Specification and Verification of Temporal HAL-API Dependencies

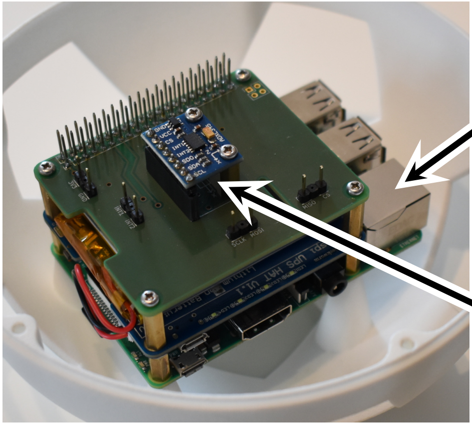
Manuel Bentele

University of Freiburg

Hahn-Schickard

15th Alpine Verification Meeting (AVM'23)
September 13, 2023

How does the Embedded System look like?



Raspberry Pi 3 Model B+
(single-board computer)

Data transfer via SPI

ADXL345 accelerometer
(sensor)

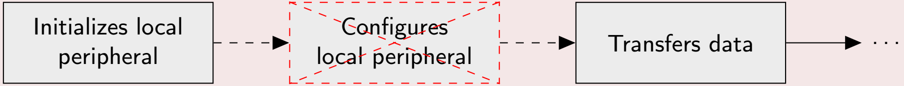
- C program on Raspberry Pi 3 Model B+ reads data from sensor using SPI

Why does the C program not transfer data properly?

C program for ADXL345 accelerometer

- Transfers data from a local to a remote SPI peripheral and vice versa
- Uses an API to control the local SPI peripheral
- Requests and receives measured acceleration data
- Is generic-error-free
- Is compilable and executable

- Does not configure the local SPI peripheral for a proper data transfer before a data transfer takes place:



How does the Serial Peripheral Interface look like?

- Interface for a synchronous serial communication
- Half- or full-duplex data transfer between SPI master and slave

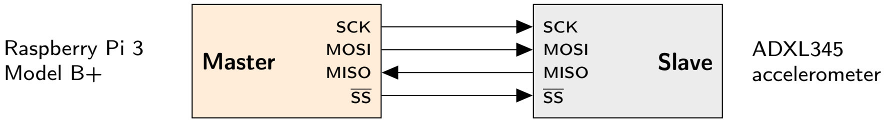


Figure: Wiring of SPI master and slave

- Operation is parameterized by configuration parameters, e.g.,
  - CPOL: Polarity of Serial Clock (SCK) during idle state
  - CPHA: Phase of SCK for data sampling
- SPI and its configuration parameters are not standardized
- Transmission errors may occur if configuration parameters are set improperly, e.g., mismatch of CPOL and CPHA from SPI master and slave

How does the Hardware Abstraction Layer for SPI look like?

- Hardware Abstraction Layer (HAL) is part of the Linux kernel
- Abstracts SPI peripherals and exposes them in user space
- *spidev* HAL-API consists of the POSIX routines
  - `open()`, `close()`
  - `read()`, `write()`
  - custom `ioctl()` routines, e.g.,
    - `ioctl(MESSAGE)` to perform a full-duplex data transfer
    - `ioctl(WR_MODE32)` to set CPOL and CPHA of SPI peripheral

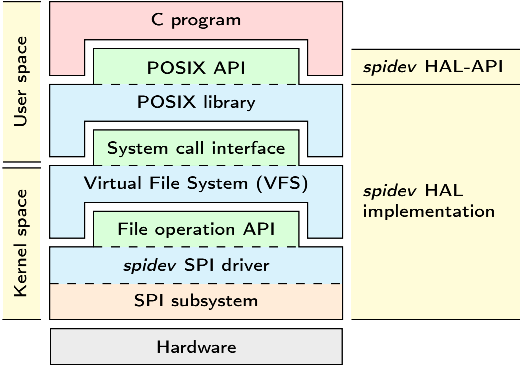


Figure: Overview of software layers from HAL *spidev*

Is there any dependency between two HAL-API routines?

Dependency

Describes the relation that some HAL-API routine *depends on* a previously performed HAL-API routine.

Example

The HAL-API routine `ioctl(MESSAGE)` *depends on* the previously performed HAL-API routine `ioctl(WR_MODE32)`.

Example

The HAL-API routine `ioctl(WR_MODE32)` *depends on* the previously performed HAL-API routine `open()`.

- Observed and extracted in total 26 dependencies from the *spidev* HAL-API

What are Temporal HAL-API Dependencies (THADs)?

Syntax

- THAD $\delta : q \lhd r$ (where q, r are HAL-API routines from HAL-API A)
- THAD δ is element of THAD relation D (binary relation over A)

Example

- THAD δ_{17} : $\texttt{ioctl(WR\_MODE32)} \lhd \texttt{ioctl(MESSAGE)}$

Semantic

- Is defined on HAL-API routine sequence $s = a_1, a_2, a_3, \ldots$
- s satisfies δ iff. $\forall i \in \mathbb{N} \bullet a_i = r^{\downarrow} \implies \exists j \in \mathbb{N} \bullet j < i \wedge a_j = q^{\uparrow}$

Example

- $s_1 = q^{\downarrow}, q^{\uparrow}, r^{\downarrow}, r^{\uparrow}$ satisfies δ? ✓ (yes)
- $s_2 = r^{\downarrow}, r^{\uparrow}, q^{\downarrow}, q^{\uparrow}$ satisfies δ? ✗ (no)

Can THADs be represented graphically?

- THADs from a THAD relation D can consitute forms
- THAD form can be represented with a directed graph

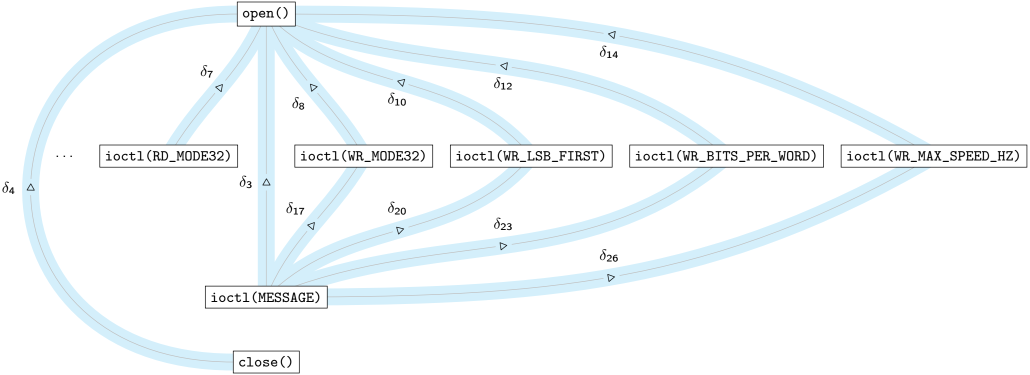


Figure: THAD form constituted by THADs from HAL-API *spidev*

How to verify THADs?

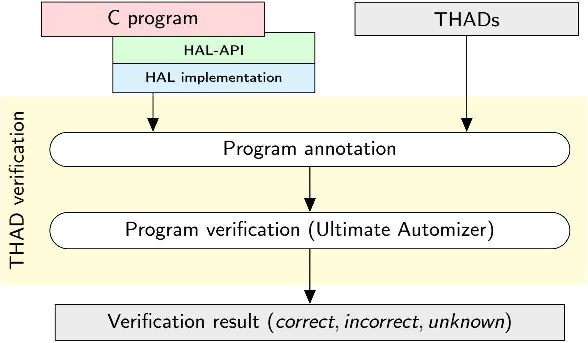


Figure: Implementation of THAD verification for C programs

How to annotate the C program?

- ANSI/ISO C Specification Language (ACSL) is used for the program annotation

Program annotation for a THAD $\delta : q \lhd r$

- Uses ACSL ghost statements (declarations, assignments, assertions)
- HAL implementation of q and r is annotated
- Is side-effect-free (according to ACSL)
- Described by the monitor automaton for δ

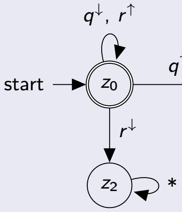


Figure: THAD monitor for δ

How to annotate the C program? (Example)

```c
/*@ ghost int state_d1 = 0; */

int open(const char *path, int oflag, ...) {
    int ret = ...;

    /*@ ghost state_d1 = 1; */
    return ret;
}

ssize_t read(int fd, void *buf, size_t nbyte) {
    /*@ assert (state_d1 == 1); */

    return ...;
}
```

Listing 1: Program annotation for THAD δ_1 : open() $\triangleleft$ read()

How to annotate the C program? (Data dependencies)

- What about data dependencies between HAL-API routines (e.g., file descriptors)?
- In theory: Extension of THADs to support parameters and return values
  $\forall \texttt{fd} \in \mathbb{N} \bullet \delta_3^{\texttt{fd}} : \texttt{fd := open("/dev/...", O\_RDWR)} \lhd \texttt{ioctl(fd, MESSAGE)}$
- In practice: Introduce an additional ghost variable to save the file descriptor

How to verify the annotated C program?

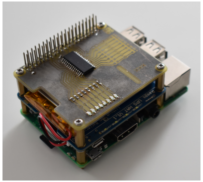
- Use a state-of-the-art software verifier (like Ultimate Automizer)
- Any verifier for C programs supporting ACSL can be used

- Verifier checks annotated C program P_D
- and outputs verification result *correct*, *incorrect*, or *unknown*

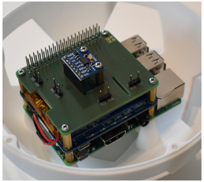
- Verification result *correct* for $P_D \implies P \models D$ (P satisfies all THADs from D)
- THAD verification with its reduction is sound

How is the evaluation of THAD verification done?

- THAD verification is applied to three real-world C programs using the *spidev* HAL-API
- Total time and memory consumption (resource usage) is measured on a commercial off-the-shelf desktop computer[1]



'I/O Expander':
Transmit data to actuator
(MCP23S17)



'Accelerometer':
Receive data from sensor
(ADXL345)

'*spidev*-Test':
Test Linux kernel's
SPI Subsystem

[1] Quad-core CPU at 3.4 GHz with 8 GB memory

How do the three C programs use the *spidev* HAL-API?

- Direct use of the *spidev* HAL-API:

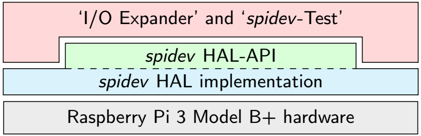


Figure: HALs used by programs 'I/O Expander' and '*spidev*-Test'

- Indirect use via third-party library, e.g., ADXL345 library:

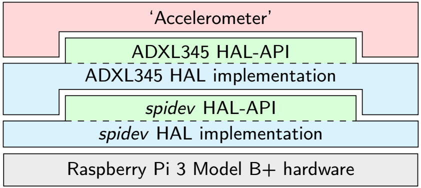


Figure: HALs used by program 'Accelerometer'

Results of Evaluation

THAD verification

- Returns correct (expected and proven) verification result
- Is applicable to C programs in the field of embedded systems
- Can also be applied to a third-party library, where the library itself uses a HAL-API
- Result is available within a reasonable time with manageable resource usage:

Annotated C program	Total time	Memory consumption
'I/O Expander'	5.74 s	383 MB
'Accelerometer'	7.64 s	458 MB
'*spidev*-Test'	26.57 s	840 MB

Table: Checking time and memory consumption of Ultimate Automizer 0.2.3 program verifications

Conclusion

What has been done?

- Created THAD syntax and semantic to formalize dependencies
- Elaborated THAD verification approach to verify THADs

Open questions?

- How easy is the THAD syntax and semantic understandable?
- Is expressiveness of THAD syntax and semantic handy?

What can be done in the future?

- Optimization and automation of THAD verification
- Refinement or extension of THAD syntax and semantic:
  - Concept of a grouped THAD
  - Regular expressions, e.g. $(a \mid b) \lhd c \lhd d$
  - Resource usage, e.g. `open()` and `close()`