

Introduction to Analog Front-end Signal Processing Experiment

H. Krüger, University of Bonn
RADHARD School 2024

Content

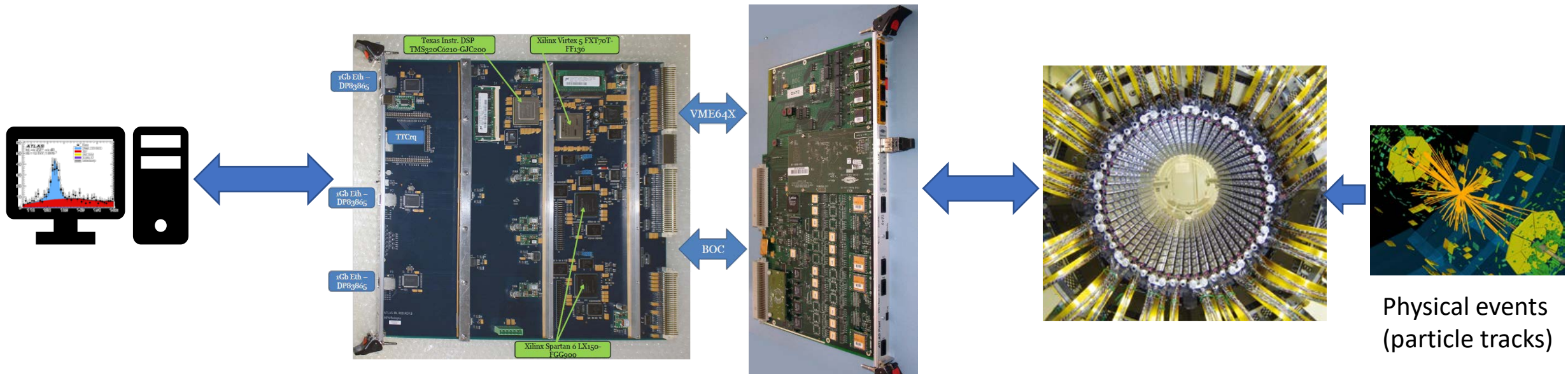
- Introduction
- Analog Front-end for semiconductor detectors
 - Function blocks
 - Electrical parameters
 - Measurement methods
 - Hardware and software
- Embedded-System-Lab
 - Hardware overview
 - Software environment
 - (General Purpose Input/Output port (GPIO)programming example)

Introduction

- The “Embedded System Lab” experiments are relatively new
- Introduced as part of the lecture “Advanced Electronics and Signal Processing”
- Upgrade/addition to the “Electronics Lab Course”, student internships etc.
- Here only the analog frontend module “AFE” will be used

Signal Path from Hardware to Software (and back)

Example: ATLAS pixel detector data acquisition system (simplified)



Computer / File server

- Detector control software
- Data processing & storage
- Analysis software

Data Acquisition Electronics

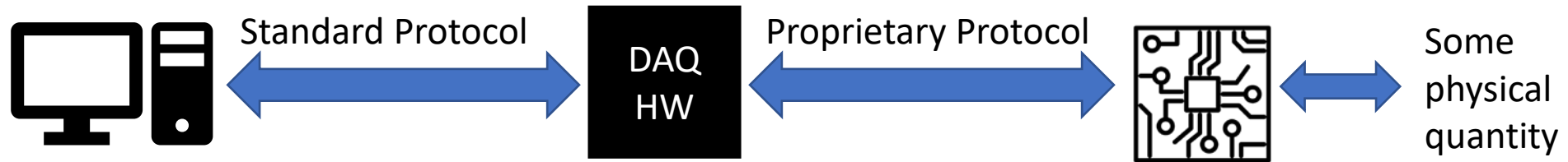
- Data aggregation & preprocessing
- FPGA boards
- Bus interface boards

Pixel Detector

- Application Specific Integrated Circuits (ASICs)

Signal Path from Hardware to Software (and back)

- Generic data acquisition system



Computer
Control & analysis software

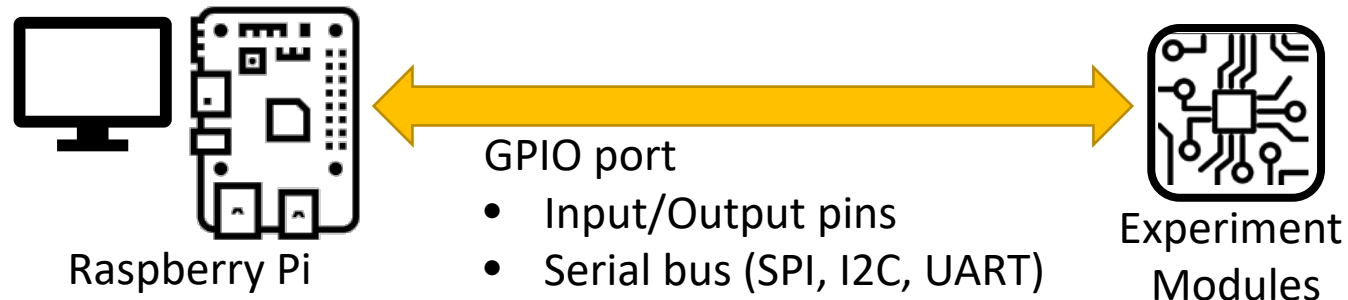
Data Acquisition Electronics

Detector Frontend Electronics

- Much of the interaction between SW and HW in a **black-box**
 - Overhead of standard data protocol (USB, TCP/IP etc.)
 - Signal preprocessing in intermediate DAQ hardware (FPGAs)
 - Added complexity by multiple SW and HW layers

Embedded-System-Lab Concept

- Use **Raspberry Pi** General Purpose Input/Output (**GPIO**) port to communicate with external electronic circuits



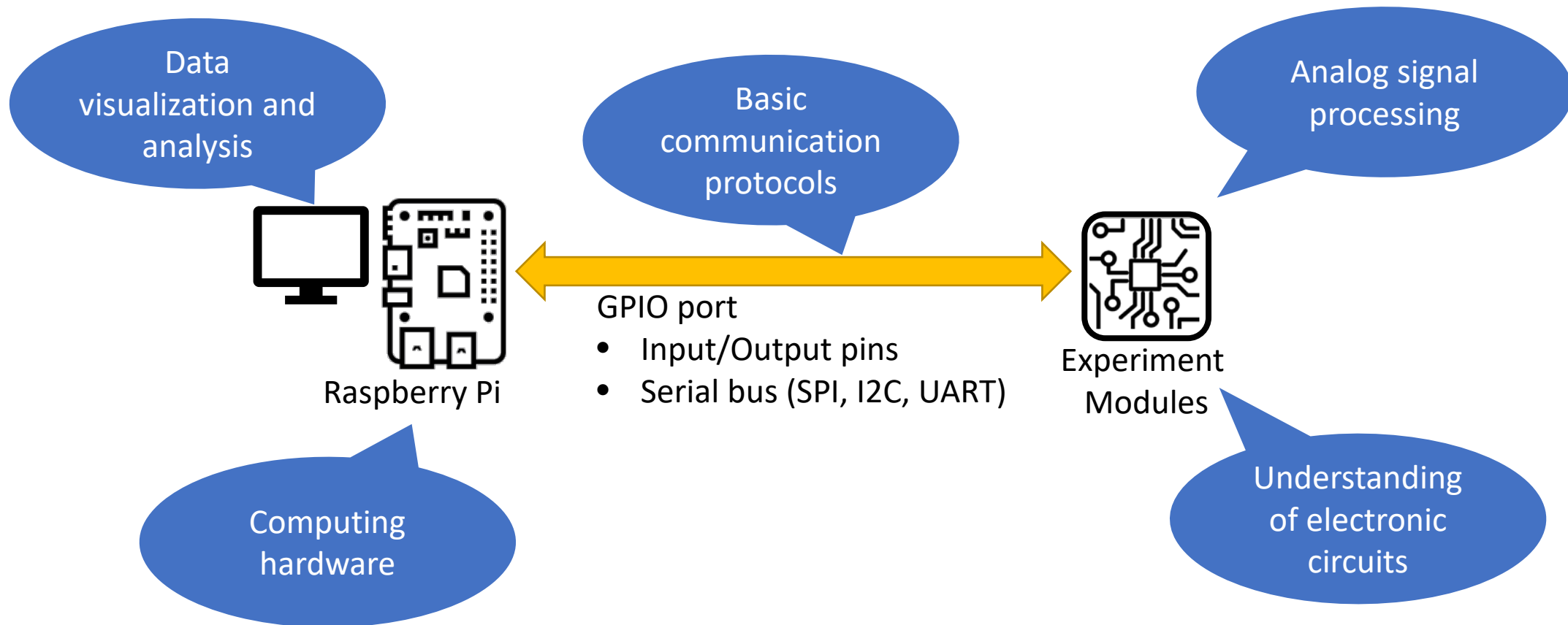
- **Hardware**

- LEDs, Buttons
- Simple (and more complex) circuits: (**modules, aka Experiments**)

- **Software**

- User code in Python (and some C) to access GPIO port
- More complex IO functions: Python modules (a bit of a black-box again...)
- Python scripts for data acquisition and analysis

Embedded-System-Lab Learning Contents



The Embedded System Lab Hardware

Embedded-System-Lab Setup

User interface

- Python scripts
- Data plots
- Web browser (documentation)

Embedded-System-Lab base board

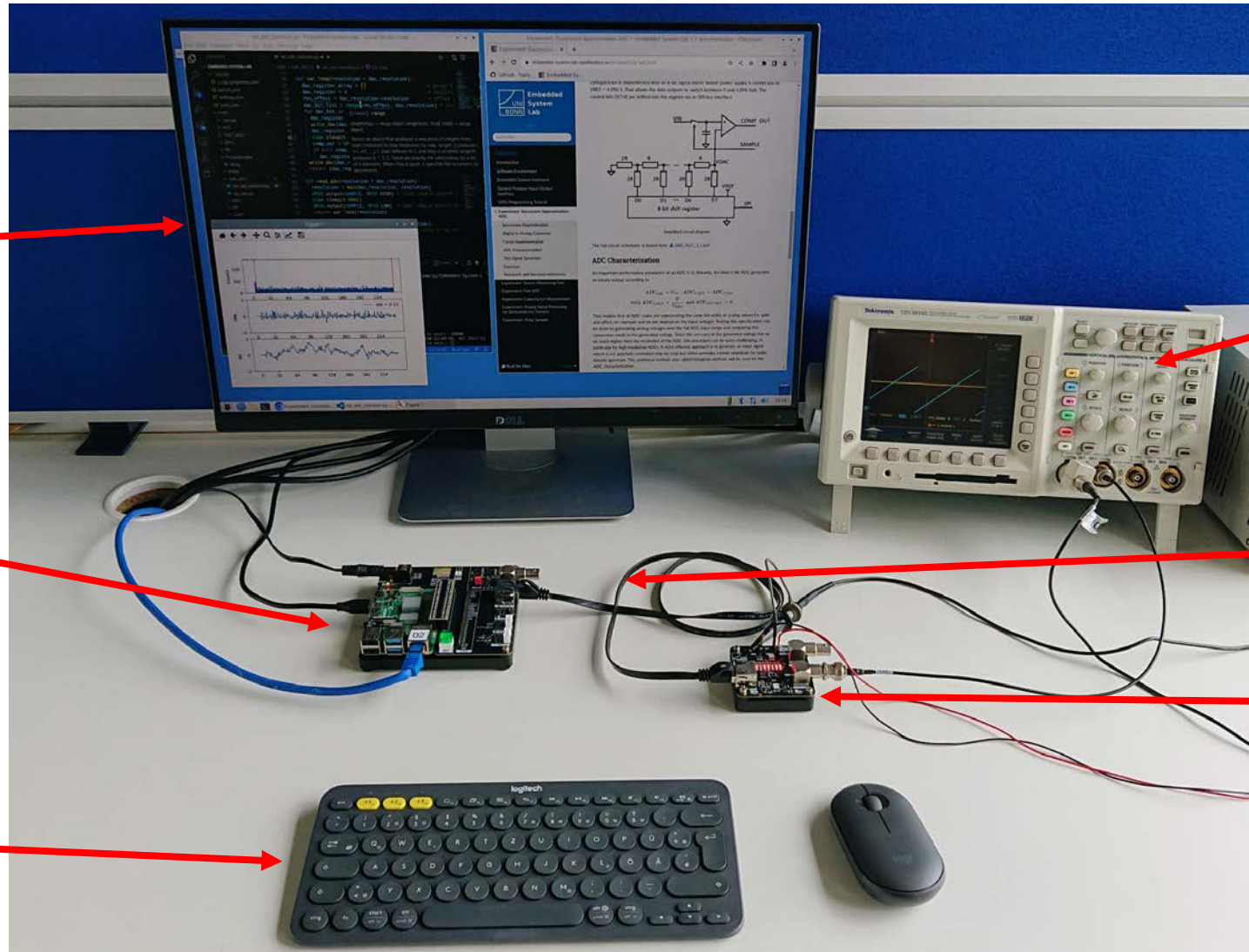
Keyboard & mouse

External instruments

- Oscilloscope
- Lab power supply
- ...

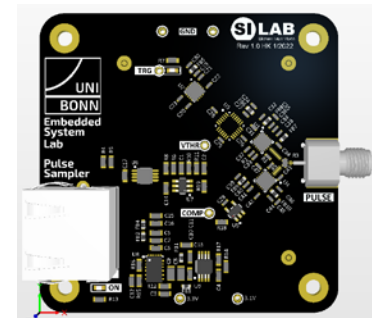
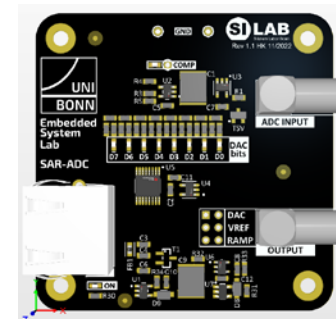
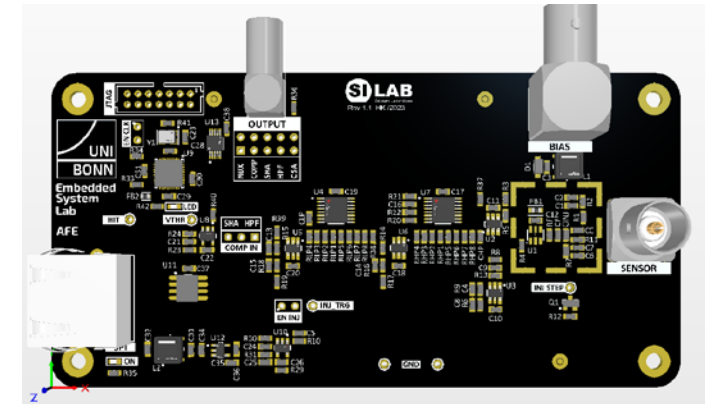
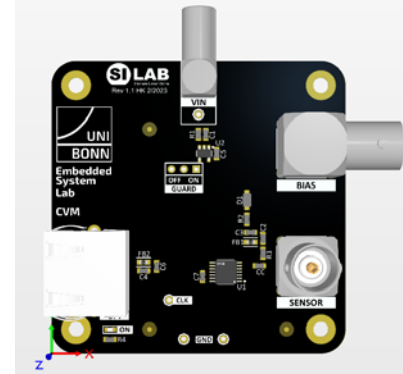
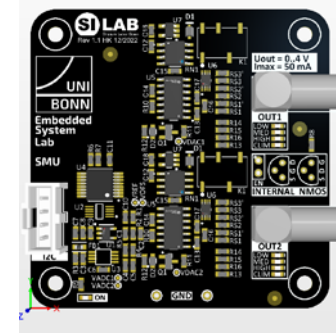
RJ-45 cable with GPIO signals

Experiment module

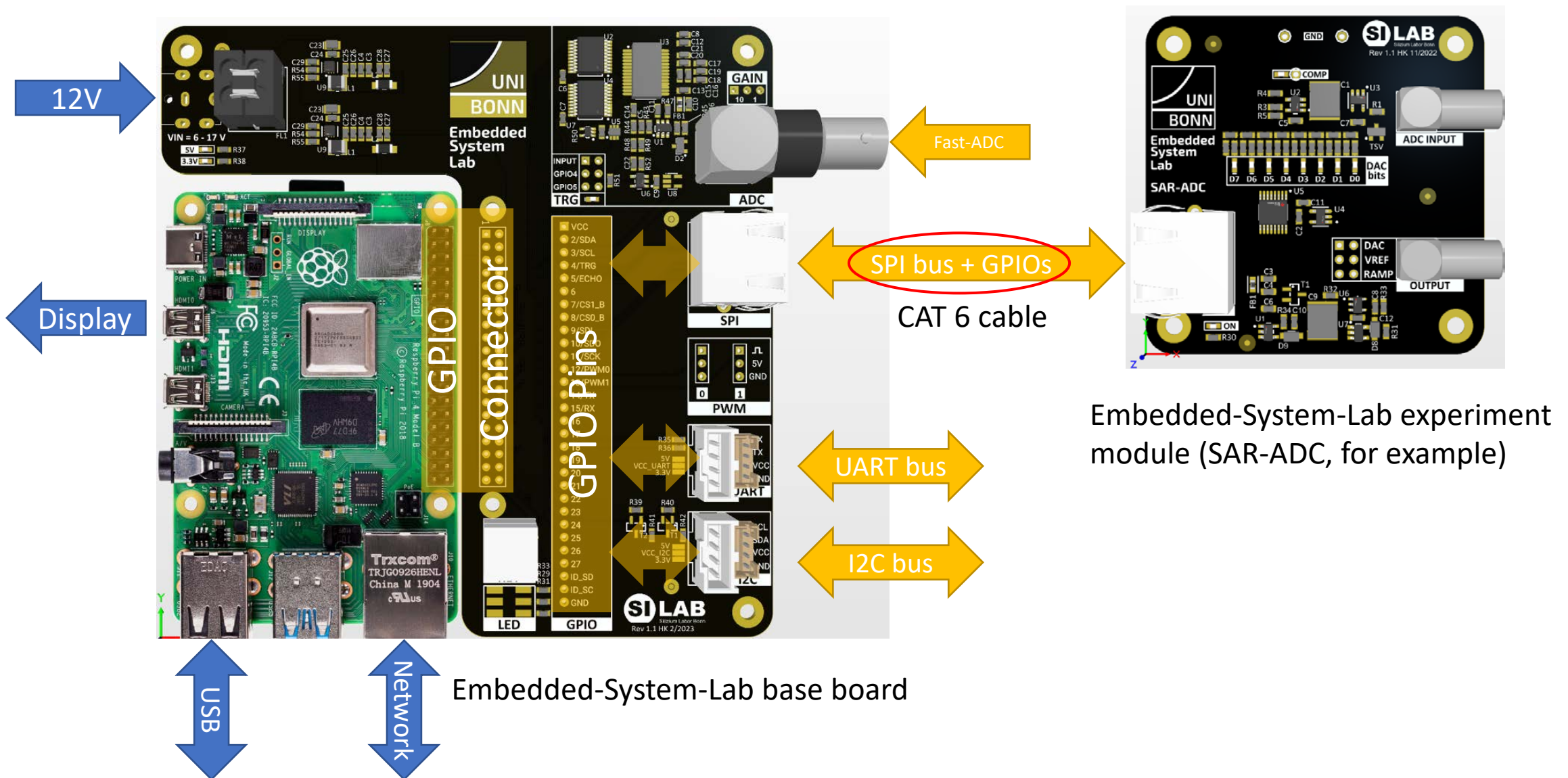


Lab-Experiment Modules

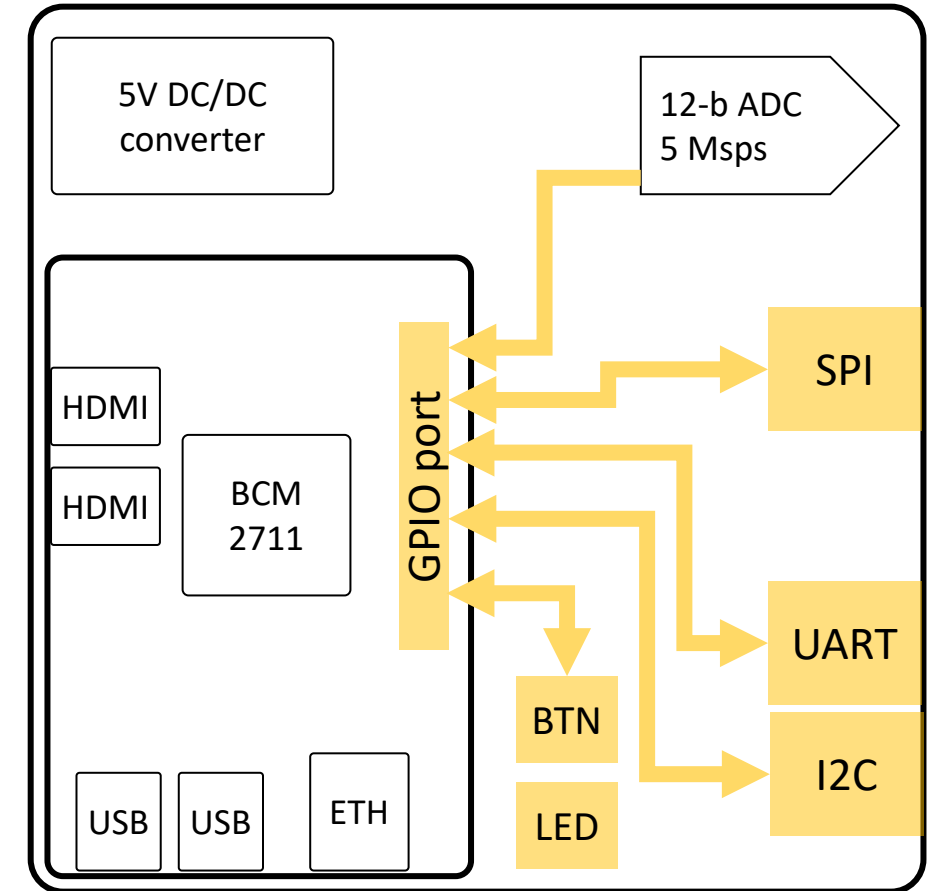
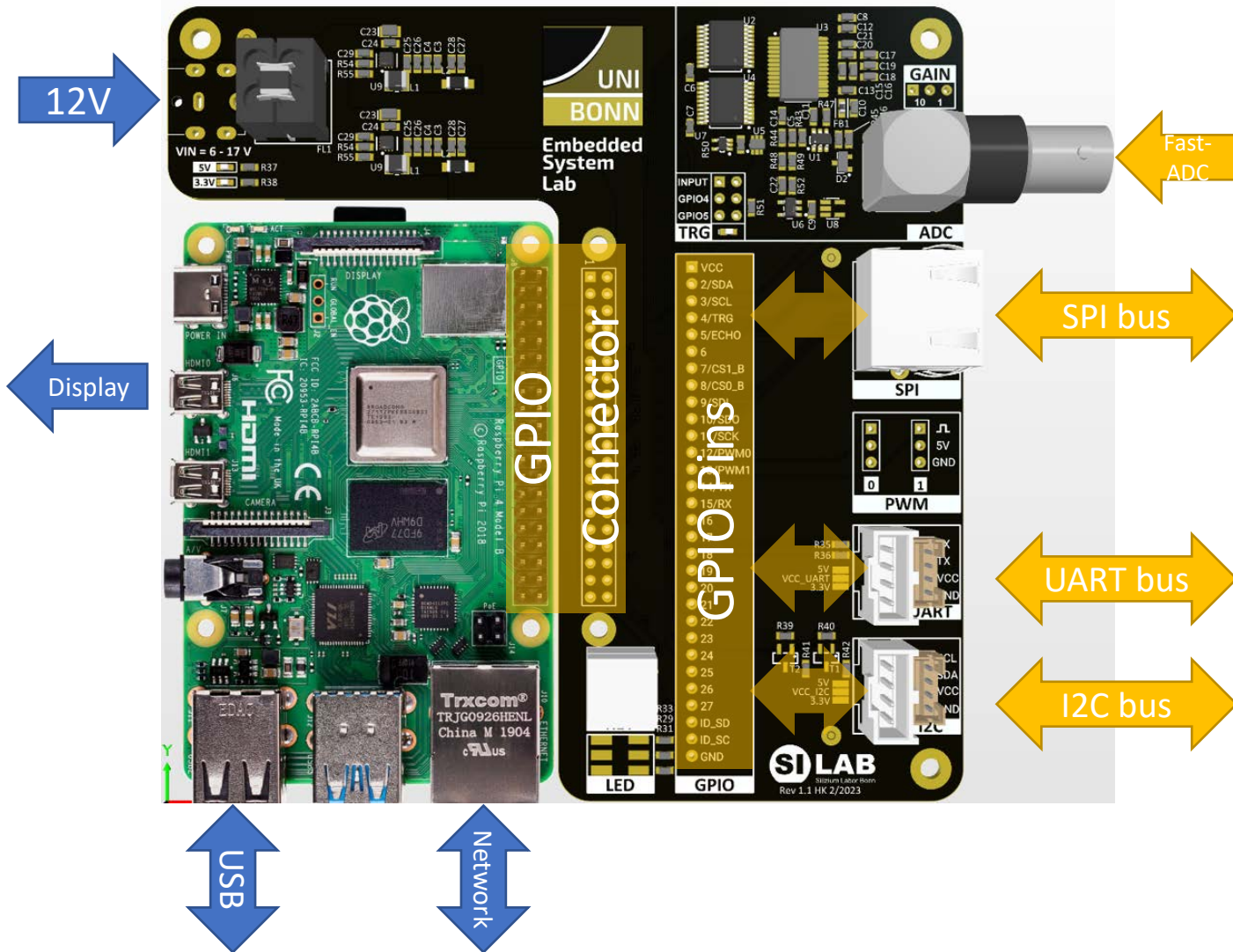
1. Source Monitoring Unit (**SMU**)
Dual channel voltage source, nA to mA current measurement
2. Precision Capacitance Measurement (**CVM**)
Current-based-capacitance-measurement
3. Analog Front-end for sensor signal processing (**AFE**)
Charge sensitive amplifier, programmable shaping amplifier and comparator + Digital logic for the AFE signal processing (**FPGA**)
Time over threshold amplitude digitization
4. Analog-Digital-Converter (**SAR-ADC**)
8-bit successive approximation ADC, based on R-2R DAC
5. Time domain reflectometry measurement (**TDR**)
Serial pulse analyzer with < 50 ps resolution



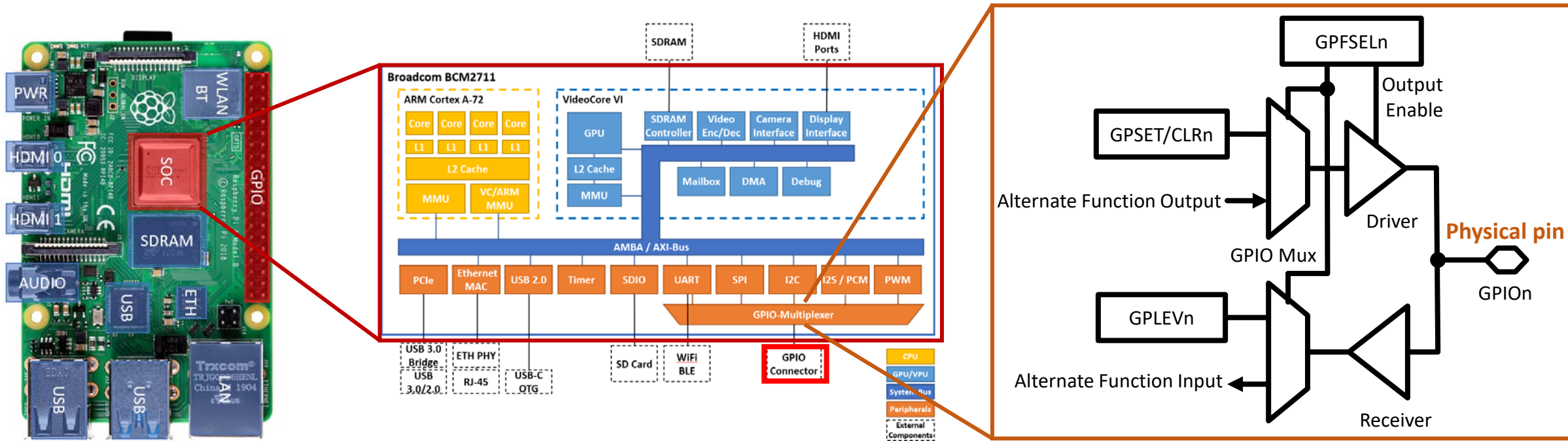
Embedded-System-Lab Base Board + Module



Embedded-System-Lab Base Board



General Purpose Input/Output Ports - GPIO



RPi 4B board

- Single board computer
- CPU, file storage
- Audio & Video
- Network

SOC block diagram

- ARM based CPU (quad core)
- VideoCore (GPU)
- Peripheral I/O blocks

GPIO block diagram

- Basic input/output function
- Alternate functions (UART, I2C, SPI...)

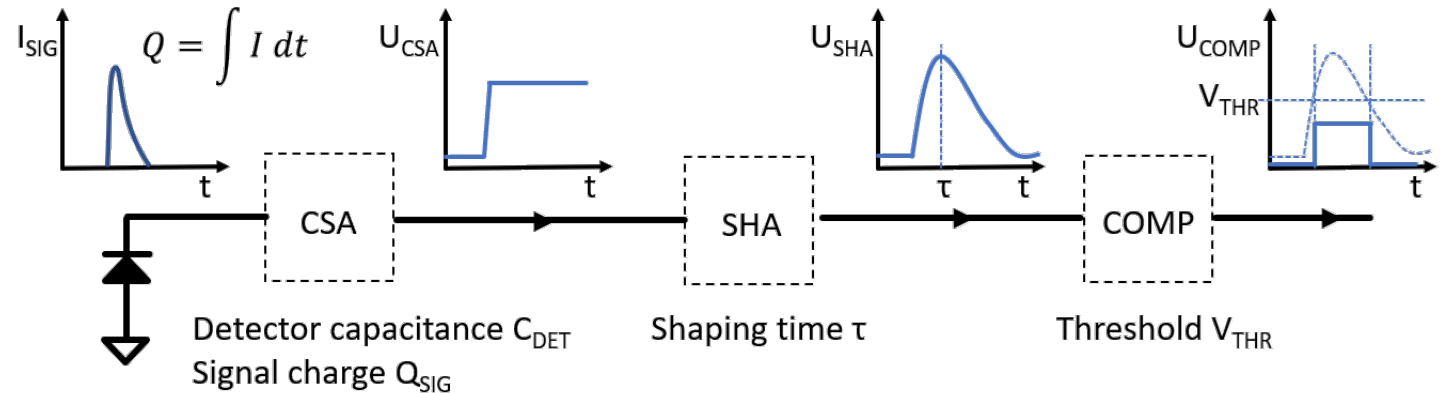
The Analog Front-end Module

Lab Exercise Analog Front-end (AFE)

- Signal chain with charge sensitive amplifier, pulse shaper, and comparator

- Parameters to control:

- Input charge Q_{SIG}
- Detector capacitance C_{DET}
- Shaping time τ
- Threshold V_{THR}



- Parameters to measure and analyze

- Pulse shape
- Threshold
- Noise
- time-other-threshold

Lab Exercise Analog Front-end (AFE)

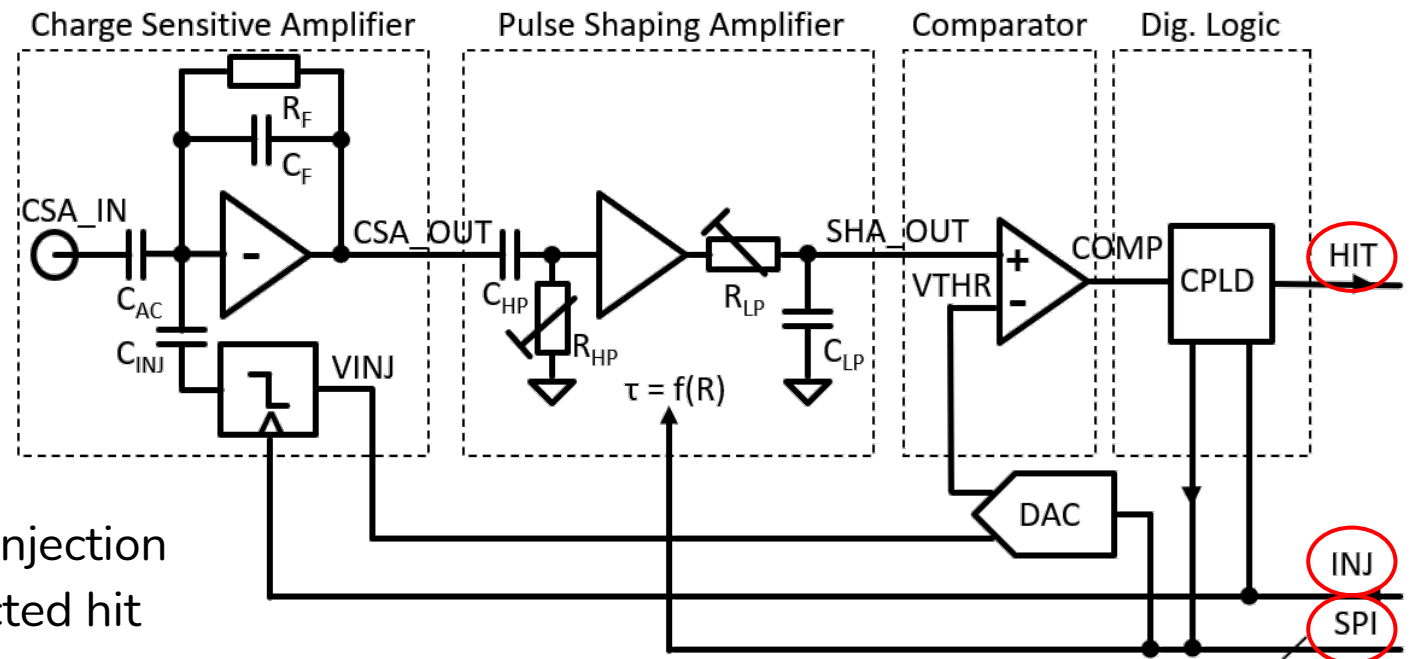
- Signal chain with charge sensitive amplifier, pulse shaper, and comparator

- Parameters to control:

- Input charge Q_{SIG}
- Detector capacitance C_{DET}
- Shaping time τ
- Threshold V_{THR}

- Control interface

- Digital input “INJ”: trigger charge injection
- Digital output “HIT”: signals detected hit
- SPI-bus (serial bus):
 - Charge injection amplitude
 - Comparator threshold
 - Shaping time
 - Time-over-threshold (comparator pulse width)



AFE Pulse Shape Analysis

- Charge injection

$$Q_{INJ} = C_{INJ} \cdot V_{INJ}$$

- CSA output (step amplitude):

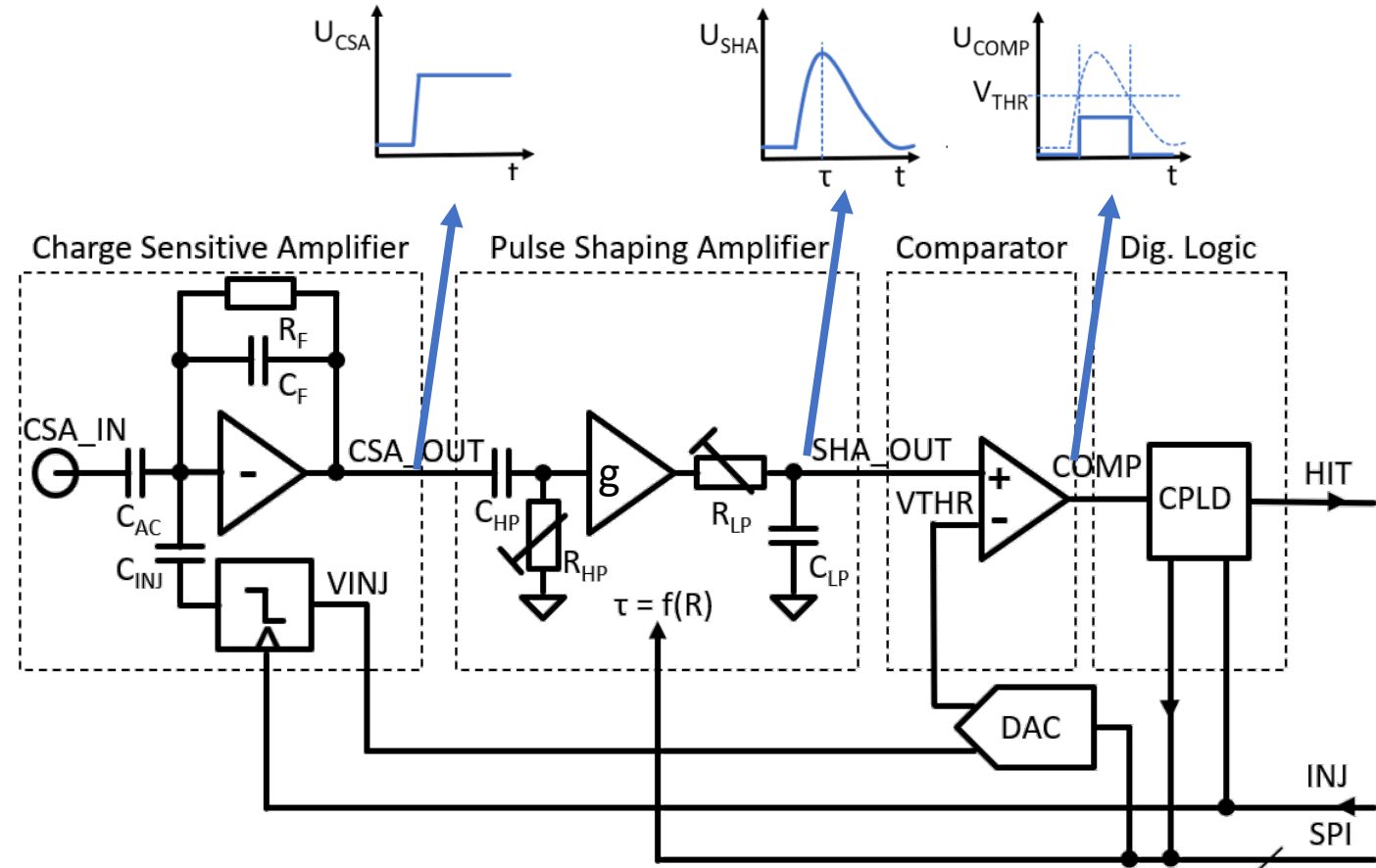
$$V_{CSA} = \frac{Q}{C_F}$$

- SHA output (1st order high-pass and 1st order low-pass filter)

$$V_{SHA}(t) = V_{CSA} \cdot g \cdot \frac{t}{\tau} \cdot e^{-\frac{t}{\tau}}$$

- Charge sensitivity

$$g_q = \frac{V_{CSA}^{peak}}{Q} = g \cdot \frac{1}{C_F} \cdot e^{-1}$$

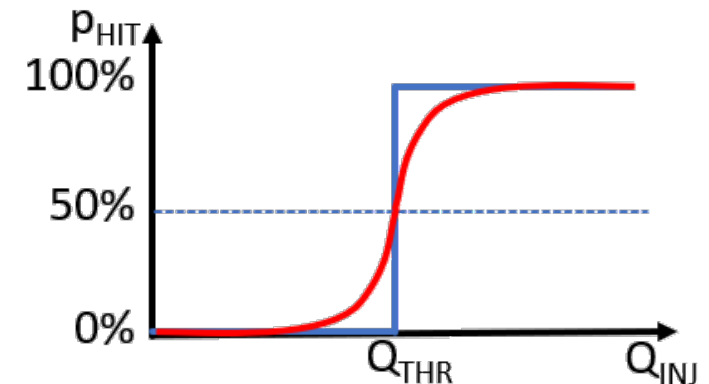
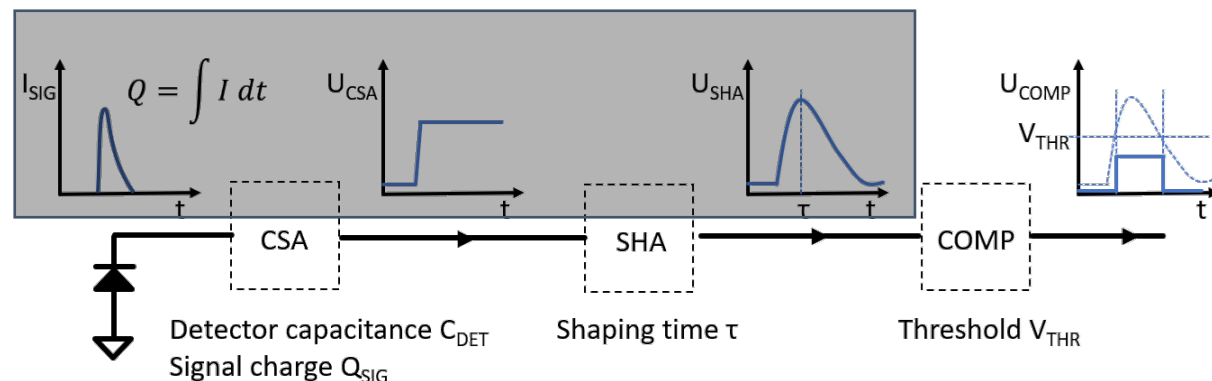


Exercises:

- Inject signal charge and study the pulse shapes (at CSA, SHA, and COMP nodes) as a function of the signal charge, shaping time and threshold
- Measure the shaping time as a function of the programmed filter time constant
- Calculate the charge sensitivity and compare with the expected value.

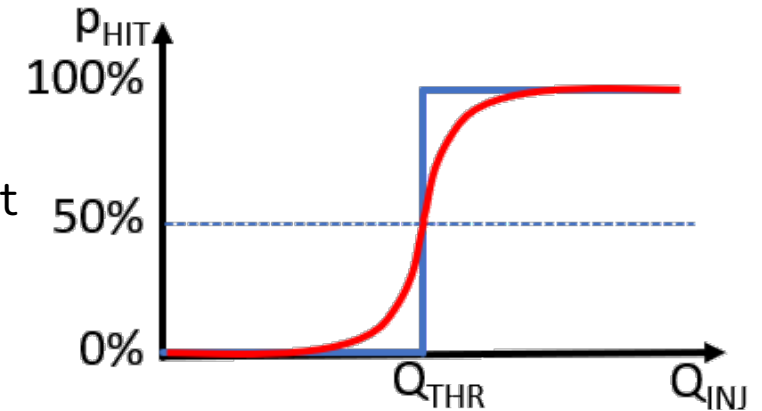
Charge injection scans

- Many real detector systems do not provide analog signal outputs (no access to characterize pulse waveforms)
- Only information of the comparator digital output available
- Threshold/noise measurement with charge injection scans
 - Sweep of the injected charge at const. threshold
 - Multiple injections ($O(100)$) per charge
 - Analyze comparator response probability (S-curves)
- Amplitude measurement by analyzing the comparator pulse width (time-over-threshold)



Charge injection scans

- Sweep of the injected charge at constant threshold
 - Ideal (noise-free) system has a step-function response (blue)
 - The **position of the step** indicates the signal charge equivalent to the set **threshold**
 - In a real system (red) amplitude fluctuations due to the noise smear out the step
 - From the **slope** of the s-curve at the 50% probability point (@threshold) the **noise** can be calculated



- Gaussian-error function is fitted to the data: $f(x) = \frac{1}{2} \left(1 + \operatorname{erf} \left(\frac{x - \mu}{\sqrt{2}\sigma} \right) \right)$, with $\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$.
 - Threshold and noise are represented by μ and σ , respectively.

Exercises:

- Measure s-curves for different threshold settings
- Measure the noise for different shaping times
- ...

AFE Example Code Snippet

Setup GPIO ports

Function for
SPI bus access

AFE configuration
parameters

Loop for 100 injections
and comparator reads

```
import time
import RPi.GPIO as GPIO
import spidev as SPI

spi = SPI.SpiDev()
spi.open(0,0) # (bus, device)

COMP = 5
GPIO.setup(COMP, GPIO.IN)
INJECT = 4
GPIO.setup(INJECT, GPIO.OUT)
GPIO.output(INJECT, GPIO.LOW)
...
...
def update_spi_regs(threshold, injected_signal, time_constant, out_mux):
...

charge = 180
threshold = 2600
time_constant = 5
monitor = 'sha'

update_spi_regs(threshold, charge, time_constant, monitor)
hit_count = 0
for i in range(100):
    GPIO.output(INJECT, GPIO.HIGH) # inject charge
    time.sleep(0.0001)
    if (GPIO.input(COMP)): # read latched comparator output
        hit_count = hit_count + 1
    GPIO.output(INJECT, GPIO.LOW) # reset charge injection and hit latch
    time.sleep(0.0001)

print('Hit probability: ', hit_count/100)

spi.close()
GPIO.cleanup()
```

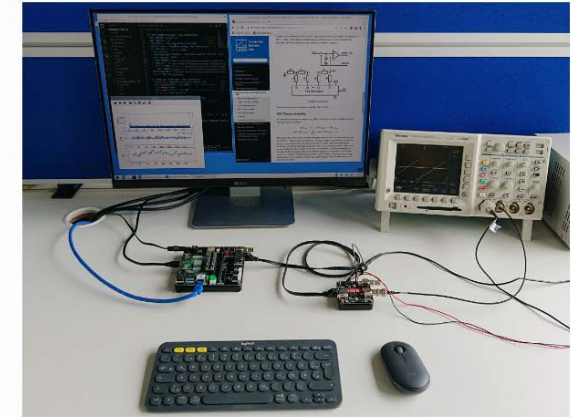
Embedded-System-Lab Online Script

- Documentation and module exercises are available online:
<https://embedded-system-lab.readthedocs.io/en/latest/introduction.html>
- General introduction to the hardware and software
- General Purpose IO interface programming
- Detailed description of each lab experiment (module)
- Lab exercises (including preparatory questions)



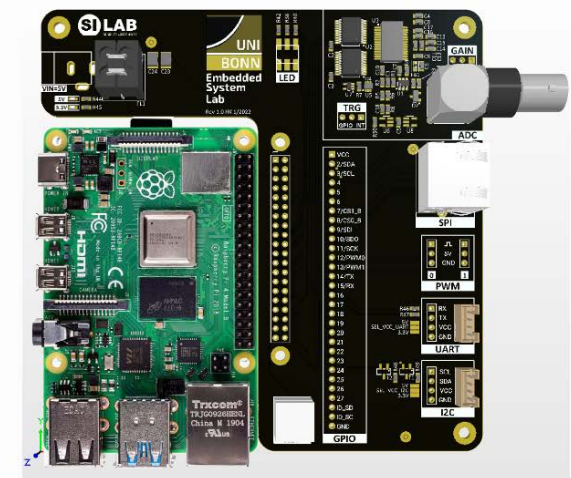
The screenshot shows the website's header with the 'UNI BONN Embedded System Lab' logo and a search bar. Below the header is a 'CONTENTS:' section with a list of topics: Introduction, Analog-to Digital Conversion using Successive Approximation, Device Parameter Extraction with I-V Curves, Analog Signal Processing Chain for Particle Detectors, Capacitance Measurement, and Transmission Line Characterisation with the TDR Method. A dark sidebar on the right lists categories like 'Software Environment', 'Embedded System Hardware', and 'General Purpose Input/Output Interface', followed by a 'GPIO Programming Tutorial' and several specific experiments such as 'Source-Monitoring-Unit and MOSFET Parameter Extraction', 'Successive Approximation ADC', 'Capacitance Measurement', 'Analog Signal Processing for Semiconductor Sensors', 'Time Domain Reflectometry', and 'Fast ADC'.

Introduction



The Embedded System Lab setup

This modular lab course gives an introduction to selected aspects of analog signal processing and data acquisition techniques. An embedded system running user programs written in Python and/or C is used to directly interact with the experiment module's hardware. The embedded system hardware is based on a Raspberry Pi single board computer which is mounted to a custom base board. The base board allows access to various interfaces (UART, I2C, SPI etc.) which are implemented via the general purpose IO ports (GPIO). In addition, the base board features a fast 12-bit ADC, which allows the Raspberry Pi to be used as a simple oscilloscope to sample waveforms for further processing, documentation, and analysis.



Embedded System Lab base board with a Raspberry 4 module

Module Description (Example: Analog Front-End)

- Each module description starts with an introduction to its basic functionality and block level diagram of the electronic circuit.

UNI BONN Embedded System Lab

latest

Search docs

CONTENTS:

- Introduction
- Software Environment
- Embedded System Hardware
- General Purpose Input/Output Interface
- GPIO Programming Tutorial
- Experiment: Source-Monitoring-Unit and MOSFET Parameter Extraction
- Experiment: Successive Approximation ADC
- Experiment: Capacitance Measurement
- Experiment: Analog Signal Processing for Semiconductor Sensors**
- Signal Processing Overview
- Circuit Implementation
- Data Acquisition and Analysis Methods
- Exercises
- Experiment: Time Domain Reflectometry
- CPLD/FPGA Programming

Read the Docs latest

Experiment: Analog Signal Processing for Semiconductor Sensors [Edit on GitHub](#)

Experiment: Analog Signal Processing for Semiconductor Sensors

Analog Front-end Module

The goal of this lab is to understand typical analog signal processing steps used for read-out of semiconductor detector charge signals, plus the associated basic data acquisition and analysis methods. In this module, a single-channel analog front-end (AFE) chain made of discrete hardware components will be used to analyze the functionality of each circuit block. In particular the characterization of the noise performance and its dependence on circuit parameters will be discussed. The electrical connections to the AFE hardware allow injection of calibration charge signals, programming of circuit parameters, and the detection of hits. On the software side, scan routines will be developed to set the circuit parameters of interest and read the AFE digital output response. Basic analysis methods will be introduced to extract performance parameters such as equivalent noise charge (ENC), charge transfer gain, linearity etc. Additionally, the fast ADC can be used to record analog waveforms for further analysis.

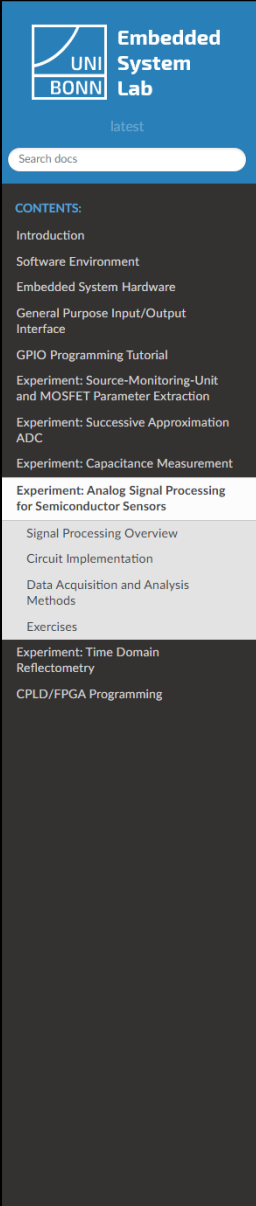
Signal Processing Overview

A typical analog read-out chain - also called analog front-end - for a semiconductor detector consists of a charge sensitive amplifier (CSA), a pulse shaping amplifier (SHA) and digitization circuit, which simplest implementation is a comparator (COMP), as shown in the picture below. The CSA converts the charge signal of a detector diode (or an injection circuit) to a voltage step according to the feedback capacitance C_f . The shaping amplifier (SHA) acts on the CSA output as a signal filter with a band-pass transfer function. By adjusting its band-pass center frequency the signal-to-noise ratio of the signal processing chain can be optimized. The comparator compares the output of the shaped signal with a programmable threshold. When the input signal is above the threshold, the comparator output goes high and flags a signal hit to the digital read-out logic.

Generic read-out chain for a semiconductor detector: charge sensitive amplifier (CSA), pulse shaping amplifier (SHA), and comparator (COMP). Shown are typical signal waveforms between the blocks and the parameters that can be controlled for each block.

Module Description (Example: Analog Front-End)

- More circuit details and relevant formulas are explained.
- Also a link to the full schematic circuit is given (i.e. [AFE_1.1.pdf](#)).



Embedded System Lab

latest

Search docs

CONTENTS:

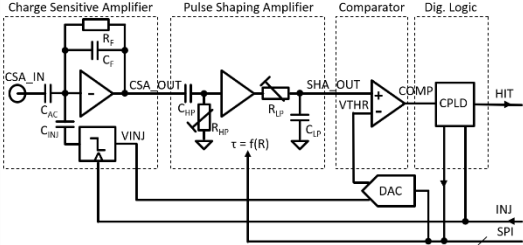
- Introduction
- Software Environment
- Embedded System Hardware
- General Purpose Input/Output Interface
- GPIO Programming Tutorial
- Experiment: Source-Monitoring-Unit and MOSFET Parameter Extraction
- Experiment: Successive Approximation ADC
- Experiment: Capacitance Measurement
- Experiment: Analog Signal Processing for Semiconductor Sensors
- Signal Processing Overview
- Circuit Implementation**
- Data Acquisition and Analysis Methods
- Exercises
- Experiment: Time Domain Reflectometry
- CPLD/FPGA Programming

Circuit Implementation

The simplified schematic in the figure below shows the implementation of the signal processing chain. The CSA is built around a low noise op-amp that is feed-back with a small capacitance C_f and a large resistance R_f . The feedback capacitance C_f defines the charge transfer gain and the resistance R_f allows for a slow discharge of C_f and setting of the dc operation point of the op-amp. The output voltage of the charge sensitive amplifier in response to an input charge Q is a step function with an amplitude given by the expression:

$$V_{CSA} = \frac{Q}{C_f}$$

For calibration and characterization measurements an injection circuit is used to generate programmable charge signals. On the rising edge of the digital INJ signal a negative charge of the size C_{INJ} times the programmable voltage step amplitude $VINJ$ is injected to the CSA input.



Simplified schematic of the analog front-end. INJ and HIT control the charge injection and digital hit readout, respectively. The SPI bus is used to program the DAC voltages V_{THR} and $VINJ$ and select the shaping amplifier time constant. The full AFE schematic is found here: [AFE_1.1.pdf](#)

The shaping amplifier consists of a first-order high pass filter (HPF) and a first-order low pass filter (LPF). Therefore such a filter is also called CR-RC shaper. The high- and low-pass filter are isolated by a voltage amplifier that adds additional signal gain to the circuit. A total gain of $g = 1000$ is achieved by using three gain stages with $g' = 10$ each. They are located at the CSA output, between the high-pass filter and the low-pass filter (signal HPF) and at the output of the shaper (SHA), respectively. The time constants of the high- and low-pass filter are controlled by selecting the resistor values for R_{HP} and R_{LP} . The control circuit sets the values such $\tau_{SHA} = \tau_{HP} = \tau_{LP}$, i.e. the time constants for low pass filter and high pass filter are equal ($C_{HP} = C_{LP} = const.$). It can be shown that in this case the pulse shape in response to an input step function with the amplitude V_{CSA} is (for $t \geq 0$)

$$V_{SHA}(t) = V_{CSA} \cdot g \cdot \frac{t}{\tau_{SHA}} \cdot e^{-\frac{t}{\tau_{SHA}}}$$

with the peak amplitude:

$$V_{SHA}^{peak} = V_{SHA}(t = \tau_{SHA}) = V_{CSA} \cdot g \cdot e^{-1} = \frac{Q}{C_f} \cdot g \cdot e^{-1}$$

where $V_{CSA} = \frac{Q}{C_f}$. The charge sensitivity of the whole signal chain can be expressed as

$$g_q = \frac{V_{SHA}^{peak}}{Q} = \frac{1}{C_f} \cdot g \cdot e^{-1}$$

and is typically given in units of $[mV/fC]$ or $[mV/electrons]$.

Module Description (Example: Analog Front-End)

- Analysis methods are explained.

Embedded System Lab
UNI BONN
latest

Search docs

CONTENTS:

- Introduction
- Software Environment
- Embedded System Hardware
- General Purpose Input/Output Interface
- GPIO Programming Tutorial
- Experiment: Source-Monitoring-Unit and MOSFET Parameter Extraction
- Experiment: Successive Approximation ADC
- Experiment: Capacitance Measurement
- Experiment: Analog Signal Processing for Semiconductor Sensors**
- Signal Processing Overview
- Circuit Implementation
- Data Acquisition and Analysis Methods
- Exercises
- Experiment: Time Domain Reflectometry
- CPLD/FPGA Programming

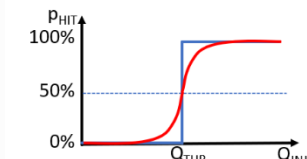
Data Acquisition and Analysis Methods

An important performance metric of a signal processing circuit is its signal-to-noise ratio (SNR), which is directly related to the efficiency and accuracy of the detection process. A noiseless system would generate a comparator hit signal with 100 % probability if the signal is above threshold and always detect no hit if the signal is below threshold. In the presence of noise, however, the step-like response function of the comparator hit probability as a function of the difference between signal and threshold is smeared out. The following figure shows the comparator response probability of a real system and an ideal system. When the injected charge is equal to the comparator threshold $Q_{INJ} = Q_{THR}$, the hit probability is 50% in both cases. In a noiseless system the hit probability immediately goes to 0 % (100 %) for lower (higher) charge. The noise smooths out this transition region. Actually the knowledge of the slope at the 50 % probability mark allows the calculation of the noise i.e. the noise is proportional to the inverse slope. Mathematically, the response curve is given by a Gaussian error-function (also known as "s-curve"). It is the convolution of a step-function (the ideal comparator response) with a Gaussian probability distribution (representing the noise).

The normalized error function describes the response probability of the comparator as a function of the signal charge in the presence of noise. The mathematical expression is given by the following equation:

$$f(x) = \frac{1}{2} \left(1 + \operatorname{erf} \left(\frac{x - \mu}{\sqrt{2}\sigma} \right) \right),$$

where μ is the mean value and σ is the standard deviation of the Gaussian distribution and erf is the error function:

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$


Response probability of the comparator as a function of the signal charge. The ideal system (noiseless, blue curve) exhibits a step function, while noise (red curve) will smear-out the transition. That results in a Gaussian error-function, which fitted parameters define the threshold (50 % transition point) and the noise (slope of the curve) of the system.

The measurement of an s-curve is based on a nested loop of injection/read-out cycles. The following steps need to be implemented in a scan routine (also called **threshold scan**):

1. Set threshold and shaping time constant to the desired values.
2. Outer loop: Define a range of injection voltage values (i.e. injection DAC values) to scan. The injection range must cover the chosen threshold, i.e. the transition from zero hits to 100 % hits must occur within the scan range.
3. Inner loop: For each charge value repeat the injection and read-out cycle (see above) a number of times (typical 100) and count the number of detected comparator signals in relation to the total number of injections.
4. Finally plot the hit probability data as a function of the injection voltage.

The dataset for the injection voltage scan will represent an s-curve that allows the extraction of the threshold and the noise. For a quantitative evaluation of the s-curve the injection voltage (i.e. DAC setting) has to be converted to the equivalent injection charge Q_{INJ} .

Module Description (Example: Analog Front-End)

- Finally, the exercise tasks are explained.
- There is always an Exercise 0 with preparatory questions. This exercise should be worked on before coming to the lab .
- The Exercise 0 questions will be discussed at the beginning of the lab.
- Each experiment comes with a basic Python script to simplify the start of your code development (“afe.py”).
- A detailed solution (“afe_solution.py) is also available. Look at it to get hints if you are stuck with your own code.

Note: There are more exercises than potentially could be worked on as part of this school. The tutors will guide you.

The screenshot shows the 'Embedded System Lab' website. The top navigation bar includes the UNI BONN logo and a search bar. A table of contents is visible on the left, with 'Exercises' highlighted. The main content area is titled 'Exercises' and contains a list of 10 tasks. The first task is 'Exercise 0. Preparatory questions'.

Embedded System Lab

latest

Search docs

CONTENTS:

- Introduction
- Software Environment
- Embedded System Hardware
- General Purpose Input/Output Interface
- GPIO Programming Tutorial
- Experiment: Source-Monitoring-Unit and MOSFET Parameter Extraction
- Experiment: Successive Approximation ADC
- Experiment: Capacitance Measurement

Experiment: Analog Signal Processing for Semiconductor Sensors

- Signal Processing Overview
- Circuit Implementation
- Data Acquisition and Analysis Methods
- Exercises

Experiment: Time Domain Reflectometry

CPLD/FPGA Programming

Exercises

The exercises are grouped into three parts. In the first part the basic functionality of the analog front-end is tested. This is accomplished by implementing a script to enable the charge injection and to observe waveforms of the charge sensitive amplifier, shaper, and comparator with an external oscilloscope and/or the fast ADC on the Raspberry Pi base board. In the second part methods to extract analog performance parameters from the digital hit information will be developed. Finally, the full analog signal processing chain will be characterized as a function of shaping time and detector capacitance.

The exercise 0 contains preparatory questions that should be answered before coming to the lab.

Exercise 0. Preparatory questions

1. The injection circuit generates a charge signal of the size $C_{inj} \cdot V_{inj}$. What is the charge in femto Coulomb generated by a voltage step of 100 mV with $C_{inj} = 0.1 \text{ pF}$? What is the charge step size for $V_{inj} = 0.05 \text{ mV}$, which corresponds to the effective LSB size of the injection voltage DAC? Also calculate these numbers in units of the elementary charge (electrons).
2. An ideal charge sensitive amplifier generates a step-like output waveform in response to an instantaneous charge signal at the input. What is the CSA output step amplitude for an input charge of 1 fC given the feedback capacitance of 1 pF? The charge sensitivity is defined as the output amplitude per input charge. What is its unit?
3. A shaping amplifier responds with a characteristic output pulse to a step-like input waveform. What is the peak pulse amplitude for an input step with a unit amplitude (i.e. 1 V)? Assume a CR-RC (high-pass + low-pass filter) with equal time constants.
4. What is the ideal charge sensitivity of the experiments analog front-end chain (CSA + SHA) i.e., peak amplitude in mV at the shaper output per fC (or electron) charge at the CSA input (Note: Use the effective feedback capacitance value $C_f = 1.39 \text{ pF}$ for your calculation)?
5. The threshold voltage to detect a signal with the comparator is set by a DAC with an LSB size of 0.5 mV. What is the equivalent LSB size in fC or electrons? Note: Use the total charge sensitivity as calculated above.
6. Draw a sketch of an amplitude histogram of an ideal noise-free system. It consist of two delta-like peaks: one for the baseline and one for the signal amplitude produced by a constant input charge. In a real system, however, noise is overlaying the ideal signals, leading to fluctuations of the baseline and signal amplitudes. Modify the amplitude histogram to reflect these fluctuations (assume a Gaussian distribution of the noise).
7. The threshold of the comparator should be set in a way, that the noise is suppressed and only the signals are detected. Draw an optimum threshold in your amplitude histogram. What would happen if the threshold was too low, what would happen if it was too high? How could the terms purity and efficiency of the detection process be defined in this context? What happens if baseline and signal fluctuations are getting too close to each other?
8. The term 'equivalent-noise-charge' (ENC) represents the number of electrons at the input of an ideal (noise-free) charge sensitive signal chain that would produce the same amplitude at the output as the noise alone would in a real system. What is the ENC value for a noise amplitude of 10 mV given the charge sensitivity calculated above?
9. How are the Gaussian distribution and the error-function related? How can one extract the width (sigma) and the mean (lambda) of the underlying Gaussian distribution from a measured error function? How is the noise calculated from the slope of the error function at the 50 % point?
10. Advanced tasks: Calculate and plot the time-over-threshold as a function of the ratio of CR-RC shaper peak amplitude and threshold voltage. You can do that either by inverting the mathematical expression for the shaper pulse waveform (-> Lambert W function) or by implementing a function representing the shaper pulse waveform in Python and numerically evaluating TOT width for a range of amplitude values at a fixed threshold. Note: this function will be useful to fit measured pulse waveforms (see the later exercises). What is the relation between the TOT and the injected charge? What is the effect of the shaping time constant on the TOT? Assume the TOT counter has a resolution of 25 ns and a maximum count of 255.

Software & File System

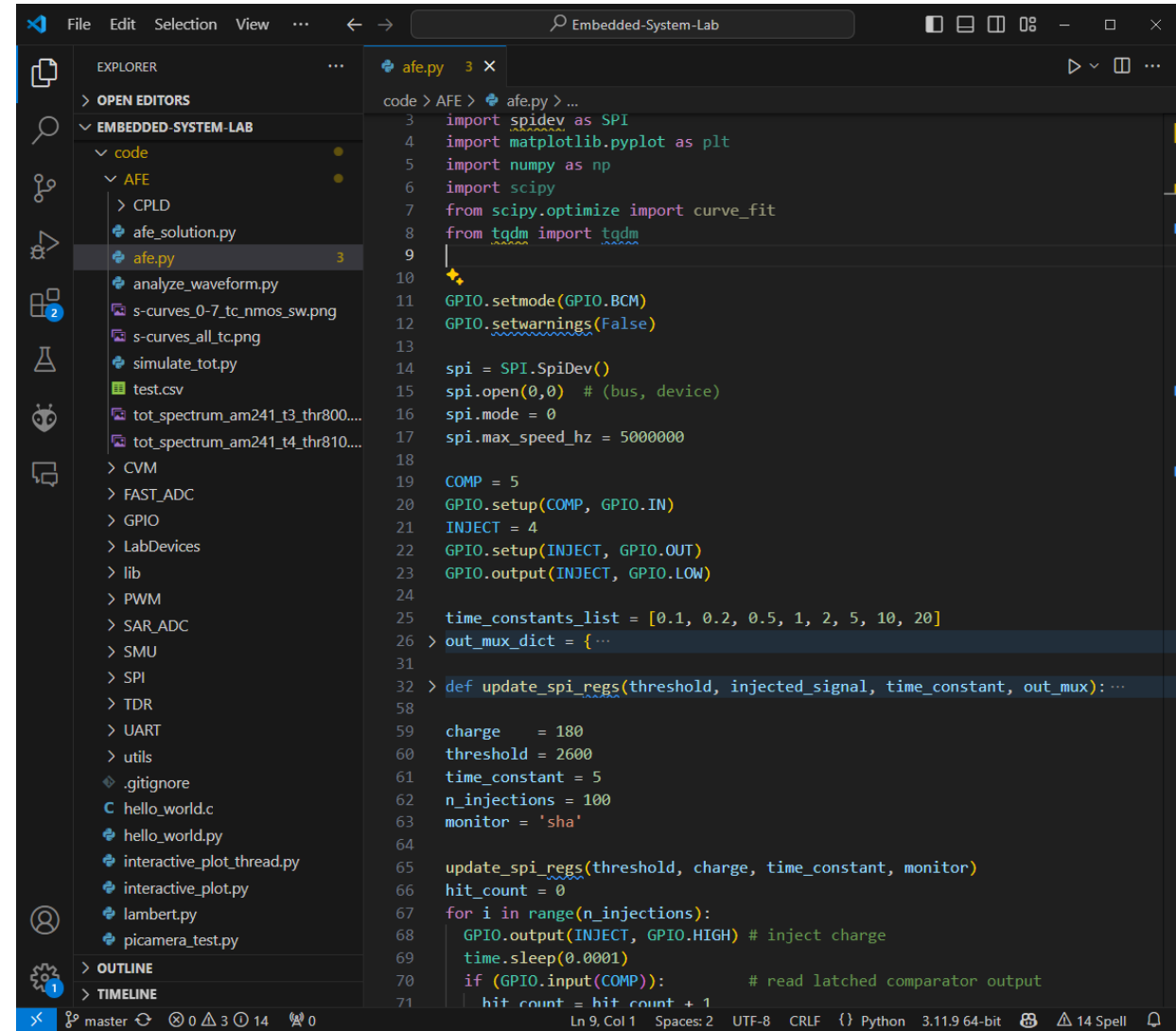
- Raspberry OS (32-bit), user name “pi”, auto-login
- Home directory structure

```
|__home  
| |__ Embedded-System-Lab (GitHub root directory)  
| | |__ code (code examples in sub-folders for each experiment)  
| | |__ docs (sources for this documentation)  
| | |__ hardware (documentation: schematics, datasheets)  
| |__ work (user working directory, not synchronized to GitHub)
```

- **Copy example code from the *code* folder to your work folder and modify it only there**

Software & File System

- Visual Code IDE integrated Python interpreter and C-compiler
- Use “git checkout --force origin/master” in the **Embedded-System-Lab** folder to get a fresh copy from git if needed
- Start with the “Hello World” examples from the **code** folder to check your Python and C environment



The screenshot shows the Visual Studio Code IDE interface. The Explorer panel on the left displays a file tree for the 'EMBEDDED-SYSTEM-LAB' project, with the 'code' folder expanded to show 'afe.py'. The main editor window displays the contents of 'afe.py', which is a Python script for SPI communication. The script includes imports for SPI, matplotlib, numpy, and scipy, and defines a function 'update_spi_regs' that configures GPIO pins and performs SPI transactions.

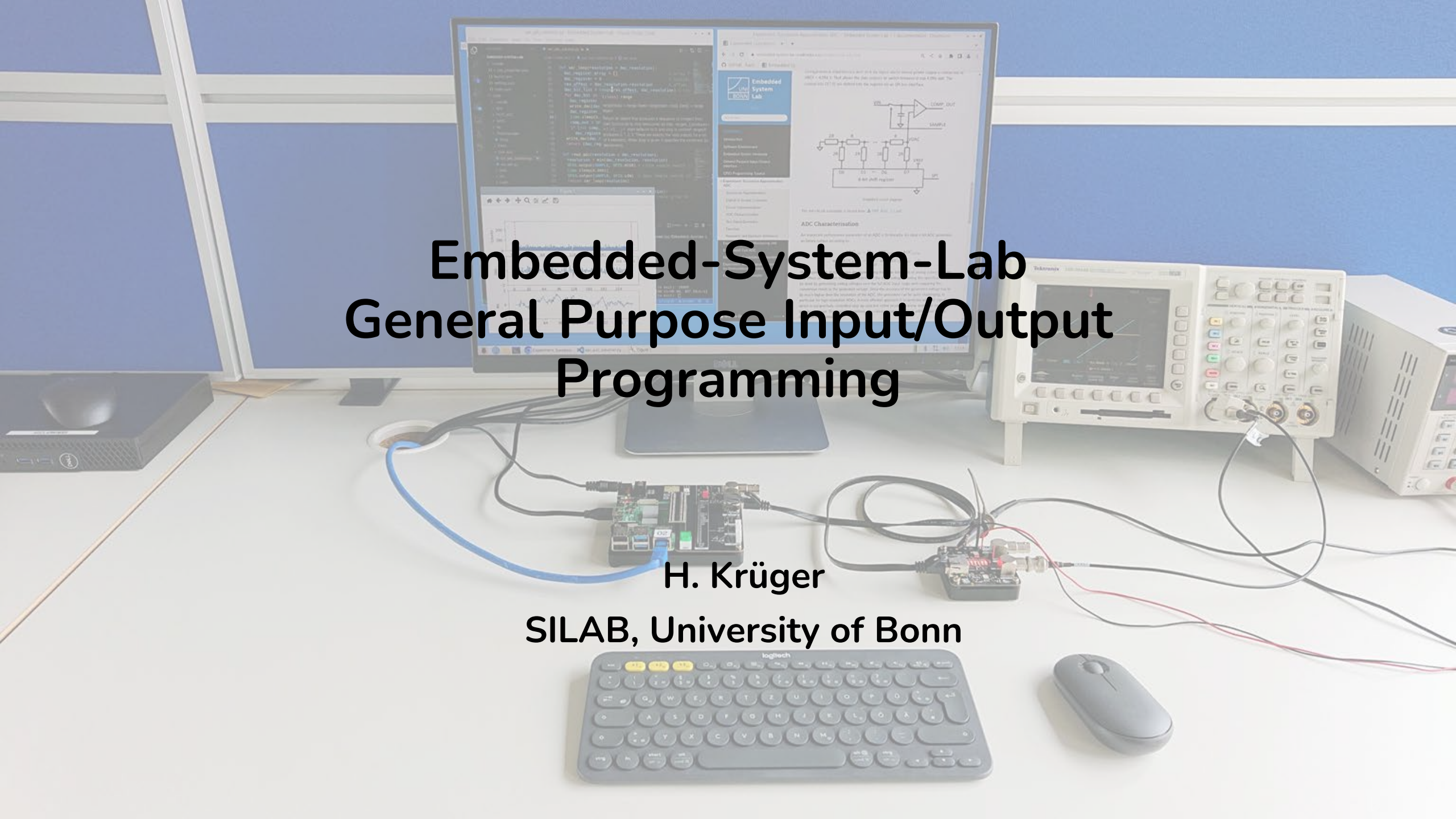
```
code > AFE > afe.py > ...
3 import spidev as SPI
4 import matplotlib.pyplot as plt
5 import numpy as np
6 import scipy
7 from scipy.optimize import curve_fit
8 from tqdm import tqdm
9
10
11 GPIO.setmode(GPIO.BCM)
12 GPIO.setwarnings(False)
13
14 spi = SPI.SpiDev()
15 spi.open(0,0) # (bus, device)
16 spi.mode = 0
17 spi.max_speed_hz = 5000000
18
19 COMP = 5
20 GPIO.setup(COMP, GPIO.IN)
21 INJECT = 4
22 GPIO.setup(INJECT, GPIO.OUT)
23 GPIO.output(INJECT, GPIO.LOW)
24
25 time_constants_list = [0.1, 0.2, 0.5, 1, 2, 5, 10, 20]
26 > out_mux_dict = { ...
31
32 > def update_spi_regs(threshold, injected_signal, time_constant, out_mux): ...
58
59 charge = 180
60 threshold = 2600
61 time_constant = 5
62 n_injections = 100
63 monitor = 'sha'
64
65 update_spi_regs(threshold, charge, time_constant, monitor)
66 hit_count = 0
67 for i in range(n_injections):
68     GPIO.output(INJECT, GPIO.HIGH) # inject charge
69     time.sleep(0.0001)
70     if (GPIO.input(COMP)): # read latched comparator output
71         hit_count = hit_count + 1
```

Lab Course Details

- We have 6 setups for two students each (total 12 students)
- Location: FTD, Room 3.013, 3rd floor
- Time: Tuesday and Wednesday from 16h to 18h
- Call 69464 from the entrance (main or back) in case the FTD doors are closed

- Preparation (Important!)
 - Go to <https://embedded-system-lab.readthedocs.io/en/latest/index.html> and read the documentation
 - Introduction
 - Software Environment
 - Embedded System Hardware
 - Read the AFE description at <https://embedded-syste-lab.readthedocs.io/en/latest/afe.html> and work on the preparatory questions found in Exercise 0.

- Source code: <https://github.com/silab-bonn/Embedded-System-Lab>

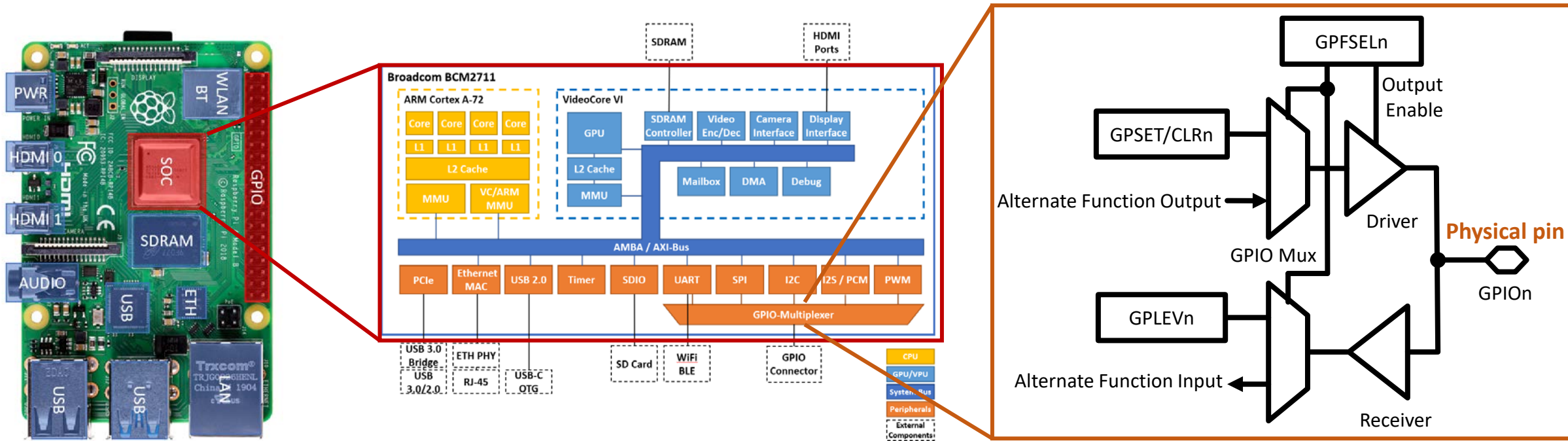


Embedded-System-Lab
General Purpose Input/Output
Programming

H. Krüger

SILAB, University of Bonn

General Purpose Input/Output Ports - GPIO



RPi 4B board

- Single board computer
- CPU, file storage
- Audio & Video
- Network

SOC block diagram

- ARM based CPU
- VideoCore
- Peripheral I/O blocks

GPIO block diagram

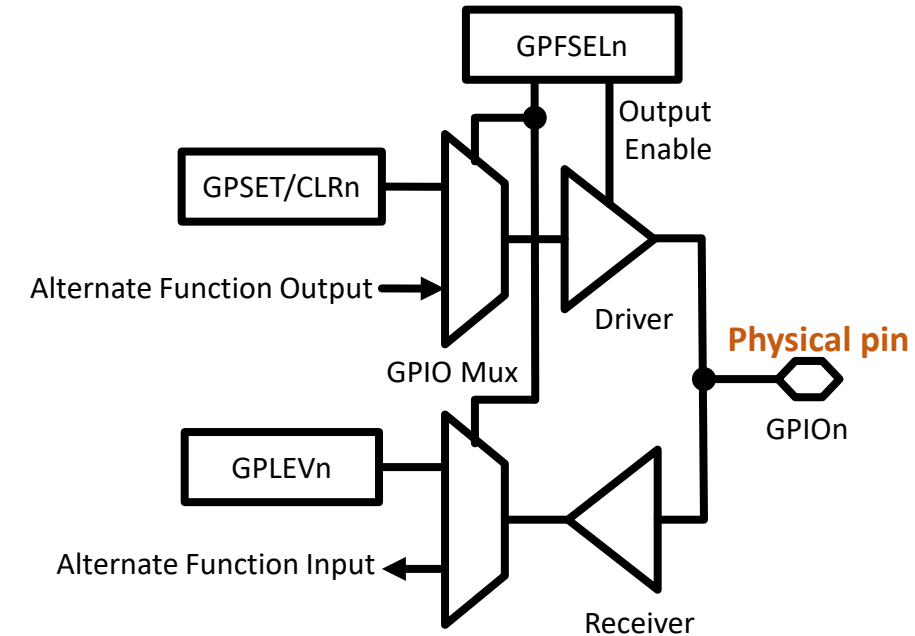
- Basic input/output function
- Alternate functions (UART, I2C, SPI...)

Programming the GPIO pins

- *GPFSEL* register control the GPIO configuration

GPIO Function Modes		
FSELn	Function	
000	Input	default
001	Output	
100	Alternate function 0	
101	Alternate function 1	
110	Alternate function 2	
111	Alternate function 3	
011	Alternate function 4	
010	Alternate function 5	

GPIO Function Select Register (<i>GPFSEL0 @ 0x7E200000</i>)					
Bit	Field Name	Description	Type	Default	
31-30	—	Reserved	R	0	
29-27	FSEL9	Function Select GPIO9	R/W	0	
26-24	FSEL8	Function Select GPIO8	R/W	0	
23-21	FSEL7	Function Select GPIO7	R/W	0	
20-18	FSEL6	Function Select GPIO6	R/W	0	
17-15	FSEL5	Function Select GPIO5	R/W	0	
14-12	FSEL4	Function Select GPIO4	R/W	0	
11-9	FSEL3	Function Select GPIO3	R/W	0	
8-6	FSEL2	Function Select GPIO2	R/W	0	
5-3	FSEL1	Function Select GPIO1	R/W	0	
2-0	FSEL0	Function Select GPIO0	R/W	0	



- *GPSET/CLR* register control the output state, *GPLEV* the input state

GPIO Pin Output Set Registers (<i>GPSET0 @ 0x7E20001C</i>)				
Bit	Field Name	Description	Type	Default
31-0	SETn	1 = set pin to logic 1	R/W	0

Set pin to high (3.3V)

GPIO Pin Output Clear Registers (<i>GPCLR0 @ 0x7E200028</i>)				
Bit	Field Name	Description	Type	Default
31-0	CLRn	1 = set pin to logic 0	R/W	0

Set pin to low (0 V)

GPIO Pin Input Level Registers (<i>GPLEV0 @ 0x7E200034</i>)				
Bit	Field Name	Description	Type	Default
31-0	LEVn	0 = pin n is low, 1 = pin n is high	R/W	0

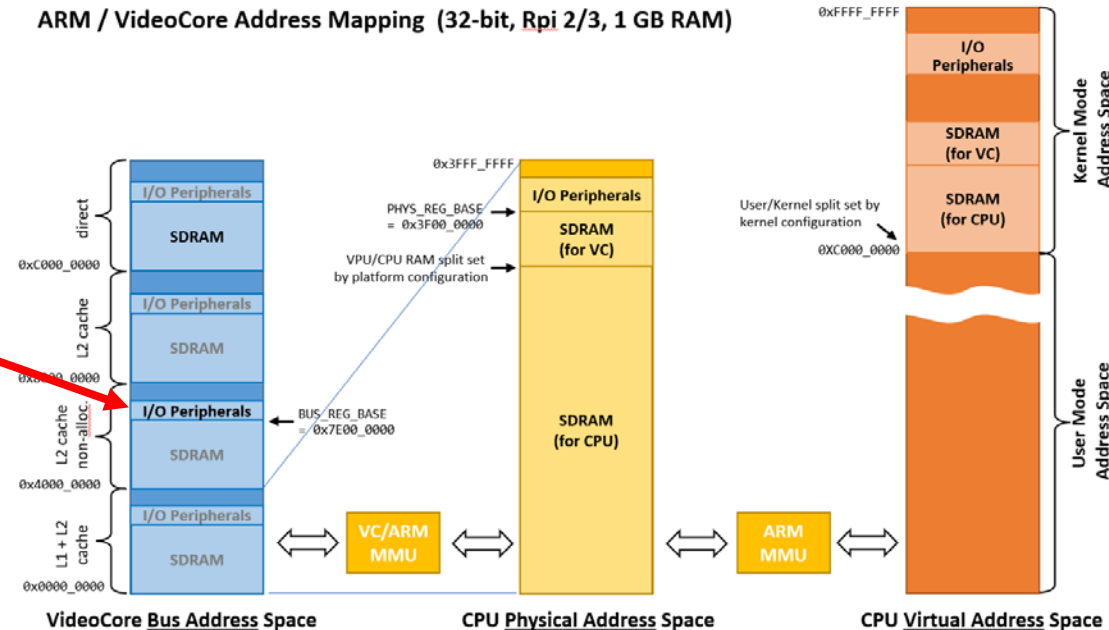
How to access the GPIO registers

- Every register has a unique address (i.e. 0x7E200000)
- An access to the register reads or writes 32 bit
- I/O peripherals registers cannot be directly accessed from **user space** (security)
- Some **mapping** needs to be done between user (**virtual**) address space, **bus** and **physical** address space

GPIO Function Select Register (GPFSEL0 @ 0x7E200000)				
Bit	Field Name	Description	Type	Default
31-30	—	Reserved	R	0
29-27	FSEL9	Function Select GPIO9	R/W	0
26-24	FSEL8	Function Select GPIO8	R/W	0
23-21	FSEL7	Function Select GPIO7	R/W	0
20-18	FSEL6	Function Select GPIO6	R/W	0
17-15	FSEL5	Function Select GPIO5	R/W	0
14-12	FSEL4	Function Select GPIO4	R/W	0
11-9	FSEL3	Function Select GPIO3	R/W	0
8-6	FSEL2	Function Select GPIO2	R/W	0
5-3	FSEL1	Function Select GPIO1	R/W	0
2-0	FSEL0	Function Select GPIO0	R/W	0

This is the address space where the configuration register are accessible

ARM / VideoCore Address Mapping (32-bit, Rpi 2/3, 1 GB RAM)



This is the address space for your code

GPIO Access Example Code (C)

```
#define BUS_REG_BASE    0x7E000000 // start address of the I/O peripheral register space on the VideoCore bus
#define PHYS_REG_BASE  0xFE000000 // start address of the I/O peripheral register space seen from the CPU bus
#define GPIO_BASE      0x7E200000 // start address of the GPIO register space on the VideoCore bus

// calculate the GPIO register physical address from the bus address
uint32_t gpio_phys_addr = GPIO_BASE - BUS_REG_BASE + PHYS_REG_BASE;

// get a handle to the physical memory space
if ((int file_descriptor = open("/dev/mem", O_RDWR|O_SYNC|O_CLOEXEC)) < 0)

// allocate virtual memory (one page size) and map the physical address to a pointer
void *gpio_virt_addr_ptr = mmap(0, 0x1000, PROT_WRITE|PROT_READ, MAP_SHARED, file_descriptor, gpio_phys_addr);

// define memory pointer to access the specific registers
gpfsel0 = (uint32_t*)((void *)gpio_virt_addr_ptr + GPIO_FSEL0);
gpset0  = (uint32_t*)((void *)gpio_virt_addr_ptr + GPIO_SET0);
gpclr0  = (uint32_t*)((void *)gpio_virt_addr_ptr + GPIO_CLR0);

// main() block: define GPIO27 as output and toggling it once and cleanup
*gpfsel2 = 0x001 << (7 * 3); // output mode: FSEL[3:0] = 0x001, GPIO27 FSEL field starts a bit 7
*gpset0  = 1 << 27;         // set output to '1'
sleep(1);
*gpclr0  = 1 << 27;         // set output to '0'
// clean-up
*gpfsel2 = 0;                // set default mode (all input)
munmap(gpio_virt_addr_ptr, 0x1000); // free allocated memory
```

C-code

- (Almost) as close to direct hardware programming as possible
- Fast and small compiled code

GPIO Access Example Code (Python)

```
# import the library and define the prefix for using its members
import RPi.GPIO as GPIO

# tell the library to use pin numbers according to the GPIO naming
GPIO.setmode(GPIO.BCM)

# define GPIO27 as an output
GPIO.setup(27, GPIO.OUT)

# toggle the output state
GPIO.output(27, GPIO.HIGH)
time.sleep(1)
GPIO.output(27, GPIO.LOW)

# set GPIO configuration back to default
GPIO.cleanup()
```

Python code

- Less lines of (user) code
- Registers access (memory mapping, register addresses etc.) hidden in precompiled libraries (black box again...)
- Module implementation uses similar C-code as shown with the previous example

Alternate GPIO Functions

- Data communication beyond simple pin toggling
- Various serial bus protocols (implemented in HW)

I2C bus

- SDA
- SCL

SPI bus

- MISO
- MOSI
- SCK
- CE

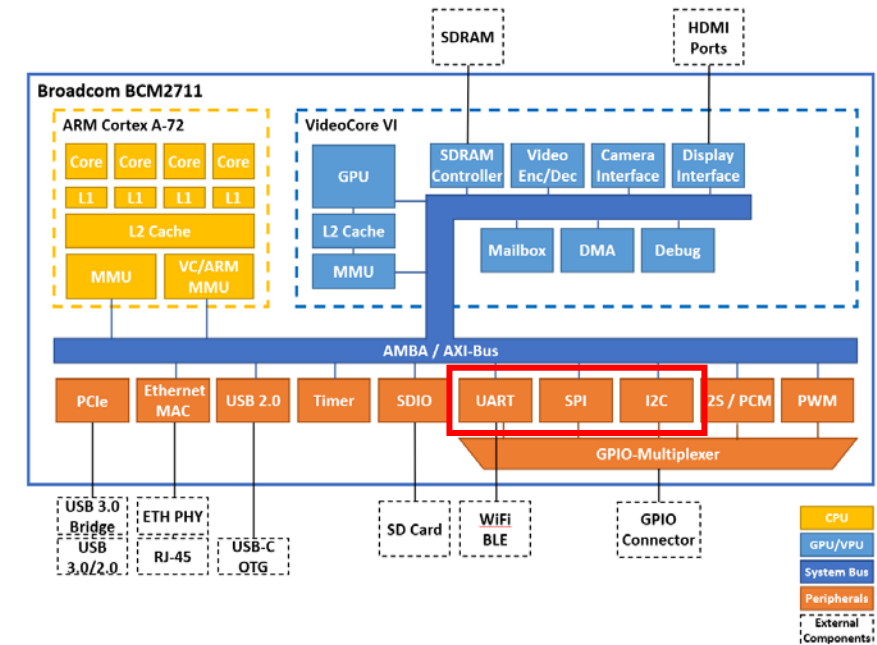
UART

- TX
- RX

	Pull	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5
GPIO0	High	SDA0	SA5	<reserved>			
GPIO1	High	SCL0	SA4	<reserved>			
GPIO2	High	SDA1	SA3	<reserved>			
GPIO3	High	SCL1	SA2	<reserved>			
GPIO4	High	<reserved>	SA1	<reserved>		ARM_TDI	
GPIO5	High	GPCLK1	SA0	<reserved>		ARM_TDO	
GPIO6	High	GPCLK2	SOE_N / SE	<reserved>		ARM_RTCK	
GPIO7	High	SPI0_CE1_N	SWE_N / SW_N	<reserved>			
GPIO8	High	SPI0_CE0_N	SD0	<reserved>			
GPIO9	Low	SPI0_MISO	SD1	<reserved>			
GPIO10	Low	SPI0_MOSI	SD2	<reserved>			
GPIO11	Low	SPI0_SCLK	SD3	<reserved>			
GPIO12	Low	PWM0	SD4	<reserved>		ARM_TMS	
GPIO13	Low	PWM1	SD5	<reserved>		ARM_TCK	
GPIO14	Low	TXD0	SD6	<reserved>		TXD1	
GPIO15	Low	RXD0	SD7	<reserved>		RXD1	
GPIO16	Low	<reserved>	SD8	<reserved>	CTS0	SPI1_CE2_N	CTS1
GPIO17	Low	<reserved>	SD9	<reserved>	RTS0	SPI1_CE1_N	RTS1
GPIO18	Low	PCM_CLK	SD10	<reserved>	BSCSL_SDA / MOSI	SPI1_CE0_N	PWM0
GPIO19	Low	PCM_FS	SD11	<reserved>	BSCSL_SCL / SCK	SPI1_MISO	PWM1
GPIO20	Low	PCM_DIN	SD12	<reserved>	BSCSL_MISO	SPI1_MOSI	GPCLK0
GPIO21	Low	PCM_DOUT	SD13	<reserved>	BSCSL_CE_N	SPI1_SCLK	GPCLK1
GPIO22	Low	<reserved>	SD14	<reserved>	SD1_CLK	ARM_TRST	
GPIO23	Low	<reserved>	SD15	<reserved>	SD1_CMD	ARM_RTCK	
GPIO24	Low	<reserved>	SD16	<reserved>	SD1_DAT0	ARM_TDO	
GPIO25	Low	<reserved>	SD17	<reserved>	SD1_DAT1	ARM_TCK	
GPIO26	Low	<reserved>	<reserved>	<reserved>	SD1_DAT2	ARM_TDI	
GPIO27	Low	<reserved>	<reserved>	<reserved>	SD1_DAT3	ARM_TMS	

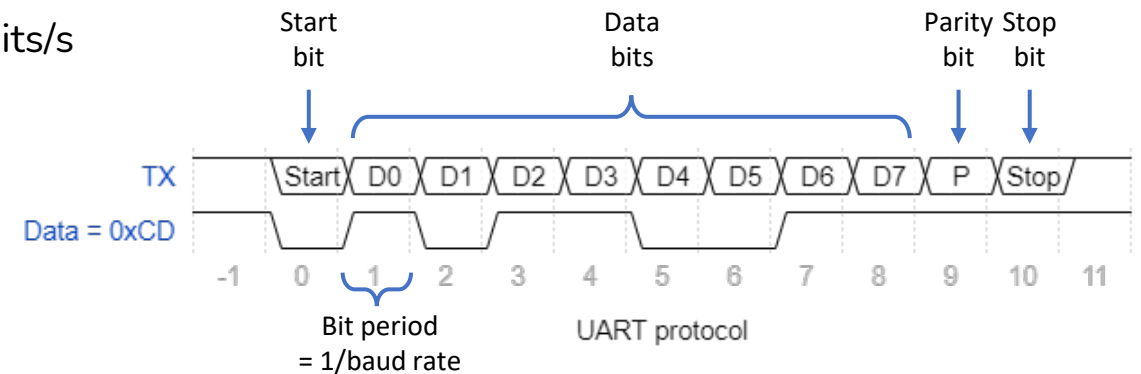
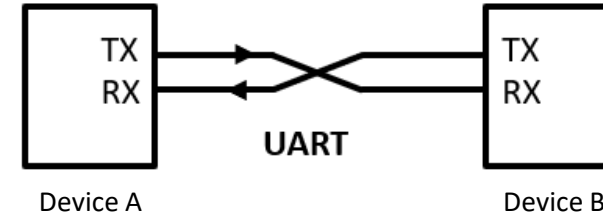
GPIO Function Modes

FSELn	Function
000	Input
001	Output
100	Alternate function 0
101	Alternate function 1
110	Alternate function 2
111	Alternate function 3
011	Alternate function 4
010	Alternate function 5



Universal Asynchronous Receive and Transmit (UART) Bus

- Serial data transfer between two devices
 - TX device A → RX device B
 - RX device A ← TX device B
 - Independent sending and receiving
- Asynchronous communication
 - Both devices need same baud rate setting
 - Typical baud rates: 9600 bit/s up to 115200 bits/s
 - Additional control bits
 - START (always low)
 - STOP (always high, var. lengths)
 - PARITY (optional: ODD or EVEN or NONE)
 - 8-bit data symbols (typical)
- Popular standard used for
 - General lab equipment
 - Microcontroller
 - Debug port
 - Terminal to mainframe communication (1970's)



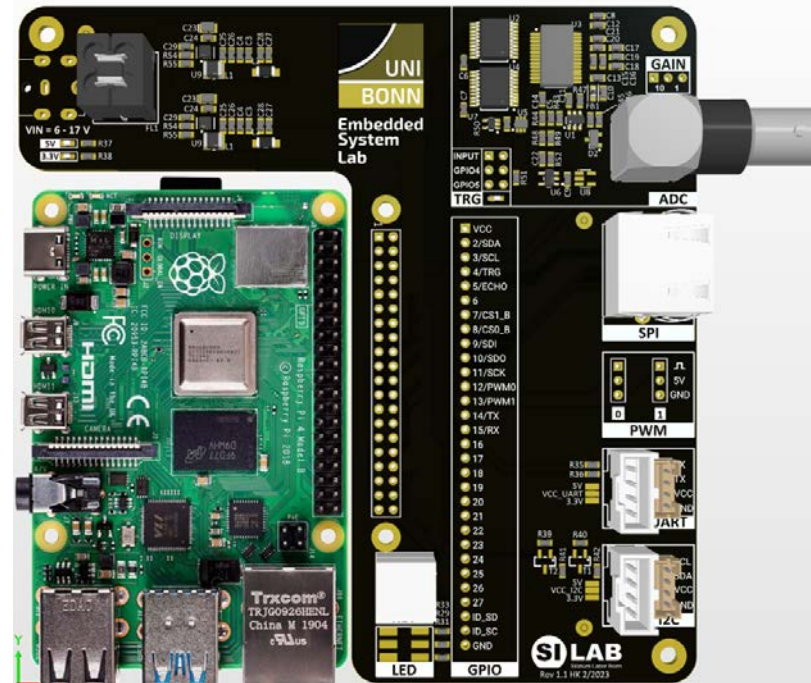
UART Example Code

```
import serial
import time

ser = serial.Serial('/dev/ttyS0', 115200)
ser.reset_input_buffer()

while True:
    if (ser.inWaiting() > 0):
        data_str = ser.read(ser.inWaiting()).decode('ascii')
        print("Received:", data_str)

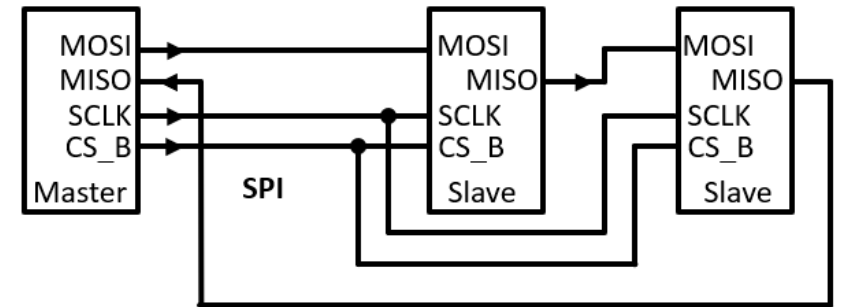
    key = input("Transmit: ").encode()
    ser.write(key)
    ser.flush()
```



Connect TX line to RX line (loopback)
Check TX signal with scope

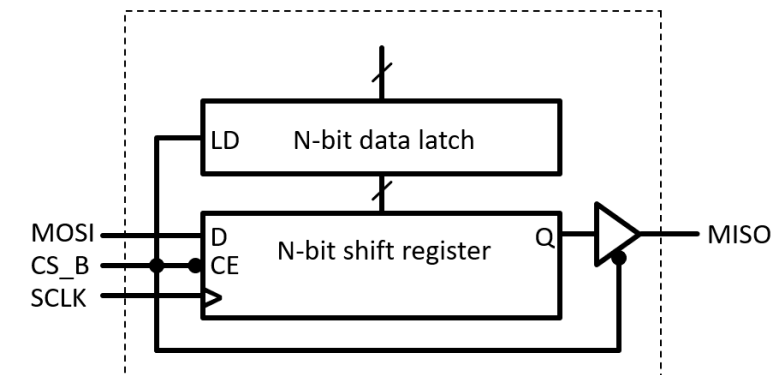
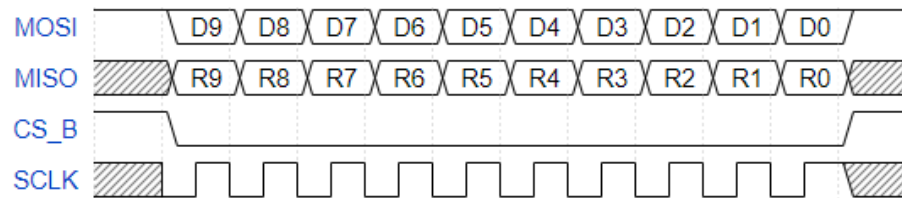
Serial Peripheral Interface (SPI) Bus

- Serial bus which typically uses four wires:
 - MOSI (SDI), data line from master to slave (master out, slave in)
 - MISO (SDO), data line from slave to master (master in, slave out)
 - SCLK, clock line from master to slave(s)
 - CS_B, chip select line (active low, one per slave, or single for daisy-chained slaves)



SPI master with two slaves

- Synchronous operation
 - Data line synchronized to a clock signal



SPI interface details

- Serial shift register
- Data latch
- Output buffer (tri-state)

SPI Bus Access Example Code (Python)

Bit-bang implementation

- Use GPIO pins in input/output mode
- Implement toggling in SW

```
import RPi.GPIO as GPIO # import the library
GPIO.setmode(GPIO.BCM) # use pin numbers according to the GPIO naming

SCK = 11
GPIO.setup(SCK, GPIO.OUT)
GPIO.output(SCK, GPIO.LOW)
SDO = 10
GPIO.setup(SDO, GPIO.OUT)
GPIO.output(SDO, GPIO.LOW)
CS0_B = 8
GPIO.setup(CS0_B, GPIO.OUT)
GPIO.output(CS0_B, GPIO.HIGH)

# start transfer
GPIO.output(CS0_B, GPIO.LOW)
for i in reversed(range(num_bits)):
    GPIO.output(SDO, 0x01 & (data >> i))
    GPIO.output(SCK, GPIO.HIGH)
    GPIO.output(SCK, GPIO.LOW)
# end of transfer
GPIO.output(CS0_B, GPIO.HIGH)

GPIO.cleanup() # set GPIO configuration back to default
```

Alternate GPIO function implementation

- Use GPIO alternate function (spidev library)
- Bit toggling implemented in dedicated peripheral HW block

```
import spidev # SPI module using GPIO alternate function

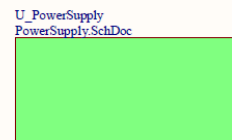
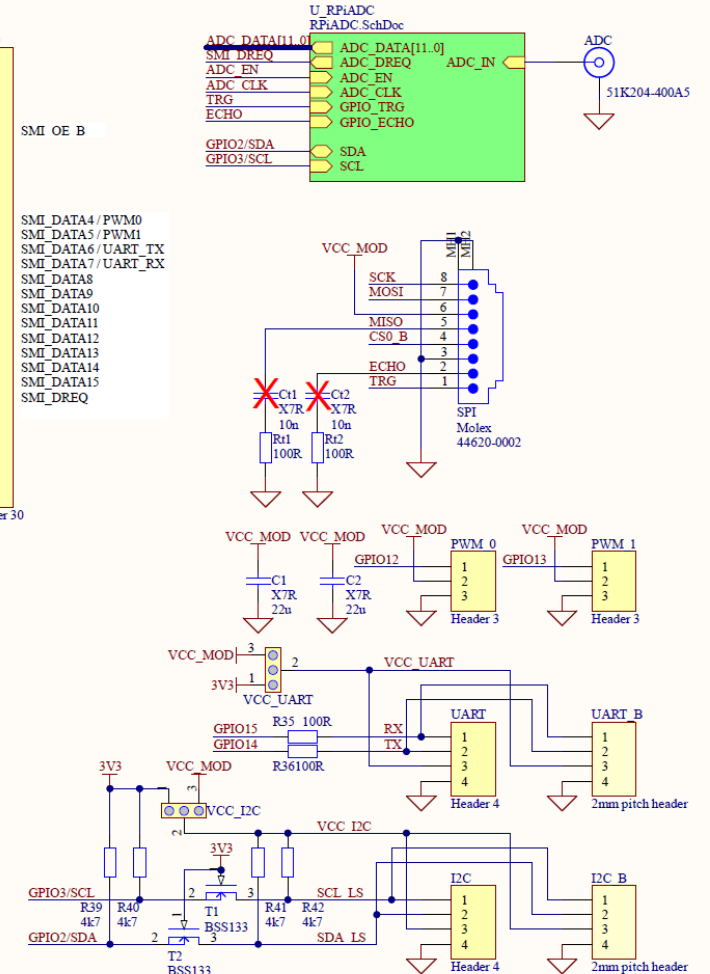
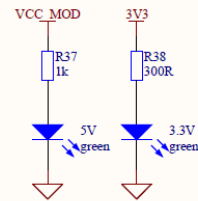
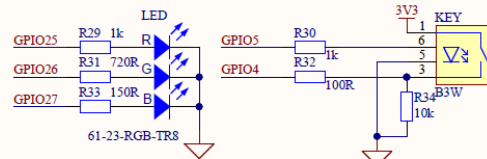
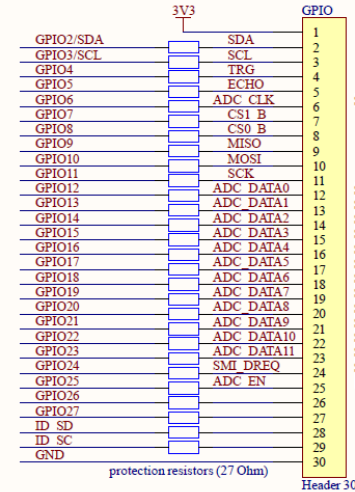
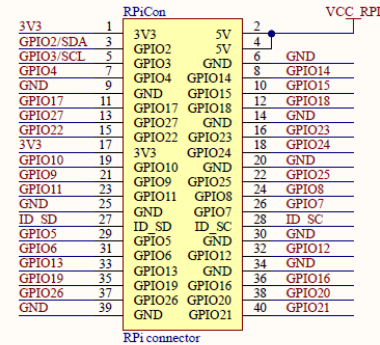
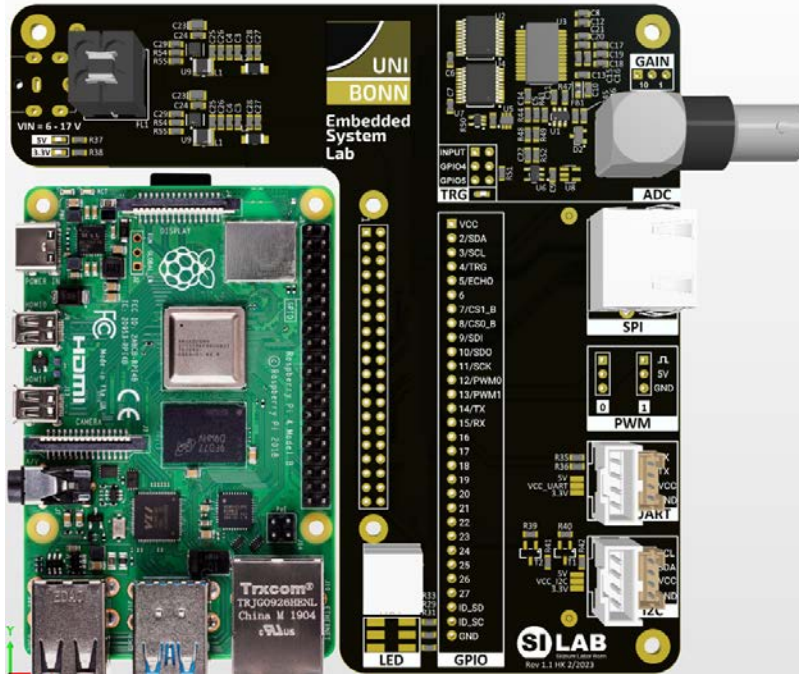
spi = spidev.SpiDev() # initialize Python module
spi.open(0,0) # open device 0 at bus 0

data_array = [0xf0, 0x00, 0x0f]
spi.xfer(data_array)

spi.close() # clean-up
```


Backup

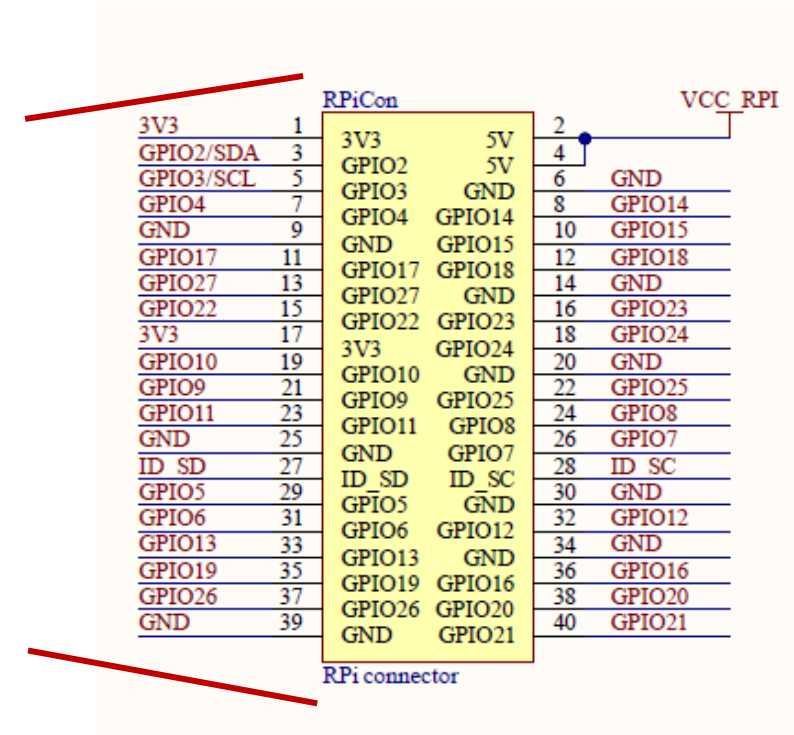
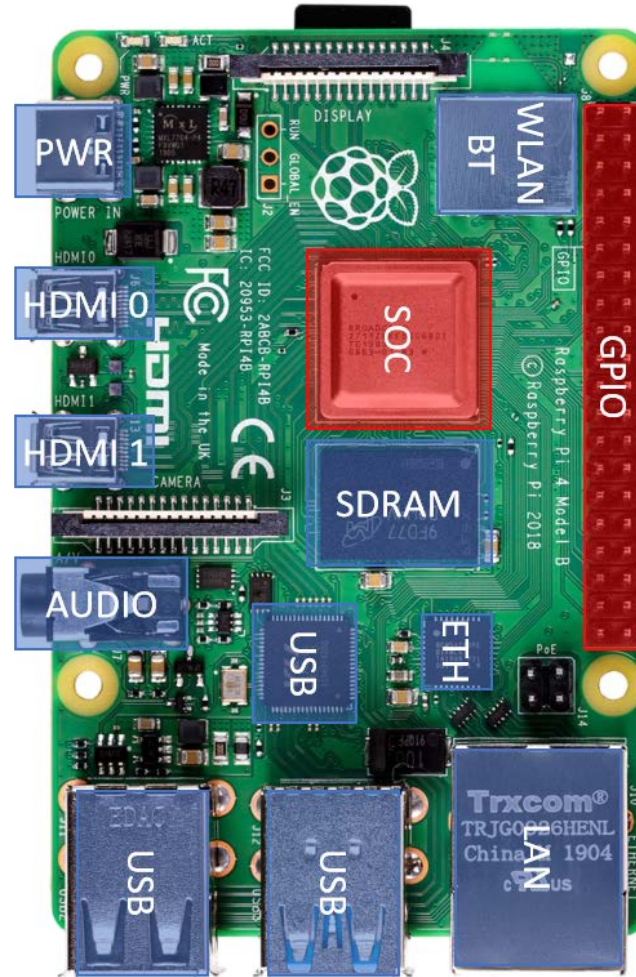
Embedded-System-Lab Base Board Schematic



Title: Raspberry Pi Base Board		SI LAB		Physikalisches Institut
Project: Embedded System Lab		Drawn by: HK	Universität Bonn	
Date: 09/02/2023		Revision: 1.1	Nußallee 12	
Time: 17:01:00		Sheet 1 of 2	53115 Bonn	
File: RPiLabBaseBoard_SchDoc				

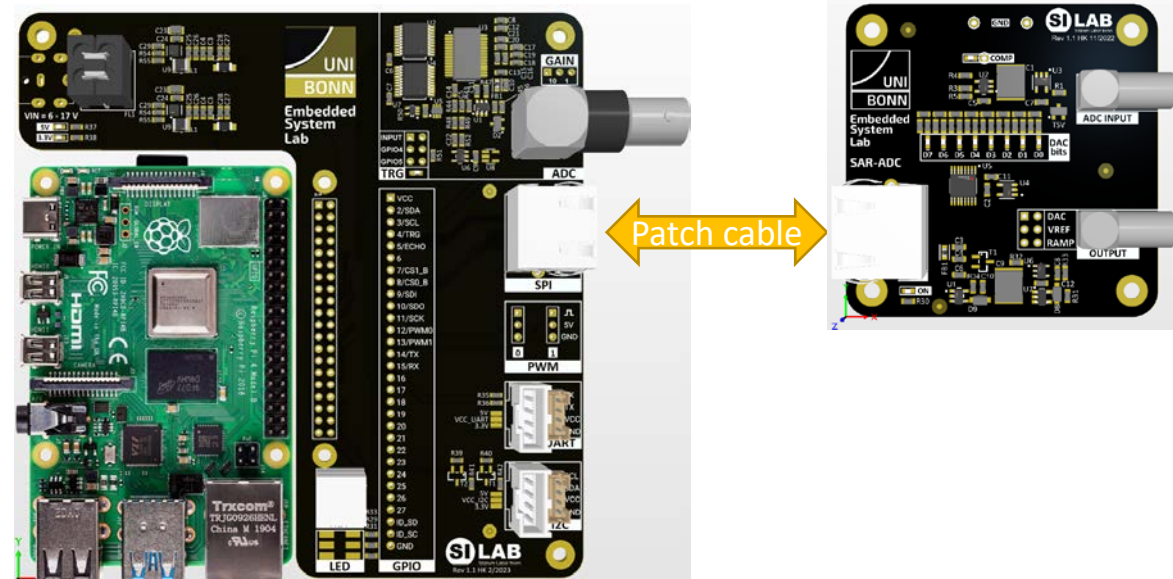
GPIO Connector

- GPIO pins 2-27
- 3.3 V CMOS levels
- 3.3 volt supply pins (outputs)
- 5 V power supply input (VCC_RPI)
- Some GPIO have special (fixed) functions (ID_SA, ID_SC)



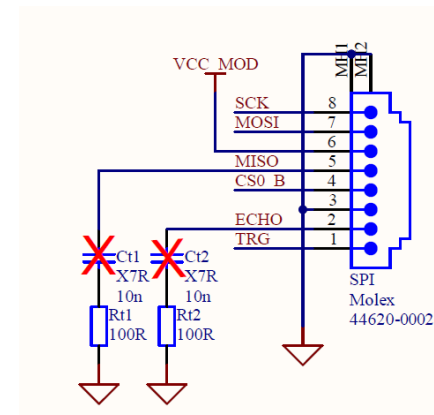
Main Interface between Base Board and Modules

- Patch cable with RJ-45 connectors
- Serial Peripheral Interface (SPI)
 - MOSI, MISO, SCLK, CS_B
- Two additional GPIO signals
 - TRG, ECHO
 - Async. Signals
- 5.5 V power supply



GPIO pin	Function Name	Description
4	TRG	Send trigger to module
5	ECHO	Receive response from module
8	CS_B	SPI chip select
9	MISO	SPI data from slave to master
10	MOSI	SPI data from master to slave
11	SCK	SPI clock

GPIO signal mapping



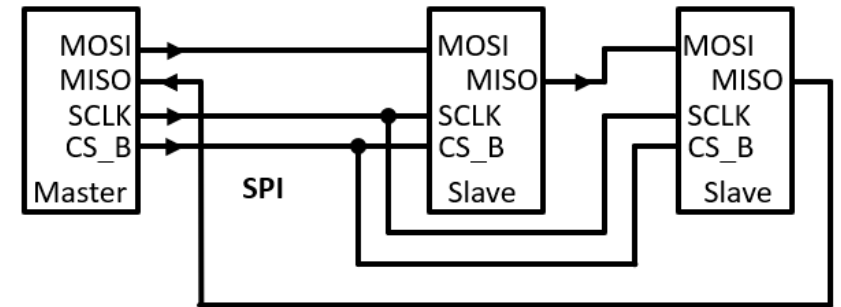
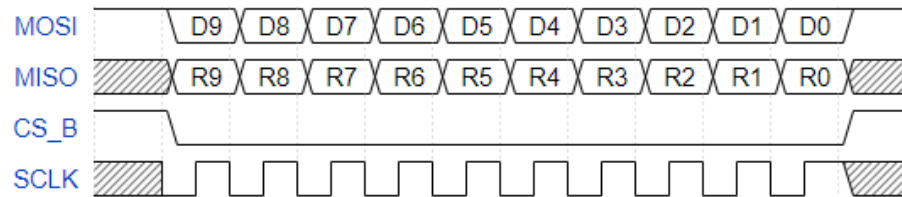
SPI connector pinout

Serial Peripheral Interface Bus - SPI

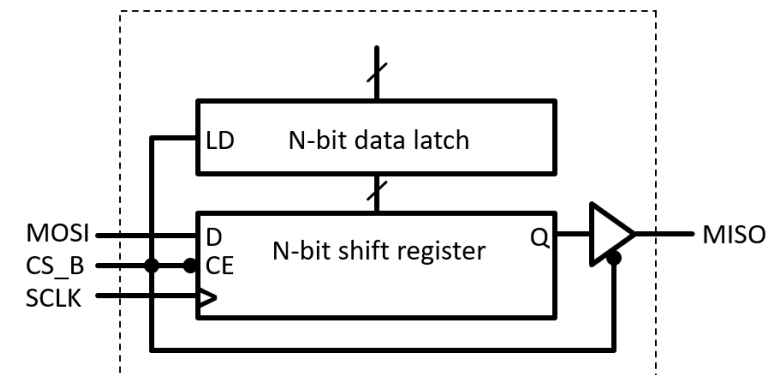
- Serial bus which typically uses four wires:
 - MOSI (SDI), data line from master to slave (master out, slave in)
 - MISO (SDO), data line from slave to master (master in, slave out)
 - SCLK, clock line from master to slave(s)
 - CS_B, chip select line (active low, one per slave, or single for daisy-chained slaves)

- Synchronous operation

- Data line synchronized to a clock signal



SPI master with two slaves



SPI interface details

- Serial shift register
- Data latch
- Output buffer (tri-state)