that semaphore 🡪 and then does what it need to and gives the semaphore back  **2.** two different tasks 🡪 one task gives the semaphore and when the semaphore appears 🡪 a second task takes it  **Example:** data coming in and task 1 is receiving that data and if the data is ready gives a semaphore 🡪 task 2 sees there is a semaphore available and takes it to process the data. Task 2 don’t have to give the semaphore back at the end  **3.** an ISR and a Task 🡪 e.g. interrupt fires if data is available on the UART bus 🡪 the the ISR stores the data in a buffer and gives a particular semaphore, because the data is ready to process 🡪 a task take that semaphore an process the data | | Mutex Fundamentals - a task is taking **ownership** of a particular resource 🡪 no other task has acces to this resource until we are done with the resource  **Example:** Mutex (key) gives access to an UART bus  **1. Task 1 Take (Lock) 2. No one else can take**  **3. Task 1 Give (Unlock**) **4. E.g. Task 2 can Take (Lock)** |
| Message Queue Fundamentals **2 different possible ways to use a message queue:**  **1. primary purpose 🡪** two different task which need to communicate to each other 🡪 queues are unidirectional 🡪 so we need in this example two different message queues  **Recommendation** 🡪 if sending more than 16 bytes worth of information across a message queue 🡪 better sending a pointer to that data instead  **2.** an ISR and a task 🡪 e.g. interrupt fires if data is available on the UART bus and then transmit the data to a task over the message queue 🡪 the task then process the data | RTOS Dangers 🡪 Priority Inversion - **Priority Inversion** occurs when a high priority task is blocked by a lower priority task 🡪 e.g. a low priority task actually has a shared resource and then the high priority task also wants this resource but don’t get the resource **Solutions:** - use mutex with priority inheritance 🡪 Lower priority task gets temporarily the priority of the blocked task, - Properly select task priorities | |
| RTOS Dangers 🡪 Deadlock and Thread Starvation - **Deadlock** occurs when two tasks need access to two or more resources to proceed but each task has only one of the resources **Solutions:** - Tasks that require two or more resources should acquire them in the same order, - use timeouts to release the first resource if the second can’t be acquired  - **Thread Starvation** occurs when a low priority thread is rarely executed due to higher priority tasks always using the CPU | |
| RTOS Dangers 🡪 Best Practices | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| RTOSIntroduction | | Operation System Services **Operating System provides Services:**  🡪 **Resource Pooling:** mutual access to hardware and resources, memory  🡪 **Scheduling:**  quasi-concurrent execution of services  🡪 **Abstraction:**  time base, memory protection (MMU)  🡪 **Middleware:**  file system, communication stacks | | | | **Different Level of Services**  🡪 Timer based Scheduler:  ’Trigger’ or ’Interrupt’ systems  🡪 Mini-Kernels: FreeRTOS, μC-OS, mbed OS, RTXC  🡪 Mid-Range RTOS with driver stacks: MQX, QNX, eCOS  🡪 Mobile and Embedded Linux: Android, iOS, Debian  🡪 Host Linux and OS: MS Windows, Mac OS X, Ubuntu |
| **Baremetal** | **RTOS** |
|  | | | |  |
| - RTOS solves synchronization problems  - RTOS provides Scalability (Skalierbarkeit 🡪 usable in many different types of devices) and Maintainability (Instandhaltbarkeit)  - RTOS need a scheduler | |
| Driver Model **Standard Operating System**  🡪 Drivers part of OS  🡪 Application not allowed to take control of hardware  🡪 Usually better protection  **Real Time Operating System (RTOS)**  🡪 Drivers as services outside OS  🡪 Application has direct access to hardware  🡪 Better performance, closer to the hardware | | | | RTOS Kernel Architecture The Kernel is the part of the operating system that is responsible for task management, and intertask communication and synchronisation.  🡪 Catching interrupts (few or all)  🡪 Needs time base: Timer/Tick interrupt  🡪 System calls and traps (SysCall gives the control from program to the kernel) :  trigger interrupt  🡪 Scheduling: System call or interrupts | | |
| RTOS Requirements - A Real-Time Operating System (RTOS) is an Operating  System (OS) intended for real-time Systems.  - **≠** “As fast as possible”!  **Requirements:**  1. The correct result  2. At the correct time  3. Independent of the current system load  4. In a deterministic and foreseeable way | | | Scheduler 🡪 Pre-emption with Tick interrupt  🡪Tick passes control to Scheduler  🡪 Scheduler can  schedule other task  🡪Opportunities for scheduler: Tick, Wait(), Yield() and Sync() | | | |
| Operating System Process States (general) 🡪 more detailed view and descriptions in FreeRTOS part of this summary   - Only one task in running state (if just one core available)  Pre-empt (vorbelegen), dispatch (abfertigen) | | | | |  | |

|  |  |  |
| --- | --- | --- |
| FreeRTOSIntroduction - Maintained by Real Time Engineers Ltd., London (**Richard Barry**)  - Open Source, free-of-charge, royalty free  - Ecosystem and commercial supported ports available:  🡪OpenRTOS: commercial supported version  🡪SafeRTOS: special version dedicated to safety critical systems  🡪Sold Reference Manual/Tutorial Book (protected PDF with watermark, no print/copy) | | FreeRTOS Licensing - Community support: sourceforge.net/projects/freertos  - Can be used in commercial applications, need to contribute back changes in the kernel  - If source code is published, FreeRTOS needs to be published too  - **Different licensing** terms for OpenRTOS, SafeRTOS and FreeRTOS+ parts |
| FreeRTOS ArchitecturePhilosophy **Kernel:**  - Small Kernel, **implemented in C**, compiled and linked with application  - Pre-emptive or cooperative scheduler mode (at compile time)  - Kernel only needs tick interrupt and software interrupt  - RTOS creates and runs in IDLE task  - Scheduler variables and task stack in dynamic memory (heap)  - Different selection of heap allocation (schemes)  **Ticks:**  - Tick or ’Ticks’ provide **time base** for RTOS  - Counter in tick interrupt  - Typically 10 ms or 1 ms tick period, default max 1 kHz  - ARM: **SysTick timer,** or low power timer (LPTMR on  Kinetis), or any periodic timer  - **All RTOS time calculations are in ticks!**  - Tick frequency/period considerations:  🡪 Interrupt and system load  🡪 Timing precision  **Tasks:**  - Possible dynamic task creation and deletion  - **Task stack and scheduler data structure in dynamic**  **memory (heap)**  - Tasks are running with stack in the ’heap’  - Interrupts are running on ’main’ stack (MSP)  - Tasks are using PSP (Process Stack Pointer)  - Software interrupt used to switch task context  - Tasks are (usually) staying in an endless loop  - RTOS always creates and runs IDLE task | Kernel and Interrupts **RTOS needs two interrupt sources:**  🡪 Tick Interrupt: **SysTick** 🡪 In pre-emptive RTOS mode that interrupt provides a way for the RTOS to pre-empt a running task and to pass control to another task.  🡪 Software Interrupt: **SVCall (**(**S**uper**V**isor **Call**) is triggered by the SVC instruction and is used by the FreeRTOS to start the scheduler. This one is not used for M0.) **, PendSV (**(**Pen**dable **S**er**V**ice) is an interrupt request is used by the OS to force a context switch if no other interrupt is active.)  - **Other interrupts** 🡪 under application control  - Need to care about re-entrancy  -**RTOS does not protect its own data structures:**  🡪 Keep interrupts enabled in RTOS  🡪 Efficiency and interrupt latency  🡪 RTOS API functions with FromISR suffix (e.g. xSemaphoreGiveFromISR) use critical sections  🡪 Special consideration: interrupt nesting | |
| ARM Cortex-M Interrupts - **Cortex-M0+** 🡪 Kernel disables global interrupts  - **Cortex-M4 🡪** using BASEPRI register to mask interrupts  🡪 **configMAX\_SYSCALL\_INTERRUPT\_PRIORITY** 🡪 the kernel only masks out interrupts up and equal to configMAX\_SYSCALL\_INTERRUPT\_PRIORITY numerical value, FreeRTOS API calls from interrupts can only be called from interrupts \*numerically equal or higher\* **(lower urgency)** than configMAX\_SYSCALL\_INTERRUPT\_PRIORITY (**remember: ARM NVIC uses zero as the highest urgency level**).  - **Interrupt setting for RTOS (lowest prio! 🡪 highest number**): SysTick, PendSV and SVCall | |
| Block Diagram - Only 10 source- plus header-files for co-routine, queue, event, timer, list and tasks  - FreeRTOS Config Header File  - 5 different heap implementations  - Port depending on architecture, tool chain and compiler |

|  |  |  |  |
| --- | --- | --- | --- |
| FreeRTOS Kernel ControlvTaskStartScheduler **void** vTaskStartScheduler(**void**)  - Starts the kernel/scheduler (Init 🡪 Running)  - Creates IDLE task (configMINIMAL\_STACK\_SIZE,  tskIDLE\_PRIORITY)  - Does not return until xTaskEndScheduler() | | FreeRTOS TasksPriority-based **Pre-emptive Scheduling** - *configUSE\_PREEMPTION* configures preemptive scheduling  - **Always run the highest-priority ready task**  - Priorities from 0 (lowest) to (*configMAX\_PRIORITIES*- 1)  - Multiple tasks with the same priority: **time-slicing**  - High priority tasks should not starve lower priority tasks | |
| vTaskEndScheduler **void** vTaskEndScheduler(**void**)  - Kernel resources will be released 🡪 All created tasks will be automatically deleted and multitasking (either preemptive or cooperative) will stop  - Task resources (queues, semaphores) are not freed (befreit)  - Many ports do not implement this function | |
| Pre-emptive Scheduling vs Cooperative Scheduling **1. Pre-emptive Scheduling:** The scheduler can take away the CPU from a  running task and make a context switch to another task  **2. Cooperative Scheduling:** The task keeps the CPU until it gives the control back to the scheduler. | |
| vTaskSuspendAll 🡪 suspend (aussetzen) **void** vTaskSuspendAll(**void**)  - Suspends the kernel/scheduler without disabling interrupts  - Context switches will not occur while the scheduler is suspended  - Can be called in a nested way | | Task States **Running:** When a task is actually executing it is said to be in the Running state. It is currently utilising the processor. If the processor on which the RTOS is running only has a single core then there can only be one task in the Running state at any given time.  **Ready:** Ready tasks are those that are able to execute (they are not in the Blocked or Suspended state) but are not currently executing because a different task of equal or higher priority is already in the Running state.  **Blocked:** A task is said to be in the Blocked state if it is currently waiting for either a temporal or external event. For example, if a task calls *vTaskDelay()* it will block (be placed into the Blocked state) until the delay period has expired 🡪 a temporal event. **Tasks can also block to wait for queue, semaphore, event group, notification or semaphore event.** Tasks in the Blocked state normally have a 'timeout' period, after which the task will be timeout, and be unblocked, even if the event the task was waiting for has not occurred. Tasks in the Blocked state do not use any processing time and cannot be selected to enter the Running state.  **Suspended:** Like tasks that are in the Blocked state, tasks in the Suspended state cannot be selected to enter the Running state, **but tasks in the Suspended state do not have a time out**. Instead, tasks only enter or exit the Suspended state when explicitly commanded to do so through the *vTaskSuspend()* and *xTaskResume()* API calls respectively. | |
| xTaskResumeAll **portBASE\_TYPE** xTaskResumeAll(**void**)  - Puts the kernel from Suspended to Active state  **- return value:**  🡪 pdTRUE: Context switch happened  🡪 pdFALSE: no context switch or still suspended (nested) | |
| taskENTER\_CRITICAL, taskEXIT\_CRITICAL - counter used to count nesting  - Some ports (ports sind Prozessortypen in diesem Kontext) always enable when count == 0  - **Do not call FreeRTOS API in Critical Section!**  The taskENTER\_CRITICAL() and taskEXIT\_CRITICAL() macros provide a basic critical section implementation that works by simply disabling interrupts, either globally (e.g. for M0+), or up to a specific interrupt priority level (e.g. for M4(F) using BASEPRI Register)  🡪 **configMAX\_SYSCALL\_INTERRUPT\_PRIORITY** | |
| taskDISABLE\_INTERRUPTS, taskENABLE\_INTERRUPTS enables and disables Interrupts 🡪 Cortex M0+ (all interrupts) and Cortex M4(F) up to BASEPRI 🡪  configMAX\_SYSCALL\_  INTERRUPT\_PRIORITY | | Time Slicing - *configUSE\_TIME\_SLICING* is default mode: time slicing between highest ready tasks with same priority at tick interrupt time.  - Processing time given to ready task with highest priority  - **Tasks with the same priority will time slice**    **Problem with Idle Task:** 🡪 see also next page for details to Idle Task  Time slicing will distribute time equally among same priority tasks (e.g. tskIDLE\_PRIORITY) 🡪 CPU cycles wasted in IDLE task  **Solution:** - With enabled *configIDLE\_SHOULD\_YIELD* IDLE task gives CPU time to other tasks 🡪 No exact time slicing any more | |
| taskYield - *taskYIELD()* is used to request a context switch to another task. However, if there are no other tasks at a higher or equal priority to the task that calls *taskYIELD()* then the RTOS scheduler will simply select the task that called taskYIELD() to run again.  - **Required for cooperative multitasking** 🡪 in preemptive multitasking the scheduler will always be running the highest priority task that is able to run, so calling taskYIELD() will never result in a switch to a higher priority task.  - Typically implemented as macro | |
| Idle Task - Idle Task allways created at scheduler start  - Priority: **tskIDLE\_PRIORITY (is zero and the lowest priority)**  - Stack size: configMINIMAL\_STACK\_SIZE  - Runs if no other task needs to run  - Frees memory of deleted tasks  Calls Idle Task Hook (whenever RTOS is idle):  🡪 Common use in Idle Hook (häufige Anwendung) 🡪 enter power saving mode  - call *taskYIELD()* (forcing context switch to another task) if using cooperative scheduler mode  - otherwise (pre-emptive mode) yield if ready tasks are available 🡪 *configIDLE\_SHOULD\_YIELD* | Blinky Task 🡪 Task Creation and Task “function” **Parameters:**  1. Task function  2. Debug name  3. Optional task parameter (pvParameters 🡪 A value that will passed into the created task as the task's parameter.) or NULL.  4. optional task handle or NULL  🡪 **Task creation can call out-of-heap error hook**  **Task function:**    - Tasks are normal functions  - parameter pvParameters pointer assigned with xTaskCreate()  - Endless loop, does not leave function (except task deletes itself)  - BlinkyTask runs until pre-empted | | |
| Task Control 🡪 Changing the Priority at runtime *UBaseType\_t* ***uxTaskPriorityGet****( TaskHandle\_t xTask );*  - Returns the priority of a task  - NULL argument 🡪 get priority of calling task  - Call only from a task context  *void* ***vTaskPrioritySet****( TaskHandle\_t xTask, UBaseType\_t uxNewPriority );*  - Changes the priority of a task  - NULL argument: priority of calling task  - Call only from a task context | | |
| Task Control 🡪 Delay void **vTaskDelay**( const TickType\_t xTicksToDelay );  - Delay a task for a given number of ticks. The actual time that the task remains blocked depends on the tick rate. The constant *portTICK\_PERIOD\_MS* can be used to calculate real time from the tick rate - with the resolution of one tick period. *vTaskDelay()* specifies a time at which the task wishes to **unblock relative to the time at which vTaskDelay() is called.**  *void* ***vTaskDelayUntil****( TickType\_t \*pxPreviousWakeTime,*  *const TickType\_t xTimeIncrement );*  - Delay a task until a specified time. This function can be used by periodic tasks to ensure a constant execution frequency. *vTaskDelayUntil()* specifies an **absolute time at which the task wishes to unblock.** |
| Task Control 🡪 Suspend and Resume *void* ***vTaskSuspend****( TaskHandle\_t xTaskToSuspend );* 🡪 Suspend any task. When suspended a task will never get any microcontroller processing time, no matter what its priority. 🡪 NULL -> Suspend the calling task  *void* ***vTaskResume****( TaskHandle\_t xTaskToResume );* 🡪 Resumes a suspended task.  *BaseType\_t* ***xTaskResumeFromISR****( TaskHandle\_t xTaskToResume );*  🡪 A function to resume a suspended task that can be called from within an ISR. | | |
| FreeRTOS Hooks - RTOS calls optional application defined callbacks (or hooks)  - Hook function implemented in application | | | Malloc Failed Hook - Enabled with  *configUSE\_MALLOC*  *\_FAILED\_HOOK*  - Called if memory allocation failed |
| Idle Hook - Enabled with *configUSE\_IDLE\_HOOK*  - Called from the IDLE task, whenever the system no user task is running  - **Ideal place to go into low power mode** 🡪 *CPU\_EnterLowPowerMode();*  - Idle hook function not allowed to call blocking API functions! | | |
| Stack OverflowFailed Hook - Enabled with  *configCHECK\_FOR\_*  *STACK\_OVERFLOWS*  - **Overflow check adds run-time cost**  - Method 1: Check SP at task swap time  - Method 2: Method 1 + checking last 16 bytes pattern |
| Tick Hook - Enabled with configUSE\_TICK\_HOOK  - Called from tick interrupt  - **Can be used instead of a dedicated timer running at tick frequency**  - Executed in interrupt context, **keep it short!**  - Only FromISR() RTOS API functions can be called | | |
| Hooks Summary 1. Concept 🡪 hooks are used as events to notify the application 2. Hooks are optional (have to be enabled if we want use it)  3. Error hooks (Malloc Failed Hook and Stack Overflow Failed Hook) 🡪 disable interrupts and halt system | | | |

|  |  |  |
| --- | --- | --- |
| FreeRTOS Heap Memory FreeRTOS is using a dynamic memory area named Heap.  **The Heap is used for:**  - Stack memory for the task  - RTOS internal structures like the TCB (Task Control Block)  - Allocation of Semaphores, Mutex and Queues  - Dynamic memory allocated by application with pvPortMalloc()  - Configure the Heap size with *configTOTAL\_HEAP\_SIZE* 🡪 size in bytes  - Efficiency: tuned for small code size  - Thread safe: to be used from multiple threads  - Deterministic: timing guaranteed within error margin (Fehlerbereich)  - Multiple allocation ’Schemes’, one selected at  configuration time | | Heap Schemes 🡪 *configFRTOS\_MEMORY\_SCHEME*  **5 predefined allocation ‘Schemes’:**  **1. Alloc only 🡪** The Scheme 1 is a very simple one. Memory and tasks only can be allocated, but not freed. This scheme works for many application types where tasks or memory is only allocated at the beginning, but never released. This scheme is deterministic (it always takes the same amount of time) and does not result in memory fragmentation.  **2. No merge 🡪** With Scheme 2 memory blocks can be deallocated, but the **freed blocks are not merged. This can result in Memory Fragmentation!** While Scheme 2 is more efficient than standard C library malloc() and free(), it is not deterministic.  **3. malloc/free 🡪** Scheme 3 is a thread-safe wrapper to the malloc() and free() standard C library functions. Using this scheme enough heap memory should be allocated for the standard library memory manager. *configTOTAL HEAP SIZE* is ignored for that scheme. The standard C library memory allocation is considered as non-deterministic.  **4. Merge free blocks** 🡪 we use this scheme in INTRO Projects.  **Scheme 4 is the same as Scheme2, but does merging of free blocks and therefore avoids memory fragmentation**. Scheme 4 it is not deterministic, but can be more efficient than the standard C heap manager.  **5. Multiple Regions** 🡪 Scheme 5 does memory block merging as in Scheme 4. It supports to have the heap memory split into multiple memory blocks which are non-adjacent (non-contiguous 🡪 nicht benachbart). *vPortDefineHeapRegions()* needs to be used to define the memory regions before using any memory allocation (which includes task creation). |
| Memory Fragmentation  **With Scheme 2 it is possible to have memory fragmentation, because freed memory blocks are not combined.** Even if there would be enough memory available, allocation of a new block can fail because free blocks are not combined  Depending on the amount of free blocks, the implementation is not deterministic 🡪 it takes a variable time until a matching block is found. Unlike Scheme 1, this scheme (Scheme 2) can be used if an application needs to delete tasks, queues, semaphores, mutexes, etc. It (Scheme 2) should not be used if the application uses 'random' or 'dynamic' sequences of allocation and deallocation as this increases the chances for memory fragmentation. | |
| FreeRTOS Queues **-** List of items, allocated in heap **–** FIFO or LIFO/Stack  **-** Queues have ability to block caller (e.g. a task) if queue full or empty **-** RTOS can resume tasks which are blocked on a queue **-** FreeRTOS: **Enqueue and Dequeue by copy operation 🡪 Queue operations in FreeRTOS are 'by value'** on 'arrays of data': the queues contain the data (not only the pointer to the data), therefore items are 'copied in' or 'copied out'. | Adding Items - xQueueSendToBack():  FIFO operation  - xQueueSendToFront():  Stack/LIFO operation  - xTicksToWait: Waiting time in case queue is full  - returns pdPASS or errQUEUE\_FULL | |
| Queue Registry - vQueueAdd  ToRegistry(): make queue name known  - vQueueUnregisterQueue(): remove queue name from list | |
| Creating and Deleting Queues - xQueueCreate(): **fixed queue size at creation time**  - Returns Queue handle, check for NULL!  - **uxQueueLength:** maximum number of items in queue  - **uxItemSize:** element item size in bytes |
| Inspecting and Removing Items - xQueueReset():  Clear queue  - xQueuePeek():  Inspecting item without removing  - xQueueReceive():  Remove item in front the queue  - pvBuffer: pointer where to copy the item  - xTicksToWait: Waiting ticks/time 🡪 0: polling, portMAX\_DELAY: wait forever | Queue Example - **Peripheral Sharing** (e.g. UART bus)  - Inter-Process  Communication **(IPC)**  - Consumer-Producer Pattern  - **Items:** characters  or messages | |

|  |  |
| --- | --- |
| FreeRTOS Semaphore and Mutex Priority Inversion 🡪 Higher priority task blocked by lower priority task  **Solution:** 🡪 Priority Inheritance 🡪 1. Lower-priority tasks inherits priority of any higher-priority task pending on resource 2. At resource release 🡪 task priority change ends | Priority Ceiling Problem with deadlocks 🡪 Nested resources and Task gets resource from a class of resources which can be hold by another task of lower/same prio  🡪 Solution with Priority Ceiling  **- Resource has Ceiling Priority: Priority of highest task priority using it**  **Priority Ceiling Rules:**  **R1:** normal scheduling (higher priority task pre-empts lower priority task)  **R2:** only get resource if task priority is higher than the ceiling of all resources currently hold by other tasks  **R3:** normal Priority Inheritance |
| **Example 1:**  - Priority Ceiling for both resources 🡪 H  **R1:** Normal pre-emption  **R2:** Ceiling of 1 is H, PH is not higher than ceiling of all resources currently hold (1, which is H) **R3:** Normal Priority Inheritance |
| Priority Inheritance Protocol - Task holding resource (PL) inherits priority of requesting  task (PH)  - **Push-Through Blocking:** task is blocked by resource usage  not of its own |
| **Example 2 🡪 nested resource usage**  - Priority Ceiling: S0 🡪 H, S1 🡪 M, S2 🡪 M |
| Nested Priority Inheritance - Nested resource usage: L2 🡪 L1 🡪 U1 🡪 U2  - **Transitive Blocking:** caused by not-directly involved semaphore (resource), accessed in a nested way by blocking tasks. | Semaphore 🡪 Does NOT implement Priority Inheritance mechanism  - Binary (single token) and counting (multiple token) semaphore  - **Used for synchronization/IPC, does not have to be released**  *- xSemaphoreCreateBinary(), xSemaphoreCreateCounting()*  *- xSemaphoreTake(), xSemaphoreGive(), xSemaphoreDelete()* |
| FreeRTOS Mutex 🡪 Implements Priority Inheritance mechanism  - Normal and recursive (can ’take’ multiple times from same task)  - **Used for resource locking, always has to be released**  *- xSemaphoreCreateMutex(), vSemaphoreCreateRecursiveMutex()*  *- xSemaphoreTake(), xSemaphoreGive(), xSemaphoreDelete()* |
| Creating Binary Semaphore 🡪 one or zero token (Zeichen)  - Creating Semaphore creates handle, Check for NULL, **Semaphore token is not ’given’ at creation time** |
| Deadlocks - **Priority Inheritance does not avoid Deadlocks**  - Typical deadlock 🡪 nested resource usage |
| Queues with no Data - API calls implemented as macros  - Macro *semSEMAPHORE\_QUEUE\_ITEM\_LENGTH* is 0  - Macro *semGIVE\_BLOCK\_TIME* is 0 |

|  |  |
| --- | --- |
| FreeRTOS Kernel Awareness - gives Information about tasks/queues | Queues - *vQueueAddToRegistry()*, - FreeRTOS Queue Registry Size > 0 |
| Kernel Awareness Views - Eclipse Menu 🡪 FreeRTOS in Debug Mode  - Stop-Mode Views 🡪 **Only if target stops, Fetches data from target (performance!)** | Runtime Statistics - Enable in FreeRTOS PE Component  - How much time a task spends 🡪  *configGENERATE\_*  *RUNTIME\_STATS*  - Counting time for each task  - Counter at task start and suspend time  - SysTick (Ticks): **limited resolution!**  - Timer (recommendation: 10x of Ticks) |

|  |  |
| --- | --- |
| Shell - We need a simple user interface Hardware Connections - Shell 🡪 Provides Command line Interface  - **Robot V1:** RTT, native USB CDC  - **Robot V2:** RTT, native USB CDC, UART over tinyK20  - **Remote:** RTT, native USB CDC, UART over tinyK20 |  |
| Shell interface |  |