# 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 conflics  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! (componentchange)  - 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 | | | | | | | | Soft Realtime Degradation, 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 | | | | |