Zephyr in Practice:
From Functional Design to Efficient Implementation

Zephyr Meetup Karlsruhe - December 4th 2025

Tobias Kästner, inovex

That’s me

Tobias Kaestner

@tobiaskaestner

@tobiaskaestner

Solution Architect Medical IoT @ inovex GmbH

#FOSS4MEDICAL

- PhD in Physics (long ago)
- SW/System Architect since 15 years
  - mainly Medical Devices
- Trainer & Technical Consultant
  - SW-Architecture, Zephyr, Yocto
- In Love w/ Zephyr since 2016
  - realised several prototype projects for life-science R&D
  - Maintainer of TiacSys-Bridle Project
  - Participant Zephyr Safety-WG

WE ARE INOVEX
INNOVATE. INTEGRATE. EXCEED.

Your partner for digital transformation

Application Development:
Embedded - Backend - Cloud Native - UX
Data & AI
Scalable IT infrastructure
Training & Coaching

LINUX ZEPHYR AOSP

inovex

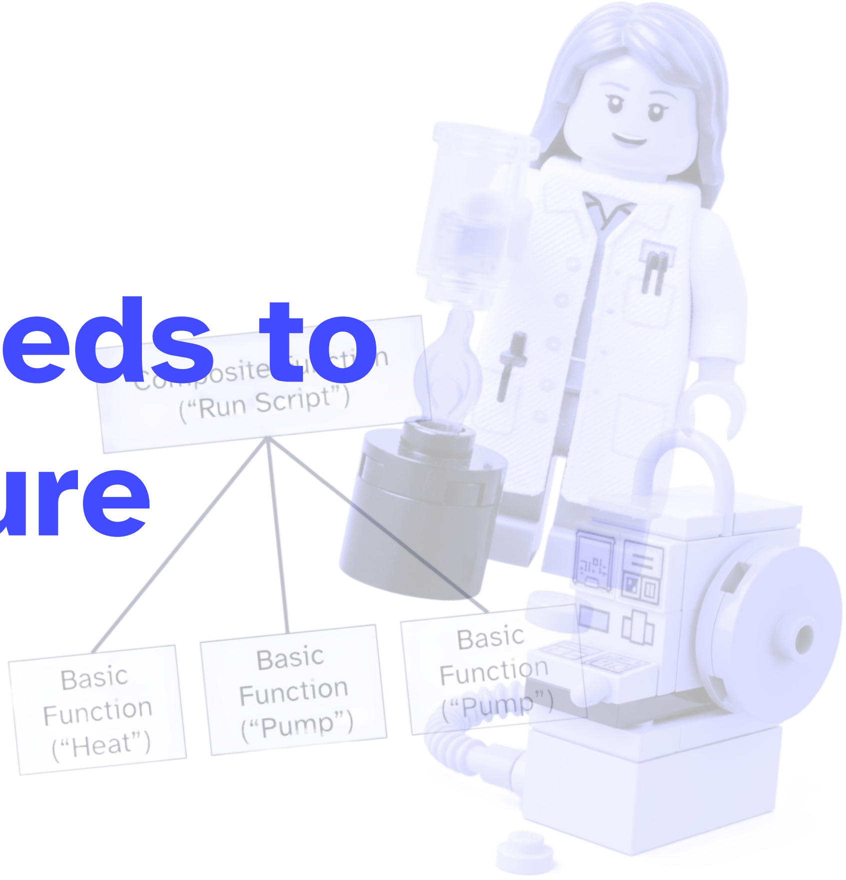
inovex.de

SOFTWARE · DATA & AI · INFRASTRUCTURE

Agenda

- From User Needs to SW Architecture
- Architecting Embedded SW with Zephyr
- Designing Embedded SW with Zephyr

From User Needs to SW Architecture



A day in the lab - The life-science domain

- Developing new diagnostic tests requires extensive research & development
    - to find correct chemical formulation
    - to determine physical parameters
    - to develop algorithms for signal extraction
- Scientists can be supported by devices to automate many/all of the required tasks

Modelling the Life-Science Domain

Doing lab experiments requires

- moving & mixing liquids
- heating & cooling reagents
- measuring signals from chemical reactions
- running prescribed protocols (assays) repeatedly

System functionalities

- pump
- heat, cool
- measure signals (electrodes, image)
- run a script

inovex

System functionalities & modalities

System functions expressed in terms of **the specific domain**

Modalities describe recurring **facets** or **aspects** of system functions

System functionalities

- pump
- heat, cool
- measure signals (electrodes, image)
- run a script

Cross-cutting modalities

How to

- invoke
- compose
- monitor/observe
- parametrize

the system functions

System functionalities & modalities

System functions expressed in terms of **the specific domain**

Modalities describe recurring **facets** or **aspects** of any system function

System functionalities

- application domain specific
- require most likely specific technical realization

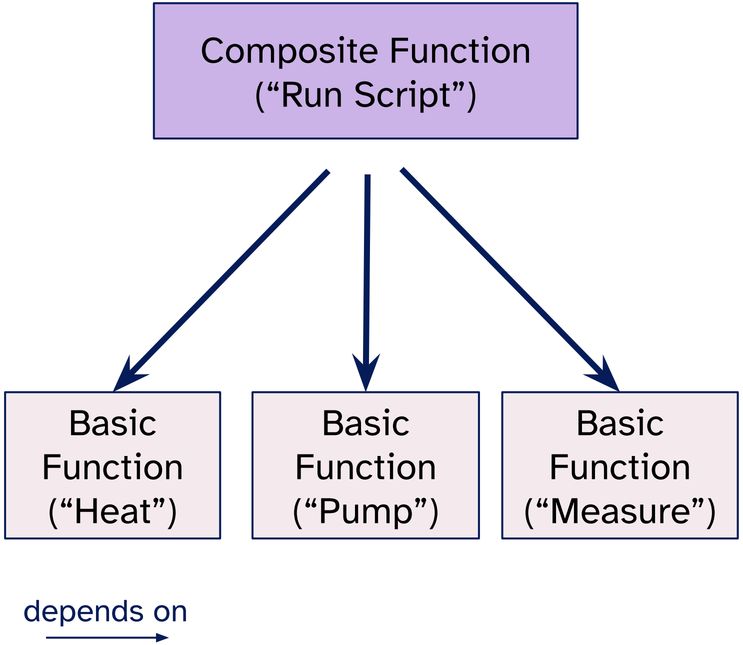
Cross-cutting modalities

- generic to most/all computerized systems
- can most likely re-use existing technical realizations

Caution: In the real world most things fall onto a spectrum, eg. “run script”

inovex

System function composability



User-level tasks are typically expressed as **composite system functions**

Decompose system functions to

- model functional dependencies
- identify **mutually independent** basic functions

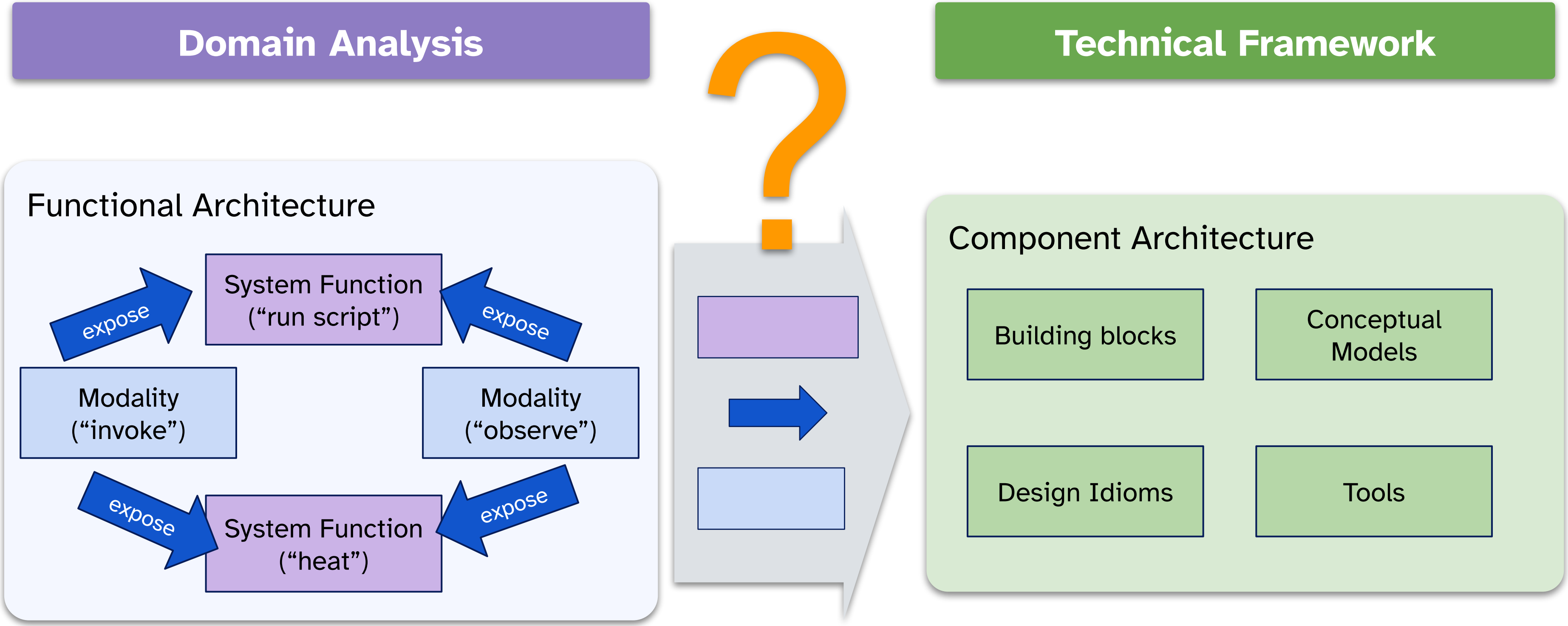
Cross-cutting modalities

Modality	System function **invocation**	System function **composition**
Technical Concept	Human Machine Interface/ Machine Machine Interface	Composability
Realization Patterns	• Command Line Interface • Graphical User Interface • Remote Procedure Call • REST API • …	• Task model • Data passing • …

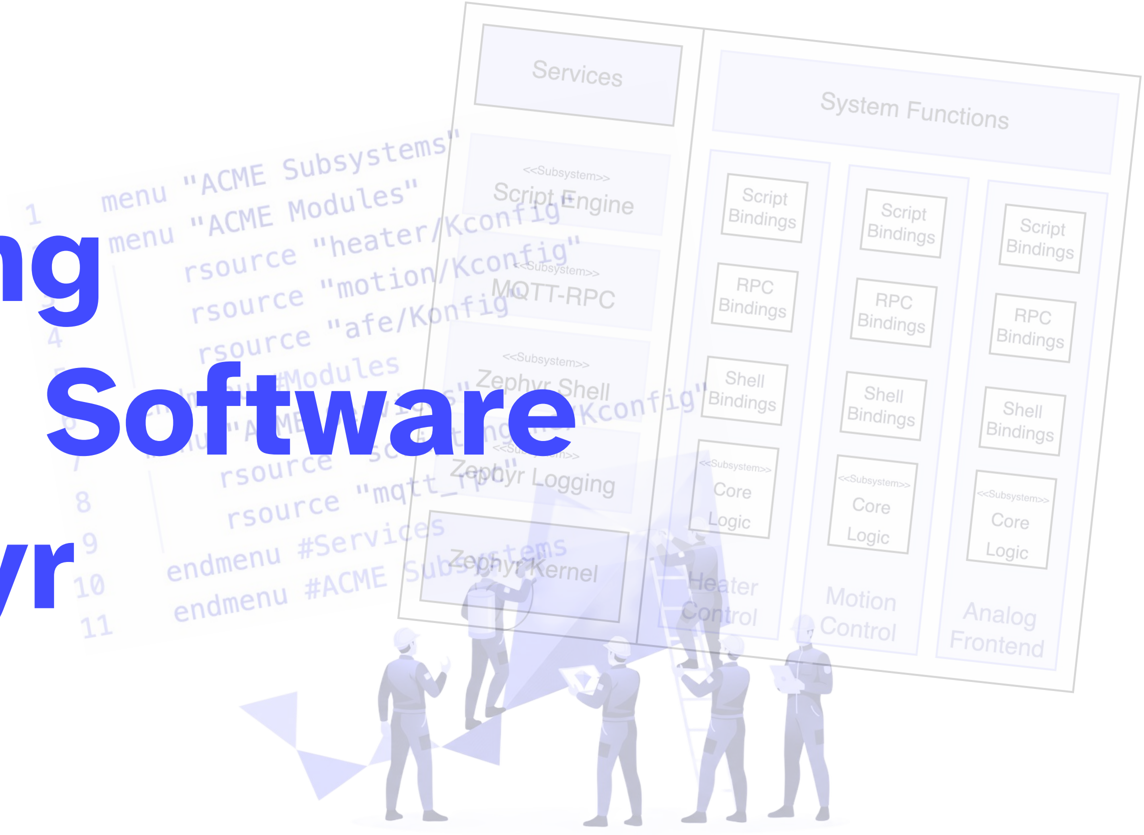
Cross-cutting modalities

Modality	System function **monitoring**	System function **parametrization**
Technical Concept	Observability	Configurability
Realization Patterns	• Logging • Telemetry • …	• Calibration data • Credentials • Access modes • …

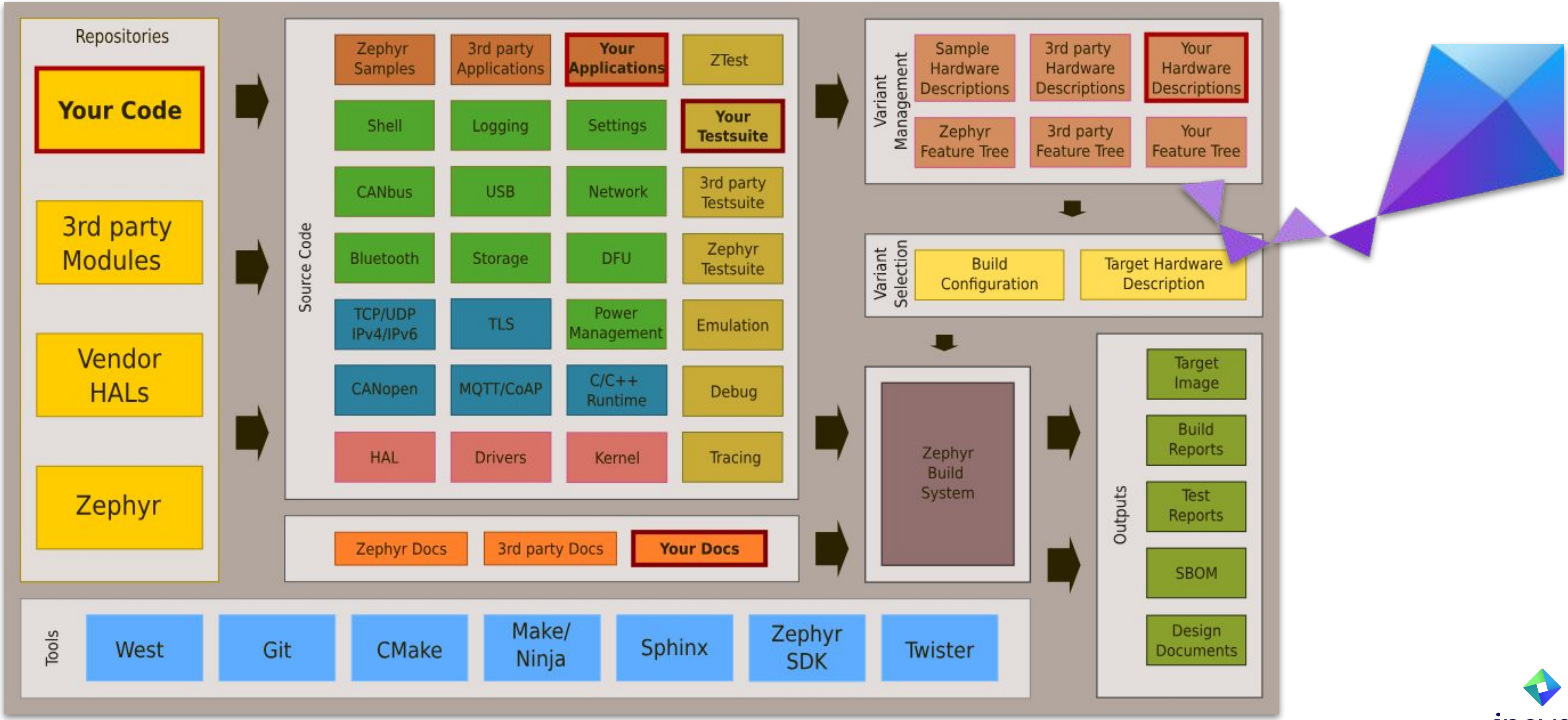
Mapping the functional architecture



Architecting Embedded Software with Zephyr



Zephyr - A modern embedded Software Framework



inovex

Zephyr's Conceptual Models

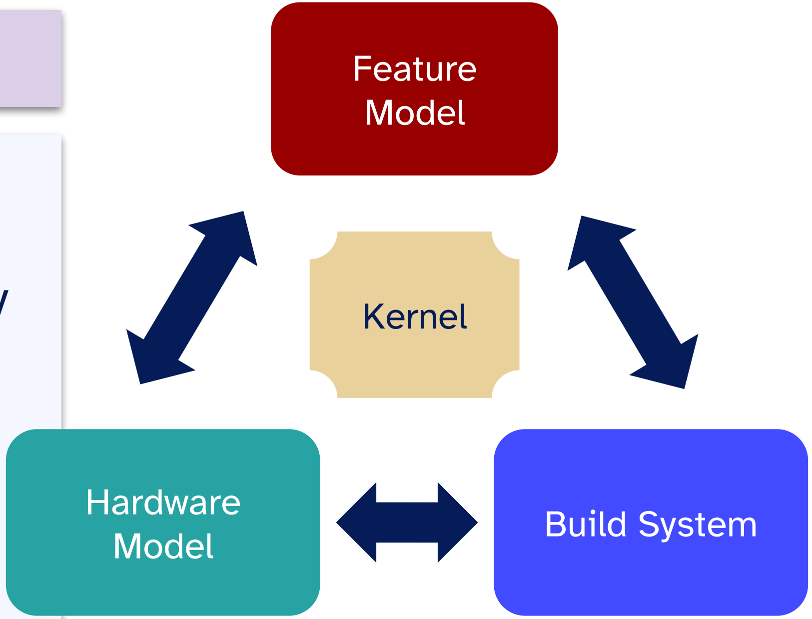
Programming Model: RTOS Kernel

Plus 3 domain-specific models

Feature Model: to select desired functionality

Hardware Model: to describe hardware properties

Build System: to describe build process



Zephyr's High-Level Building Blocks

Applications

Subsystems - have own runtime context via Tasks, Work Queues

Libs - collections of functions for synchronous invocation

Drivers - implement HW-specific details against common device APIs

Kernel - Basic RTOS primitives for synchronization, data passing

Tests

SW-Architecture needs to **create** or **re-use** these **building blocks** to **express** the **functional architecture** (functions, modalities).

inovex

(Some of) Zephyr's Design Idioms

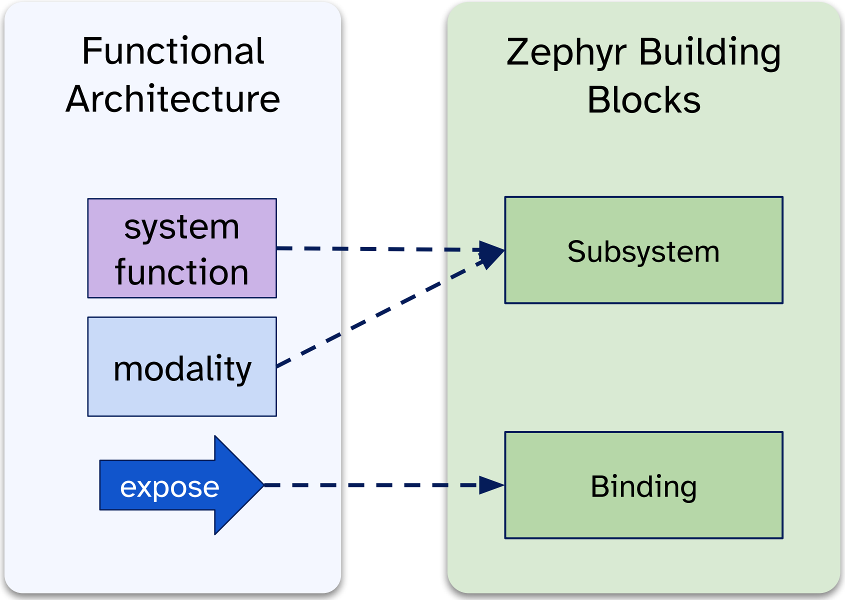
- **RTOS API**
  - implied by Programming model
- **CPP (C-Pre-Processor) driven Code Generation**
  - non-typed meta-programming
- **APIs from Function Pointer Structs**
  - decouples interface users from implementors
- **Iterable Sections**
  - build-time resolvable plugin mechanism

inovex

Mapping the functional architecture

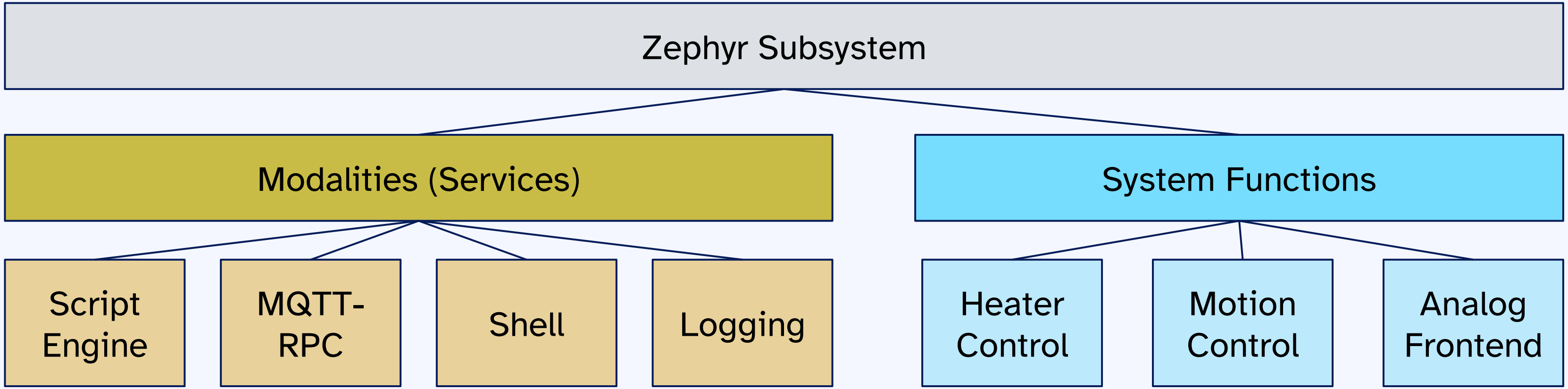
- Modalities map naturally to (existing) Zephyr subsystems
- System functions become additional subsystems
- Services expose aspects of system functions via bindings

Mapping **preserves structural relationships** of functional architecture



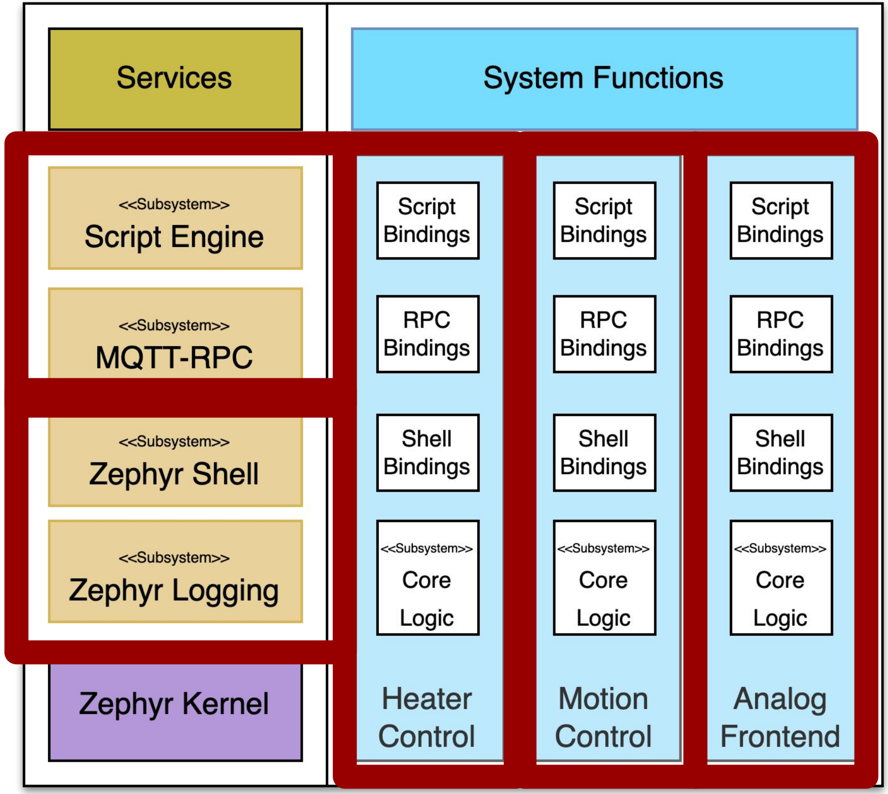
Functional Architecture & Component Architecture

Our Example: Test rig for life-science experiments



- Decomposition into System-Level Functions & System Services
- Modalities & Functions mutually independent from each other

Functional Architecture & Component Architecture

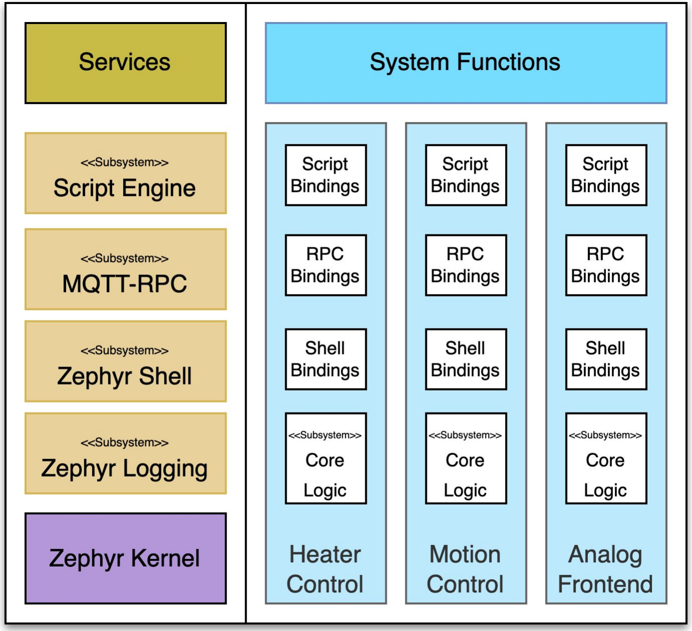


- Zephyr provides many services already
  - Shell, Logging, Settings, …
- Each Zephyr service also provides extension points
  - SHELL_CMD,
  - LOG_MODULE_DEFINE,
  - SETTINGS_STATIC_HANDLER_DEFINE, …

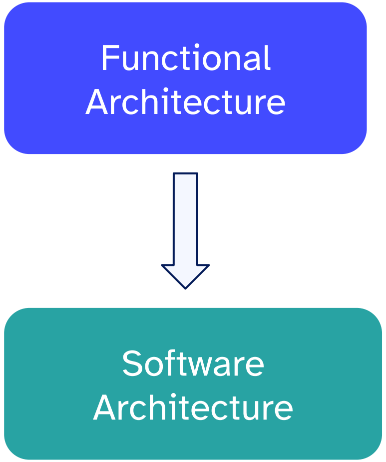
- Use conceptual model and idioms to implement system function specific subsystems

inovex

Modeling Software Features w/ Kconfig




```
menu "ACME Subsystems"
menu "ACME Modules"
    rsource "heater/Kconfig"
    rsource "motion/Kconfig"
    rsource "afe/Konfig"
endmenu #Modules
menu "ACME Services"
    rsource "scriptengine/Kconfig"
    rsource "mqtt_rpc"
endmenu #Services
endmenu #ACME Subsystems
```



Modeling Software Features w/ Kconfig

Subsystems mutually independent

- enable/disable

Bindings depend on service providing corresponding extension point

Feature tree mapped to build system

```
menuconfig ACME_SUBSYS_HEATER # option to toggle the entire subsystem on/off
    bool "Heater subsystem"
    help
        The Heater subsystem is responsible for measuring and controlling
        the temperature.

if ACME_SUBSYS_HEATER

    config ACME_SUBSYS_HEATER_THREAD_STACK_SIZE
        int "Stack size of subsystem thread"
        default 2048

    config ACME_SUBSYS_HEATER_MQTT_RPC
        bool "Enable MQTT-RPC bindings for $(subsys-str) subsystem"
        depends on ACME_MQTT_RPC

    config ACME_SUBSYS_HEATER_SHELL
        bool "Enable shell bindings for $(subsys-str) subsystem"
        depends on SHELL

```

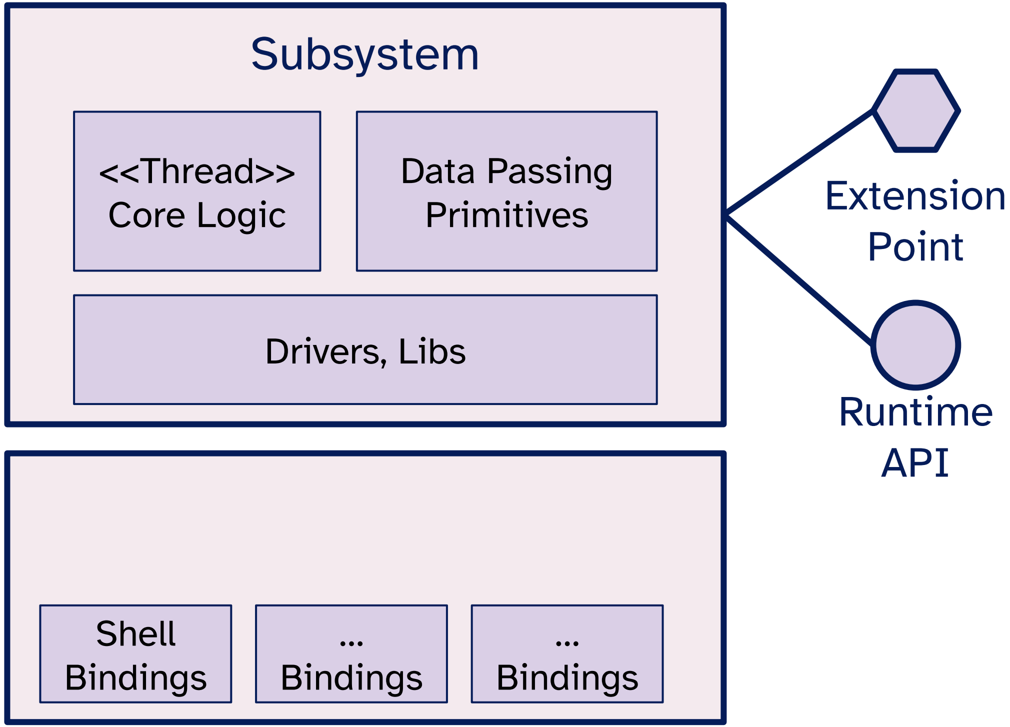
```
zephyr_library_named(acme-heater)

zephyr_library_sources(heater.c)
zephyr_library_sources_ifdef(CONFIG_ACME_SUBSYS_HEATER_SHELL heater_shell.c)
zephyr_library_sources_ifdef(CONFIG_ACME_SUBSYS_HEATER_MQTT_RPC heater_mqttrpc.c)
zephyr_library_sources_ifdef(CONFIG_ACME_SUBSYS_HEATER_SCOPE heater_scope.c)

```

Designing Software with Zephyr

Designing Subsystems for Zephyr



- Subsystems provide own runtime context

- Runtime API wraps Data Passing Primitives to interact with core logic
- Bindings "hook" into other subsystems
    - primary place to use the Runtime API

- (Optional) Extension Points for custom modalities

Designing Subsystems for Zephyr

EXPLORER

- OPEN EDITORS
- SAMPLES
  - .vscode
  - boards
  - dts
  - include / acme
    - heater.h
    - scriptengine.h
  - shields
  - subsys
    - m_afe
    - m_heater
      - CMakeLists.txt
      - heater_mqttrpc.c
      - heater_script.c
      - heater_shell.c
      - heater.c
      - Kconfig
    - s_script
      - CMakeLists.txt
      - iterables.ld
      - scriptengine.c
    - CMakeLists.txt
    - Kconfig
    - Kconfig.template.su...

heater_shell.c — subsys > m_heater > heater_shell.c

```c
#include <zephyr/shell.h>
#include <acme/heater.h>

static int cmd_heater_on(const struct shell *sh,
                         size_t argc, char **argv) {
    return heater_on();
}

static int cmd_heater_off(const struct shell *sh,
                          size_t argc, char **argv) {
    return heater_off();
}

SHELL_STATIC_SUBCMD_SET_CREATE(sub_heater,
    SHELL_CMD(on, NULL, "Turn on heater"),
    SHELL_CMD(off, NULL, "Turn off heater"),
    SHELL_SUBCMD_SET_END);

SHELL_CMD_REGISTER(heater, &sub_heater,
                   "Heater commands, NULL");
```

heater_script.c — subsys > m_heater > heater_script.c

```c
#include <acme/scriptengine.h>
#include <acme/heater.h>

static int l_heater_on(lua_State *L){
    int result = heater_on();
    lua_pushinteger(L, result);
    return 1;
}

SCRIPT_ENGINE_REGISTER_FUNC(heater_on, l_heater_on);
```

scriptengine.h — include > acme > scriptengine.h

```c
#include <zephyr/sys/iterable_sections.h>
#include <acme/lib/lua/lua.h>

// many other things here

const struct  {
    const char *name;
    int (*)(lua_State *L) register_func;
} script_function_entry;

#define SCRIPT_ENGINE_REGISTER_FUNC(_name, _register_func) \
    static const STRUCT_SECTION_ITERABLE(script_function_entry, _name) = {\
        .name = STRINGIFY(_name),    \
        .register_func = _register_func \
    }
```

Extension Point

heater.h — include > acme > heater.h

```c
#include <zephyr/kernel.h>

//**Public API of the Heater Subsystem */

int heater_on();
int heater_off();
```

Runtime API

scriptengine.c — subsys > s_script > scriptengine.c

```c
static int register_functions()
{
    STRUCT_SECTION_FOREACH(script_function_entry, entry){
        lua_pushcfunction(scriptengine.L, entry.register_func);
        lua_setglobal(scriptengine.L, entry.name);
    }
}
```

Applications as Configuration Management Containers

prj.conf

```
CONFIG_ACME_SERVICE_SCRIPTING=y
CONFIG_ACME_SERVICE_MQTTRPC=y

CONFIG_ACME_SUBSYS_MOTION=y
CONFIG_ACME_SUBSYS_HEATER=y
CONFIG_ACME_SUBSYS_AFE=y
```

dev-overlay.conf

```
CONFIG_LOGGING=y
CONFIG_SHELL=y
```

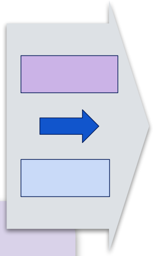
main.c

```c
int main(int argc, char** argv){
        return 0;
}
```

- Any Zephyr application is a concrete instance of the feature model
- Relevant features described in **`prj.conf`**
- Configuration fragments can be merged at build-time:
  - context-of-use (**`prod`** vs **`dev`** vs **`test`**)
  - hardware (**`board`**-specific overlays)

Ideally, applications do not contain any additional code

Summing up



Domain Analysis

Identify System Functions & Modalities

Mapping

Map to existing Zephyr subsystems/custom subsystems and their extension points

Design/Implement custom subsystems

Use Zephyr's Models & Idioms to implement additional subsystems, implement bindings

Configure Application

Realize applications as context-aware instances of the underlying feature model

Conclusions

- Starting an embedded systems design from its functional decomposition bears many benefits
  - clearly analysed (functional) dependencies
  - consistent, domain-oriented terminology
- Zephyr supports the work of SW architects with
  - advanced models and design idioms
  - a rich set of existing functionalities
- When designing with Zephyr always consider
  - feature model and build system integration
  - re-using existing subsystems

Thank You

Check out our Zephyr Hands-On Trainings

Find out more
https://www.inovex.de/de/training/zephyr-basic-training/

Dr. Tobias Kästner
Solution Architect Medical IoT

tobias.kaestner@inovex.de

+49 152 3314 8940

Allee am Röthelheimpark 11,
91052 Erlangen

Tobias Kaestner

@tobiaskaestner

@tobiaskaestner